EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Securing the iphion IPTV network

van Selst, J.

Award date:
2009

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computing Science
Computer Science: Security Group

Securing the iphion IPTV network

by Johan van Selst

Confidential until September 2010

Supervisors:

dr. N. Zannone (TUE)
dr. J.I. den Hartog (TUE)
J.P.P.A. Saman (iphion b.v.)

Acknowledgements

I would like to thank the following people for their assistance:

Nicola Zannone (TUE) and Jean-Paul Saman (iphion), my direct supervisors, who have read my work countless times and provided valuable suggestions and feedback throughout the project,

Jerry den Hartog (TUE), Willem Jan Withagen, Merien ten Houten and Peter de With (iphion) for giving me the opportunity to do this thesis,

Finne Boonen, Peter Minten, Erwin Riemens, Sven Berkvens-Matthijsse, Marc Olzheim, Martin Poelstra, Jilles Tjoelker and Jan Derk Gerlings, my iphion colleagues who are developing a great IPTV system,

Jaap van der Woude, for motivating me to continue my studies and finish my masters' degree,

and Wynke Stulemeijer, for her unconditional support.

Contents

1 Introduction **13**

1.1 System overview 14

1.2 Existing infrastructure 16

1.3 Contribution 17

1.4 Structure of the thesis 18

2 Requirements analysis of the iphion system **21**

2.1 Background 22

2.1.1 (Mis)use cases 22

2.1.2 Goal modelling 23

2.1.3 Goal reasoning 26

2.2 iphion use and misuse cases 28

2.3 iphion goal model 30

2.3.1 Security 32

2.3.2 Dependability 41

2.3.3 Usability 47

2.3.4 Inter-goal contributions 49

2.4 iphion goal analysis 52

2.4.1 Summary of alternatives 53

2.4.2 Configurations analysis 55

2.5 Summary 56

3 Secure communications with TLS **61**
3.1 Transport Layer Security 61
3.1.1 TLS authentication 62
3.1.2 Setup of a Public Key Infrastructure 64
3.2 Deploying TLS at iphion 65
3.2.1 Communications using TLS 66
3.2.2 Secure authentication 68
3.3 Setup of the iphion PKI 71
3.4 Key exposure threats and risks 74
3.5 Key creation, signing and usage policy 78
3.5.1 Development keys 79
3.5.2 Key generation 80
3.5.3 Key backups 81
3.5.4 Key revocation 83
3.6 TLS software 84
3.7 Summary 86

4 System integrity and secure updates **87**
4.1 Secure boot mechanism 88
4.2 Software updates 90
4.2.1 Application updates (ipkg) 91
4.2.2 Signed executables 91
4.2.3 Binary filesystem updates 94
4.3 Rescue boot procedure 95
4.4 iphion player 96
4.4.1 Secure boot mechanism 96
4.4.2 Software updates 98
4.4.3 Rescue boot procedure 99
4.4.4 Cryptographic keys 102
4.4.5 Bootloader modifications 103
4.4.6 Rescue image features 104
4.4.7 Legal issues 104
4.4.8 Summary 105

5 Content distribution **107**
5.1 Communication streams 107
5.2 Authentication 109
5.3 Integrity 111
5.4 Confidentiality 114
5.5 Summary 115

6 Conclusions **119**
6.1 Summary 119
6.2 Recommendations 122
6.3 Further work 123

Appendices

A Glossary **127**

B Use and misuse cases **133**
B.1 iphion player 133
B.1.1 Use cases 133
B.1.2 Misuse cases 136
B.1.3 Mitigation cases 141
B.2 iphion servers 142
B.2.1 Use cases 142
B.2.2 Misuse cases 146
B.2.3 Mitigation cases 150

C Key creation manual **153**
C.1 Setting up a secure certificate authority 153
C.2 Set up the U-boot certificate 161
C.3 Generating client certificates 164
C.4 Generating server certificates 164
C.5 Generating signed image files for U-boot 165
C.6 Making copies of CDs or sets of CDs for other people (not U-boot) 166
C.7 Making copies of the U-boot CD 166

Bibliography **169**

List of Figures

1.1 simplified iphion network overview 14
1.2 iphion network overview 17

2.1 Misuse case example diagram 23
2.2 Modelling tool example diagram 26
2.3 Reasoning tool example diagram 27
2.4 iphion player use and misuse cases 28
2.5 iphion server park use and misuse cases 30
2.6 Main security goals 32
2.7 Security goal model tree 42
2.8 Main dependability goals 43
2.9 Dependability goal model tree 47
2.10 Main usability goals 48
2.11 Usability goal model tree 50
2.12 Full goal tree model 58
2.13 Reasoning tool results 59
2.13 Reasoning tool results (cont.) 60

3.1 Simplified TLS handshake protocol. 64
3.2 X.509 certificate hierarchical overview 65
3.3 Client-server communications using TLS 68
3.4 Shared-secret authentication check 70
3.5 iphion certificate hierarchy 73
3.6 2D barcode representing a part of an RSA private key 83

4.1 Secure boot architecture 89
4.2 Rescue boot procedure 101

5.1 A simple iPAP network diagram 108
5.2 iphion Token and Key Protocol 111
5.3 Flow of tokens and keys 116

6.1 iphion communications security 121

List of Tables

2.1 Misuse case table overview 24
2.2 iphion player use case: Authenticate 29
2.3 iphion server misuse case: Obtain content 31
2.4 Input configuration values 56
2.5 Goal result configuration values 57

3.1 iphion intermediate certificate use 73
3.2 iphion key pair storage 74
3.3 iphion key pair misuse 76
3.4 NIST SP 800-57 key strength comparison 78

1. Introduction

In an Internet Protocol Television (IPTV) system a digital television service is delivered over a public network infrastructure using the Internet Protocol. Due to the bandwidth required for high quality television streams, IPTV is best suited for broadband connections or local area networks. Currently, IPTV is mostly used in controlled private networks such as hotels (hospitality market).

The start-up company *iphion* aims at bringing IPTV to the homes via the public Internet. Their IPTV service will be just as easy to use as 'conventional' television, but with the added benefits of interactive Internet technology. The most important advantages of IPTV are the excellent visual quality, potential low costs for the user and flexible channel access schemes.

Providing a high-quality IPTV experience to thousands of users simultaneously involves the transmission of a huge mount of data to all recipients with a very short delay. Just delivering this data in time, poses an interesting technical challenge. Rather than sending a full copy of the stream to each customer (unicast), *iphion* decided to use a collaborative delivery network, where local relays and receivers may send on parts of the data to other local peers, a method similar to the exchange of data via peer-to-peer networks [Poe08].

The security of the content delivery network is not a trivial task. The TV receivers are located in an environment outside *iphion*'s control (at somebody's home, connected to the Internet, possibly behind a firewall), but will still need to communicate secure, fast and reliable with other similar devices and with the *iphion* servers. In addition, television content providers strongly insist that their material is never made available to anybody – not even the customer – before it is delivered to the television in raw (uncompressed) form.

In this chapter we start with an overview of the collaborative IPTV system that is envisioned by *iphion*. Then we discuss the existing infrastructure: aspects that have already been implemented or are currently under development. This infrastructure sets the context for the thesis.

Section 1.3 describes the specific goals of the masters' project. We conclude this chapter with an outline of the rest of the thesis.

1.1 System overview

Before we can talk about securing information and communications, it is important to get an idea of what kind of data is used in the system and how it is communicated, both internally and externally.

Streaming multimedia content will be provided by various partners and suppliers, such as SBS and RTL Television. Each content provider will offer one or more television channels. Ideally the content would be received in a format that is directly suitable for streaming via *iphion*'s content delivery platform, but in some cases the data may need to be transcoded.

The multimedia content streams will be sent from a central *iphion* server to the customers via private channels over the public internet. At home the customers have the *iphion* player set-top box that they bought, which is connected both to their internet link and to their television set. The player receives the multimedia content streams and other information, such as an electronic program guide, and displays this information at the users' request on their television screen. A simplified overview of the communications path is given in Figure 1.1.

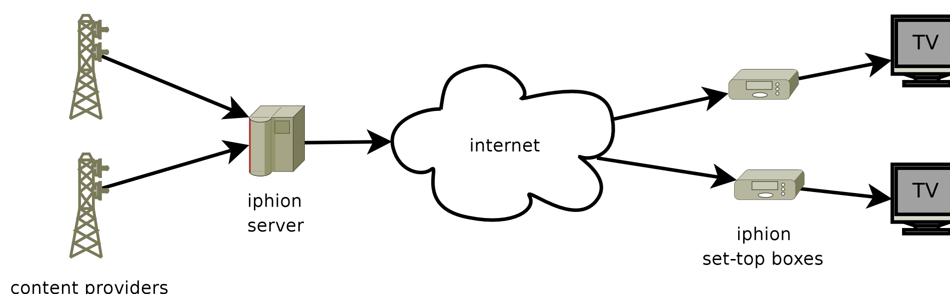


Figure 1.1: simplified iphion network overview

In reality it is slightly more complicated: the multimedia content streams will be transmitted via a delivery network of *iphion* controled nodes that relay the data to all the customers that request it. Relay servers will be placed on strategic locations within the network, for example at an Internet Exchange point such as the AMS-IX or BNIX — or in the server room of an ISP to provide a proxy for all the customers in their network. Customers' set-top boxes can also relay chunks of data to other clients in their area that are waiting for the same content (this is the collaborative aspect of the data distribution). By sending the data in chunks via various paths,

congestion of specific links can be avoided, while speed and reliability may be improved.

The regular television content (specifically that from foreign broadcasters) may be merged with commercials from local advertisers. These commercials would replace the commercial slots in the original broadcast. This could be used for example to broadcast an Australian sports channel with Dutch commercials, or to present customers in the Netherlands and in Belgium with the same shows, but with different advertisements from local companies during the commercial breaks. This feature is only available for channels where the content provider explicitly supports it.

Multimedia data that is not sent out live, will be stored on *iphion* servers and only sent-out to individual customers upon request. However, this feature will probably not be available at the start. The exact details of how Video-on-Demand will be deployed are not clear yet. With a limited amount of timeslots and pre-selected videos, the regular content delivery network might be used to distribute this data as well.

Apart from pure streaming multimedia content, *iphion* receives and distributes data from other sources as well. This will include information such as program guide details, interactive television, teletext and other digital services, but also software updates for the system. Most of this data will not be send out via the distribution network, but is communicated directly with individual customers in an interactive fashion. The customers can select specific information whenever they feel like it. To obtain this information, the set-top box directly contacts a specific *iphion* server.

The set-top boxes will not only obtain data from the *iphion* servers, but will send information back as well. This includes technical data about what version it runs, how the system operates, what kind of special events occur (errors, hack attempts etc.), but also regular statistical data about how the system and the services are used.

Viewing statistics about the requested multimedia content and other services will be collected by the company. This may be used to provide customers with personal recommendations (related TV shows, timetable changes, etc.). The content providers are also very interested in how many people tune in to their channels and when viewers join or leave a specific channel. Aggregated viewing statistics will be made available by *iphion* to the content providers via a website.

There will also be a website for the customers where they can log in and customise personal settings. Here they might order additional services or features. Additionally, settings like the order in which TV channels appear on their set-top box or the configuration of ‘parental control’ features could be modified via this interface.

1.2 Existing infrastructure

A large part of the *iphion* collaborative IPTV system has already been created. An infrastructure has been developed primarily to meet the requirements for scalability and efficiency. Security requirements are not ignored or compromised in this design, but they will have to work inside the framework that is already set up.

Video data will be streamed by a central server: this server encodes the stream for transmission over the network and splits the result into small packets that may be transmitted individually. The underlying format will be an MPEG Transport Stream [oapmhi00].

The central broadcasting server distributes the data to the *iphion* relay servers. The number of these servers will grow with an expanding network. The relays (or repeaters) will send on data to the clients, the *iphion* set-top boxes (STB). The broadcaster, repeaters and STBs take part in a peer-to-peer network as nodes. Nodes have a dynamic ranking (depending on uplink speed and relative distance to the source) and each node tries to receive data from higher-ranked peers.

Data is transmitted on-demand (i.e. pulled rather than pushed): any node can contact a central service to find out which nodes currently offer the data that it is interested in and then request a stream from them. When nobody is interested in a specific stream, it won't be send out. The STBs will only request and relay traffic for the active channel (that the user is watching), so that the bandwidth can be used fully to provide high quality video and audio.

Data packets in the network are transmitted using UDP. A mechanism to deal with transport errors, congestion control and packet re-ordering has been designed on application level. The STBs have special provisions to operate from behind firewalls and NAT-routers (Network Address Translation).

The collaborative network is only used for distribution of audio and video content: all other data is handled through separate channels. These channels generally use a direct TCP connection to a central server – as the network grows, there may be multiple servers for each service. These servers are accessed by clients to consult channel information, find peering partners, browse the electronic program guide (EPG), download software package updates and obtain other information that doesn't require high bandwidth. Figure 1.2 gives a simplified overview of the designed *iphion* network infrastructure.

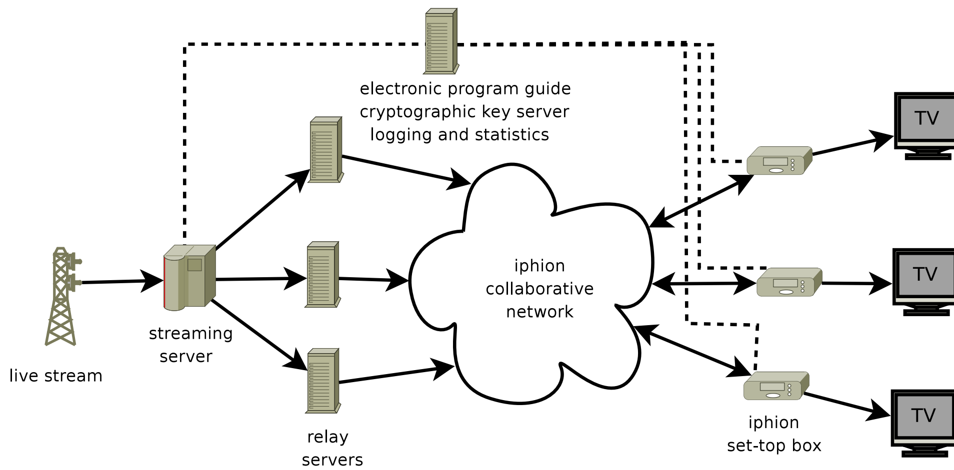


Figure 1.2: iphion network overview

1.3 Contribution

The objective of the thesis is to analyse and improve the security aspects of the *iphion* collaborative IPTV network. The analysis includes deciding what needs to be secured: Chart the communication streams, the stored data and potential risks and then decide what needs to be protected. Additionally, we need to decide how the data should be secured: Information security is not just about *confidentiality* of the data, but also requires *availability* of the data where and when it is needed and guarantees about the *integrity* of the presented information.

For some information streams integrity will be much more important than confidentiality, while for other data the availability of relevant decryption keys at the right time and place, will be a real challenge. In many cases several alternative solutions will be available and we will determine which is best suited to each situation.

Based on our requirements analysis, we aim to give concrete suggestions on how to improve the security of the *iphion* system. This includes the selection and design of suitable communication protocols, procedures for the creation and deployment of cryptographic keys and assistance with the implementation of the suggested improvements. In the thesis, three aspects of the system are analysed in great detail.

For the authentication of clients and servers in the system, we suggest the use of Transport Layer Security (TLS) in combination with a identification scheme based on public-key cryptography. We design a Public-Key Infrastructure (PKI) to implement this. TLS based communications

facilitate authentication, but also offer data integrity and confidentiality (encryption) support. We suggest the use of TLS where possible and identify the communication streams where TLS is not feasible.

To ensure the integrity of the software running on the *iphion* player set-top boxes we suggest a booting mechanism that performs integrity checks with each step (on-chip, bootloader, kernel, application software). These checks are based on digital signatures, also supported by public-key cryptography. Updates of the software running on the set-top box should preserve this integrity at all times. A rescue mechanism is designed to restore the application software install if its integrity is somehow compromised.

The confidentiality, integrity and authenticity of multimedia content streams in the *iphion* network can not be established via TLS, because of the incompatible set-up of the delivery network (a packet-oriented collaborative infrastructure). We introduce alternative mechanisms to guarantee the required security of the content data transmissions.

1.4 Structure of the thesis

This thesis is structured as follows:

- **Chapter 2** presents the requirements analysis of the system under development. The focus will be on the security requirements – but these cannot be analysed without taking the other functional and non-functional requirements of the system into account as well. From these requirements we work towards an implementation and decide on the best design alternatives.

  The following chapters each focus on specific aspects of the system. For some parts of the system we work out exactly how this aspect can be secured in practice. We deal with encountered implementation restrictions and explore the protocols that can be used: both for the software and for the operational aspects (e.g. who needs access to which encryption keys and how are these to be stored).

- **Chapter 3** describes how to use Transport Layer Security (TLS) for client and server authentication, data integrity and encryption. To support TLS communications, we set up a Public Key Infrastructure (PKI) for *iphion* with keys and certificates for all communication partners, using a centralised trust hierarchy. A usage policy details how this infrastructure can be deployed in practice.

- **Chapter 4** introduces the infrastructure that allows *iphion* to check and preserve the integrity of the installed software on the set-top boxes. This

integrity will be preserved through regular software updates. For the event that the software integrity is compromised (for example when a regular update fails to complete successfully), we devise a rescue procedure that will restore a 'compromised' set-top box to a valid, integer *iphion* software installation.

- **Chapter 5** focuses on the *iphion* Peer Assisted Protocol (iPAP) that is used to deliver streaming multimedia content to the set-top boxes. These communications do not use TCP and therefore TLS cannot be used for the data exchange. We create an alternative set-up to handle peer authentication, preserve data integrity and deal with the encryption of the content data in the iPAP network.
- **Chapter 6** presents a summary of the work that has been done and describes how this has helped to create a more secure system. This chapter will also include a description of what has not been done – either because it was deemed unnecessary or because of other constraints – and what tasks are left for future analysis and implementation work.

2. Requirements analysis of the iphion system

The *iphion* collaborative IPTV system serves a clear purpose: to display television broadcasts. To clarify in more detail what is needed for good operation of this system, a list of requirements has been specified. This includes requirements from the service provider, *iphion*, from the content providers (national TV networks) and from the users who will use the system. The objective of this chapter is to determine the requirements of the system and translate these into specific implementation tasks.

We start by looking at the functional requirements of the system: what is it supposed to do. This analysis is mostly done by discussing the system with the designers and developers at iphion. These requirements are formalised using the **Use Case** model [SC02].

Along with the use cases, it is interesting to consider how the system might be abused by people. This will help to clarify the security considerations of the system and show what additional system functions will be needed to mitigate these attacks. An extension to the traditional Use Cases, the **Misuse Case** model [SO05], will be used to describe this aspect.

Alternative design decisions can be adopted to protect the system against abuse. Each decision has a different impact on the qualities of the system. This demands a trade-off analysis in order to select the best alternative. For this purpose we use a **Goal Modelling** technique, which allows for the representation of software qualities together with design alternatives [YM94].

When a clear overview of the system has been outlined, formal **Goal Analysis** [GMNS03] will be performed using software tools, to interpret this model and use it to clarify and guide implementation choices. The analysis should identify options that make it impossible (or difficult) to reach certain goals. When conflicting aspects are found, a choice may be made between quality requirements (e.g. choosing for security or for usability). This analysis can help us choose good implementation solutions.

2.1 Background

2.1.1 (Mis)use cases

In software engineering **use cases** models are used to describe a system's behaviour in relation to its environment (the users of the system). These models are **UML diagrams** [RHCF05] that illustrate how a system is supposed to function. Each case can be recorded in detail using standard templates, thus leading to a formal definition of the functional requirements of a system [SC02]. Use case descriptions are particularly useful when one has a good idea about what a system is supposed to do, yet it lacks a formal description of the requirements.

In the use case descriptions, the **actors** are the users, or external entities that interact with the system. And each **use case** describes a specific action, or sequence of actions, that the system should perform. These actions will be triggered by the actors.

Standard use case descriptions focus on functional requirements and do not consider non-functional requirements, such as security requirements. The conventional use case descriptions can be extended to not only describe the regular (positive) use cases, but also the negative use cases (**misuse cases**) that specify behaviour *not* desired by the proposed system. From this the security requirements may be elicited [SO05].

Misuse cases constitute threats to specific use cases: when the actions misuse case would be execute, it could undermine or disable the desired functionality described in a use case. For example, the use case *order goods* that describes a function of an online store, could be 'threatened' by the misue case *steal credit card info* (Figure 2.1). The actor in a misuse case is generally a malicious outsider who wants to attack the system. However, this actor could also be a regular user who wants to abuse the system.

We add extra use cases to describe the actions that can be taken to mitigate the attack potential of a misuse case. These use cases are called **mitigation cases**. In the previous example, the misuse case could be mitigated by the additional use case *protect card info*.

A graphical overview of the misuse cases will be presented using UML diagrams. The basic elements of misuse cases are: actors, cases and relationships. Actors are the users of a system, such as iphion customers and partners; but crooks attempting to hack the system are actors as well. Actors are depicted as a person in the diagram (even if it may be used to describe a company). The cases are either use cases (white boxes) or misuse cases (black boxes). The mitigation cases are use cases as well and are therefore also represented by white boxes.

The relationships (shown as arrows) describe the dependencies between actors who may trigger an action and the corresponding use case. Use cases and misuse cases also have relationships between them: a misuse case generally **threatens** one or more use cases; use cases can **mitigate** a specific misuse case (these use cases are referred to as mitigation cases). Specific use cases may **include** other use cases, for example to mitigate threats to this specific case. The relationship arrows between use/misuse cases will always include a text to describe the type of relation: threaten, include or mitigate.

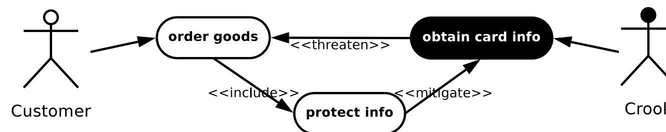


Figure 2.1: Misuse case example diagram

After that a more formal description of each use case and misuse case will follow, using a standard template. This details the actions that compound a use case, the event that triggers the use case and the threats to the use case that can be identified.

For the misuse cases the description will also include the stakeholders and risks of an attack; the potential misuser profile (well-informed insider, unskilled external opportunist, etc.); and references to the mitigation cases that may prevent the attack. An overview of all items listed in the use case description is outlined in Table 2.1.

2.1.2 Goal modelling

Goal-oriented approaches offer a way to describe the requirements of a system. A **goal** describes a desired state of affair. **Softgoals** are used to model non-functional requirements for a software system [GMNS03]. Softgoals are goals that do not have a clear-cut criterion for their satisfaction: they are considered satisfied when there is sufficient positive and little negative evidence for this claim.

These softgoals can be refined and split up into subgoals that describe specific aspects of a general principle. These subgoals can be fulfilled by means of tasks. A **task** represents a particular course of action that produces a desired effect [MMZ07]. Essentially they are used to describe the concrete procedure that accomplishes a goal. For example, the softgoal ‘data confidentiality’ may be (partially) refined into the task ‘encrypt data communications’, which could be implemented using ‘symmetric AES encryption’.

Entry	Description
Name:	Use case identifier
Summary:	Short description of the use/misuse case.
Basic path:	The actions that the actor(s) and the system go through to harm the proposed system.
Alternative paths:	Actions that are not accounted for by the basic path, but are still sufficiently similar to be described as variants of the basic path.
Exception path:	Actions that interrupt the basic path and lead to a different result without completing the basic path.
Trigger:	The states or events in the system or its environment that may initiate the use/misuse case. For some cases, the trigger is just the predicate True, indicating that this event may occur at any time.
Assumptions:	The states in the system's environment that make the use case possible.
Precondition:	The system's state that make the use case possible.
Postcondition:	The resulting change in state after the completion of the basic path.
Threats[a]:	Misuse cases that threaten this specific use case.
Mitigation points[b]:	Those actions in a basic or alternative path where misuse can be mitigated.
Mitigation guarantee[b]:	The guaranteed outcome of mitigating a misuse case. If mitigation points are not yet specified in detail, the mitigation guarantee describes the level of security required from the mitigation security use cases that will be designed later.
Related business rule[b]:	The business rules that will be violated by the specified misuse.
Potential misuser profile[b]:	This field describes whatever can be assumed about the misuser, for example, whether the misuser acts intentionally or inadvertently; whether the misuser is an insider or outsider; and how technically skilled the misuser must be.
Stakeholders and risks[b]:	The major risks for each stakeholder involved in this misuse case. This may be an abstract textual description, e.g. "the system is unavailable for several hours".

[a]Only included in the use case template
[b]Only included in the misuse case template

Table 2.1: Misuse case table overview

AND/OR-decomposition is used to describe the relation between a goal and its subgoals [ALF93]. If a goal is AND-decomposed, then all the subgoals must be met in order for the goal to be satisfied. OR-decomposition is used to indicate that at least one of the subgoals must be satisfied.

Unfortunately the decomposition-approach for modelling and analysing goals does not work for many domains, where goals can't be formally defined and the relationships among them cannot be captured by semantically well-defined relations such as the AND/OR-decomposition [GMNS03].

Abstract goals such as *user satisfaction* cannot be fully defined with formal predicates and the relationship with other (sub)goals is hard to capture using only AND/OR-relations; even though there may be necessary conditions that must be met for this goal to be satisfied. Furthermore such an abstract goal can also be related to other general goals, such as the *effectiveness* of a system. The latter contributes towards satisfaction of the former goal, but this contribution is partial and qualitative.

For another example, let's assume that we have a goal that can satisfy a *security* aspect, such as *user identification*. The implementation fulfilling this goal is likely to have negative effects on *usability* goals, such as *efficiency*. But this too is just a partial contribution: satisfaction of the former goal does not deny satisfaction of the other, although it may make it harder to obtain. Generally we do not strive to an optimal engineering solution for efficiency, but rather strive for an adequate compromise, accepting a solution that is "good enough" in (partial) satisfaction of multiple conflicting goals.

The original goal model can be extended with qualitative relations [GKMP04]. This makes it possible to describe subgoals that contribute to the satisfaction of a goal, but do not guarantee fulfilment of the goal: these contributions can be partial and qualitative. The added qualitative relations may indicate either positive (+) or negative (-) contributions.

The intuitive meaning of these contributions is that the satisfaction of goal contributes positively (negatively) to the satisfaction of another goal. A formal definition is given in [GMS05].

Graphically, softgoals are represented as clouds in goal diagrams, tasks as hexagons and resources as rectangles. In the rationale diagram, contributions are specified as arrows with a plus or minus sign (see Figure 2.2).

A goal graph can be seen as a forest of and/or-trees whose nodes are connected by contribution relation arcs. Root goals are roots of and/or-trees, whilst leaf goals are either leaves or nodes which are not part of the trees.

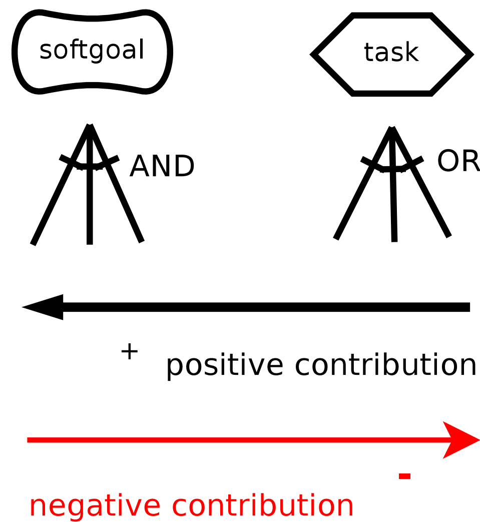


Figure 2.2: Modelling tool example diagram

2.1.3 Goal reasoning

Initial values represent the evidence available about the satisfaction and the denial of a specific goal, namely evidence about the state of the goal. For each goal we considered three values representing the current evidence of satisfiabilty and deniability of a goal: *Full*, *Partial* and *None*. We admit also conflicting situations in which we have both evidence for satisfaction and denial of a goal.

Given a goal graph and an initial values assignment of the so-called *input goals* (typically leaf goals), **forward reasoning** focuses on the forward propagation of these initial values to all other goals in the graph.

After the forward propagation of the initial values, the user can look at the final values of the *target goals* (typically root goals). The desired outcome is satisfaction of all of the target goals. Forward reasoning is used to evaluate the impact of the adoption of different alternatives with respect to the softgoals of the system-to-be.

Backward reasoning focuses on the backward search of the possible input values leading to some desired final value, under desired constraints. We set the desired final values of the target goals, and we want to find possible initial assignments to the input goals which would cause the desired final values of the target goals by forward propagation.

The backward reasoning is used to analyze goal models and find the set of goals at the minimum costs that if achieved they can guarantee the

achievement of the desired top goals and softgoals. In other words, we find among the alternatives of the goal model those with the minimal cost that allow us to obtain our desired goals.

Analysis tools are available to automate part of the goal reasoning process. We will be using the *Serenity Si* plugin* for Eclipse [Bon07]. These *Si** tools offer means to draw goal graphs in Eclipse, using the previously described composition- and contribution-relationships.

For the goal analysis we use the Serenity Goal-Risk Solver tool [Asn08] that works with the Serenity plugins in order to facilitate forward reasoning.

In this analysis a (sub)subgoal is considered satisfied (indicated in a green colour; as shown in Figure 2.3) if all dependant subgoals are satisfied in the case of AND-decomposition; or if at least one of the subgoals is satisfied in the case of OR-decomposition.

A subgoal with only positive contributions is also considered satisfied; a subgoal with only negative contributions is considered to be denied (indicated in red). If a subgoal has both positive and negative contributions, it is considered both partially satisfied and partially denied (indicated in yellow). Since we are doing qualitative analysis, rather than quantitative, there will be no percentage or threshold for what is 'good'. In general, yellow cases warrant closer inspection outside the scope of this tool.

Finally, a goal may also have no contributions whatsoever (indicated in white), which implies that it is neither satisfied nor denied; but since we strive to satisfy all goals, this is not a desired result either. The reasoning for the satisfaction/denial of tasks is the same as that for subgoals.

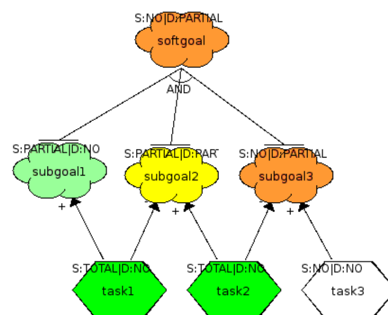


Figure 2.3: Reasoning tool example diagram

As input values for the leaf nodes we will use '1' to indicate that an implementation is chosen (satisfied) and '0' for those that are not used. The best result would be if the top goals then colour green when running the automated analysis. If any goal turns red (goal denied) or white (no

contributions), then we have done something wrong. If it turns yellow (both positive and negative contributions), then this is cause for further analysis.

2.2 iphion use and misuse cases

Use cases and misuse cases for the *iphion* system have been identified by reading the available documentation about the existing and planned system, and mostly by interviewing the management and developers at *iphion*. They presented a clear view of what kind of misuse of the box – and especially of the content and other data in the system – should be prevented.

The **iphion player** is the set-top box that customers will buy from iphion. The customer places this box in his home and connects it to their television and their internet uplink. The box will fetch video content and display it via the television set. These are the main use cases for the users of this device.

Authentication (of the device) is required to determine exactly which content may be displayed; software updates are required to implement new features or additional security safeguards. Encryption and validation of all data communication, should prevent both unauthorised use of sensitive data by third-parties and the use of untrusted input data by regular iphion players.

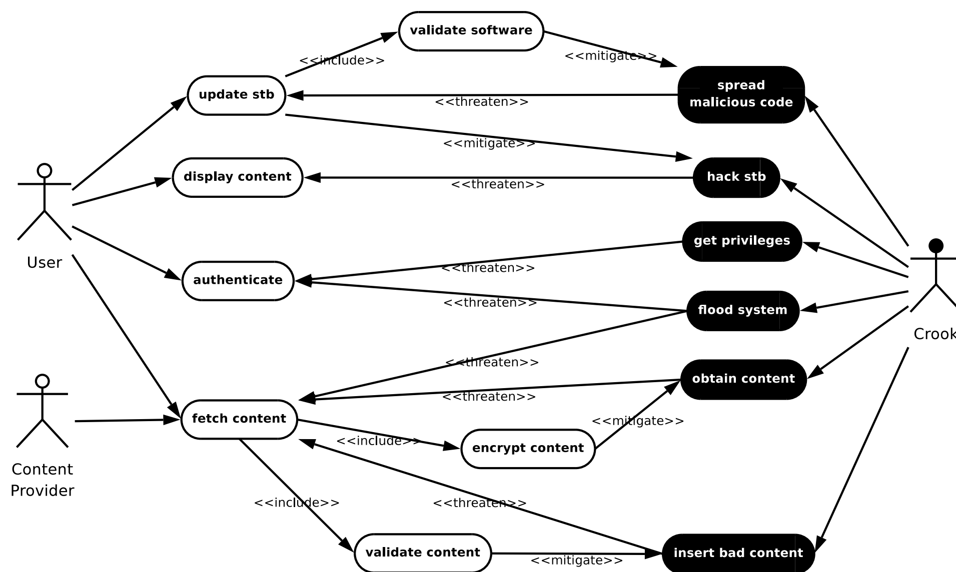


Figure 2.4: iphion player use and misuse cases

Figure 2.4 shows the use and misuse cases that have been identified for the *iphion* player environment. All these use cases have been described in detail using the standard template (from Table 2.1). We only present the details of one of these use cases here, in Table 2.2, the others are listed in Appendix B.1 in full.

Name:	**Authenticate**
Summary:	The iphion player (client) authenticates itself to the server and receives a secure access token for further communication.
Basic path:	1. The client sets up a secure connection to the server. 2. Client and server exchange credentials (securely) and verify each other's identity. 3. Server sends a signed session token to the client that it may use to identify itself to other actors in the system.
Alternative paths:	
Exception path:	If either client or server supplies invalid credentials, the authentication is aborted and no token is given.
Trigger:	Whenever the client sends an authentication request. It will do this when coming online or switching to another channel.
Assumptions:	Authentication mechanism should protect against replay and man-in-the-middle attacks. How this is done, is described in 3.2.2.
Precondition:	Clients and servers can check credentials of other actors.
Postcondition:	The client is now registered and is (the only one) in possession of a token that grants this specific client access to exchange information with others for a limited period.
Threats:	1. An attacker can try to steal credentials in order to obtain access for himself (B.1.2.3). Note that obtaining a token isn't enough, because all requests must also be signed with the client's private key. 2. An attacker can try to flood the system so that it cannot obtain access (B.1.2.4).

Table 2.2: iphion player use case: Authenticate

It should be noted that there is no separate mitigation case to counter *get privileges* attacks on the *authentication* process. As noted in this case description (B.1.2.3), the authentication process itself (B.1.1.1) should be resilient against such attacks: a proper authentication mechanism will not grant privileges to unauthorised users. There is no mitigation against *flooding attacks* either: an attack that sends more data than the network path to a specific player can handle, **will** cause an interruption of the service. Such an attack might be countered by filters in the network that are placed in front of the connection path bottleneck (typically the link between a customer's home and the network provider), but this is the domain of the ISP and not of *iphion*.

The **iphion servers** obtain video content from the content providers and redistribute it to the appropriate iphion players. A player should only receive the content that it has asked for. The servers will provide additional

'meta data' as well, such as an electronic program guide (to the players), account information (to the customers) and viewing statistics (to the content providers).

Crooks that attack the iphion servers can be both third-parties or customers who have direct access to an iphion player set-top box. The types of attack that can be launched against the server are pretty similar to those threatening the players: denial-of-service attacks, interception of data (passively or by actively submitting fake data first) and trying to trick a server into inappropriate behaviour; possibly even trying to take full control of the server.

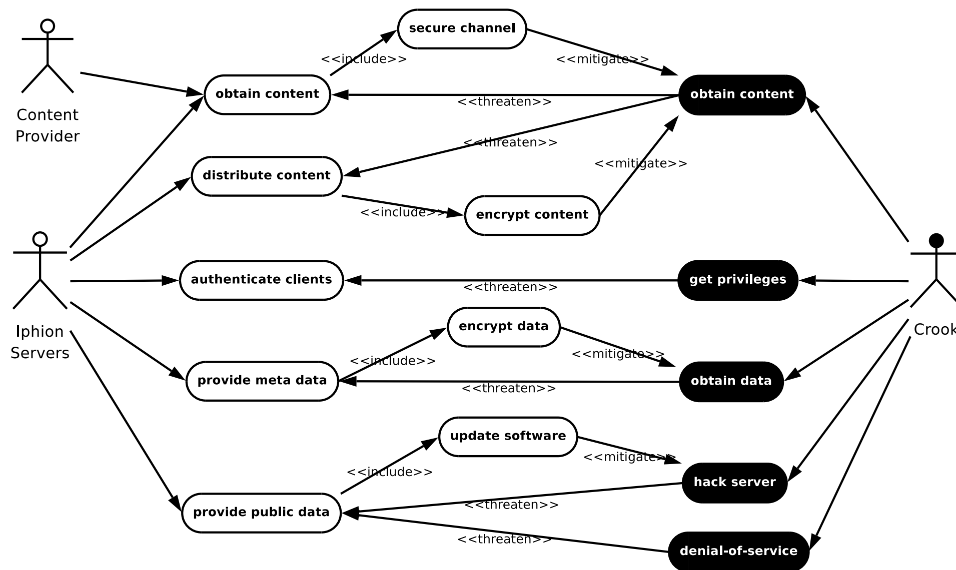


Figure 2.5: iphion server park use and misuse cases

An overview of the identified use cases for the *iphion* servers is shown in Figure 2.5 and each case has been described in detail in the Appendix B.2 for further reference. To get an impression of these descriptions, the **obtain content** misuse case is included here in Table 2.3.

2.3 iphion goal model

In this section we apply the goal modelling approach described in Section 2.1.2 to capture and analyse the requirements of the *iphion* collaborative IPTV network. For this analysis, we use a goal graph where the main goals that we focus on form the roots of a tree.

We focus on three softgoals: **security**, **dependability** and **usability**. There are other goals, such as those dealing with the overall cost of the

Name:	**Obtain content**
Summary:	Multimedia content is send from content providers to the iphion servers and then further distributed via iphion's collaborative network. At either stage an attacker may try to obtain this content.
Basic path:	1. A content provider sends data to the iphion server. 2. An attacker manages to intercept this information.
Alternative paths:	1. An iphion server sends out multimedia data to the iphion network. 2. An attacker manages to intercept this information.
Exception path:	
Trigger:	Whenever multimedia data is send by the content providers (B.2.1.1) or iphion servers (B.2.1.2). This happens continuously.
Assumptions:	The attacker can intercept or relay data streams on its own system.
Precondition:	-
Postcondition:	Restricted multimedia content data ends up with an unauthorised party.
Mitigation points:	1. When obtaining media from content providers (B.2.1.1), a secure channel should be used for the media transport (B.2.3.1). 2. When distributing media from the iphion servers (B.2.1.2), all the media should be encrypted (B.2.3.4).
Mitigation guarantee:	Multimedia content is only available to authorised parties.
Related business rule:	Content is only accessible for iphion customers.
Potential misuser profile:	Skilled: The attacker must be able to intercept and/or reroute internet data streams.
Stakeholders and risks:	Iphion: Full content access might be obtained by people who never paid for it.

Table 2.3: iphion server misuse case: Obtain content

system, but for now these are considered to be less important and are not included in the analysis. These goals are decomposed into subgoals, which may be refined even further.

The decomposition and refinement of goals into subgoals can be continued until we have tangible goals that can be satisfied through an appropriate course of action. These courses of action, or tasks, correspond with the use cases and mitigation use cases that were described in Section 2.2.

Use cases describe specific tasks that need to be achieved to obtain certain goals, but they are high level descriptions. We will also describe the concrete methods to fulfil (implement) these tasks. For such a specific implementation there may be several concrete methods to choose from; so any implementation may be decomposed yet further into sub-methods, leading to additional implementation choices.

Our goal model is thus divided into three separate layers: **Goals**, that describe the non-functional requirements, **Tasks**, that correspond with

the use cases, and concrete **Implementations**, that represent the design solutions.

The (sub)goals are satisfied if the related tasks are satisfied. Specific use cases may have positive or negative contributions to more than one goal. The implementations of a task will offer a positive contribution toward satisfaction of the use case. But a specific implementation may also have a positive or negative influence on the satisfaction of other goals.

For example, a softgoal *communication confidentiality* might be satisfied by a task *encrypt communication*, which could be implemented with *Transport Layer Security, TLS*, for which concrete implementation choices might be the application of *RSA encryption* or *Elliptic Curve encryption*.

2.3.1 Security

A common definition of information security is also cited in US federal law, as part of the E-Governance Act, [FIS02]:

> "protecting information and information systems from unauthorised access, use, disclosure, disruption, modification, or destruction in order to provide—
> - **integrity**, which means guarding against improper information modification or destruction, and includes ensuring information non-repudiation and authenticity
> - **confidentiality**, which means preserving authorised restrictions on access and disclosure, including means for protecting personal privacy and proprietary information;
> - **availability**, which means ensuring timely and reliable access to and use of information"

A graphical representation of the security goal decomposition into these three aspects is depicted in Figure 2.6.

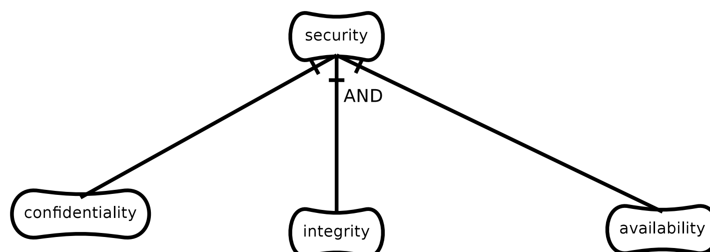


Figure 2.6: Main security goals

2.3.1.1 Integrity

To uphold system integrity, unauthorised system alterations must be prevented. This refers to both the software systems running on clients and servers, but also to the data that is exchanged between parties.

Software integrity

Software on both servers and clients will need to be protected against malicious alterations (hack server B.2.2.4, hack STB B.1.2.2). Software can not remain static for ever, but needs to be updated with bug fixes and new features. Care must be taken to validate and test software packages before deploying them: accidentally installing a package from an unreliable source – or even a broken package from a trusted source – may have grave consequences. Updating software on the servers (B.2.3.2) is relatively easy as they are in a controlled, secure environment. But updating software on the clients (B.1.1.2) requires an automated *validation mechanism* to guarantee that only trusted updates can be installed (B.1.3.1).

Content integrity

Content integrity in the system must be maintained: the content that is relayed by the network peers and displayed by the *iphion* players must be the content that *iphion* wants to distribute, and not something else inserted by a malicious party (B.1.2.6). To guarantee content integrity, components should *validate content* before using it (B.1.3.2).

Digital signatures

A common method for the validation of digital data is to use secure digital signatures [KL07]. A digital cryptographic signature of the data (checksum) will guarantee that it has not been changed (tampered with) since the data was signed. Verification of the signature, using public key cryptography, should also guarantee that the data as it is was indeed signed by the owner of the corresponding private key. Of course if a signature does not match the identity (public key) of the signer that was expected, or the data content does not match the signed checksum, then signature validation fails and the data must not be used.

There are several possibilities for the infrastructure to facilitate the processing of digital signatures. The simple way is to have a dedicated trusted party sign all the data (the TTP or **Trusted Third Party**). Validation of signatures can then be done either by this party as well, or by the individual recipients using a pre-distributed public key of the signer. This

first alternative requires that all actors maintain a secure communications channel with the trusted party. This introduces a real bottleneck in all secure communications, so only the second alternative is feasible.

A more flexible signing infrastructure allows each individual actor to generate its own signature. Since there are many actors in the system (iphion client and server machines), and new actors are introduced frequently, it is not convenient to distribute everybody's public key to all others. A better structure would be a distributed trust system, where individual signing keys are signed by a trusted authority and actors can exchange their own keys to others would be better suited. This hierarchical model is called a **Public Key Infrastructure** (PKI) [Wei01].

The de-facto standard implementation of a such a PKI used for secure communications on the internet is the X.509 standard [IT08]. The X.509 infrastructure include standards for key generation, the storage of signed keys, signature creation and validation. X.509 is also used for secure communication over the internet via **Transport Layer Security** (TLS, better known under the 'older' name SSL; [DA99]). TLS includes provisions for authenticating of communication partners and establishing secure connections using temporary encryption keys.

Setting up a PKI infrastructure requires much more work than using a single trusted party (TTP); however, once rolled out, it scales very well to larger infrastructures and both theory and practice are well-established and widely used in the industry. A further advantage is that X.509 combined with TLS also offers a good solution for actor authentication and establishing confidential (internet) communications. It is likely less work to set up an X.509 PKI infrastructure and use it in combination with the large range of existing tools, than to set up a TTP for data validation and develop our own tools to handle this.

2.3.1.2 Confidentiality

To prevent unauthorised disclosure of information, the system should ensure that users are authenticated, that the relevant information is protected and that access is only granted to authorised users.

Because multimedia content is distributed rather differently than any other data in the system, the method of protecting it will be very different as well. Confidentiality is split up into the following parts.

Multimedia content confidentiality

When distributing multimedia from the *iphion* servers to the players over the internet, it will need to be encrypted to preserve confidentiality (B.1.3.3).

Between the content provider and the *iphion* servers, it is also possible to keep multimedia confidentiality by using a secure channel as transport, B.2.3.1.

Meta data confidentiality

Meta data is always exchanged between a single client and a single server. This data can be protected either by using a secure channel (B.2.3.1) or by encrypting it directly (B.2.3.3).

Authentication

The task of **user authentication** corresponds with the use case *authenticate* (B.1.1.1). For authentication purposes it is enough to know the identity of an actor. This can be facilitated by giving all actors an *unique user identification number* (UUID). The trick is in communicating this ID with a prospective partner in a secure way, so that the partner can be reasonably sure that he is not speaking with somebody else who is pretending to be you. One way to do this is by not transmitting the ID itself, but using it as a key in a challenge-response authentication handshake [Sim96]. This type of authentication is also frequently used as a secure replacement for plain-text password authentication in internet protocols (IMAP, HTTP).

Another way to do authentication is by using **client certificates**, where the actor presents its credentials in the from of a certificate that contains its name, unique identification number and public key which can be used for public-key data encryption and digital signatures. This certificate would be digitally signed by an authoritative trusted party. The X.509 standards defines a common format that can be used to do this [IT08]. This is commonly used in actor authentication over the internet in combination with the SSL/TLS (Secure Socket Layer, Transport Layer Security) communication standards [DA99]. Since this is exactly the sort of application we are interested in, this is a good mechanism should we opt for certificate authentication. In particular, if we decide to use SSL/TLS for secure transports, it would be a natural choice to use X.509 certificates for authentication as well. This motivation will be illustrated with a positive contribution arrow in the security model, Figure 2.7.

A UUID might be implemented in the set-top box by fixing this number in a *polyfuse register* (read-only hardware) of the device. This has the advantage that it will be practically impossible to modify or fake this ID on the device. In combination with the built-in crypto hardware and a secret key that can be stored in unreadable *polyfuse registers*, we can design an authentication mechanism that reliably determine whether the *iphion* servers are really communicating with a bona fide *iphion* set-top box (see

Section 3.2.2). Unlike the UUID, the secret key could be shared by all the devices (in a certain batch). However, the use of UUIDs fixed in hardware has quite an impact on mass-production, as it means that the production process would be slightly different for each product.

X.509 client certificates are too large to be stored in a similar fashion. However these could be stored into *NOR* (read-only memory) on the set-top box instead. Unlike the polyfuse registers, the NOR has plenty room for such data, and cannot be easily tampered with. The memory might be physically removed from a device, but thanks to digital signatures on the memory and hardware checks, the device will not function properly if the NOR has been modified. Storing unique keys in NOR still means that the production process is slightly different for each box, but programming only NOR rather than polyfuse registers is much easier and cheaper. To confirm that the private key is indeed used by an *iphion* STB, the authentication check could be combined with the shared secret key described in the previous paragraph.

Content encryption

The multimedia content is delivered to the clients in a continuous stream of data packets. The content will be encrypted by the *iphion* encryption service and will be decrypted by individual clients that receive the content. Not all clients may be authorised to receive all content channels and the content must never be available unencrypted to other parties then authenticated *iphion* clients.

The choice for a content encryption method (what to encrypt) is not directly linked to the choice for a content encryption algorithm: each encryption scheme can be combined with each algorithm. Therefore, we will discuss these two aspects of content encryption separately.

Content encryption method

Encryption can be applied in several ways. The simplest way is deploying a *pre-shared key* (PSK) that can be used by all clients [AMV96]. This has the advantage that content can easily be exchanged and send further between relay servers and between peers themselves without having to decrypt and encrypt it.

Another method is to encrypt data *dynamically per client*: each client would have its own decryption key and all the data can only be read by a specific client. The advantage of this method is that it is easy to guarantee that clients will not be able to read data that is not intended for them. A disadvantage is that relaying of data requires re-encryption of it. This is

also the encryption scheme that would be used by SSL connections; however because of the way in which content data is distributed in the system, the use of SSL is not an option here.

The third method is to encrypt data *dynamically per channel*: each channel would use its own encryption key. Since data access is restricted to which actors have access to which channels, this will still make it possible to ensure that clients only get access to the data that they are allowed to read. Data can still be relayed unchanged, however only on a per-channel basis.

In each of these three cases, encryption keys can, and should be updated frequently. To obtain the necessary key, clients could authenticate themselves with a general authorisation service which would give them the required decryption key or set of keys that can be used to decrypt the data that they are allowed to read. When using the same key everywhere, this key should be rotated more frequently then when all players use individual decryption keys.

Content encryption algorithm

Regardless of the chosen method for content encryption, there is another choice in the algorithm used to implement encryption of the multimedia stream. The two serious contesters here are *AES* (the Advanced Encryption Standard for shared cryptography [NI01b]) and *DVB-CSA* (Common Scrambling Algorithm, a European standard for Digital Video Broadcasting [Com96]). AES is an open standard that has seen much scrutiny by international cryptographic experts and is at this moment considered the standard choice for both regulators and the crypto-industry. CSA has not been inspected so closely, but some reports suggest that it is not as secure as originally claimed [WW04].

DVB-CSA is the de-facto standard for the European broadcasting industry and although it is weaker, it has not been 'broken'. CSA is not an open standard, it is protected by patents and even the specification is only available through a license agreement [Cus07].

Meta data encryption

To obtain meta data information, individual clients connect directly to a centralised *iphion* server. There will be several servers for different types of data: a server with software updates, another one for the electronic program guide, etc. One way to send data encrypted to the client is by completely encrypting the channel over which the communication is sent (see Section 2.3.1.2), another way is to encrypt only the sensitive blocks of data.

The application of a secure channel is not even a feasible option for all the communications. In particular, communication that must be done from the bootloader, can only use a simple encryption scheme. The bootloader which is locked in read-only memory should be able to fetch software to perform a *rescue operation* when the software that is installed on the set-up box somehow becomes unusable (as described in Section 2.3.2.3).

For the direct encryption of meta data the same options are available as for multimedia content encryption (Section 2.3.1.2): use a *pre-shared key* or use *individual keys per client*. However, there is no distinction between content channels in this communication.

For data that is only accessible to individual clients (such as account information), general shared keys are not a good option. If we are using SSL for authentication of clients, it may be convenient to use SSL for encryption as well: SSL offers a secure data communication channel (see Section 2.3.1.2).

For the bootloader, only fixed keys (stored in ROM) may be used, but for other applications key distribution can be performed just like it is done for the content encryption. Using individual client keys for the bootloader will make the production of the set-up box more expensive and offers little security advantage, as the rescue software that would be encrypted is basically the same for all clients. The rescue mechanism is discussed in-depth in Chapter 2.3.2.3 *Updating set-top box software*.

There is no discussion about the algorithm used for the encryption of meta data. AES (the Advanced Encryption Standard, [NI01b]) is considered the best option for the cases where algorithm isn't pre-determined by the selected encryption method.

Secure channel

A secure data communications channel may be used for the distribution of multimedia content and meta data content. In this case there is a point-to-point connection between two actors and all the communication over this channel will be encrypted. This is a different concept to the encryption of data packets as discussed above.

Using secure channels for internet communication is very common. Companies use it to communicate securely between offices and people also use it for secure online banking and shopping.

There are two common mechanisms to set-up a secure communications channel on the internet: using a *Virtual Private Network* (VPN) or using *Transport Layer Security* (TLS). A VPN basically creates a large flat network on top of the existing infrastructure; but additional access controls

for individual services will need to be added on top of this. Authorisation will need to be done twice: first for joining the network and again when connecting to a specific service. TLS is useful for securing individual connections and has a little more overhead for each new connection: generally setting up a new connection means that authentication has to be done again as well.

Neither a VPN solution, nor a TLS infrastructure can be used for **content distribution**. For one because setting up direct secure TCP connections between *iphion* players will be difficult, but more importantly because it should be possible redistribute encrypted content that is received from *iphion* servers or players 'as is' to others: without changing or adding new encryption.

2.3.1.3 Availability

Keeping all sensitive data stored in a vault is very secure, but rather useless. Data must be available to authorised users and it should be available directly when and where it's needed.

Disruption of the information streams is something to be avoided. We make a distinction between the different data flows.

Content availability

The multimedia content is obtained from the content providers (B.2.1.1) and is continuously distributed to the clients via the *iphion* collaborative network (B.2.1.2, B.1.1.3).

Meta data availability

Meta data includes *iphion* account information, user settings, viewing statistics, etc. This data must be available from the *iphion* servers upon request (B.2.1.4).

Public data availability

To attract new partners and customers, *iphion* also has an infrastructure to provide non-sensitive services, such as a public website and email contact addresses (B.2.1.5).

Content delivery network

Delivery of all the multimedia data to the *iphion* player devices is the task of the content delivery network. This network should be secure, efficient and reliable. Several standard mechanisms are available to handle distribution of the data: *Multicast* is likely the most efficient solution, since data would have to be sent only once to each provider and multicast routers would then relay it to all the interested customers, copying the data stream only when it is needed. Unfortunately multicast routing is not widely supported by the existing internet infrastructure: most ISPs do not support this service. *Unicast* is the other 'extreme': send all data to every individual player. This causes a huge impact on the bandwidth cost at the sending end (the *iphion* server park) as for every client the data stream must be copied and transmitted again. The implementation can be improved somewhat by using localized relay servers (located for example at a large ISP) that receive data only once and re-distributed it to all local clients.

Another alternative is to let the players retransmit part of the data to other (local) players as well. This type of peer-assisted content distribution is used by *peer-to-peer* networks, which are popular for the distribution of large data files such as films and software images over the internet. By splitting up the data into chunks, a player could retrieve different chunks of data from different sources simultaneously. This would require all sources to have the same data (same encryption) and a format that can be easily split up into parts. There are a lot of choices left in a peer-assisted protocol: such as which transport to use (*TCP* or *UDP*), how to sort peers (fastest first), how to prioritise packets (for near-real-time streaming) and what to do with lost packets (in a video stream, dropping a few packets is okay). The background of the collaborative IPTV distribution network that *iphion* will be using is discussed at length in [Poe08].

Providing meta data

The distribution of meta data does not share many of the restrictions of the multimedia content. The amount of data is relatively limited and time-constraints on the delivery are more relaxed. Confidentiality, integrity and reliability of the data are still issues of course, but these can be solved by conventional means of providing data on the internet. All the data that the players need to access can be made available via *secure webservers*.

Secure HTTP [Res00] offers standard means of encryption and authentication (either by passwords or by X.509 certificates). And the techniques for availability scaling by adding redundant servers or local proxies are well known.

The use of secure HTTP is not limited to (internal) protocols used for communication between *iphion* players and servers. This method can also be used to provide meta data information directly to the customers (such as account information) and to content providers and partners (viewing statistics).

Providing public data

Public information that will be published to potential customers and partners will be published via conventional means: an *iphion* website. As mentioned above, keeping data available by scaling webservice solutions is a pretty standard task nowadays and this should not present any real difficulties.

2.3.2 Dependability

The original definition of dependability is 'the ability to deliver service that can justifiably be trusted'. This definition stresses the need for justification of trust. An alternate definition provides the criterion for deciding if the service is dependable: 'the dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable' [ALRL04].

Dependability encompasses the following attributes:

- **availability** - readiness for correct service.
- **integrity** - absence of improper system alterations.
- **safety** - absence of catastrophic consequences on the user(s) and the environment
- **reliability** - continuity of correct service.
- **maintainability** - ability to undergo modifications and repairs.

The decomposition of dependability into these five subgoals is illustrated in Figure 2.8. The previous chapter on **Security** already discussed the two subgoals availability (in Section 2.3.1.3) and integrity (in Section 2.3.1.1). The other dependability subgoals are described in this chapter.

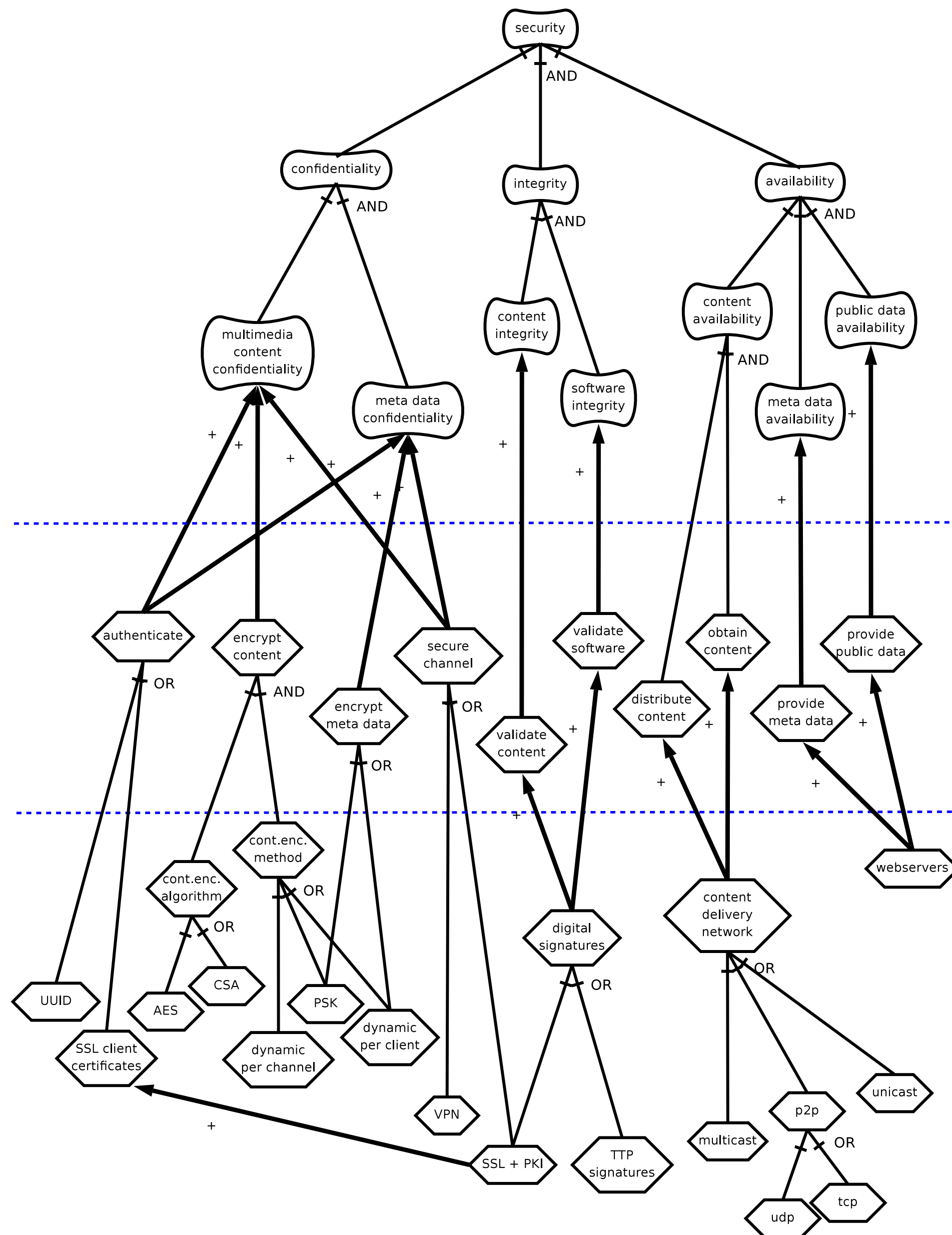


Figure 2.7: Security goal model tree

2.3.2.1 Safety

It is always possible that something goes wrong with the system, either accidentally or by malicious intent. In this case the effects for other parts of the system should be limited. Safety aspects don't concentrate on preventing bad things from happening, but on minimizing the impact when

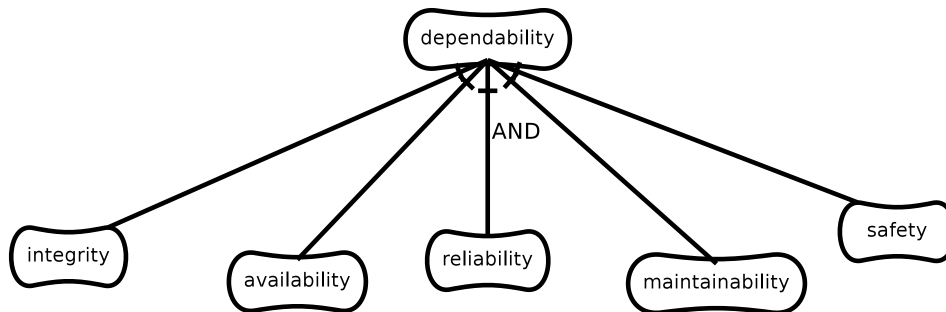


Figure 2.8: Main dependability goals

something does go wrong.

In the communication between peers in the system, corrupted messages might cause disruption of the communication or even cause certain peers to stop communication at all. This can be done by messages that either terminate a session (e.g. a shutdown message), that trigger a bug in the system (e.g. a buffer overflow) or that cause disproportional load on the system (e.g. bad video information or cryptographic calculations), etc.

Protocol hardening

To counter these disruptions, the communication protocol should be designed such that it keeps working even if a single node starts acting strangely. Peers should also inspect packages superficially before spending a lot of time processing them (check protocol version and headers before cryptographic signature, validate signature before handling video content, etc.). And since it's never possible to make a perfect protocol the first time round, it must be possible to upgrade the communication protocol later if problems are found or if better insights in how to handle certain events surface later. All these measures are part of the general label **protocol hardening**.

Update set-top box

It is important to keep in mind that protocol updates can only been handled if the software can be updated. However, part of the system that will be installed on set-top boxes, will be fixed in hardware – or at least in read-only memory. This is required for security of the system (the secure chain or trust should start with something that cannot be modified by others) and for maintainability of the STB (when a remote software update goes wrong, there should be a rescue mechanism left to return the box to a functional state).

Choices need to be made: which parts of the system should be solidly fixed and which parts may be modified by software updates that may provide continued safety. This is discussed in more detail in the section about **maintainability** (2.3.2.3), of which the set-top box updating mechanism also forms an important aspect.

2.3.2.2 Reliability

Even during normal situations, conditions occur that may affect the performance and functionality of the service. From time to time software and hardware will need to be upgraded to prevent or fix problems. Network glitches may occur, servers may need to be moved around when the network grows and clients may move from one ISP to another.

If the system is **reliable** then the service will continue to work whenever possible even when something unexpected happens or when routine maintenance needs to be done. Of course not all interruptions may be prevented: when a house loses power completely, the *iphion* player goes down and when an ISP loses network connectivity, the clients in that network cannot receive multimedia content. But on the server end, server can be set up with redundancy and in the content delivery network, connections should be able to switch over automatically to other peers when one of the peers becomes unreachable.

Service continuity

Within the range of service continuity there is a distinction between the different types of service: the player itself, the content delivery service and the availability of meta data.

To guarantee that the set-top box keeps working, it should have the ability to perform software upgrades. This means that when problems are discovered, they can be countered or prevented with an **STB update**. The update mechanism should be robust, so that it always leaves the player in a functional state when the update is done and the downtime should be minimal: although a reboot can not be avoided in some cases, the service should continue as long as possible. This mechanism is described in-depth in a later chapter.

The collaborative **content delivery network** is designed so that content can be delivered reliably, in real-time by deploying a network of both *iphion* servers and re-distribution by peers (other users) in the network. Whenever connected peers stop delivery, a player will automatically search for and switch over to other peers. This mechanism is described in [Poe08]. The

iphion content delivery servers will be set up with redundancy, so that other servers can take over when one of the machines has a problem.

For the delivery of meta data, regular **web servers** will be used. Privileged data will be served via a secure (`https`) connection, other data via regular `http` services. The methods of setting up reliable (redundant) web services are well-known and will be deployed by *iphion* to facilitate service continuity for the distribution of meta data to the *iphion* players.

2.3.2.3 Maintainability

Even though a lot of time is spent on planning of the infrastructure, the hardware and the software involved, they will not be perfect at the time of the launch. And even if it were perfect, then progressing demands and insights or the wish for new features and the issues of scaling to a larger deployment base would still trigger the need for regular updates and **maintenance**.

If the *iphion* servers are set up in a reliable (redundant) way, then it should be possible, to take one of them offline and perform maintenance on hardware or software without affecting any of services – as these should be taken over by other servers.

Update set-top box

Upgrading a set-top box will directly affect the service of this machine and interrupt the user experience. Still **STB upgrades** must be possible, to fix (potential) problems, add new features or improve performance. The system software and of the protocol that is used to communicate with others may both change over time.

The hardware itself can not be upgraded as easily. Once a box is sold, the user will be stuck with that hardware and the only foreseen 'upgrade' is to replace the entire system when something should break in the hardware. There should be operational procedures to replace broken hardware, but hopefully it won't happen often.

However there will be new hardware revisions and new boxes that offer more features than what's available on the initial release. So any software update mechanism should be able to deal with different hardware installs that require different software to run (device drivers and the like).

There are scenarios in which a regular software update mechanism may not be enough to fix a set-top box: when the updating software itself will no longer function correctly - or when the system disk will be corrupted so

badly that it no longer boots, then it will be impossible to perform the usual software update process.

This kind of corruption could have several causes:

- A bad software update which installed new software that doesn't work well (in this specific hardware environment).
- A (power) interruption during the software upgrade process: a half-installed system will not work.
- A user intentionally fiddling with the software on the system (e.g. by removing the flash drive and accessing it externally).
- A hacker/virus/worm that somehow managed to write to the disk with system software.

Several software solutions could be used to tackle this problem. Of course none of it will help if there is a problem with broken hardware; but working around software issues should be possible.

- With **network boot**, the STB fetches software from a remote server when it is switched on. This is pretty reliable (as long as the internet connectivity works, but that is required for normal operations anyway). However main problems with this solution are that the booting time will be rather long – much longer than what people experience with a regular TV – and that it will be nearly impossible to update the protocol used for the booting procedure.
- Using a **lightweight client**, only a small software environment is pre-installed on the STB which can never be updated. The software to obtain, decode and distribute content will run on a remote server. This option is rather unpractical in the designed environment and suffers from the same problem that upgrading the communication protocol is near-impossible without adaptable software on the STB.
- When **(automatic) software updates** are deployed, then all the software is pre-installed on the STB anyway. But a small hard-coded rescue environment is installed that can obtain a fresh install whenever the software becomes corrupted. As with the network boot, this means a part of the protocol can never be properly changed.

This last case is a compromise between running with a read-only image and fetching a full system image on every boot. It is likely the most complicated solution and hard to secure, but may offer a flexible and fast system if it is done properly. The solution has little overhead as long as everything works as it should, and things only get complicated when something goes wrong.

The regular software updating mechanism and the details of the software rescue procedure are outlined in Chapter 4.

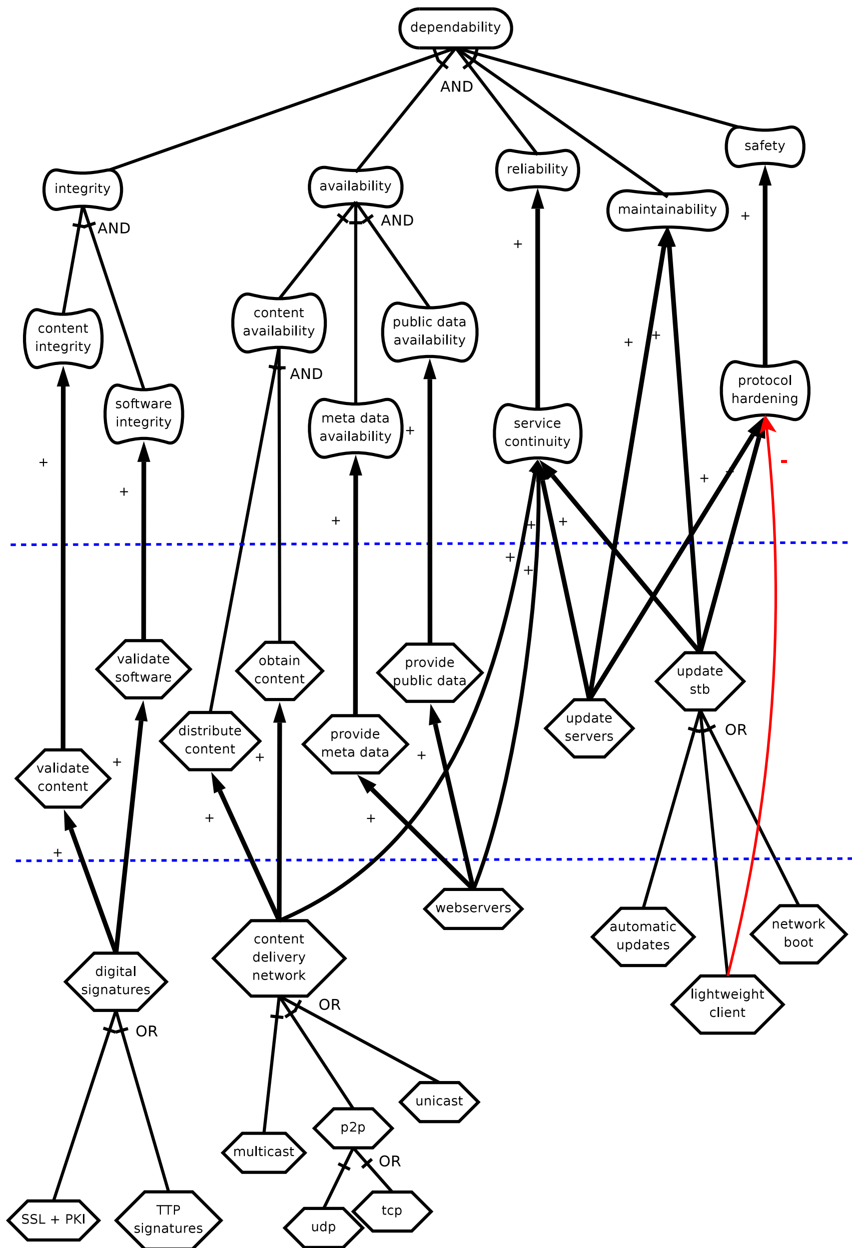


Figure 2.9: Dependability goal model tree

2.3.3 Usability

Usability is the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use [ohsi98, ohsi99]:

- **satisfaction** - freedom from discomfort, and positive attitudes towards the use of the product.
- **efficiency** - resources expended in relation to the accuracy and completeness with which users achieve goals.
- **effectiveness** - accuracy and completeness with which users achieve specified goals.

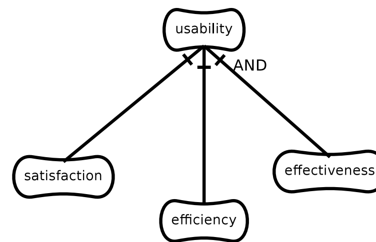


Figure 2.10: Main usability goals

2.3.3.1 Satisfaction

Satisfaction is a very important aspect. Only if customers and parters are satisfied with the product and service of *iphion*, can the project become a success. Therefore, it is important to get an idea of what others will expect of the product. This of course directly relates to the user base that *iphion* wants to target with its product.

To the **customers** *iphion* will be trying to sell something that can replace their television receiver. For this purpose, the customer system should function like existing television set extensions do (i.e. similar to a digital cable modem or satellite decoder). This means for example that start-up time and zap-delay should be short and that a reasonable range of channels should be available from the start; to give results that correspond to the existing **television experience** of the users.

By the **content providers**, *iphion* will be regarded as a broadcaster (cable company or satellite provider). Content providers will want satisfied users, they want some **viewing statistics** and they demand the guarantee that their content is protected from misuse (B.2.2.1). This can be guaranteed by either using a private network for delivery – like cable companies do – or by content encrypting such as satellite companies use. Since we are using the existing internet infrastructure for delivery, content encryption is the only option for *iphion*.

There are several ways to do this, but it will be easiest to convince content providers to trust an encryption system that they are already familiar with.

In practice this is DVB-CSA, which is widely used for satellite television distribution. Digital Video Broadcasting (DVB) is the general name for the set of European standards for digital multimedia broadcasts (via air, cable and satellite) and the Common Scrambling Algorithm (CSA) is the standard encryption algorithm for DVB transmissions [Com96].

2.3.3.2 Efficiency

The system should be operating efficiently – at least in relation to the resources spent from the user viewpoint. It doesn't matter if the *iphion* player uses resources such as time (of the user to operate the player), energy (to operate the set-top box) and bandwidth (to obtain and redistribute data), as long as this stays reasonable and delivers what the user wants: a good way to watch television.

The main indication for efficiency to the user, is (near) real-time delivery of the multimedia content (video and audio). As long as all the data can be processed and delivered to the television with a **short latency**, the perceived efficiency will be sufficient. *iphion*'s challenge is to deliver this target without spending excessive resources.

Any sensible **content delivery network** will ensure efficient means of getting multimedia content to the users. The most efficient solution seems to be multicast, however lack of support in the internet infrastructure make this solution hard to implement [Poe08]. The 'next best thing' would be peer-assisted content delivery (either via UDP or TCP). However to efficiently distribute content via unicast only, would require a larger investment on bandwidth on the server side, as this means the servers will have to send full duplicates of all channel content to each individual client that is 'tuned in'.

2.3.3.3 Effectiveness

Effectiveness of the system basically means that it should deliver what it promises: in this case it means that the customer who buys an *iphion* set-top box should be able to use it for watching television channels. The related use case that satisfies this goal is **display content** (B.1.1.4).

2.3.4 Inter-goal contributions

After considering security, dependability and usability individually, it should be noted that there are some inter-dependencies as well: tasks and choices that are made to satisfy one of these goals, may affect the other goals

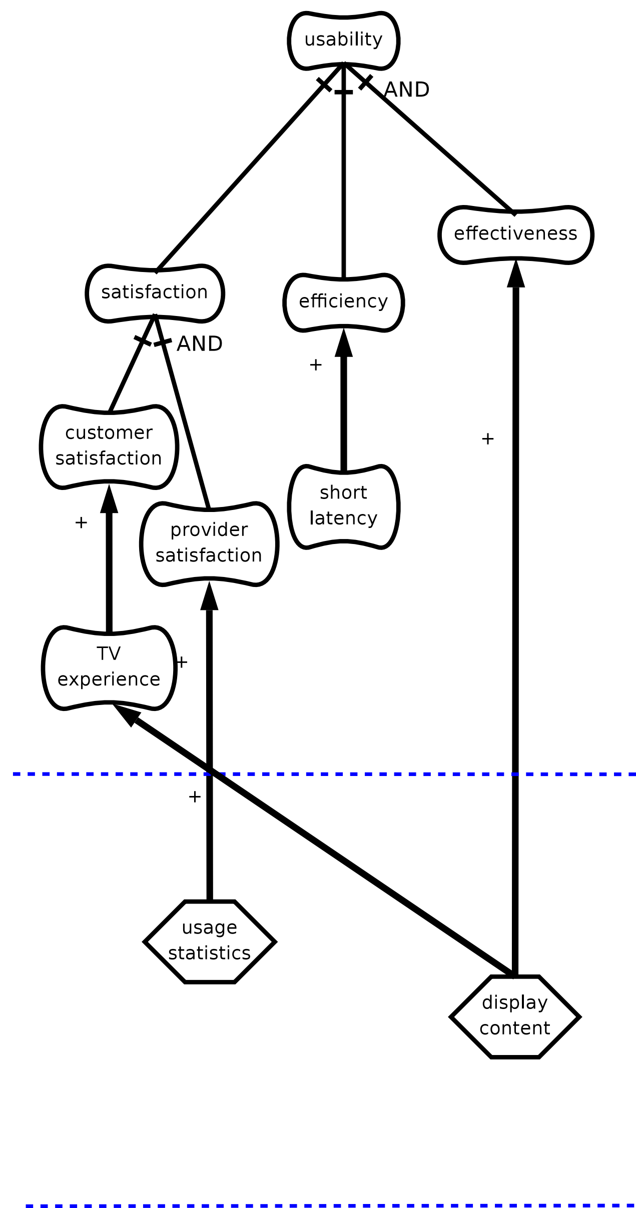


Figure 2.11: Usability goal model tree

as well. When these contributions are negative, they should be examined more closely.

Content encryption

The addition content encryption is a good example of how security can affect usability. Encrypting all the multimedia data requires additional processing

power from both servers, regardless of the chosen implementation, and may also increase the bandwidth needed to distribute all the data. This clearly has a negative impact on the efficiency of the system, in particular the **short latency** subgoal. This means that the other aspects of the system which grant a positive contribution to the latency, that is the *content delivery network* should be good enough to compensate for this, to still offer acceptable latency.

Content encryption algorithm

As discussed in Section 2.3.3.1, there is a clear preference from the content providers for implementing content encryption via DVB-CSA – even though this may not be the most secure solution. This is due to the fact that content providers are familiar with CSA: it is the standard for television distribution via satellite broadcasts in Europe and experiences there are generally satisfactory. Providers are reluctant to accept an unfamiliar alternative. Since content encryption is mostly done because the content providers demand it, *iphion* will implement the encryption algorithm they desire.

In the graphical model this preference is indicated by a negative contribution towards the **content provider satisfaction** when choosing anything other CSA. When performing the goal analysis, we will see that the choice for AES leads to partial denial of the **satisfaction** subgoal. However, this preference may shift in the future when new standards become available.

Virtual private network

Apart from the technical issues discussed in Section 2.3.1.2, there is also an expectancy issue concerning the use of a VPN: The phrase VPN implies a shared network (under *iphion*'s control). Users might not be happy to learn that *iphion* has a private network that extends to servers behind their personal firewall. Even though the actual use of this VPN would not technically differ from an implementation with TLS connections, and only be used to make the *iphion* player work as specified, the notion of a VPN may very well be perceived quite differently by the customers – and as such have a negative impact on *user satisfaction*. This expected uneasiness is much lower with TLS, because this is purely application-based it gives users the impression that they have more control.

Network boot

One way of making sure that the *iphion* players always run an up-to-date version of the software is by letting the boxes fetch a fresh copy of the

software from a central server every time they boot. This is a relatively simple solution from a security point of view. However a main disadvantage of it is that the time needed to start up a device will increase considerably. In fact, it would take much longer than other such devices that customers are used to, such as the television, a cable modem, satellite decoder or DVD player. One of the usability subgoals was to mimic the **television experience** and a netboot would have a negative effect on this expectation.

Cost

We add **minimise cost** as a softgoal in our analysis: when alternative options cannot be decided based on security, dependability and usability aspects alone, the cost of each option (in hardware, bandwidth, implementation effort, patent licenses, etc.) will often be the deciding factor that settles the choice. Most implementation tasks will have a negative contribution towards minimising cost (-), but some choices may have a much worse effect than others, which will be indicated with double negatives in the overview (--). The negative contributions have been described in the previous sections. The two main positive contributions to minimise the cost will be more paying **customers**, a likely consequence of satisfied customers, and more **advertisement revenue**, which will be boosted by television-like content display.

The resulting goal model that will be used for our analysis, including all inter-goal contributions is shown in Figure 2.12.

2.4 iphion goal analysis

Now that we have a goal graph, we will use it for our analysis. We will do forward goal-risk analysis using the tools described in Section 2.1.3. First we will draw the entire diagram in the Serenity tool and then we will analyse the satisfaction of all modelled goals by using different input configurations.

For the goal analysis, there will be a few changes in the diagram with respect to figure, presented in Section 2.3. When decomposition of different elements leads to the same alternatives, these alternatives would be shown as a single item in Figure 2.12; for example the implementation using *pre-shared keys* would be used in the decomposition of both *content encryption* and *meta data encryption*. This simplifies the overview, but when doing the analysis it is important to treat these as distinct solutions: the implementation choice for content encryption is not (necessarily) linked to the choice for meta data encryption and the trade-off arguments should be considered separately.

For simplification the choice of an encryption algorithm for the *content encryption* is not repeated three times in the overview, once for each

encryption method. Since the algorithm choice analysis is the same, regardless of the chosen method, it will only be shown and discussed once.

2.4.1 Summary of alternatives

We will now consider all the alternative choices that are shown in the model (as OR-decompositions) and give a short analysis of the available options. For each case we hope to find a best alternative: either by automated goal analysis via the tools, or by other arguments if our model is not conclusive.

For **authentication** we have a choice between using UUIDs and SSL (X.509) client certificates. Either choice will give the same result in goal satisfaction. And although choices elsewhere may have a further positive effect on the SSL implementation, there is no negative effect without this.

Initially *iphion* planned to use unique user identification numbers (UUID) for authentication of the *iphion* players (combined with a shared secret key). The content delivery protocol was already designed to use a unique ID (peerid) for client identification. However the cost of storing this ID in polyfuse hardware proved too much. This was also underscored by the automated analysis as illustrated in Figure 2.13. Once SSL/TLS became the preferred authentication method for meta data, it was decided to put a private X.509 key on the STB instead. The X.509 client certificate ID number will be used as the ID in the content delivery protocol, but the client certificate will be used for authentication; although the additional secret key is still required as well for the content delivery protocol. This means that copying the X.509 key from NOR is not enough to exchange content. The authentication process is discussed further in Section 3.2.2.

For **content encryption** the three choices are (2.3.1.2): dynamic per channel, dynamic per client or using pre-shared keys (PSK). Once again, either choice may satisfy our goals. In this case, the reason to choose dynamic encryption on a per channel basis was based on the development costs and simplification of the security model. Per client encryption would be more expensive to implement (the cost scales linear with the number of customers). Using shared keys for all data would make it more important to guarantee that the right channels only arrive with the properly authorised users; this opens up more possibilities for interception attacks. Of course even when using per channel encryption, channel data should still only arrive at the correct (authorised) destination; but this is more a usability requirement than a security requirement, since only authorised users would be able to decode the content.

The next choice is which **content encryption algorithm** to use. The choice for dynamic encryption per client (in the previous paragraph) does

not influence this choice. The two serious contesters here are AES and CSA. AES may offer better encryption, but CSA is still acceptable as well. The content providers have a strong preference for DVB-CSA (as explained in 2.3.1.2) and that is the deciding factor here.

For the implementation of a **secure channel** the choice is between a *VPN* (Virtual Private Network) and *SSL+PKI* (a Public Key Infrastructure using the Secure Sockets Layer). The choice for SSL has a positive effect on other aspects of the system that could use an SSL-based solution. However, the VPN alternative does not have negative effects on any of the security goals either. The VPN does negatively effect the customer satisfaction, which is clearly shown by the reasoning tool as well (Figure 2.13).

Validation of multimedia content and software updates will be done using a **digital signatures** scheme: either via a *Trusted Third Party* or by using a *Public Key Infrastructure*. Neither option causes problems for other parts of the system and in practice the difference between those two methods is quite small. By itself the TTP option could be a simpler solution (see Section 2.3.1.1). But if we are going to use SSL+PKI for other parts of the system anyway, it would make sense to use this mechanism to implement digital signatures as well: the work to add digital signature functionality based on a PKI when all the PKI ground work has been done, is minimal.

Considering the number of places in this system where authentication, data validation and encryption would benefit from a SSL implementation with a supporting public key infrastructure, this looks rather promising. I gave a presentation at *iphion* on how such a public key infrastructure might be set up and used in the system [vS09]. It was then decided to implement the required infrastructure. Public and private keys will be issued to all clients and servers in the system under a central certification authority. Each key can be used to digitally sign data as well and the certificates and signatures can be verified using the PKI. A full description of the public key infrastructure for *iphion* is given in Chapter ??.

In the implementation choice for the **content delivery network**, one option (unicast) is clearly more expensive than the others: bandwidth usage scales linearly with the number of clients. As for the other options, there is no clear winner to be determined by the aspects that we consider. The arguments that do determine the best option are largely outside the scope of this study, but they have been covered extensively in [Poe08]. The option that was chosen is a *peer-assisted* solution using *UDP* as transport layer.

The last choice that we consider is how to keep the software running on the **set-top box up-to-date**. The three options are using automatic updates, a light-weight client or a network boot mechanism. The *light-weight client* solution make protocol hardening difficult, because the part of the protocol that handles the initial communication between the client and its computing

server can never be modified. A *network boot* would have a negative impact on the 'television experience' of the customers, as hit would cause long start-up delays. So the best option is a system that providers for (automatic) *software updates* on the STB itself. The effects of the light-weight client and the automatic update alternatives are shown in Figure 2.13.

2.4.2 Configurations analysis

This section presents the analysis of some configurations in the goal-analysing tools. Each configuration represents a set of choices for the alternatives discussed above. Although all configurations have been considered, we only include just three possible configurations here, to illustrate the results. Table 2.4 lists the input values for the tested configurations. In this overview the value **1** indicates a selected alternative for the input values. The result of the automated analysis is shown in Table 2.5 and graphically in Figure 2.13.

The first configuration $Conf_1$ uses a light-weight client as the alternative for maintainability of the set-top box, the other configurations use automatic updates. $Conf_2$ uses a virtual private network solution for the confidential distribution of meta data and unique IDs (UUID) for client authentication, while the others use the SSL solution for confidentiality and authentication. Finally, $Conf_3$ is the alternative that gives the best over-all results for each goal in our system.

We conclude this section with a reflection on the configuration that gives the best results, as shown in Figure 2.13(c). It is good to see that there are no more red (goal denied) or orange (partially denied) boxes left. But there are some yellow results (partially satisfied, partially denied), for which our qualitative analysis does not give a conclusive result.

Our analysis indicates that latency is still affected by the cost of content encryption. Here is a trade-off between security and usability, and it had been determined that the system really needs this security aspect. This latency problem will be countered by the efficiency of the content delivery network, but our general analysis cannot tell if this is good enough: the final implementations should take care of this. It is also clear from the analysis that many implementations have a negative effect on minimising the cost of the system – as one would expect. Where otherwise equal options were presented, we did choose the one that helps minimising the overall cost of the system. A closer analysis of the costs of the *iphion* system is outside the scope if this document and will be done by others.

Input values	$Conf_1$	$Conf_2$	$Conf_3$
authentication			
UUID	0	1	0
SSL client certificates	1	0	1
content encryption algorithm			
CSA	1	1	1
AES	0	0	0
content encryption scheme			
pre-shared key	0	0	0
dynamic per client	1	1	1
dynamic per channel	0	0	0
encrypt meta data			
pre-shared key	0	0	0
dynamic per client	1	1	1
secure channel			
VPN	0	1	0
SSL + PKI	1	0	1
digital signatures			
TTP signatures	0	1	0
SSL signatures	1	0	1
content delivery network			
unicast	0	0	0
multicast	0	0	0
peer-assisted	1	1	1
webservers	1	1	1
update set-top box			
automatic updates	0	1	1
light-weight client	1	0	0
network boot	0	0	0
update servers	1	1	1
usage statistics	1	1	1
display content	1	1	1

Table 2.4: Input configuration values

2.5 Summary

In this chapter the general requirements for the *iphion* system have been elicited and analysed. The misuse case model has been used to identify security requirements and the goal model was used to identify the relationships between requirements – sometimes positive and sometimes negative. As a result of this some implementation choices have been shown

Softgoals	$Conf_1$	$Conf_2$	$Conf_3$
security			
confidentiality	S	S	S
integrity	S	S	S
availability	S	S	S
dependability			
reliability	S	S	S
safety	S/D	S/D	S
usability			
satisfaction	S	S/D	S
efficiency	S/D	S/D	S/D
effectiveness	S	S	S
cost			
more customers	S/D	S/D	S
advertisement revenue	S	S	S

Table 2.5: Goal result configuration values

to be preferred over other options. But in other cases the selection was not determined purely by the requirements that were considered. These choices will be made by other arguments – for example based on the ease of implementation, overall cost or simply by informed executive decisions.

Even when a general decision has been made to go with a specific implementation option and actually move forward to implement that, there are still a lot of considerations and choices left open. For example the choice to implement software updates of the set-top box by using automatic updates, says very little about the actual mechanism that will be used to do this. What we *have* shown earlier is that this choice should not affect other parts of the system in a negative way. But to make sure that the actual implementation will indeed be functional and secure, this aspect will need to be investigated in much more detail.

In the next chapters we will discuss these implementations in detail and offer a more in-depth analysis of how the infrastructure to handle these tasks may be designed. Our analysis will focus on the security aspects – but not disregard the other requirements – and should lead to detailed descriptions of specific solutions that could be applied and implemented by *iphion*.

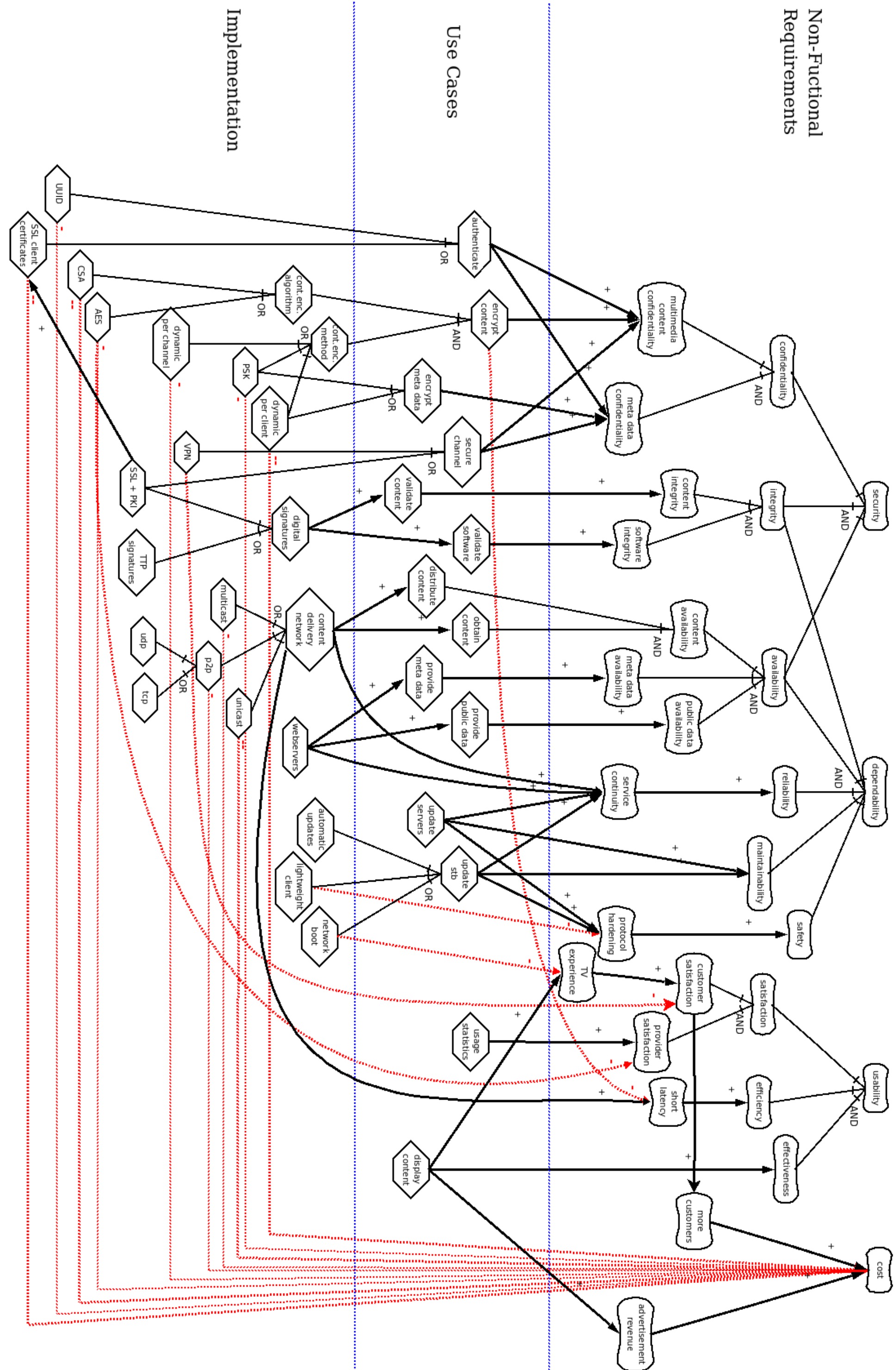
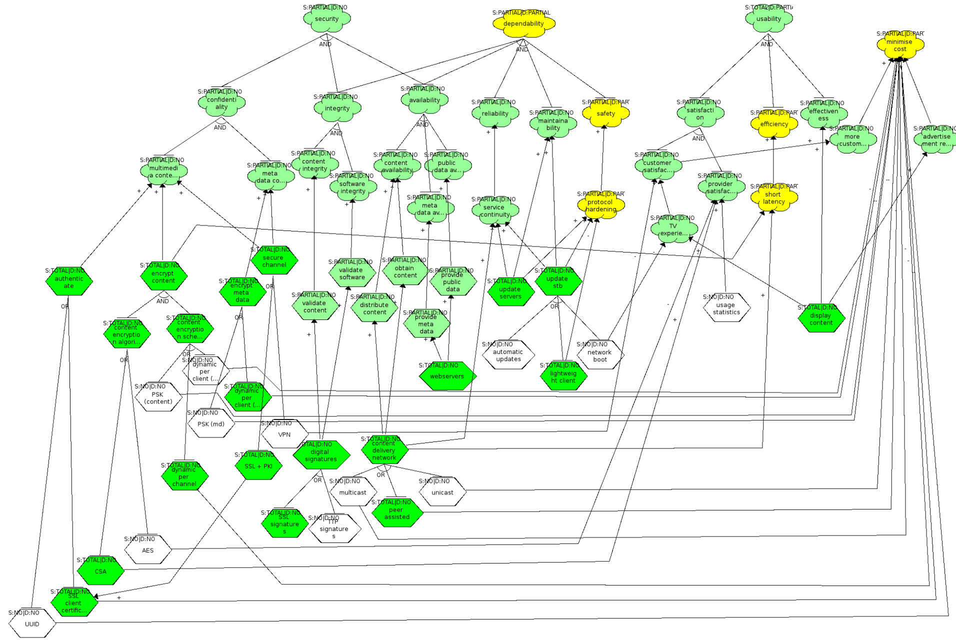
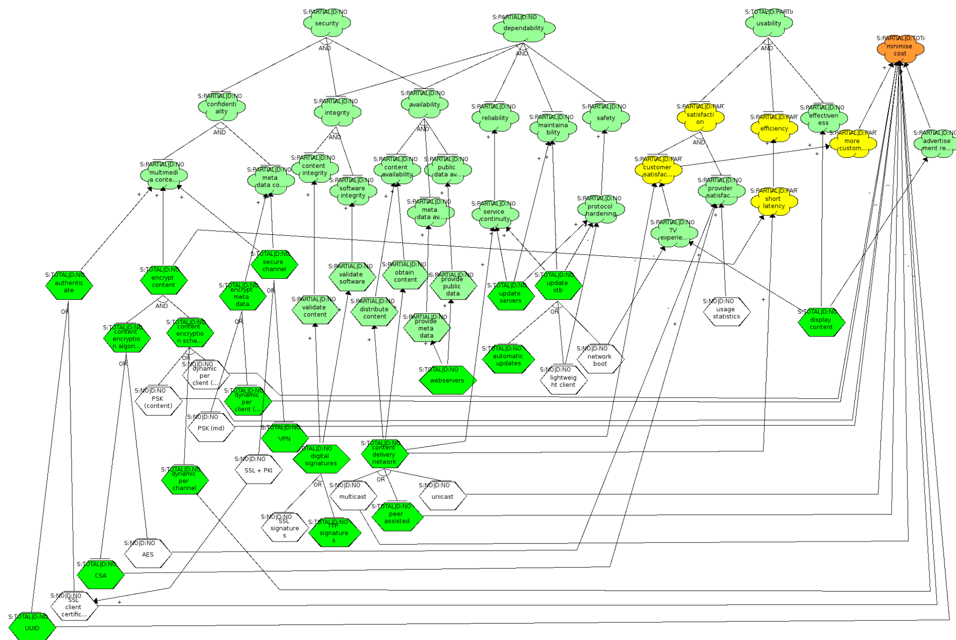


Figure 2.12: Full goal tree model

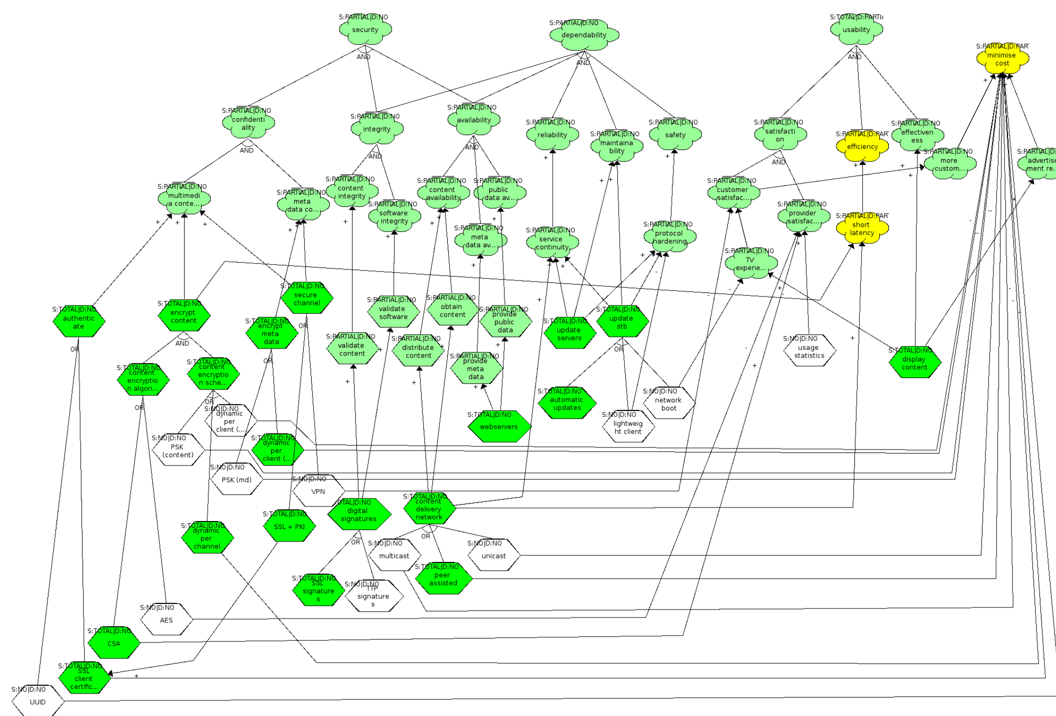


(a) $Conf_1$



(b) $Conf_2$

Figure 2.13: Reasoning tool results



(c) $Conf_3$

Figure 2.13: Goal model reasoning tool: Showing the results for the sample configurations described in Table 2.4

3. Secure communications with TLS

Based on the requirements analysis we decided to to use Secure Sockets Layer (SSL) in combination with a Public Key Infrastructure (PKI) to facilitate secure channels that will guarantee meta data confidentiality in the system. In this section we describe out how SSL works and discuss how it can be deployed in the *iphion* environment. First of all, it should be noted that SSL is actually the old name and that this protocol is nowadays known as Transport Layer Security (TLS): we will use this name in the rest of the chapter.

We will also discuss how TLS certificates (or rather X.509 certificates) can be used in the authentication of *iphion* players and servers. Authentication is required for both meta data and multimedia content confidentiality.

The chapter first describes the background TLS communications and authentication in general (Section 3.1) and then describes how TLS may be deployed at *iphion*, both for secure authentication and confidential communications (Section 3.2). We describe the set up of a Public Key Infrastructure (PKI) to support the TLS infrastructure (Section 3.3), the risks of key loss or compromise (Section 3.4) and a policy to securely create, sign and deploy the keys (Section 3.5). We conclude this chapter with an overview of the available software tools to handle TLS communications (Section 3.6).

3.1 Transport Layer Security

Transport Layer Security, or TLS, is the de-facto standard for secure internet communications [DA99]. It is the successor of SSL (Secure Sockets Layer). TLS operates on top of TCP/IP in internet connections. A client first starts a regular TCP connection to a server and after that it starts a TLS handshake. From that point on, all data sent over the TCP connection will be encrypted with a session key, so that only the client and server can

decipher it. Regular application data will be exchanged over this encrypted communication channel.

Apart from data encryption, TLS can also handle authentication of the communication partners via public key cryptography, in the initial handshake when a connection is established. TLS can be used to authenticate the clients, so that the server can decide which clients get access to which resources. This protocol can also be used for two-way authentication, where the server also authenticates itself to the client. To take full advantage of this, all communication partners, clients and servers, should have their own private keys and a corresponding public key. The public keys can be signed by a trusted authority, the so-called certificate authority (CA).

In the rest of this section we describe in more detail how TLS works. First we will look at the authentication handshake mechanism of TLS, which includes the exchange of a session key for secure communications. Then we will discuss how a Public Key Infrastructure may be used to ease TLS deployment.

3.1.1 TLS authentication

Authentication via TLS communication uses public key cryptography. Each communication partner has its own keypair, consisting of a private (secret) key part and a corresponding public key part. Although they represent different parts of the same keys, these parts are generally referred to as a public and private key. In public key systems, asymmetric cryptography is used, where data cannot be decrypted with the key that was used to encrypt it (in contrast with symmetric cryptography). Common algorithms used for public key cryptography are RSA [Inc02], DSA [NI09] and ECDSA [X905].

With public key cryptography, the public key may be used by anyone to encrypt data, so that only the owner of the corresponding private key can decipher it. The owner of the private key can also use this to create a digital signature for any piece of data, that can be checked by others. With the signed data and the public key, anybody can verify that the signature must be created by the person who holds the private key.

When a TLS communication is initialised using public key cryptography, client and server first exchange public keys. The client then generates a session key and sends this to the server in a message that is encrypted with the server's public key. This session key is usually a key for symmetric cryptography (since that is faster than asymmetric crypto). Client and server can then exchange confidential data using this key.

Authentication of the client and server is done when sending the public key. Along with the public key details, identity information about the sender is

given as well. This can be an internet hostname or another unique identifier for the node. In the *iphion* case, each node will have a unique *Peer ID* that is used for identification.

To prevent attackers from creating own keys and presenting these with fake *iphion* identification, this public information (the public key and identification details together) are signed using a digital signature from a trusted party, the certification authority. The signed public information is generally called the certificate. The key that is used to sign the certificates needs to be known by both parties in advance. This signature signs both the public key and the textual identification details of both the certificate owner and the signature issuer. So if any bit on the certificate is changed, a signature validation will fail and the entire certificate is considered invalid and will never be accepted.

Checking the certificate of a communication partner alone, is not enough to be certain that you are communication with the entity whose identification is listed in the certificate. After all this certificate is public information and anybody can copy it. Therefore it is important to check that the communication partner is also in possession of the corresponding private key. This can be done by sending an encrypted message and verifying that the other end can decrypt and use this message. As long as all nodes in the system keep their private keys well guarded, a communication partner can be reasonably sure that it is indeed communicating with the node who is in possession of the one private key and whose identity is listed on the certificate. This way communication partners can reliably authenticate themselves.

The full handshake that establishes a secure TLS session, is shown in Figure 3.1 as a Message Sequence Chart [IT04]. The actors in this figure are server s, client c and certificate authority a. Nc and Ns are fresh random values; $Pk(u), Sk(u)$ represent the public and secret key of u; $ID(u)$ is the identity of u; $\{M\}_K$ indicates encryption ($m_{5,7}$) or signing ($m_{2,3,4,7}$) of message M using key K and $\mathcal{H}(M)$ denotes the hash value of message M. For every signed message, the recipient should immediately check the digital signature and abort the connection if it doesn't match. The action **MS** indicates calculation of the Master Secret from the values of PMS, Nc and Ns. After m_6 all further communication is signed and encrypted using keys derived from the Master Secret.

A full TLS handshake includes authentication of both communication partners and the agreement on a fresh Master Secret key. This secret is used to create the session keys for secure hashing (to preserve data integrity) and encryption (to preserve data confidentiality).

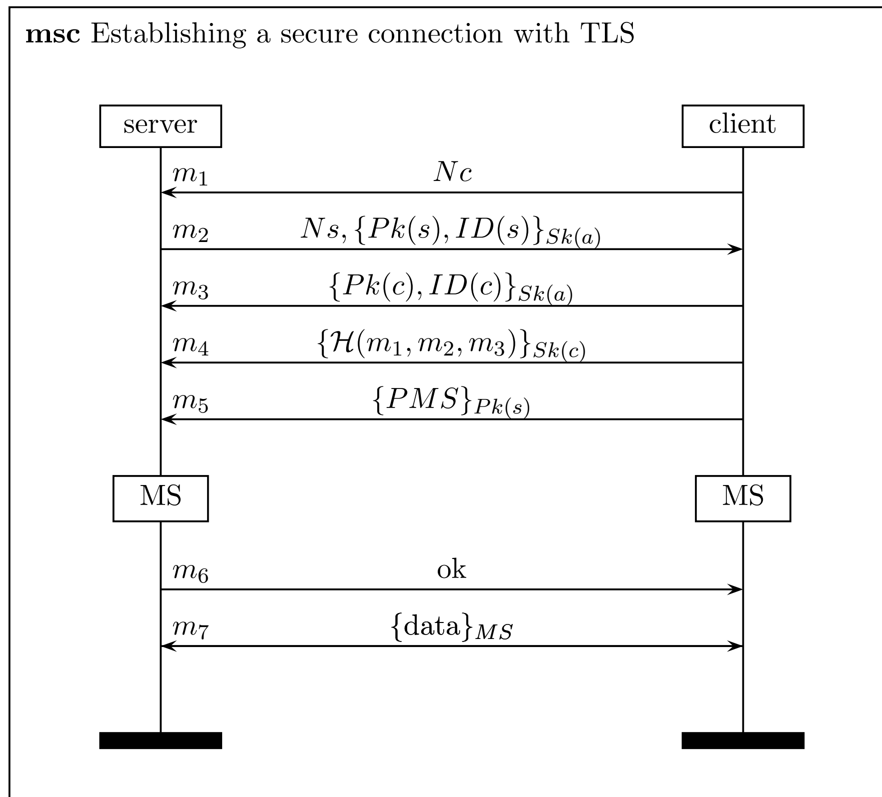


Figure 3.1: Simplified TLS handshake protocol.

3.1.2 Setup of a Public Key Infrastructure

A Public Key Infrastructure (PKI) is used to manage trust relationships and cryptographic keys within an organisation. The PKI defines a set of policies and procedures needed to create, manage, store, distribute, use and revoke digital certificates. A digital certificate contains the public key of a node, combined with the identity of that node and a signature from a trusted Certificate Authority (CA). The corresponding private key should be kept secret and only be in possession of the named node.

The trust model in a PKI is strictly hierarchical. Every certificate is signed with a single trust signature from a node that is above it in the PKI hierarchy. This can be either the Root Certificate Authority (RCA) or an Intermediate Certificate Authority. Ultimately the chain of trust signatures always leads up to the Root CA (which signs the certificates of the main Intermediate CAs). The Root CA certificate itself is self-signed. Because the path ends there, every node in the system should be aware of the public certificate of the Root CA: this is needed to check the chain of trust for any other certificate that's encountered.

Replacing the certificate of the Root CA may be a very troublesome operation. It also means that all the certificates in the system that have been issued previously can no longer be validated against the single new Root CA certificate. Therefore, either two Root certificates must be checked; or all old certificates that have been issued must be replaced. Although there should be a plan to do a Root certificate roll-over when it is required, it's better if this situation can be avoided, by keeping the root certificate safe. For issuing end-point certificates, an intermediate certificate could be used, and the root certificate would only be needed when a new intermediate certificate needs to be generated. The intermediate keys would not be used for authentication or encryption via TLS, but only as singing keys: to sign the authentication keys of entities in the system (see Figure 3.2).

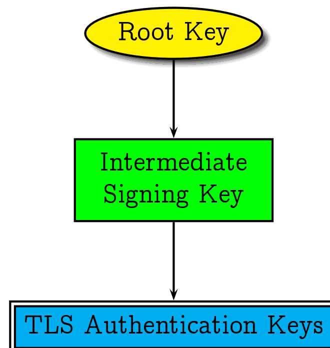


Figure 3.2: X.509 certificate hierarchical overview

3.2 Deploying TLS at iphion

At *iphion* TLS could be used to facilitate communication confidentiality and the authentication of communication partners. However, TLS alone is not enough to handle all secure communications for the system. There are some areas where a TLS solution can not be used, either because the system is not using TCP for communications[1] (for example in the rescue loader and the content distribution), because full TLS support is not possible due to technical limitations (e.g. in the bootloader environment that is too restricted), or because we are not doing point-to-point communications (e.g. in the content distribution). So before discussing how a TLS solution should be set up, it is important to first identify where and how exactly it can be used; and where other solutions are needed.

[1]However, a TLS variation, Datagram Transport Layer Security, could be used for point-to-point UDP connections.

In regular internet use, often only the servers use a key-pair for authentication, while the clients remain anonymous (e.g. https webservers), or use password authentication instead (e.g. secure imap servers). However for our purpose it is better if every communication partner has its own key-pair, so that authentication can (and must) always be done both ways via TLS.

In the following sections we will first determine the areas where TLS can be applied in the *iphion* system, both to guarantee data confidentiality and to authenticate communication partners. We will also list the locations where TLS is not an option and other solutions will be needed. After this we describe how exactly TLS may be deployed.

3.2.1 Communications using TLS

First we need to establish where in the communication TLS can and should be used – and where it is not appropriate. TLS can take care of several aspects of secure communication: authentication (of the communication partner), confidentiality (encryption) and integrity (data validation). These aspects are generally needed together in the system.

We focus on the communication between the *iphion* servers and clients (set-top boxes). A main part of this communication consists of the distribution of multimedia content. For this distribution itself, TLS is not a feasible solution. This content is distributed over UDP, while TLS only works over TCP and an important aspects of this distribution is that packets may go missing without the need for resending them; and that it should be possible to relay data onwards to other peers without having to encrypt it again for this specific recipient. Furthermore, the chosen set-top box (STB) does not have the computing power required to handle the overhead of TLS encapsulation of the full multimedia data stream — although it will have to handle data integrity checks and descrambling in some fashion[2]. Each of these aspects disqualifies TLS as an option for content distribution.

However TLS can be used in the distribution of the private keys that will be needed to decrypt the content data on the STB. These keys are valid for a limited (short) period and every player that receives multimedia data will need to have the appropriate decryption keys. The keys are issued by the *iphion* **token server**, which authenticates all clients that connect and issues them decryption keys for the channels they are authorised to view.

TLS can also be used for other meta data that the clients will request from the *iphion* servers: the electronic program guide, account status information etc. Communication of this meta data uses the classic client-server model:

[2] An earlier design used SHA-1 message digests for the multimedia stream, but the STB software could not keep up with that. This was later changed to MD5-HMAC.

a player connects to the server from which it wants some information, client and server each authenticate the other party, the client sends its request and the server responds with the appropriate information. If everything goes well, the connection can then be shut down again. For different types of meta data information, different servers will be used.

The clients will use the same mechanisms to report operational issues to the *iphion* servers (start-up notifications, problem reporting, statistical usage information). The communication is always initiated by the client that connects to the appropriate server; although the server response may contain additional queries or requests for the client. The client has to initiate communications because it may be turned on and off whenever the customer feels like it; it may be placed behind a restrictive firewall (or even a NAT home-router) and it may even use a new IP address every time it is switched on.

Software updates work similarly: each client will periodically check whether updates are available (for this specific box) and download them when appropriate. This will also use a TLS connection; however additional integrity checks are used in this case. To prevent hackers from installing software on a player's flash chip, which might disrupt the network, the installed file system is digitally signed and a file system will only be used when this signature can be verified. This mechanism is discussed in detail in Chapter 4.

A special case of secure updates is triggered when an *iphion* player can no longer be started up in the normal fashion — for example because the kernel or root filesystem's digital signature can not be verified. In that case the player's bootloader (which is stored in read-only memory) initiates a rescue mechanism. First a small rescue operation environment is downloaded from an *iphion* server using TFTP (that is without TLS). This environment will then try to download and install all the data that is needed to restore the player to a functional box, using regular software update mechanisms. This layered approach is due to technical limitations: the boot loader must be very small and with the chosen hardware platform this means that a TCP stack and TLS implementation are not possible here. TFTP is a UDP-based protocol without sliding windows (download speed depends on round-trip times), so even with a high-bandwidth connection downloading this image may take a while, because the round-trip time for home connections generally isn't very short. Therefore, the rescue image should be kept small. The operational and security aspects of this rescue procedure are further explained in Section 4.3.

The *iphion* players only communicate with other *iphion* players via the content distribution network. This is a UDP protocol, so there is no TLS-based data transport involved and the authentication of partners cannot

use TLS either. However such communications should also be secure and preserve the desired aspects confidentiality, integrity and availability. The *iphion* content distribution protocol (iPAP) guarantees these requirements. Players authenticate themselves using public key cryptography, with certificates issued by the **token server**, all data is digitally signed using HMAC with session keys and the multimedia data is encrypted with DVB-CSA, also using keys that are replaced frequently.

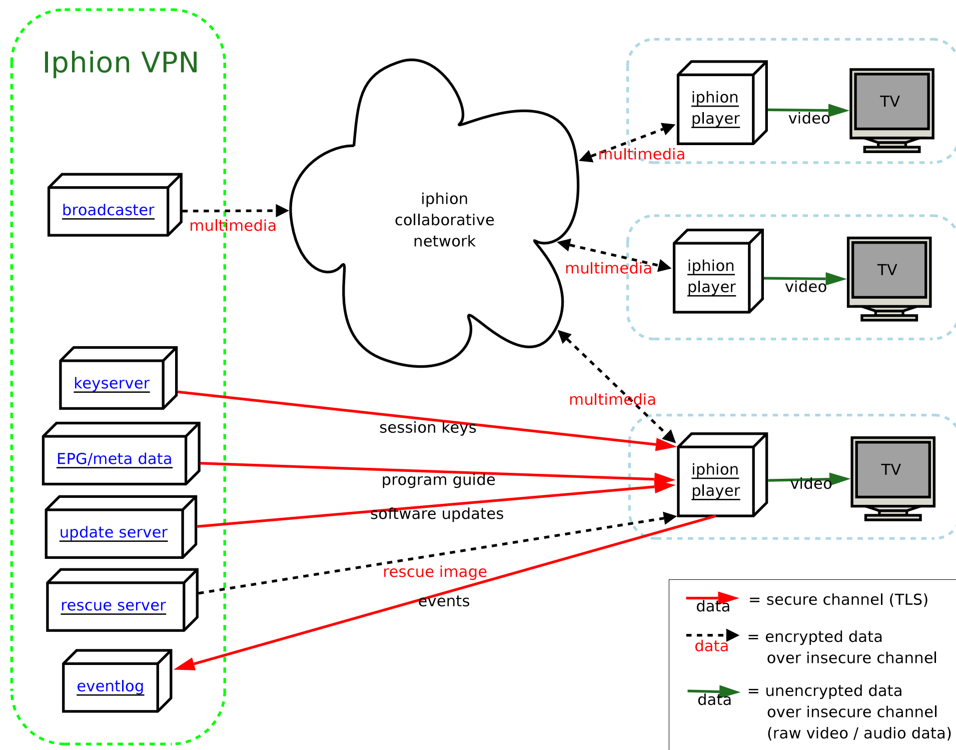


Figure 3.3: Client-server communications using TLS

Figure 3.3 shows an overview of the communication between the *iphion* servers and *iphion* set-top boxes and highlights the places where TLS can be used to secure communications as well as the other communications, where TLS is not an option and alternatives need to be used instead.

3.2.2 Secure authentication

TLS will also be used to authenticate communications partners in the *iphion* system. However, a regular TLS authentication is not enough to reliably authenticate *iphion* players: their credentials are stored on the box that is given to customers and this information may be copied to other systems.

In this section we describe the threat in detail and also offer a solution to reliably authenticate all partners in the *iphion* system.

A connection set up with TLS in the *iphion* system can guarantee that you set up a secure connection with somebody in possession of a certificate issued by the *iphion* certificate authority and the corresponding private key. To make this work, every node (computer) in the *iphion* network will need its own private and public keypair. This includes the set-top boxes that are sold to customers and placed in somebody's home environment. The secret key is installed on this system in such a way that the software can use it without manual intervention (i.e. no passphrase protection). But when the STB software is compromised or memory storage is accessed directly (for example by connecting it to other hardware), the secret key might be read out. In theory this could be used by an attacker to copy the key to other locations and set up unauthorised (and untrusted) clones.

It is not simple to launch such an attack, since the STB will only run software that is digitally signed by *iphion*, thanks to secure boot-strapping mechanisms (Section 4.1). Moreover, the *iphion* servers should notice it when multiple clients connect from different locations while using the same credentials. Such heuristics do not offer sufficient guarantees – and it doesn't tell you which one is the 'original' *iphion* player. If clients using cloned keys are detected, the key must be 'blacklisted' by the token server, so that it can no longer be used to obtain tokens to access to the *iphion* network. In that case the original box will be banned as well. At this point, manual intervention will be needed to determine which is the original hardware and which is a clone. The company will need a procedure to handle this situation, for example by replacing the banned STB.

The only way to have a secure key that cannot be compromised or copied is by having it stored (in hardware) in a location that can never be modified or ever read out after it was written. Ideally it should not only be impossible to access this information from software, but one shouldn't be able to read this information with hardware tools either. If this hardware device also supports basic cryptographic operations using this key (encryption, decryption, signing and signature validation) then it would be possible to use it in software without ever needing the private key in accessible memory. The software could send data to the cryptographic hardware, tell it to encrypt or decrypt and then read out the result.

The selected hardware platform (NXP STB 225) does include such a cryptographic feature. It can store a key in polyfuse hardware; fuses are blown when the key is written into a write-only register and it should then be completely inaccessible and tamper-proof. These registers can either store three 64-bit TDES keys or one 128-bit AES key (so only for symmetric encryption).

Now before getting into the gory details of how such a key can be programmed during production and later used from software, let's work out how we can employ this feature for secure authentication.

The main goal that we want to accomplish with the extra key is a verification (on the server side) that the connecting client is controlled by a genuine *iphion* player set-top box and not some clone (e.g. a PC). If the client can prove that it can use the embedded secret key, which is also known to the server, in a TLS session, then we can be reasonably sure that the software is indeed running on *iphion* hardware.

To verify this, a simple challenge-response system could be used once a TLS connection has been established. The server sends a challenge to the client (a fresh random value), which it should encrypt and send back. To counter man-in-the-middle attacks (should anyone manage to obtain the copy of a valid private key), the identity of the client could be included in this encrypted message as well. The server can then check if the message is really encrypted by the client he is supposed to be communicating with. As an extra check, authentication of the server can be verified in the same way, so that the *iphion* player can be sure he is communicating directly with the real *iphion* server.

This shared-secret authentication protocol is presented in Figure 3.4. The actors are server s and client c who share knowledge of a secret key K. Nc and Ns are fresh random values; $ID(u)$ is the identity of u; $\{M\}_K$ indicates encryption of message M using key K. Both partners should check if the received random values and identities match with their expected values.

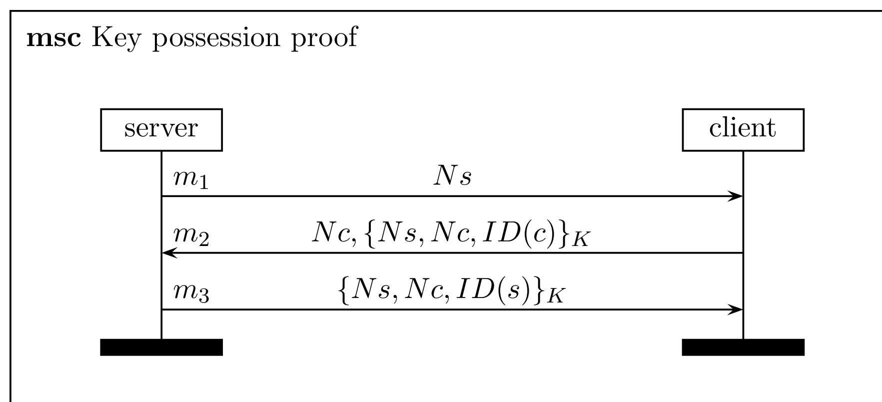


Figure 3.4: Shared-secret authentication check

Note that even with this setup, all that the server knows is that he client has access to a genuine *iphion* client. It is still possible that somebody is running software on a PC (or even multiple PCs) that only offloads the challenge-response check to the *iphion* set-top box. This is a form of man-

in-the-middle attack that cannot be detected by the authentication server, but only by the client. However, the *iphion* server will notice when multiple clients use the same identification certificates. Operators can then block the use of this certificate. In Chapter 4 we will describe a mechanism to establish client integrity. The man-in-the-middle scenario can be prevented if we can be sure that clients run *iphion* software that enforce server authentication as well.

3.3 Setup of the iphion PKI

The certificates used for TLS authentication must be signed by a trusted (higher level) key. The set-up of a certificate signature tree is strictly hierarchical: at the top is the Root key for *iphion*; below that the intermediate key, which is used to sign the client certificates. There may be multiple intermediate keys – even multiple levels of intermediate keys. A question is whether more than one intermediate certificate would be needed in our system. This section analyses the required X.509 certificates that *iphion* will be using and determines the certificate hierarchy that should be used to support this.

To determine what kind of intermediate keys are needed, we'll make an inventory of how the client keys will be used; how and when they are generated; and when keys might need to be replaced (expiry, revocation). If this leads to a clear functional distinction, then it would make sense to have multiple intermediate certificate authorities. But if keys are created by the same people and the same method each time, then a single signing key will do.

Keys for the identification of clients (both public and private key parts) will be stored on the *iphion* player box in a read-only NOR memory chip. These keys need to be stored unencrypted, as they need to be accessible by the system, without manual intervention. They could be stored encrypted and be automatically decrypted when needed, but since this decryption key would also be stored in the same location (at least somewhere on the box, and not on the network), such an encryption would only obfuscate the use and not add any real security. The ID of a client is a unique number, that is also used as its PeerID in the content distribution protocol [Min09]. These keys should never change during the lifetime of a player, and it would arguably be best if it would not be possible to overwrite it at all. This means that the lifetime (expiry date) of the certificate should be no less than the expected lifetime of the hardware as part of the *iphion* system.

Client keys are generated when the devices are produced. Adding the key-pair to NOR is part of the production process. That means that the keys

are generally produced in large batches (100 or more) at once. If a client key is compromised (e.g. used by a cloned device), it should be possible to disallow further use. This may be done via a certificate revocation list (CRL), or other blacklist mechanism. Only the *iphion* servers need to know which client keys are banned, because clients authenticate themselves via short-lived authorisation tokens from the **token server** when talking to other clients directly.

Private keys for the identification of *iphion* servers will be stored on the respective servers. These keys will be stored on standard writeable hard disks, at least initially. For practical purposes, most of these keys may be stored unencrypted, as it would be impractical to manually intervene whenever a service is restarted (e.g. upon a reboot when a machine has crashed). Better ways to store server keys (e.g. RSA hardware tokens) may be installed later, although probably only for the most critical systems. The ID of a server is its internet hostname — as is common with TLS communications. These hostnames will likely include a keyword describing the functionality of a service (token server, EPG server) and a protocol release number. So the server that offers the electronic program guide for clients running version 1.3, might get the internet hostname 'epg.1-3.iphion.net'. The use of a version number in the hostname, not only has practical implications for legacy compatibility the protocol itself, but it also implies that the lifetime of server certificates will be limited to the duration of a development and release cycle.

Server keys are generated when a new server is installed; or as part of a release cycle, when new hostnames are introduced. So these keys are generated individually or in small batches. If a server key is compromised (e.g. intruders obtain temporarily access to a machine), it should be possible to disallow further use. This may be done by using a blacklist (or CRL) that clients and servers can access via the internet (OCSP), or that is stored on the clients. Forcing a new release (number), will also get all clients and servers to stop accepting a server certificate with the old version number in the hostname. Servers communicate with other servers via TLS as well as with the *iphion* players.

However *iphion* servers may also communicate securely with other computers (clients) that are outside *iphion*'s control. Customers, but also content and advertising partners, may access information from *iphion* that is made available specifically to them: account details, viewing statistics etc. Most of this information will be provided via secure websites (HTTP over TLS). Authentication of the clients via *iphion*-issued certificates would not be practical in this case. It would even be hard to get the partners to accept *iphion*-issued certificates for the servers without causing confusion. This is because browsers will loudly complain when they encounter a TLS certificate that was signed by a company whose root certificate is not included in the

browser distribution. Getting *iphion*'s root certificate included here, would be a very expensive exercise. So it is probably best to have the key pairs for these services signed by an external company, whose root certificate is already included in popular browsers. For the authentication of the clients (partners), it is more practical to use passphrases generated by *iphion*, rather than X.509 certificates.

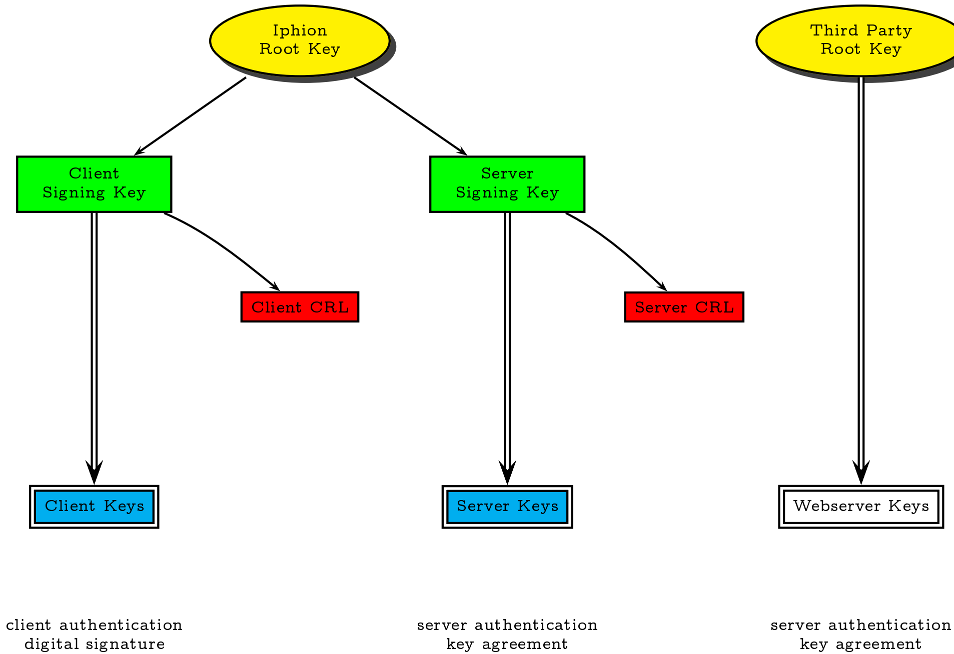


Figure 3.5: iphion certificate hierarchy

Since the use of client and server keys, is indeed quite different, we will use distinct intermediate certificate authorities to issue these certificates. This also means that clients may check if all TLS certificates they encounter are issued by the server signing authority. Even if a client keypair is compromised, it could never be used to impersonate an *iphion* server, since these have a distinct chain of trust. Figure 3.5 illustrates the different certificates that will be used in the *iphion* system. The usage of each certificate is listed in Table 3.1.

Key	Purpose	Use frequency
Root Key	Sign intermediate keys	Hardly ever
Server Signing Key	Sign server certificates	New server install
Client Signing Key	Sign client certificates	Occasional large batch

Table 3.1: iphion intermediate certificate use

3.4 Key exposure threats and risks

The loss of private keys that are used for TLS authentication or certificate signing are lost, may have a large impact on the *iphion* system. It would allow other parties to impersonate *iphion* machines disrupt large parts of the network or obtain multimedia content without authorisation.

Some keys are easier to obtain than others, but the impact on the rest of the system is different as well. The largest impact would be loss of the root authority key; but the easiest to obtain would be a client authentication key, which is stored on the distributed boxes. First we will assess usage and storage of all keys (see the overview in Table 3.2) and then we will discuss the dangers and risks, and what steps can be taken to recover from such a loss.

Key type	**Purpose**	**Private key**	**Public key**
Iphion Root Key	Only for signing intermediate signing keys	Must be kept off-line at all times	Stored on iphion player filesystem and all servers
Client Signing Key	To sign all client certificates	Can be kept off-line at all times; used whenever new (batch of) players is produced	Stored on all iphion servers
Server Signing Key	To sign all server certificates (EPG, rescue, ...)	Can be kept off-line at all times; used whenever a new server is installed	Stored on iphion player filesystem
Client Keys	To identify an iphion player in TLS connections	Stored in NOR (fixed)	Stored in NOR (fixed)
Server Keys	To identify an iphion (TLS) server	Stored on the server	Stored on the server
Webserver Keys	To identify an iphion (HTTPS) webserver; used by partners and customers	Stored on the server	Stored on the server

Table 3.2: iphion key pair storage

The public certificate of the root certificate authority must be pre-installed everywhere: this is the root for all trust chains. Although the chain of public intermediate keys can also be sent along in a TLS handshake, parties should have these stored locally as well and rely on their own version. The public root key will be referenced in all secure communications. The private part of the root key is only needed on rare occasions: whenever a new intermediate

certificate authority key needs to be installed (signed). Since the number of intermediate CAs is very limited (2) and their life-time very long, this shouldn't be needed for long time after the initial creation. However it is important that the key remains accessible: with functional and secure backups and if it is encrypted with a passphrase, this access code should still be reproducible when it's needed.

The intermediate keys will be needed from time to time. They are used when new server or client certificates need to be generated. This generation and signing can be done on a device that is not connected to the network: only the resulting client/server keys need to be copied to the computers on which they will be installed. The public certificate of the server signing authority will also be installed on all clients. Clients should check if server certificates that they encounter are indeed signed by the server certificate authority: a client certificate could never be used in a server. Likewise, servers should check that client certificates are signed by the client certificate authority. It is theoretically possible to replace the keys for the intermediate CAs in the future – however since client certificates cannot be replaced via software updates, it is unlikely that this will ever happen. Additional intermediate certificate authorities might be added in the future though, if and when they are needed.

The client and server keys and certificates are only needed on the machine where they will be used. There is no need to copy public certificates to other machines, as it can be automatically send along when a TLS connection is established. The communication partner can then check the certificate signature to verify that it is a genuine *iphion* certificate. Client keys are stored in hardware and cannot be replaced, but server keys can and will be replaced. Server keys are linked to internet hostnames, which in turn relate to a specific service and release or protocol version. When this hostname changes (with each major release), the certificates will be replaced as well. This limits the life-time of the keys, and thus the impact if these keys were compromised. The impact of client key compromise is limited by the combination with a shared secret key in hardware, which is also required in order to do secure authentication for the *iphion* content delivery network (as discussed in Section 3.2.2).

Table 3.3 gives an overview of the consequences of any key compromise. Here we assume the loss of only a single key. The impact of a compromise might become much worse, if it is combined with the lose of other secret keys; such as the software signing key; or the identification secret use for authentication in the content distribution network. Since the way these keys are used and stored is rather different, it is unlikely that they are compromised in the system at the same time. But it is possible that the safe location of the secure key backups is compromised and that all keys are lost at once. In that case the only option will be to replace all the private

Key type	**Threat**	**Impact**	**Remedy**
Iphion Root Key	The key may be compromised in storage or during the brief period in which it is used.	Servers and clients can be created that look valid (mostly useful for meta-data). Software updates and content distribution are not affected as these require additional keys.	***Recall customer hardware:*** The keys on all players and servers will need to be replaced.
Client Signing Key	The key is compromised in storage or during the brief period in which it is used.	Valid looking clients can be created and used to obtain confidential meta data (no content).	***Recall customer hardware:*** The keys on all players must be replaced; on the servers the intermediate CA certificate must be updated.
Server Signing Key	The key is compromised in storage or during the brief period in which it is used.	Valid looking servers can be created. Clients may be tricked into divulging confidential information.	Replace all server certificates and issue a software update to players to distribute the new intermediate CA certificate.
Client Keys	The NOR is read out (doable for a reasonably able attacker).	Unauthorised clients can get access to confidential information.	Block the key on the server side and replace the box.
Server Keys	The server in question gets compromised.	Valid looking servers can be set up: but only for the specific service and revision that a key is linked to.	Issue a software update to players.
Webserver Keys	The server in question gets compromised.	Valid looking servers can be set up: but only for a specific hostname.	Revoke the key with the issuer. Since most browsers don't use OCSP by default, the old key may remain trusted.

Table 3.3: iphion key pair misuse

keys in the system.

In practice it will be rather difficult to replace the keys that are used by

the *iphion* players for any large batch (e.g. when the client signing key is compromised). It is technically possible to recall a player and replace the NOR, but this can not be done via remote software updates. This will probably mean that a new signing key will be generated and used for new batches while the old keys phased out. How fast it would be phased out would depend on the circumstances at the time of the compromise. The system should be able to handle multiple intermediate certificate authorities. If multiple intermediate CAs are temporarily required after an incident, then this should indeed be temporarily: *iphion* should always strive to completely remove use of the tainted key. After all the system is only as secure as its weakest link: as long as a compromised key remains in use, the system is vulnerable to misuse of this key.

The risk is not limited to leaking copies of private keys: losing access to a private key will have a grave impact as well – especially for the keys that can not be easily replaced, the root and intermediate CA keys. To prevent this, copies of the important keys are needed. These copies will need to be protected, not just with encryption, but also physically: The copies could be stored in a safe at someone's house, rather than lying on a desk in the office. The problem with encryption of the important keys is that the decryption key (or passphrase) could still be lost. Since the important keys see only little use, this is a real risk. Also the media on which the keys are stored, might become unusable after time. For recovery, it's probably best to store unencrypted versions on a reliable (non-digital) medium as well, such as a print-out of the key on paper. Obviously this version should be well secured.

Of course the keys should also be cryptographically secure for their intended purpuse as well. Since the boxes are supposed to 'live' for several years with fixed keys, it is important to use cryptographic algorithms that are expected to remain secure during the entire lifetime of the system. Unfortunately the *iphion* players come with small MIPS CPUs, which are slower than anything you'll find in a modern PC. Besides, public-key cryptography operations are very expensive (for example when compared to symmetric cryptography). Operations using keys that are more secure (using larger key lengths), will be even more expensive. So we are looking for keys that won't be broken by brute-force attacks for several years yet, but also keys that are not too large to have a serious negative impact on the performance of the *iphion* players.

It is hard to compare the strength of cryptographic algorithms that use different mathematical properties. It is even harder to predict how long an algorithm will remain secure. But the recommendations of NIST (the US National Institute of Standards and Technology) set the standard for the cryptographic algorithms that all US federal agencies should use to protect their data, and these recommendations are generally followed by others as well [BBB+06]. These recommendations assert for example that

RSA and DSA with 1024-bit keys should **not** be used after 2010 – but 2048-bit keys should be fine until 2030. RSA is probably the best known, and most widely used algorithm for public-key cryptography these days. When creating certificates, the signing key that is used to sign other certificates should never use weaker cryptographic algorithms than what is used by the keys that are getting signed: Using 4096-bit client certificate keys, signed by a 1024-bit root key makes little sense, as an successful attack on the root key makes all the client keys 'worthless' as well.

3.5 Key creation, signing and usage policy

iphion will be using X.509 certificates for client-server communications via TLS, this covers most of the client-server communications, as shown in Figure 3.3. To secure the other client-server communications, public-key algorithms will be used as well, but without TLS; this will be discussed later, for the content distribution protocol in Chapter 5 and for the rescue procedure in Chapter 4. For all secure server-server communications, apart from the content distribution, TLS will be deployed as well.

For the client and server certificates 2048-bit RSA keys will be used. A 2048-bit keysize is the minimal recommended secure size from 2010 according to NIST [BBB+06]. When doing performance test on the *iphion* player platform, RSA operations proved to be much faster than DSA (in signing and verification) and even faster than ECDSA (elliptic curve crypto) in verification as well, which will be the most common operation. ECDSA is in fact faster in other operations, but it has also seen less scrutiny via cryptanalysis and is used much less in real-world applications than RSA.

The public-key cryptography will be combined with symmetric cryptography (for data encryption) and secure hashes (e.g. for digital signatures). These should all use cryptographic protocols of similar cryptographic strength when combined (or at least not weaker). Once again the NIST report lists good candidates: AES-128 and SHA-256 (see Table 3.4).

bits of security	**symmetric**	**RSA / DSA**	**ECDSA**	**digital sign. hash**
80	2TDEA	1024	160-223	SHA-1[a]
112	3TDEA	2048	225-255	SHA-224
128	AES-128	3072	256-383	SHA-256
192	AES-192	7680	384-511	SHA-384
256	AES-256	15360	512+	SHA-512

[a]Weaknesses have been found in the SHA-1 algorithm: The 2006 assessment of SHA-1's strength against collisions is about 69 bits. In April 2009, this was reduced to 52 bits.

Table 3.4: NIST SP 800-57 key strength comparison

For the keys of the root and intermediate certificate authorities, the key sizes should be at least as strong. Since these keys are expected to last even longer and since they will not be used as intensively by the *iphion* players and other computers in the system, it would be a good idea to use larger key sizes. However, it doesn't make sense to choose extremely large numbers: it should be enough to resist (brute force) attacks for many years to come, but it should still offer reasonable performance and be acceptable for all the cryptographic software that *iphion* plans to use. NIST recommends not using RSA keys larger than 4096 bits, which is also the limit for some software (although OpenSSL can handle larger keys).

The digital signatures that are made with these keys, should use a hash algorithm that corresponds in cryptographic strength with the chosen public key algorithm. Using a stronger hash function would only delay computation, for no additional security and deploying a weaker hash function would decrease security – and in the extreme case allow people to reuse the existing signatures with their own certificates or messages. For instance if we have a 2048-bit server key, whose certificate is signed by a 3072-bit RSA intermediate key, then this signature should use at least a SHA-256 message digest (or comparable algorithm).

As mentioned above, a 2048-bit RSA key is the minimum recommended size from 2010. Performance tests with the selected hardware platform indicates that this is usable for authentication and meta data integrity, confidentiality (but too slow for streaming multimedia integrity checks, as discussed in Chapter 5). We will be using this algorithm for server and client keys in the *iphion* system. As long as we choose the intermediate and root keys sufficiently large, we can always add stronger keys for peers later, within the existing system.

The root key will use the maximum recommended keysize, 4096-bit RSA. And for the intermediate keys (client signing key and server signing key), we go for a compromise between speed and security and select 3072-bit keys. Note that clients (the *iphion* players) will be doing RSA encryption operations using the server keys (2048-bit), signing operations using the client keys (also 2048-bit) and only the faster signature verification operations using intermediate key (once for each connection; 3072-bit) and the *iphion* root key (only once per application; 4096-bit).

3.5.1 Development keys

The root- and intermediate keys will only be used very rarely for normal operations. However during development they will be used much more. There will also be servers and clients used for testing in an environment that is not quite as secure as the production environment.

Most importantly, in the testing environment, *iphion* clients and servers will be fitted with debugging tools. These debugging tools give easy access to confidential information that is well-guarded in the production environment (security keys, source code, multimedia content).

Therefore we have decided to set up a full, separated, X.509 certificate hierarchy for the development environment at *iphion*. This environment will use the same types of keys and similar secure procedures for creation and storage as the production keys. The advantage of this is that during development everybody involved in the process gains experience that can later be applied to the production process as well. It may even be that procedures are improved further based on feedback gathered during development, so that the production process works even better.

Although the development keys are 'less secure', they should not be used carelessly. Following sound procedures for secure handling of keys will lead to less problems when handling the production keys.

3.5.2 Key generation

Secure keys that are never needed on the network during normal operations (the root and intermediate keys), should not be stored on the network during their creation either. It would be best to have a dedicated, secure device that generates all the keys and handles signatures (issues certificates). This device does not need to be connected to the network, does not need to have a hard disk even, but it does need an input device that contains the required software and the required signing keys and an output device to store the generated and signed public/private key pairs.

We appropriate a dedicated computer for this purpose, without hard disk and without network interface, but with a USB storage interface, a CD reader/writer and a printer. The CDs will contain the (encrypted) private signing keys of the root/intermediate certificate authority, a USB stick will contain the necessary software and the issued keys and certificates will be written to USB storage. The printer is used to make backups of the private keys

Since a computer with few external interfaces that is only booted for key generation/signing will collect little real entropy[3] (required to generate random keys), the USB input will always contain a file with random data from a network server. This prevents operations from getting stalled by the kernel, when there is not enough entropy available to generate real random

[3]Entropy for the pseudo-random number generator is generally collected from external I/O, such as the timing of interrupts from the network card, a microphone, keyboard and mouse events.

numbers; although such operation would continue once the operating system gathers enough entropy input later (e.g. by typing random characters on the keyboard). This external entropy data is merged with the entropy that the dedicated computer has gathered from other sources, before it does any cryptographic operations.

As explained in Section 3.3 the private Root and Intermediate keys each serve a clearly distinct purpose and they are generally not used at the same time. Therefore we will be storing each key on a separate device (CD-ROM). When for instance the Client Signing Key is needed to issue client certificates, the other keys can remain safely in secure storage.

So for each of the intermediate keys, we will need a generation process that securely creates this CD-ROM (and any backups). For the generation of the intermediate certificates, the root key will be needed as well. These CDs will be created using operating system images that were created specifically for this task. These systems are based on a Ubuntu (Linux) live system install and will start up with a menu to execute the required tasks. The CDs themselves will also contain bootable live systems, with menus to perform tasks that use the specific private keys (e.g. the Client Authority CD will contain a menu option to create a batch of *iphion* player keys).

The main managers of *iphion* will get a copy of the three certificate authority CDs. Each copy will store the private keys encrypted on disc, protected with a passphrase that can be entered by the person who will guard the CD-ROM. The reason to create multiple copies is for redundancy (availability): only one of the managers needs to be present with a CD for a specific operation; and production won't be interrupted if one of them goes away on vacation. Each person will be using his own passphrases, both to make these easier to remember (it might be a while before the CD is used and it wouldn't be useful to add a post-it with the passphrase); and to reduce the chance that someone would 'loan' their copies to someone else or leave them in the office for general use. The CD menus will include an option to create an additional copy of a CD (optionally with a different passphrase).

It was decided that there will be three managers who keep personal copies of the important keys (in secure storage). There will be different sets for the keys used in development and the keys used in production. The CD-ROMs have been created and distributed in May. The full key generation procedure is detailed in Appendix C.

3.5.3 Key backups

It would be a grave problem if one of the certificate authority keys were lost for *iphion*. Basically this would mean that no new certificates could be

installed, and thus no new servers or *iphion* player devices could be created.

If this happens, a new X.509 hierarchy (subtree) would need to be set up, and all existing devices would have to get to know the new keys that are involved, without discarding the old keys. The practical implications would be almost as worse as when a key would be compromised. And the operation might be rather expensive.

Although multiple copies of the secure keys makes it harder to keep each copy safe (and the least protected copy is the most important), backups are definitely required to guarantee continued operations. Since CDs might get corrupted after time or in accidents and passphrases might be forgotten, it is useful to keep a copy by more conventional means. A key printout on paper (without encryption) might seem rather insecure at first, but actually it is quite well understood how a piece of paper might be protected against misuse (theft) and accidents (fire). Also it doesn't suffer from technological progress: the encrypted CDs might need to be replaced within 10 years, because the CD technology becomes old-fashioned and is abandoned (not just the device, but also the filesystem and the file format themselves might change). The paper could even be stored in a bank or with a notary, as it would hardly ever be used: only in the case that the electronic copies would fail, is this last resort required.

Typing over multiple pages of hexadecimal characters from paper to recover a usable electronic version is not a trivial task, with a great chance of making mistakes. Therefore we wrote a version to let the printout include progressing checksum values for each line and each page. This way the location of a typo can easily be traced and fixed when the message is copied. Next we tried to automate this process by scanning the paper and using OCR software to recover the data. This actually produced quite good results (especially when we reduced the character set further). Looking for better ways to recover a large set of binary data from paper, we came up with the idea to use 2D data matrix barcodes [idct06]. In this form we could represent each keypair with one or more pictures rather than a very long list of hexadecimal numbers – and the computer would do a flawless restore when we put the image in a scanner. An example created with `libdmtx` from a demo key (not used by *iphion*) is shown in Figure 3.5.3.

The barcodes still rely on specific software, but because of the good results in converting it back to electronic data, we decided to create both versions: a full textual printout of the keys (in hexadecimal characters) and a condensed image in the form of a 2D barcode. These pieces of paper can be generated by the CD menus and are to be stored securely in a safe, where they can remain unless the electronic copies of our keys are lost in an incident.

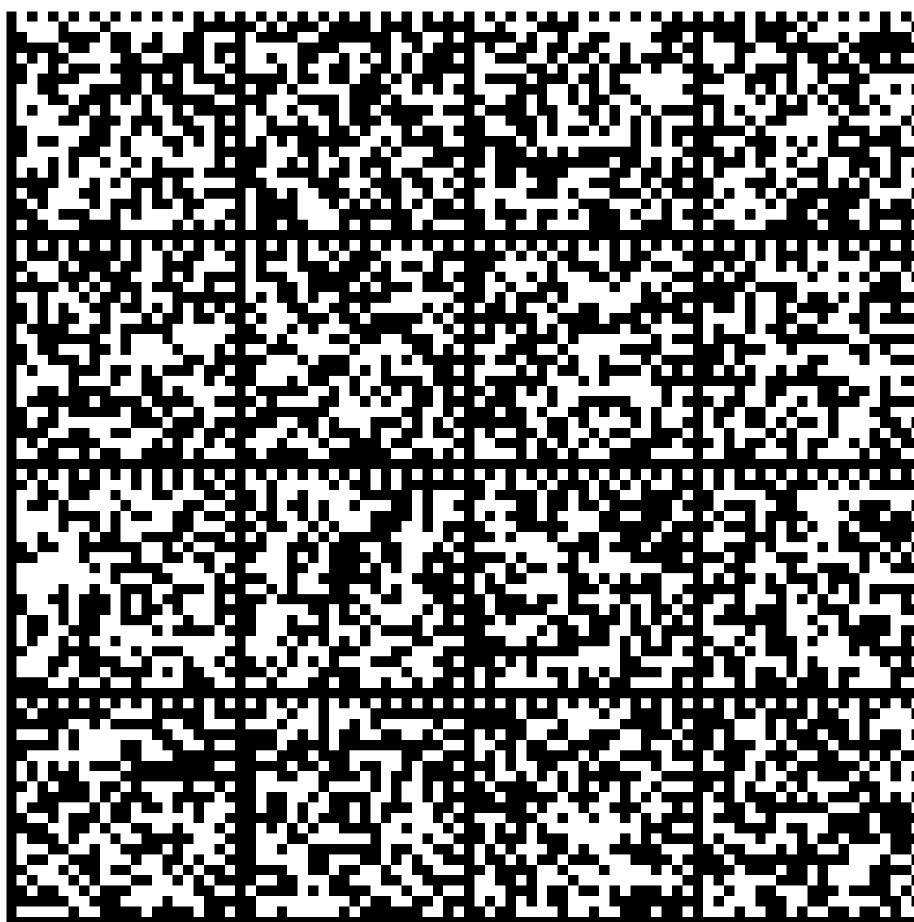


Figure 3.6: 2D barcode representing a part of an RSA private key

3.5.4 Key revocation

If a client or server will be taken offline permanently, the corresponding certificates should be invalidated as well. The reason to take a machine or service offline might be because it has been compromised, but it might also be for operational reasons. In a Public Key Infrastructure certificates may be invalidated by the authority that had signed the certificate. This authority issues a so called Certificate Revocation List (CRL) that includes the certificate serial numbers of all the invalidated certificates. This list and all updates to it will by signed by the relevant certificate authority (either client CA or server CA).

Every client and server in the system will check this list during every TLS handshake and abort a connection if a revoked certificate is offered.

Therefore, all clients and servers must have access to an up-to-date version of this CRL. There are basically two ways to access this information: either by doing an online check with a central CRL server or by using local files that are updated whenever certificates are added to this list.

The Online Certificate Status Protocol (OCSP) may be used to facilitate real-time online certificate checks [MAM+99]. Both *iphion* clients and servers could use this service to check the validity of certificates offered by their communication partners.

The main disadvantage of an online check is that it creates a single point of failure and an additional communication bottleneck: if this service is unreachable, no TLS communications ill be possible in the *iphion* system. The system already includes other critical services, but adding more should not be done lightly.

The main problem with local lists is that it will be difficult to keep all the distributed copies up-to-date. A new CRL could be included in a file system image for the clients, or application data package for servers, but these images are only updated with the occasional software release. Taking (old) services offline permanently may frequently be combined with the roll-out of a new release, but to make a CRL update possible only in combination with a software release is needlessly restrictive.

Note that clients will not directly communicate with each other via TLS, so the clients would only need to consult the server CRL. Servers should check both the client and server CRL. One server in particular: the token and key server, that grants clients tokens to participate in the collaborative content distribution exchange (iPAP) and decryption keys to descramble the delivered content, must always have an up-to-date list of the client CRL. Revocation information on the other servers and clients is slightly less critical.

Periodic updates to invalidate obsoleted certificates are considered sufficient for now. More analysis about the possible impact and extensive testing in development are necessary before an OCSP service may be rolled out and deployed for the *iphion* system in the future.

3.6 TLS software

Support for the TLS protocol suite is available in many modern communications libraries and for nearly all programming languages. Rather than trying to implement TLS authentication and encryption ourselves, it makes sense to use existing tools and libraries for this job. Using existing software is more secure –as it's hard to get everything correct when building your own

cryptographic solutions– and cost-efficient – because it reduces development time.

The OpenSSL library is very popular: OpenSSL is an open source library that has been around for quite a while, it is actively maintained and is released under a permissive license. The core of OpenSSL has been audited and approved by the U.S. government for use by federal agencies and contractors, via the FIPS 140-2 validation process [NI01a]. The OpenSSL toolkit also includes tools to generate X.509 keys and certificates and handle signing, validation, encryption and revocation of such certificates. Other cryptographic algorithms that are not used in TLS, are available in the OpenSSL library as well.

Open source alternatives include GnuTLS, which is newer and includes less features (e.g. no session support, no elliptic curve cryptography); and libnss (Network Security Services) which is the successor of the original implementation of Netscape's SSL. Both GnuTLS and libnss are libraries that specifically provide a TLS toolkit, while OpenSSL has evolved into a more general cryptographic toolkit. This means that OpenSSL is also better suited for use in the areas where TLS will not suffice – such as the content distribution network where a Diffie-Hellman key exchange is done. OpenSSL is also the only one with the FIPS certification, although libnss has also entered the certification process [Pro09].

OpenSSL had the additional advantage that it is included by default in the development environment that is used for the *iphion* players (Open-Embedded Linux). Unfortunately the version that they use, is rather old (OpenSSL 0.9.7e from 2005) and has several issues that have led us to upgrade this to a more recent version (0.9.8g) anyway (OpenSSL 0.9.7 cannot load DER encoded certificates via the general API; and it cannot handle SHA-2 signed certificates). For the certificate generation procedure we even had to upgrade to OpenSSL on our server to version 1.0 (0.9.8 cannot handle dates beyond 2038 on 32-bit architectures).

Neither OpenSSL 1.0, nor the stable releases of GnuTLS and libnss, support the latest TLS v1.2 standard that includes higher-grade cryptographic algorithms than those used in previous TLS versions [DR08]. However it should be no problem to update the *iphion* applications to a newer OpenSSL version that does include support, when it becomes available in the future.

Another program that did not readily support SHA-2 signed certificates was `curl`: an OpenSSL-based application to retrieve data from a webserver. We have added support for this ourselves and I have submitted a patch to the developers that will be included in the next release.

3.7 Summary

This chapter described how Transport Layer Security (TLS) may be deployed at *iphion* to authenticate communication partners and achieve confidential communications for much of the data exchanges in the system. However, the use of TLS is limited to TCP connections and will not be used for the exchange of multimedia content data via *iphion*'s content delivery network.

We have set up a secure certificate hierarchy so that TLS communication partners can reliably identify genuine *iphion* servers and clients without the need to keep a record of all individual identification certificates. On the *iphion* player devices, these certificates will be stored in read-only hardware so that these are linked to specific hardware devices. As an additional authentication mechanism we use hardware crypto support from the STB, so that the private key alone is not enough to authenticate an *iphion* client: this guarantees that the client keys can only be used in combination with the *iphion* client hardware.

We have also designed a protocol for secure generation, storage, backup, usage and replacement of the keys and certificates. The keys that are at the top of our certificate hierarchy will use more expensive cryptographic operations, be used less frequently and be stored more securely than the keys lower in the hierarchy. The root and intermediate signing keys will never be stored on a network-connected machine. We briefly analysed the impact should any of the keys get lost or be compromised and we suggested possible remedies in case this ever happens. We did not analyse the costs resulting from a possible key compromise.

Finally we discussed the software support to implement the designed TLS solutions. Although TLS is very standard and generally well-supported, there are still some minor issues with our chosen approach that require minor modifications for some standard tools.

4. System integrity and secure updates

In eliciting the requirements of the *iphion* system, it became clear that we will need to support software updates of *iphion* players (Section 2.3.2.3). Software updates will be used to install protocol updates, new features and security improvements. To guarantee integrity of the distribution network and confidentiality of the multimedia content, we should make sure that the players will only run genuine *iphion* software and not some modified version.

In this chapter we will describe a method to implement this requirement on the *iphion* player hardware, the NXP STB 225 model. We will also describe a method to recover from a situation when software integrity has been compromised. This might happen if an software update is aborted before completion or when unauthorized modifications have been made to the software.

An *iphion* player should be able to join the *iphion* content distribution network and display a basic set of channels without the need for external registration, smartcards or other keys or passwords provided by the user. In theory, a customer should be able to buy an *iphion* box of the shelf in a shop, plug it in to his internet connection and television set and be able to watch television channels. When a player connects for the first time, it will be automatically registered by the *iphion* servers.

This implies that all cryptographic keys and other security controls to get started need to be present on the box itself and available to the software that is running on it. As a consequence if somebody can figure out how *iphion*'s software works and replace it with his own software, he will be able to directly access the cryptographic keys and thereby the multimedia content that is available to the player. Obviously, *iphion* will try to avoid this scenario.

It is relatively easy to secure software so that it can never be modified, but it gets harder when there is also a clear demand for automated remote-controlled software updates. Since the user will have full control over the

player and its network environment, he might be able to point the box to an unauthorised source for the software updates. A customer might even be able to open the box and replace the software that is stored on the hardware (flash memory) directly.

Apart from a mechanism to fetch and install software updates, we will also need a mechanism to be sure that the *iphion* player will only run authorised software; rather than just starting with whatever is installed. In the next section we will discuss how system integrity may be checked and guaranteed. In the following sections we will describe how updates can be done without losing this integrity.

The initial sections of this chapter describe our research into the possible options for ensuring software integrity. In the first section we first describe the security features for software validation, that are readily available in the set-top box hardware and standard installation. In the following section we describe three alternative methods to secure software updates. After that discuss how a rescue boot operation should work.

In the final section of this chapter we describe how these methods may be applied to improve the security of the *iphion* set-top box. This section includes a discussion of the additional functionality that needs to be incorporated in the system to make this the presented implementation work well.

4.1 Secure boot mechanism

The STB 225 includes a facility to ensure only verified code can be executed. Secure boot ensures that only authorised software can be run on the device. If the software in the device differs by only a single bit, the verification will fail and the software will refuse to start. The ability to authorise software is dependent on ownership of a private key. The corresponding public key is then embedded in the software image itself.

Secure boot is not enabled by default and can only be enabled with assistance of the hardware providers, as it secures the entire execution chain, starting with the hard-coded chip that is provided by NXP. Secure boot uses cryptography to establish a chain of trust between the software elements. There are five such software elements in the platform:

1. On-chip ROM
2. First-stage bootloader: Aboot
3. Second-stage bootloader: U-Boot
4. Linux Kernel
5. Filesystem

The on-chip ROM is hard-coded into the chip and as such cannot be changed. This provides the root of the chain of trust. Aboot is a very small and simple bootloader: it detects RAM and flash chip settings and it will execute U-boot, which is stored in flash memory (NAND). The second stage bootloader is more elaborate: U-boot can read and write flash partitions and even has limited networking capabilities. U-boot will locate the Linux kernel and root file system partitions. It loads the kernel into memory and starts it with the correct arguments.

Secure boot works by ensuring that every element of the software verifies the integrity of the following element before allowing that element to be loaded and run. So the on-chip ROM verifies Aboot before loading and executing it, Aboot then verifies U-Boot before that is loaded and executed and so on. In the STB 225 software, U-Boot verifies both the Linux kernel and one or more filesystem images.

Changes are detected based on digital message digests. The digest is generated using the SHA-1 algorithm, which is a cryptographic hashing function. SHA-1 creates a 160-bit representation of an arbitrary length piece of data. Although this means cryptographic hashing functions are many-to-one functions (i.e. multiple source data will map to the same hash), the algorithm has the property that a single bit change in the source will produce a very different hash. Thus it is computationally infeasible to find another source image that will produce the same digest (and in the case of software, even harder to find an executable source image with the same digest).

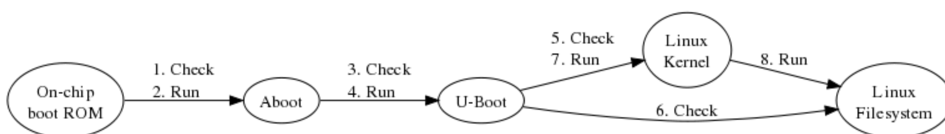


Figure 4.1: Secure boot architecture

In order to protect the digest against tampering, it is cryptographically signed. The algorithm used by the bootloader is RSA, with a 1024-bit keys. RSA is based on public-key cryptography, in which the signing (encryption) key is different from the verification (decryption) key. Thus the verification key cannot be used to recreate the signature, and so it is secure to embed this key in the device - only the private key needs remain confidential.

Most stages can be signed using a different key if needed, although, the kernel and file systems need to be signed using the same key since U-Boot verifies both these stages. The primary bootloader will be signed by the chip producer, the secondary bootloader by *iphion*. This is a one-time operation as and these will never change after production. The primary

bootloader will include the public key needed to validate the signature on the secondary bootloader. The secondary bootloader will include the key needed to validate the system images (kernel and root filesystem).

This means that the kernel and filesystem images can be replaced after production, as long as the new image is also signed with the same RSA key. However, this key itself may not be changed as it is embedded in the bootloader. U-boot will need the exact public key that corresponds with the private key that is used to sign filesystem images: it does not support signed certificates and can not use the hierarchical delegation of trust within the public key infrastructure that we have set up (described in Section 3.3).

During a normal boot sequence, U-boot will perform the following steps:

1. check the U-boot header (crc32 and RSA-signature) of the kernel image in flash and loads it into memory on success.
2. check the U-boot header (crc32 and RSA-signature) of the root filesystem image from flash.
3. execute the kernel in memory, which will load and mount the root filesystem image

If any of these steps fail, execution will be aborted and the installed data images will not be used. The system will then enter the so-called **rescue mode**, which is described in Section 4.3.

After researching how the secure boot mechanism of the STB 225 device works, I gave a presentation about this topic to *iphion* management and developers. After the presentation it was decided that we will be using all the mentioned security features that are supported by the device. The following sections will describe the options to do software updates and rescue operations. After that we explain how all these features may be used in practice: how to update and rescue an *iphion* player system using the outlined secure boot features to assure and maintain system integrity.

4.2 Software updates

For regular software updates, we have a lot of flexibility. These updates can be done from a normal operational environment, that is a complete Linux system. In the following sections we consider alternative methods such as per-application update packages and binary filesystem patches. We will also consider methods to ensure that only software that is distributed by *iphion* may be installed and executed.

4.2.1 Application updates (ipkg)

The Itsy Package Management System (**ipkg**) is a lightweight software package management system designed for embedded devices [WAH06]. The ipkg method of software updates is already used at *iphion* for the prototype player devices. These are based on PC hardware, unlike the final *iphion* players which will be set-top boxes: initially based on the NXP STB 225 platform, with a flash chip for data storage and a MIPS-based primary CPU.

ipkg is based on the package management of modern Linux distributions, such as Debian. All the software applications are available as separate packages and each package may be upgraded individually. The package manager not only takes care of removing the old version and installing the new files, but also of shutting down the applications from the package before upgrading and restarting it afterwards. In this way applications may even be upgraded without the need to reboot the whole system – which reduces impact for the user. Of course, the upgrade of certain packages (such as kernel device drivers) may still require a system reboot.

Unfortunately there are some short-comings of ipkg that became prominent when preparing deployment for the set-top box clients.

- ipkg is not sufficiently reliable: it does not always leave the system in a consistent state, especially when the upgrade is interrupted because of power losses or something similar.
- ipkg would have to modify a filesystem where software is already running, possibly causing problems with the running software.
- We will want to use all the security features that NXP implements in the STBs chip and the security features that U-boot offers. To use these features to their utmost extent, we need to be able to generate signatures for everything that is not dynamic on the system. The non-dynamic parts include the kernel and the root filesystem.
- We would like to have all STBs in exactly the same state. This can not be guaranteed with ipkg as we have no control over where updates are written in the filesystem partition.

4.2.2 Signed executables

For the Linux kernel we can use a signed and encrypted kernel image that is updated only in a secure way and can be checked and validated by U-boot every time before the kernel is loaded.

However we can not use a signed, read-only filesystem for our Linux installation; primarily because we will want to upgrade individual packages or files on occasion. Updating the entire filesystem for every minor change would make updates slow and would require a full machine reboot for every update.

These packages should be downloaded from an authenticated source over a secure connection. However this will not make it impossible for users to read and write to the local filesystem. After all, the enduser will have full control over the hardware (and the network).

So ideally, the kernel should only execute files of which we can be sure that they are distributed by us. One way to do that is by using signed executables. When signing executables every file includes a signed checksum that can only be generated by a private key in our possession. The checksum of the binary (SHA-1) and the validity of the signature (RSA) should be checked by the Linux kernel itself before the binary is executed (using a hard-coded public key). This signature information may be included in the ELF headers of the executables.

When signing executables, all shared libraries that are loaded should be signed as well. Obviously tampering with a library may change the behaviour of a program just as effectively as changing the program itself.

All Linux kernel modules that are loaded should be validated as well. A kernel module would be able to trivially circumvent any checks that we want to enforce in the kernel - or it could just run arbitrary code, like a userland program might.

All scripts should be checked as well: if we ship out a command interpreter (shell) or general language interpreter (Python, Perl), then scripts in these languages are as much a security risk as any binary executable. Checking the validity of a script's checksum and signature should ideally be done by the interpreter itself. This will probably require some custom hacks as well.

4.2.2.1 Trusted executables

There is no support for signed executables, libraries or modules in current version of the Linux kernel. At the moment there aren't even any kernel functions that offer public-key cryptography (such as RSA). However several projects have added such functionality in the past. The most promising project is DigSig: this project has fully functional releases and is used by others [Can05].

DigSig (GPLv2) offers a Linux kernel module that can validate all ELF executables and shared libraries in a system just before they are executed. The crypto functions are based on GnuPG and the signing scripts can use

gpg keyrings and commands. The suite includes scripts to facilitate signing all executables and libraries in a directory tree.

DigSig uses SHA-1 checksums and RSA-1024 (or 2048 bit) signatures. Generally this requires a SHA-1 and RSA calculation every time a executable is started - usually multiple operations since shared libraries are checked as well. However some of this data may be cached in kernel memory: this speeds up loading of unmodified files a lot. See the website for some more performance statistics.

At the moment DigSig is only available as kernel module and has both a soft (only warn) and hard (prevent execution) mode. To prevent any chicken-and-egg problem, the module should probably be adapted to make it linked directly into the kernel with a fixed (hardcoded) RSA keys. This also makes sure that the module cannot be unloaded.

4.2.2.2 Trusted kernel modules

The second step is only using signed kernel modules. A custom kernel module can possible influence the system worse than any userland executable. The easiest way to prevent this is by not allowing any kernel modules. However modern Linux installs generally to rely on modules and the STB 225 includes quite some NXP-specific modules as well

There is currently no Linux kernel-support for validating kernel modules. However Red Hat's Fedora distribution has been using signed kernel modules for years (2004-2008).

Fedora-7 and -8 had this feature enabled by default (Fedora-8 is still supported). Red Hat's 'modsign' kernel patch [How07] will only warn about insecure modules - not prevent usage; however this is probably easy enough to change. Mid-2008 these patches from Red Hat were sent upstream for inclusion in the general Linux kernel. However kernel maintainers didn't see much advantage in it and never included the feature. Newer Fedora releases (9 and 10) no longer include the modsign patches.

Modsign uses the GnuPG cryptographic library [Koc09] with SHA-1 checksums and it assumes a hardcoded (public) RSA key in the kernel. Performance is less of an issue here because modules are only loaded once - generally at boot-time. Booting will be slower when many modules are used. Like DigSig it uses additional ELF headers to store the signature information. Scripts to sign modules are included in the patches.

4.2.2.3 Trusted scripts

There will be no general interpreters installed on the set-top box. However there will be a general command interpreter, the /bin/sh shell. This is a limited ash clone (by busybox) that includes job control and mathematical functions.

The shell is required for general Linux functions (boot scripts etc.). It is possible to modify the shell so that it can only be used to run signed scripts and won't run as a general command-line interpreter. However, it's doubtful if this is worth the effort...

4.2.3 Binary filesystem updates

Instead of signing all individual packages or executables, we can sign the entire filesystem image with a digital signature. This will be easy to check, even before a kernel or filesystem is loaded. If as much as a single bit differs, then the filesystem image would be rejected. Here it doesn't matter if this difference occurs in a binary executable, a script or any other data on the filesystem. This method is simpler and arguably more secure than the two alternatives discussed previously.

An important advantage of distributing and updating identical filesystem images is that we can guarantee that each STB is in exactly the same state. If we know the software revision of a system, we immediately know exactly what is installed and where in the partition everything is.

Binary filesystem updates are intended to keep the filesystem on all clients in a consistent state. When an update is required, we will only need to modify the filesystem blocks where changes are needed.

However, there are also some drawbacks to this alternative:

- We do not want to fetch a full filesystem for each update. Patches should be generated that are as small as possible, to minimise the delay for users. Not writing the entire filesystem, but only specific blocks, also lets the flash memory live longer as the total number of block rewrites is limited.
- A reboot is required after every update. Updates are not limited to individual packages, but may affect other applications as well, because write actions are done in filesystem blocks. And the filesystem integrity should be checked again by the bootloader when the update has finished.
- To prevent having to shut down all the software before on a system during the upgrade process, we will use an initramfs (memory filesystem) on the STB, so that the flash is available for writing while software keeps running. It is not a major issue as most files in the initramfs will be in use anyway on a running system and therefore loaded in memory already.

  This means that the memory filesystem does not waste a lot of memory space by keeping unused files.
- Developers will need to write additional software to create, optimise (block-based) and apply patches for the flash filesystem.

Even with these drawbacks, this still is the best alternative, both regarding integrity guarantees and for maintainability.

4.3 Rescue boot procedure

The rescue boot procedure will overwrite the system disk images of an *iphion* player device with a fresh image that is obtained from *iphion* via the network. This is not the normal method to upgrade an *iphion* box (described in Section 4.2), but rather is intended as a safety net in case the regular method fails. The rescue boot procedure that is described in this section will use the STB 225 secure boot provisions that are outlined in Section 4.1.

The bootloader will initiate the rescue operation in one of two cases: if the kernel or root filesystem on the machine itself are corrupted or if the rescue procedure is manually triggered by the operator. In the *iphion* player design a user can force the rescue procedure by holding down the power button for at least 5 seconds. This feature will probably not be in the manual, but may be suggested by the helpdesk.

A system image is considered corrupted if the corresponding digital signature (stored in the file system header) no longer matches the actual contents of a kernel or file system image. This corruption can occur when a regular update procedure has gone wrong (for example by network problems or a power interruption) or when the file system has been changed by unauthorised parties: either the owner of the device or external hackers. If a system image has been changed, the bootloader will always refuse to boot using this image.

When the rescue boot procedure is triggered, the U-boot bootloader will download, validate and execute a rescue image. Because of technical limitations of the bootloader (like the lack of a TCP stack), this will only be a small execution environment. This rescue kernel will then proceed to download the full system images for the *iphion* player. These images will be validated as well, before writing them to disk, replacing the corrupted data. The player will then reboot and boot the freshly installed system image.

4.4 iphion player

In this section we will discuss how the secure boot features outlined earlier (Section 4.1) will be used in combination with software updates (Section 4.2) and a rescue mechanism (Section 4.3) for the *iphion* player systems.

For the deployment of these features, several additional facilities need to be implemented. The required features will be discussed in the second part of this section. This include the generation and use of extra cryptographic keys, modifications to the existing bootloader and the development of a rescue environment (kernel and applications).

We conclude the section with a brief consideration of the legal issues that affect the protection of the installed software. Conflicting licenses of open source and proprietary software might lead to unwanted situations if not handled carefully.

4.4.1 Secure boot mechanism

To facilitate a secure bootstrapping order, every boot step will need to be validated before it is executed. Booting starts with a hardware chip that only does self-checks, but is not externally validated during start-up. This chip will validate the first-stage bootloader, which is also provided by the hardware supplier (NXP). This step, like everything that follows, includes checking the checksum of the contents and a digital signature. When this does not match, execution will halt and the player box is essentially useless and the player box will need to be returned.

The first-stage bootloader (Aboot) will validate the second-stage bootloader (U-boot). Aboot includes the public key (provided by *iphion*) that is needed to check the digital signature of U-boot. The second-stage bootloader will be created and signed by *iphion*, in collaboration with Prodrive[1] and *iphion* together).

The second stage bootloader will load the filesystem and Linux kernels that are used for normal operations of the machine. These system images will be created by *iphion* and therefore U-boot will need to include the relevant *iphion* public keys. The private image signing key will always remain in possession of *iphion*: nobody outside the company ever need access. Obviously, this means that U-boot has to be created with the relevant keys before it can be signed by *iphion* and embedded into the player device by Prodrive.

[1] Prodrive is the partner company that assembles the hardware. This includes the read-only memory that has the bootloaders.

For the decryption of a rescue image the bootloader will need an additional cryptographic key. This will be a symmetric decryption key (AES) that is the same on all the *iphion* players (of a specific generation). This key will be embedded in the bootloader, together with the public key to validate system images. The rescue image will be signed with the same key as the normal system images.

The *iphion* player devices are produced in (large) batches. Within a batch all devices have the same hardware specifications and will use the same public key in the bootloader. However when a new batch is created, another key may be used. The software update mechanism should be adapted so that it can deal with specific images for specific batch revisions. Since the hardware may change with each new batch as well, these specific images should be implemented anyway. Changing the signing key also means that the impact remains limited if a key would ever be compromised. It does increase the complexity and the administration of the supporting systems.

4.4.1.1 Signing filesystems

Both the kernel and the application filesystem image will be signed using a digital signature that can be validated by the bootloader. If either signature does not match with the image contents or the public key that is known by the bootloader, the system images will not be used and the bootloader will start an on-line rescue procedure instead (Section 4.3).

The private key to sign system images can be kept offline. It is only needed when a new software version is released for the *iphion* players. For the signing procedure we can use the secure environment setup that is also used for signing client and server X.509 certificates (Section 3.3).

The image signing key can be treated as an intermediate certificate, just like the client and server certificate signing keys. However there is no need to include this key in our PKI, as the bootloader will not be able to check the certificate chain anyway. We extend our signing policy to include the image signing key and a signing procedure. This procedure is completely analogous to the certificate signing procedure. An additional CD-ROM with the private key and the appropriate backups are included as well. The updated procedure is explained in Appendix C.

The public key that corresponds with the private image signing key will be included in the bootloader that is fixed on all *iphion* player devices. This will be hard-coded and cannot be modified after production of the system. However new production batches (hardware revisions) may use other signing keys. Since system images are generally linked to hardware revisions anyway, changing this key for future revisions shouldn't cause any

problems. However old keys cannot be revoked while they are still used on old devices that remain active in the *iphion* network.

The signed filesystem images can be stored online. They will be used in the production of new players, they are used to create binary patches from old revisions, and they are placed on the rescue server in case a player needs to fetch a full fresh system image.

4.4.1.2 Applying binary updates

Given filesystem images for different software revisions, we can generate binary patches that only included the changes between two revisions. These patches can then be used by the binary filesystem updates mentioned above. By only including the differences, the time required to download and apply (write) this information is greatly reduced.

Still the differences can be quite large. To optimise this process, we try to minimise the patch in the number of data blocks that are rewritten in the flash partition. When applying a patch, data can only be erased and written in full blocks: to even change a single byte, the whole block is erased and written again. The total number of erase and write operation during the lifetime of flash storage is limited, even though the limit is pretty large. Still it is worth the effort to optimise the patch not by minimising the number of bytes, but the number of blocks that need to be replaced.

Tools to create, optimise and apply these binary patches have been written and tested at *iphion*. When a new software release is created, patch files can be created to update not just from the latest version, but from older versions as well. These patch files can be stored on the update server `update.iphion.nl`, from where they may be retrieved by the update application running on any *iphion* player system. This retrieval will use TLS authentication and encryption (via https). The update server will keep a log that shows which patch has been downloaded by which player. However the server cannot be sure that a client upgrade was completed successfully, until the next time that this client contacts the server.

4.4.2 Software updates

Different methods to distribute software updates will be deployed at *iphion*. Signed filesystems with binary patches (Section 4.2.3) will be the standard method to update software on the *iphion* set-top boxes. For the *iphion* servers, software updates using individual signed packages for each application (Section ??) will be used. This is also the method that is used on the older *iphion* player machines, which are Intel PC-based rather than NXP set-top boxes.

When public cryptographic keys need to be updated (added or revoked), then these keys will also be included in the regular updates. These updates should always be signed with trusted keys in the *iphion* Public Key Infrastructure, so that the 'chain of trust' is always maintained.

4.4.3 Rescue boot procedure

The second stage bootloader providers a rather limited environment. Usually this bootloader will determine the location of the Linux kernel and the root file system flash partition, validate the signatures of both system images and then boot the Linux kernel with the correct parameters. In this case the bootloader does not do anything network-related.

The U-boot bootloader includes a hard-coded RSA public key that can be used to check the digital signature. This digital signature in fact signs the SHA-1 checksum of the content — this is the normal procedure with digital signatures [NI09]. So the bootloader will do both a checksum calculation and an RSA validation for the kernel and the rootfs system images. The bootloader and the RSA public key are stored in read-only flash memory (NOR) on the system.

If the rescue mode has been triggered, then U-boot will use its networking capabilities: The bootloader obtains an IP address for the client using DHCP[2] and downloads a rescue image for the specific client (based on product revision number) over TFTP from a central server, `rescue.iphion.net`. The downloaded image is compressed, digitally signed and encrypted.

The bootloader does not have a TCP stack, but only a rather limited understanding of the Internet Protocol (IP) and support for User Datagram Protocol (UDP). This means that Transmission Control Protocol (TCP) based protocols such as FTP and HTTP can not be used to transfer data. TLS (Section 3.1) requires TCP-support as well, and also stronger cryptographic capabilities, so that is not an option here either. We can use UDP-based protocols, such as DNS (to find the IP address of the *iphion* rescue server) and TFTP (to transfer data).

The Trivial File Transfer Protocol (TFTP) only requires UDP and was specifically designed for bootstapping limited environments, [Sol92]. TFTP is usually used for local networks only: it does not support authentication or encryption in itself and it doesn't handle large files and retransmissions very efficiently. So it is rather slow and limited for our use. We can improve the performance by using the block size option (in the original protocol revision this is fixed at 512 bytes, but new extensions make this configurable). Tests

[2] A very limited implementation of DHCP, that is best described as BOOTP (RFC 951).

show that running with a 1024 has a clear positive effect on the speed. Since the TFTP operation won't be used often, the rescue image can be kept small, and we can add digital signatures and encryption to the data, it will be sufficient for our purpose.

To support multiple versions of player hardware via the same TFTP server, the file name that is requested by the client will include some information about the client system. In the rescue image naming convention the *iphion* product number (pp), generation number (gg) and hardware revision number (rr) are encoded. The rescue image consists of a kernel and rootfs (initramfs) concatenated back to back.

The data that is downloaded will be cryptographically signed with the same key that is used for stored file system images. This is the only public key that the bootloader will use. If the signature check fails, the rescue operation will be aborted and the process will start all over again. There is no alternative to try: if local file systems are corrupted and the rescue procedure fails in any way, the *iphion* player will just keep retrying to download and install a fresh rescue image — until it hopefully succeeds at last.

The rescue image is not just signed, but also encrypted. This prevents attackers from simply obtaining a copy of the rescue image 'off the wire'. Although there is no information in this image that is considered highly confidential, it should not be too simple for attackers to obtain the data either. They may still find a way to read the data from the *iphion* player once it has been decrypted; but that kind of attack is quite a bit harder to achieve than simply sniffing the network. For encryption of the data, symmetric encryption will be used with a shared (fixed) key that is stored hard-coded in U-boot, together with the public RSA key. For encryption we use 128bit AES, which is a reasonably good encryption method (strength comparable to 1024bit RSA, [BBB+06]) and the only algorithm natively supported by our bootloader.

Once the rescue image has been successfully validated and decrypted, it still needs to be decompressed before it may be used. The image is transferred with compression in order to minimise the download delay. The given order (validate, decrypt, decompress) is not only secure, but by checking the signature first we can also avoid wasting computation time on corrupted or maliciously injected data.

The unencrypted system image will be loaded into memory. The bootloader then executes the kernel in this image from its present location. After it has successfully initialised it will unpack an included root file system (initramfs) and mount it as memory filesystem. If this step successfully completed the rescue image is up and running. A graphical overview of the rescue boot procedure is outlined in Figure 4.2.

In the rescue environment the firmware download application will be started after the normal boot operations. This application then downloads the real kernel and rootfs image from one of *iphion*'s servers over a secure (TLS) connection and writes them to flash. Since we now use TLS authentication as well, this may even be a image that is crafted specifically for this box. After installation, the system will reboot and execute the newly installed system. The actions of the normal secure boot sequence will then be performed (Section 4.1).

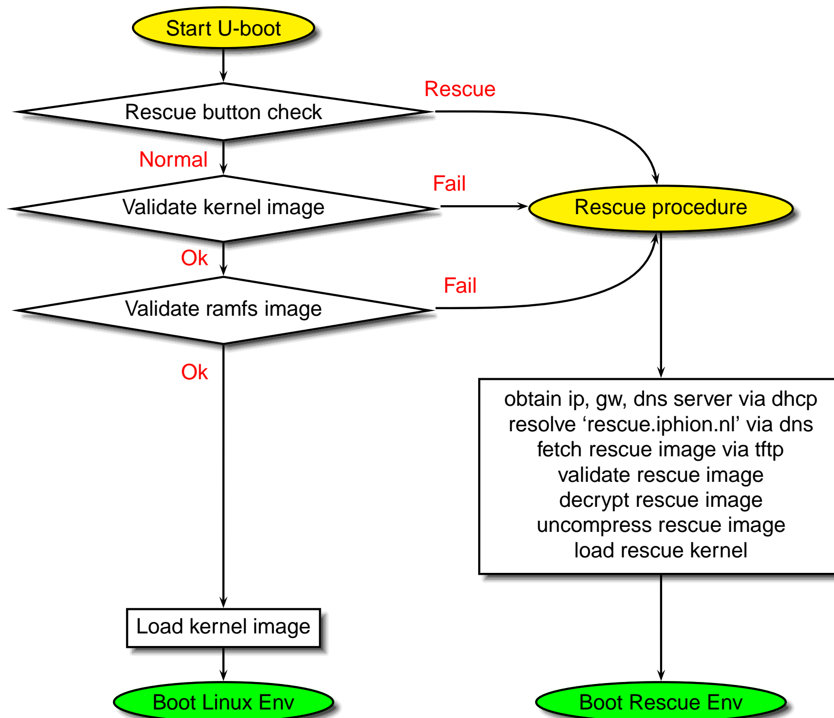


Figure 4.2: Rescue boot procedure

When the bootloader downloads a rescue image, there is no authentication of the communication partners: neither of the client nor of the server. This leaves the protocol open to man-in-the-middle and downgrade attacks. Should *iphion* ever release a (signed) rescue image with a serious security vulnerability, then users can keep a copy of that image and boot it at any later time to exploit this vulnerability. Users can not use modified rescue images, because the cryptographic signature is checked by the bootloader. As the rescue image is a rather simple environment that fetches the full-featured filesystems for the player (using two-way authentication), it should be possible to keep this rescue image secure. Adding authentication features requires considerable changes to the bootloader, which might introduce other problems in this critical part of the system. Therefore, it was decided

not to do this in the initial version (see also Section 4.4.5 about other bootloader modifications).

4.4.4 Cryptographic keys

Several cryptographic keys are added to the system to validate and protect the operations described in the previous sections. The staged booting steps use public key cryptography to validate each following step before execution. The key that is used to sign the second-stage bootloader (U-boot) and the key that signs the filesystem images will be provided and protected by *iphion*. The corresponding public keys are stored (fixed) on the player device (in read-only flash). And all software that is installed must be on a filesystem secured by the image signing key.

Another key provided by *iphion* is used for the encryption of rescue images that are distributed over an unprotected channel (via TFTP). This is a shared symmetric key (AES) that is stored on the box, also in the read-only flash chip. This key does not really provide confidentiality, but merely obfuscates the transported data. Both the key and the decrypted image can be read from the player device by a determined attacker.

The U-boot signing key will not be used for anything else. The corresponding key is stored in the Aboot bootloader and only used there. The image signing and the rescue image encryption keys will be stored in the U-boot bootloader. U-boot is also the only part of the system that needs to access these keys. These keys can be the same for all players using the same hardware, which is very convenient for mass production.

The bootloader versions that are provided by NXP for this system already includes provisions to include these keys. For digital signatures a 1024-bit RSA key will be used, for encryption and decryption a 128-bit AES key. The public part of the RSA key will be included in the bootloader, the private part is used to digitally sign system images. The AES key will be used both for encryption (when rescue images are created) and decryption (by the bootloader). We can just enable these features in U-boot and don't need to change any code for it ourselves. We do need a protocol to securely generate and store these keys and build a system image digitally signed with these keys.

We extend the secure TLS key generation process (Section 3.5) with additional options to generate these two keys on a secure computer as well. These keys are not directly related to the public key infrastructure and they will be stored on a separate device (CD-ROM). Extra menu options are added to create both the RSA and AES key, write these to a CD and create backups. For regular use a menu option is added to sign rescue file system

images in the appropriate format that the bootloader will understand. The full key creation and storage procedure is described in Appendix C.

The AES key that is used by the bootloader to decrypt rescue file images is very different from the AES key that is stored securely in chip hardware and used for improved authentication of the *iphion* players (??). The key stored in U-boot will only be used by the bootloader and is not considered quite as secure.

Both keys in the bootloader may safely be replaced by other keys in future hardware revisions. They are only used to check file system images and these images will always be linked to specific hardware revisions anyway. But once player devices are produced with specific keys, these cannot be replaced: the keys are stored in the bootloader which is located in read-only flash memory.

4.4.5 Bootloader modifications

Even though we mostly use functionality that is already present in the NXP version of the U-boot bootloader, some modifications are still required to use the secure boot and rescue scenarios described in the previous sections. We try to minimise the necessary modification, to keep our system more standard (thus better supported) and minimise the chance of breaking something critical in the bootloader.

Four custom features are still considered necessary:

- Add a simple DNS client (to resolve `rescue.iphion.nl`)
- Set TFTP packet size option to something larger than 512 bytes
- Include custom RSA and AES keys
- Add a static picture to inform the user while downloading the rescue images

Luckily, for the DNS implementation a well-tested patch has already been created by a third party and adding the TFTP blocksize option is rather trivial. The inclusion of cryptographic keys and a picture is not really a modification, as this is supported by the standard image. However these are hardcoded and inclusion means that files need to be changed and the bootloader recompiled.

The modifications to the bootloader will be coordinated between *iphion* and Prodrive, the company that supplies the assembled hardware (including the bootloader). The U-boot installation itself will be digitally signed as well,

with a key that is included in the primary bootloader. If the bootloader somehow gets corrupted later, it will not be loaded by the device and the box is essentially 'bricked' and will need to be returned physically. This bootloader signing is done by *iphion* and the corresponding public key will be incorporated in the primary bootloader by the chip manufacturer, NXP. The keys that are stored in the primary and secondary bootloaders are fully under the control of *iphion*: no other company will get access to the corresponding private keys.

4.4.6 Rescue image features

Once a rescue image has been loaded on an *iphion* player system, this rescue environment will perform the steps necessary to return the system into a functional state. It will download and restore any corrupted file system images and will reboot the system when everything is installed.

The rescue image should include at least the following features:

1. Authenticate with an *iphion* server (using on-box X.509 certificates)
2. Download kernel and file system images over secure link (e.g. https)
3. Write the digitally signed kernel and filesystem image image to flash (NAND)
4. Reboot

The rescue system can actually do a bit more. This is a fully functional system environment, which means that it can use all the device drivers and can play audio and video. This should be used to inform the user about the progress of the rescue operation (download, validate, install).

The private keys that are used to sign the images (corresponding with the public keys available in the bootloader and in the rescue image) should not be kept on a system that is online in the network. All the images can be signed offline when they are produced, and before they are uploaded to the rescue server.

4.4.7 Legal issues

There is also a legal threat to the software security of the set-top box. While many providers of software libraries and hardware chips alike are adamant that their work should remain untouched and confidential, open source software suppliers frequently insist that their work (and all derived works)

must remain open and accessible to everybody. With a single supplier this apparent conflict of interests would be no problem. However, *iphion* would like to use both open source and proprietary software libraries in its system; along with a lot of software that they developed themselves. This can only work if the software licenses are compatible. Some of the STB software and hardware is also protected by patents, this means that there are no real alternative implementations or licenses available.

A popular open source license, the **GPLv3** insists that when their software is distributed (for example installed on a set-top box), it must still be possible for the customers to obtain a copy of the source of this code (and any derived works) and actually let users replace the installed software on the set-top box, with their own code [FSF07]. In practice this means that *iphion* would either have to give up on filesystem signatures, of effectively publicise the signing key. Such a modified machine might compromise the functionality of the *iphion* network and it might leak information that must remain confidential. This is basically a legal problem, and there is no real technical solution. To safeguard integrity and confidentiality of the system, the licenses of all the software (and any hardware as well) that is used should be considered carefully and it must not be used if it is deemed a threat. For these reasons, *iphion* has chosen to avoid GPLv3-licensed software completely in the code base of its set-top boxes.

Some other open source licenses stipulate that the source of any derived work must also be open source. This also needs to be handled with care. There is no problem with opening up some parts of the system that are written by *iphion*; but this is not an option for other parts. Especially parts by others where licenses forbid publishing and the parts that are considered very important ('trade secrets') or that become an easy target for attacks when published (although there should be no such code in the *iphion* system). This license issue may be avoided by integrating parts of the software system in ways that does not create a derived work, but keeps them as stand-alone parts. It is something to keep in mind during development though.

4.4.8 Summary

In this chapter we described a secure mechanism to boot and update software on the *iphion* player devices. This mechanism assures integrity of the software that gets installed and thereby maintains confidentiality of the authentication key that is embedded in the *iphion* player hardware. Without this key parties cannot join the *iphion* content distribution network and they will not receive any decryption keys needed to descramble the distributed multimedia content.

Apart from regular software updates, we also worked out a secure mechanism to restore a box to a functional state when the software on it becomes corrupted (either accidentally or by tampering with it). This makes the software updating system more robust prevents boxes from getting unusable ('bricked') too easily.

To facilitate these procedures we introduced additional cryptographic keys that will be used during the booting and updating operations. The previously described key generation and management procedures have been extended to include provisions for these keys as well (Appendix C).

The outlined mechanism requires some additional software: the bootloader will need some minor modifications (Section 4.4.5), applications to create and install software updates need to be created (Section 4.4.1.2) and a system to perform a full rescue operation for the *iphion* player is required as well (Section 4.4.6). This software has now been created by the development team at *iphion*.

The added software integrity checks do not guarantee that the *iphion* players will only run validated *iphion* software. The filesystem signatures are only checked at boot-time. An attacker might manages to hack into the player while the system is running, by exploiting bugs in the installed software. He may then install and run his own software. This may continue unnoticed until the next reboot. If additional software has been installed on the filesystem, it will be removed with a rescue procedure when the player restarts. But once a bug has been found, it would be trivial to exploit this again after each restart.

5. Content distribution

As we have seen in Chapter 2, content availability and content confidentiality are important aspects of the system requirements. Streaming high quality multimedia content to many customers requires fast and dependable distribution of a lot of data. Keeping this data protected against unauthorized use implies encryption during transport and proper authentication of those who will be allowed to decrypt and use this information (Section 2.3.1.2).

The *iphion* content delivery network is responsible for the timely delivery of multimedia data to all clients in the network. The network consists of *iphion* content servers that transmit and relay traffic and all *iphion* clients that receive and optionally forward traffic to other clients. Data is distributed in a collaborative (peer-to-peer like) fashion, rather than sent via unicast streams from the servers to all the clients.

In this chapter we will take a closer look at the communications protocol that is used in the content delivery network. In the first section, we briefly describe the use of these communications. In the following three sections we describe how security aspects are handled in this protocol: We focus on authentication, integrity and confidentiality respectively.

5.1 Communication streams

The content delivery network strives to deliver continuous multimedia streams to a lot of clients in a very short time. The multimedia data is divided into separate channels, like regular television channels. All users on the same channel will receive the same content. For each channel there is one central streaming server that injects the data into the network. This broadcaster will only send the data to dedicated *iphion* servers (repeaters). These repeaters will be placed at strategic locations in the *iphion* service network (e.g. at an internet access provider). The repeaters will forward the data to the clients (*iphion* players).

To speed up content delivery and avoid congestion, data will be split up into packets which may be delivered to the user via different paths. These

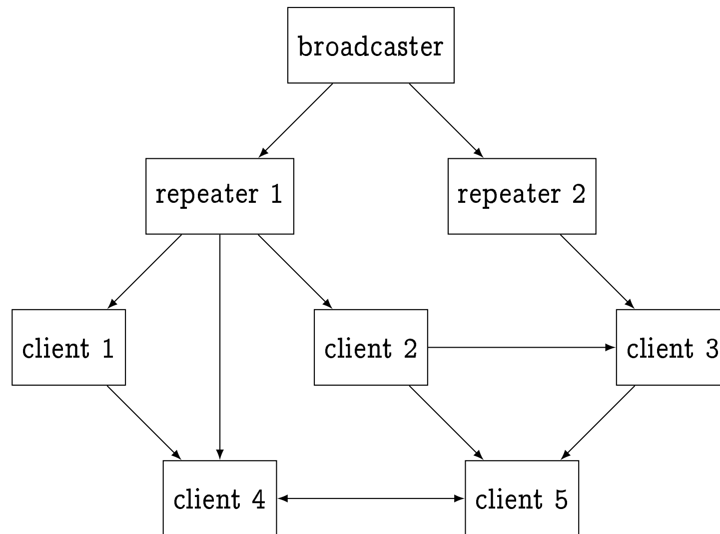


Figure 5.1: A simple iPAP network diagram

packets are the standard units of an MPEG-2 Transport Stream (MPEG-TS) [oapmhi00]. The packets are delivered via a peer-assisted protocol: All clients may forward received packets to other local clients as well, rather than all of them fetching the same content from a remote *iphion* server. So a client may receive packets for a single stream from different sources (repeaters or other clients). Figure 5.1 shows an example of the content data streams in a part of the network.

All multimedia data may be forwarded unmodified to all clients: there is no client-specific data in the content itself. And although the protocol strives to deliver all the data to all the clients (with retransmits via the same or alternative routes where necessary), it may happen that a client doesn't receive all the data in time. In this case the client will display what it has and may need to skip audio or video frames when there are bits of data missing. Packets may also be received out-of-order, in which case they will be reordered by the recipient. This distribution network is dynamic, especially at the client end: players may be turned off or change to another channel at any time.

When a client wants to start fetching content, it will first need to find out how to contact *iphion* servers that relay data streams for the selected channel. Standard channel list information is available from a central *iphion* server, the 'profile server'. This list includes information about the *iphion* repeaters offering channel streams. Typically a client will retrieve this list via TLS (https) right after start-up. The channel names from this list will

be presented to the user who may then select a channel to watch.

When a channel has been selected, the client will contact one of the *iphion* relay servers listed for that channel. In response to a channel join message, the relay server will present the client with a selection of active peering partners from which it may obtain the channel's multimedia stream. It is up to the client to contact the peers (these can be either repeaters and other clients) and query them for the actual content. The client may later query the relay server again to obtain an updated list of possible peers.

The *iphion* Peer Assisted Protocol (iPAP) is the UDP-based protocol that is used for all the control and data messages related to the actual content exchange [Min09]. The use of UDP implies that all communications are packet based, rather than stream based like TCP. UDP messages have less overhead than TCP, but UDP does not include automatic message integrity checking, packet retransmits and data ordering. All communications between clients and relay servers and between clients themselves use iPAP messages over UDP. To maintain a functional network the source and the integrity of all these messages should be verifiable. Fake control messages might disrupt the content delivery protocol that is highly dependant on collaboration among peers.

Multimedia content should only be exchanged between clients that are authorised to handle (receive) the multimedia stream of a specific channel. However, in addition to the authentication controls in the iPAP protocol, the content will also be encrypted. This means that attackers intercepting some of the content during transmission will not be able to use it. The content won't even be stored in plain format on the *iphion* player: it will be decrypted using hardware features right before it is decoded and displayed. This makes it hard, even for an *iphion* customer, to obtain the data in a re-usable format. The keys that are required for content decryption will be distributed out-of-band, i.e. not via the collaborative iPAP network.

5.2 Authentication

When a client has selected a specific channel, it may not contact the relevant repeater directly. It first needs to contact a central authentication server that will check whether this client is authorised to join the selected channel. If the server is satisfied with the client's credentials, it will grant the client a temporarily authentication token that it need to present in all its communications with repeaters and other clients.

The authentication server is called the *iphion* Tokens and Key Server (iTKP). Communication with this server is not done via iPAP, but via a direct TLS link between client and server. Apart from the regular TLS

authentication, the extended authentication using the secret STB-embedded key (Section 3.2.2) will always be used in this registration step as well.

The iTKP server will need to perform the same AES secret key operation as the STB in order to verify the challenge-response authentication. The server could use an hardware device that can perform AES operation without disclosing the secret key to the software; just like the clients do. We could even use a dedicated STB platform as a backend for this, or if that is not fast enough, another hardware AES device (e.g. PCI or USB based). In this way, even if the iTKP server gets compromised, an attacker would only have temporary access to AES operations, but wouldn't be able to steal this key and use it to 'clone' an *iphion* player machine in software. The same holds true for the STB clients: As long as an attacker has access to the device (via a backdoor or other attack), he can participate in the protocol by using the STB's AES operations for his own purposes, even without knowing the key.

The iTKP server will use a central database to look up the channel access rights of each client. It will also check if a client has been administratively blocked (temporarily or permanent; for whatever reason). If the iTKP server does not grant access, then it will not be possible for the client to join the requested channel and it will not be able to take part in any content exchange.

If the client is authorised, it will receive a temporarily authorisation token that is signed by the iTKP server and that is valid only for this client and a specific channel group. The token will be accepted by all *iphion* repeaters and other clients who also have content for the listed channel group. Client tokens will not be accepted by the broadcaster though: only *iphion* repeaters may connect directly with a broadcaster. Whenever this token expires or the client wishes to change to a channel that is not included in the selected group, the client needs to reconnect with the iTKP server and go through the authentication procedure once again.

Figure 5.2 shows a graphical overview of the iTKP authentication and token request. First the standard and extended TLS authentication steps are preformed: after each step both client and server validate each others credentials. Then the client sends a request to indicate it wants to join a specific channel. The request includes the client peer identifier $ID(c)$, a client's temporary public key $PK(c)$, and the channel group identifier $chan$. These three items will also be included in the token, together with an expiration time stamp $expiry$. This token is signed by the iTKP server. In the following sections we will extend this exchange further with additional information from the server, such as the temporary content decryption keys (Section 5.4).

The expiry time of a token will be relatively short, initially this will be set at 10 minutes for both players and repeaters. But this value may be changed

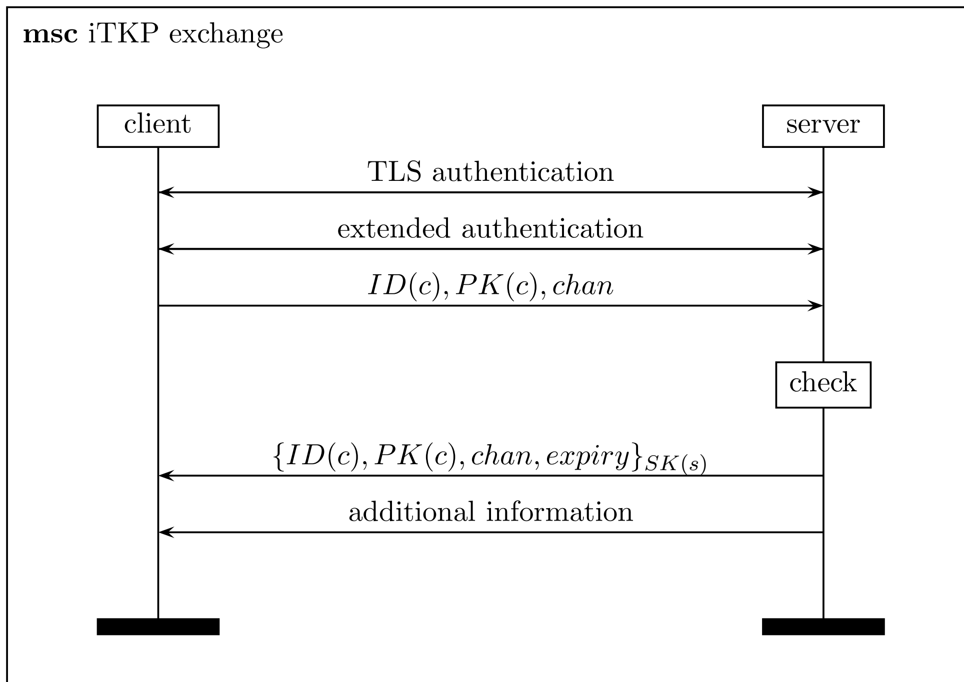


Figure 5.2: iphion Token and Key Protocol

in the future as *iphion* gains experience with this system. The token expiry should not be extended because the iTKP server cannot handle the traffic: the correct solution for that would be to upgrade the hardware or add more servers. Tokens are only used for the authentication of clients and repeaters in the iPAP network.

To make sure that the expiry timestamps work as designed, all peers in the network should have synchronised clocks. All servers and clients will be running the Network Time Protocol (NTP) to keep their system clocks synchronised. *iphion* will be running its own NTP servers, so that synchronisation will not depend on external servers and networks. It is more important that all peers keep the same time, than that this time exactly matches Coordinated Universal Time (UTC).

5.3 Integrity

Validation of message integrity and its original source is important in the content distribution protocol (iPAP). If one can feed clients or repeaters misinformation about what other peers are offering or requesting, then they will start sending the wrong data to the wrong peers. It is important that both repeaters and clients only act upon messages that are genuine and

discard fake or outdated data. Invalid messages should be reported to a central logging server ('eventlogger').

To preserve the integrity of messages between peers we use keyed-Hash Message Authentication Codes (HMAC) [KBC97]. The HMAC of a message is calculated from a cryptographic hash function of the data and a secret key that is only know to the two communication partners. HMACs use symmetric cryptography which is much faster than the asymmetric (public key) cryptography that is generally used for digital signatures. Since the processing power of the set-top boxes is rather limited and there is a lot of data to be exchanged for multimedia streaming, this reduced overhead is very welcome.

This leaves the question of how to securely establish a temporary shared key between two peers that is needed to use HMAC. We accomplish this using the **Diffie-Hellman (DH) key agreement protocol** [Res99]. This enables two communication partners to agree on a fresh secret key, without ever having to send this key over the network.

For speed and simplicity, the public DH parameters will be pre-generated by the iTKP server and are send along with a successful client registration. The DH input values that are determined by the peers with need to be exchanged via an initial message that allows data and source validation, but does not use an HMAC itself. Since this is only one message, we will use a more expensive public key-based digital signature here: the peer signs this initial message with its own private key.

The corresponding public key could be sent along in this message as well. Rather than using the 'hard-coded' X.509 certificates that are used in TLS communications, we will be using distinct key pairs here. This has a couple of advantages: it allows us to use temporary client keys that can be replaced periodically; this lets us use smaller keysizes (for faster operations) and we don't need to send full X.509 certificates. Initially we plan to use 1024-bit RSA keys for this operation. Such keys are large enough to be secure and yet small enough to work well on the set-top boxes. Since these keys are not hard-coded like the box identifiers, but dynamically generated, it will be easy to replace these keys with others (larger key sizes) later.

The client's public key should be signed by a trusted authority. In this case that will be the iTKP server. The client public key will be presented to this server during registration and the full public key will be included in the authentication token that is generated and signed by the iTKP server. When a client presents this token in its signed communication with a new peer, the peer can match the signature of this message with the public key in the token and the signature on the token with the public key of the iTKP server. Every peer will already know the public key of the iTKP server from its earlier contact with this server. This key *will* be a part of our established

X.509 certificate hierarchy. The public certificate of the iTKP server will be signed by *iphion*'s server signing authority.

Note that although all information in the initial crypto set-up message between two peers is public, it cannot be re-used in later replay attacks. The public values for the DH key exchange correspond with private values that are only known by the identified peers. A third party listening in would not be able to calculate the secret key that will be used in HMAC generation [Res99]. Therefore, an attacker cannot use this information to later inject fake messages: Because the HMACs would be incorrect, the injected messages would still be ignored.

Once an HMAC key has been securely established by two peers, all further communications between these peers will use messages that include a keyed hash of this message (the HMAC). The other party will verify this HMAC when it receive a message and must ignore the message if the HMAC does not match the content, or if the used key has expired. The established key expires once the validation of the token that was used to establish it expires. At that point a fresh DH key agreement setup is required.

The HMAC is calculated from both the agreed key and the message digest of the data. For the earlier PC-based prototypes, *iphion* used the SHA-1 message digest algorithm. However this proved to be too expensive for the set-top boxes: they could not process the incoming packets fast enough. Therefore, *iphion* will be using the **MD5** message digest algorithm [Riv92] on the first generation set-top boxes. MD5 is an algorithm that can be executed fast enough by the STB and although it has some known issues with pre-image resilience, there are no known attacks against use in HMAC operations [SLdW08]. TLS uses both these algorithms as well: each TLS message includes either a MD5-HMAC or SHA1-HMAC for message integrity checks [DA99]. If MD5 later proves no longer to be sufficient for this application, then the protocol can be changed via a software update on both clients and servers, replacing MD5-HMAC with something better.

The Diffie-Hellman key exchange between peers and the use of MD5-HMAC for message integrity enables clients to verify that a message has been sent by a specific peer and was delivered intact. It makes data injection and modification from untrusted sources impossible. However, it does not tell us anything about the original source, when a message gets relayed — this source being the broadcaster, in the case of multimedia content data. A compromised client might inject its own data and its peers would not be able to tell. A rogue peer could theoretically even scramble injected data using the active content keys; or descramble the original content and forward it without encryption. Of course, the mismatch with content from other peers will be noticed quickly. And once a rogue peer is detected, it will immediately get banned and won't receive any more communication tokens.

Still, it would be better if a signature from the original source were included in the data, so that *iphion* players could recognize and automatically reject injected multimedia content.

5.4 Confidentiality

The control messages that are exchanged in the iPAP protocol are not considered confidential. However the multimedia content that is distributed in MPEG-TS blocks via iPAP should remain confidential. The multimedia data should only be made available to authorized users. Earlier in Section 5.2 we established how such users may be authenticated.

All nodes in the content delivery network will trust the temporary authorization tokens and exchange data with authenticated peers. However it remains possible that attackers intercept this data (either in-traffic or at a set-top box) and forward it to an unauthorized computer. To make sure that this is not enough, the content itself will be encrypted as well.

In Section 2.3.1.2 we considered alternative implementations for content encryption and later decided to use the Common Scrambling Algorithm (CSA) for data encryption (Section 2.4.1). The chosen *iphion* player devices have hardware support to decrypt CSA scrambling. This means the feature can be implemented with manageable overhead.

The distinction between scrambling and encryption is not very important here and we will use both terms to refer to the same operation. Nowadays, ‘scrambling’ usually refers to operations on analog signals rather than digital data, with digital audio and video thrown in for historical reasons. However some properties of CSA scrambling are relevant: most of the meta data in the MPEG-TS stream is not encrypted (stream id, sequence numbers, flags, etc.); all data blocks will be encrypted and decrypted independent of each other; and the encrypted data has exactly same size as the decrypted version.

One of the arguments against CSA was that it was not believed to be as strong cryptographically as other algorithms, such as AES. If we are going to deploy this, it is important to still rotate the keys frequently. To facilitate key rotation, messages encrypted with CSA include a key indicator bit: This signifies that either the ‘odd’ or the ‘even’ key is used. This flag toggles whenever the key is changed. The *iphion* players should know in advance what the next ‘even’ or ‘odd’ key will be that they should be using.

When DVB-CSA is used for satellite broadcasts, then it is customary to generate the key from two input sources (key parts): one part that is kept for longer periods (say, a month) and distributed out-of-band (this can be on

a physical smartcard sent by the post) and a part that rotates very frequently (for instance every 10 seconds) and is distributed in-band (included in the data stream). This means that if the key and its components are discovered at any point, then an attacker will be able to decipher the content stream for the duration of that month, because all the short-time key updates will be available to him as well. However, because only a small amount of data is encrypted using the same key, it is still hard to launch a successful attack.

This does not apply to our situation: because we are not limited to a single data channel, we can use unrelated key-updates and distribute them out-of-band via secure (online) channels to the player. There is no need to include any keys in the data stream itself. An advantage of this is that it allows us to use longer-lived keys: for example keys expiring in minutes rather than seconds. Although there are currently no known (documented) attacks to break CSA encryption, there is also little public cryptanalysis available on the algorithm. We will be keeping the key-rotation as a precaution.

Keys will be issued to the clients by the *iphion* Token and Key Protocol server (iTKP). Initially key rotation will be done with the same frequency as authentication token expiries (every 10 minutes). Although the protocol does not rely on these two to be related, it will be convenient if clients only need to contact the server once every 10 minutes to renew both token and key. The authentication mechanism used by iTKP for issuing tokens is the same as for issuing keys: TLS authentication combined with a challenge-response check using the unreadable hardware encryption key (Section 3.2.2).

The keys themselves will not be created by the iTKP server, but by a dedicated secure service, the **keymaster**. The keymaster periodically generates new CSA keys for each channel (64-bit random numbers[1]). The keymaster forwards these keys via secure channels to the **scrambler** and the iTKP server. The per-channel scrambler performs the actual DVB-CSA encryption. This is done before data is send to the channel broadcaster and distributed via iPAP. Figure 5.3 illustrates the distribution flow of the generated keys and authentication tokens in the *iphion* network. This simplified overview does not show the token exchange between peers in the iPAP network nor the complex collaborative content stream exchanges.

5.5 Summary

Multimedia content confidentiality is one of the main requirements from the content suppliers who will use the *iphion* network. The content should

[1] Part of the 64-bit CSA key is used as checksum and may be derived from the message: only 48 bits are truly unknown

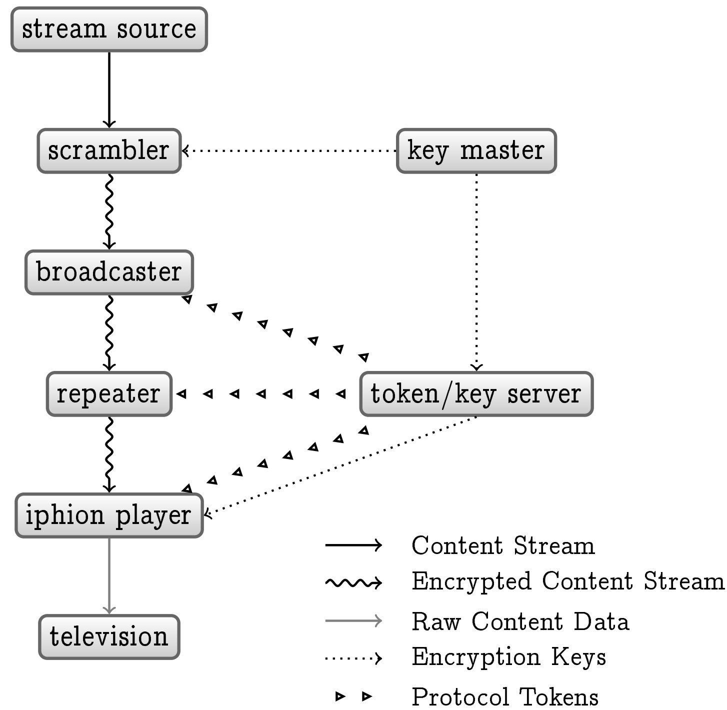


Figure 5.3: Flow of tokens and keys

only be made available to authorized customers and then only for the designated purpose (displaying it on the television screen). The signal that is sent to the television is completely clear, raw data that can be copied, stored or relayed — without restrictions via High-bandwidth Digital Content Protection (HDCP) or similar protocols. However the compressed high quality MPEG-encoded audio and video streams will not be available to the customers: these will be sent encrypted to the *iphion* player devices, where they will be decrypted and decoded in hardware before sending the result to the video output.

Content distribution is handled by the *iphion* Peer Assisted Protocol (iPAP) [Min09]. This protocol has been extended with authentication tokens that are issued by a central authentication server. These temporary tokens are granted only to clients that are allowed to access a specific channel. When establishing an iPAP communication, peers can quickly verify that their communication partners are authorized to handle specific channel data via this token, without the need to consult a central database or server. HMAC codes are used with all iPAP messages to show that the messages indeed originate from authorized peers and they have not been tampered with in-transit.

For the decryption of the multimedia content, *iphion* players also need

to obtain channel-specific temporary cryptographic decryption keys. Unlike the tokens, these keys are never used in the iPAP protocol, but only on each individual client that has to obtain these keys from the central keyserver. Both key and token distribution rely on TLS with the extended device authentication that was described in Section 3.2.2.

6. Conclusions

This thesis started with the analysis of the security requirements for the *iphion* collaborative IPTV network. We have described risks that threaten the network and devised solutions that may mitigate those risks. Where multiple solutions were available, we analysed which alternative would best suit the specific situation.

Several of the selected security solutions were worked out in detail. We analysed how a general method might be applied in the particular environment of the *iphion* system, then designed a concrete path for the implementation of this solution and assisted in the completion of the actual production. The topics that were analysed thoroughly are TLS authentication and communications, set-top box software integrity and the security of streaming multimedia content.

In the next section, we describe the results of the security requirements analysis and the proposed solutions. Following that, we give concrete recommendations to mitigate some security risks there were disclosed in our analysis. And finally, we highlight other parts of the system that were not yet addressed in detail by our analysis. Work on these aspects should continue in order to identify risks and establish proper mitigation measures.

6.1 Summary

iphion realizes that security is an essential requirement for their IPTV system to enter in the business. Our analysis addresses the security aspects of the system. We started with eliciting the security requirements. Initially, we look at what the system should do (functional requirements). Each of these actions may be attacked by 'crooks' who seek to disrupt the system or obtain privileged data. This leads to additional requirements, needed to mitigate those attacks. These mitigation cases should be regarded in their proper context, not just from a security viewpoint, but also considering other aspects of the system, such as usability and dependability and of course financial aspects. Fulfilling the requirements of each of these goals

requires a compromise in the implementation choices. Through qualitative analysis we tried to find the solutions that best satisfy all goals (Chapter 2).

The solutions that we came up with after the goal analysis are still very general and leave much room for choices in the actual implementation. We did a more detailed analysis of several proposed solutions. In the following three chapters we describe concrete designs for the security of several components in the *iphion* system.

To preserve the integrity and confidentiality of meta data communications, Transport Layer Security (TLS) will be used. Meta data is the term used for all information that is not directly related to the exchange of multimedia content, such as profile data, electronic program guides, software updates and status reports. TLS will also be used to authenticate the clients and servers in the system, with use of a public key infrastructure (PKI) and personal keys for each of the actors. Extended authentication includes an additional step on top of the regular TLS authentication that requires cryptographic hardware support of the *iphion* player device. This makes it much harder to fake an identity. Not all servers will authenticate themselves using this extra step, nor will they be able to perform this check for clients (Chapter 3).

Software integrity is difficult to guarantee in a situation where a potential attacker has full control over a system environment (the *iphion* player) and its network. An attacker might disassemble the hardware and read or write data to the flash chip directly, thus bypassing all checks of running software. However, we can check the integrity of this data storage at start-up and can make sure that the device only boots with an *iphion*-approved (signed) filesystem. All software updates (both incremental and in rescue mode) should be digitally signed as well. Attackers may still be able to replace the client software, but only with *iphion*-issued system versions. This does not prevent an attacker from exploiting vulnerabilities in the *iphion* software itself: the system should be designed to prevent and detect possible attacks (Chapter 4).

Maintaining multimedia content confidentiality and integrity in the *iphion* system poses some interesting challenges. One problem is the unorthodox way the content data is delivered: via a distributed collaborative network rather than a direct TCP stream from a server. Another issue is the limited processing power of the *iphion* player platform that makes it practically impossible to use public key cryptography for encryption or digital signatures for the content stream. We achieve our goals by using a central authentication server and temporary keys, both for peer-to-peer communication integrity and for content data encryption (Chapter 5).

Figure 6.1 shows an overview of the communication streams in the *iphion* network. It illustrates the differences between the data that is exchanged

via secure channels (i.e. TLS) and the communications that are secured by other means: the multimedia content streams and the client rescue images.

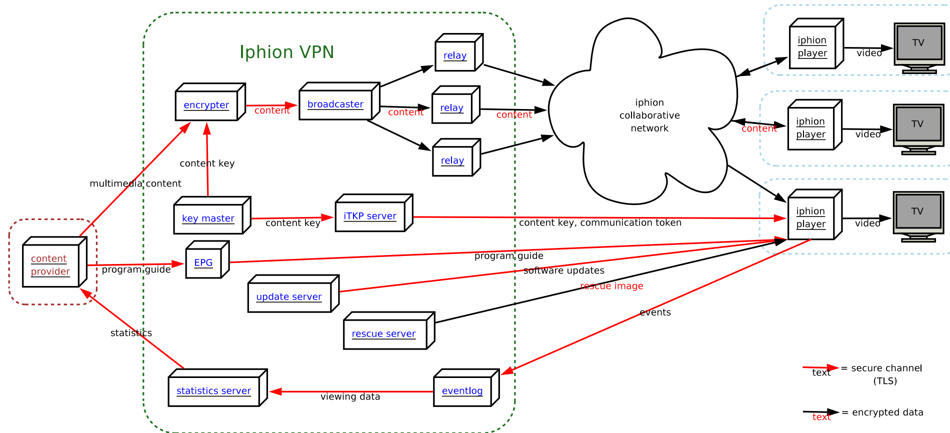


Figure 6.1: iphion communications security

During the work on this thesis, two presentations have been given for the company in which intermediate results were presented, along with several of the recommendations listed in Section 6.2. One presentation focused on the application of TLS and the implementation of a Public Key Infrastructure for the *iphion* system. The other presentation analysed the secure boot features of the NXP set-top box and presented recommendations about the use of these features, including signed file system images and the outlined rescue boot procedure.

Throughout this project there has been a good dialogue and collaboration with the developers at *iphion* — and in some cases with developers at Prodrive as well (the set-top box hardware supplier). Although this project was more about analysis and design rather than implementation, some code has been produced as a by-product and was committed directly into the company's code repository. Bugs reported regarding the cryptographic code of the NXP hardware drivers and the curl https library, lead to fixes in those projects. Other work that did not make it into this thesis, including practicality tests, performance measurements and literature citations, has been published on the company's intranet site.

The result of the good collaboration with *iphion* developers is that many of the recommendations listed in the next section have already been implemented by the company. As a consequence the first batch of *iphion* player devices was recently produced and delivered. These include the required hardware support for the secure boot features described in Section 4.4.1 — and a client key pair in NOR flash as well as the hardware AES key, both used for extended authentication outlined in Section 3.2.2.

Still under development is, for example, the iTKP protocol for client authentication and the secure distribution of tokens and keys used to distribute and decrypt multimedia content data.

6.2 Recommendations

The detailed analysis of three aspects of the system security, leads to several concrete recommendations. The other aspects that were not covered in this report should be investigated further. Specific suggestions for further analysis will be listed in the next section.

- Set up a public key infrastructure and issue certificates for all nodes in the network to be used for authentication via TLS; this includes a separate certificate hierarchy for the development environment and the production environment. Use a separate signing authority for client and server certificates.
- Honour the policy document that describes the secure use of these keys and the procedures for creating, signing and storing certificates and private keys (Appendix C).
- Use TLS authentication, integrity checking and encryption for all client-server communications that use TCP.
- Use the on-chip crypto features of the set-top box for extended authentication of the iphion players and the iphion iTKP server.
- Consider the use of this feature for the authentication of other client-server communications as well.

- Use all of the secure boot features of the set-top box, so that every boot step validates integrity of the next step before executing it.
- Specifically, use digital signatures for the bootloader software, bootloader environment, operating system kernel and the filesystem containing software and configuration data. Generate and store the required keys for these signatures responsibly.
- Use a secure software update mechanism that can preserve the digital signatures: with each update the file system signatures must be updated as well.
- Use a software rescue procedure (where the entire system is reinstalled) that also checks digital signatures of the rescue image and checks and preserves digital filesystem signatures.

- Use different signing keys when deploying new hardware revisions in the future.

- Encrypt all streaming multimedia content data via DVB-CSA. Consider switching to DVB-AES later when it is more widely used and supported.
- Use temporary, short-lived content encryption keys, handed out by a central client authentication server.
- Use temporary authentication tokens from a central server for peer authentication in the iPAP protocol, because the set-top boxes cannot handle certificate-based authentication fast enough.
- Use MD5-HMAC integrity codes on all iPAP messages and use Diffie-Hellman key agreement to establish a key for MD5-HMAC, because the set-top boxes cannot handle digital signatures based on public key cryptography fast enough. The set-top boxes cannot handle SHA1-HMAC either.

6.3 Further work

This thesis has mostly focused on the security of the *iphion* player. However, security of the servers is at least as important. In some ways it may be considered even more important, because a single compromised server might affect a large part of the clients. There were several reasons to focus on the clients first though. A lot of the player security depends on features that need to be available in hardware (private keys, bootloader features), which means this needs to be planned before the players can go into production. While on the servers, most of the security features will be implemented completely in software (or generic add-on hardware), which means it is easier to improve and update later on. Another reason is that general internet server security — firewalls, secure communications, software updates, package management, etc. — is a more common and well understood problem. Even though the current *iphion* server security may not have been analysed in-depth, people are satisfied that at least some sensible basic protection is in place.

Still, there is more work to be done on server security. In the future *iphion* servers will be placed at several remote locations in varying environments and they will need to communicate securely with each other — and be manageable from *iphion* headquarters. For software updates, the servers will use application updates via a Debian-like package management system. These updates should be protected with digital signatures from *iphion* as well.

New server machine installations may not become a daily event, but the addition of new services might. It may become impractical to follow the current secure certificate signing policy for each server certificate. Therefore we have been considering pre-generated server certificates, specifically for broadcasters and repeaters, that can be stored encrypted in a temporary storage until they are needed. To keep a clear policy distinction, these certificates may be issued by a new intermediate certificate authority.

Another aspect of the *iphion* certificate hierarchy that needs further work is the certificate revocation procedure. At the moment certificate revocation lists (CRLs) can only be distributed via software updates. This is not a very flexible method and might not suffice if revocations are needed frequently – although it is not expected that they will be. In fact, for remotely taking an abusive or malfunctioning client out of the network, it is enough to ban the client in the iTKP server: without tokens or decryption keys the player can no longer participate in the content exchange network, nor can it play (decrypt) any multimedia content.

This thesis did not address the handling of customer data. The company may collect a lot of data about customer behaviour: at what times do people watch television, what programs do they watch, when exactly do they switch channels etc. *iphion* plans use this data to provide personalised customer suggestions in the future (e.g. about upcoming television shows). Customers will consider this information personal data and expect *iphion* to handle it responsibly (i.e. not use it for anything other than what is required for the technical functionality of the network).

However this data is also of interest to *iphion*'s partners: broadcasting companies would like to see aggregated viewing statistics and advertisers are very interested in when exactly people do or do not 'zap' away to other channels during commercial breaks. The company will need to decide what data to collect exactly, what to present to third parties and how to guarantee that specific data cannot be traced back to individual users. This may become harder if *iphion* decides to offer pay-per-view features in the future as well. A clear privacy policy should be drafted and presented to the customers.

Besides intentional use, there may be unintended use of the data as well. Customers will know with whom (which IP addresses) they are exchanging data: this means that these people are watching the same channel at the same time. These IP addresses might also be traced back to individuals (putting an address into a search engine is a good start). Although peering in the *iphion* network is based on network locality rather than geographic locality, these two frequently coincide. So one might learn what the neighbours are watching. Another problem is the internal database with viewing information: even if *iphion* does use it responsibly, a hacker

who might obtain access could still abuse the information. For example for burglars it might be interesting to know when people turn on the television (when they get home) and turn it off again (when they go to sleep). This kind of behaviour might also be of interest to law enforcement agencies, who could confiscate a copy of the collected data.

A. Glossary

AES Advanced Encryption Standard, an U.S. standard (FIPS 197) symmetric encryption algorithm for data encryption. It's considered fast and secure by modern standards; suitable for large sets of data. Also known as **Rijndael**.

broadcaster See **streaming server**.

CSA Common Scrambling Algorithm, an European standard encryption algorithm (ETSI 289) used in DVB transmissions for the encryption of multimedia streams. CSA was specifically designed for fast DVB encryption and decryption and it is the most commonly used algorithm in this context.

decoder The iphion service running on an iphion player that decodes a multimedia stream as received from the iphion network to a format that can be displayed on a regular television. This service will be implemented by dedicated STB hardware.

decrypter The iphion service running on an iphion player that decrypts a multimedia stream received from the iphion network. This will be implemented by dedicated STB hardware.

DVB Digital Video Broadcasting, a set of open standards for the distribution of digital television. DVB standards are issued by ETSI and the DVB protocols are mostly used in Europe. Its North-American counterpart is ATSC.

encoder The iphion service that transcodes a multimedia stream as provided by a content supplier into a multimedia stream fit for distribution over the iphion network: an MPEG-TS formatted stream. There will likely be a dedicated encoder for every TV channel. If the content provider sends us data in the correct format for distribution over

the iphion network (hopefully most will), then no encoder will be needed for this channel.

encrypter The iphion service that encrypts a multimedia stream (MPEG-TS) using the chosen encryption format (CSA) and private keys that are generated by the key master. All channels will likely have their own encrypter.

eventlog The iphion service that collects activitiy data from the iphion players. This includes both regular operational information (such as channel change requests) for statistical analysis as exception reports that may indicate a problem with the device or the network.

EPG Electronic Program Guide, information about all available TV program information. This is collected by the EPG service from information made available by the broadcasters and presented in a GUI by the iphion players.

ETSI European Telecommunications Standards Institute, a standardisation organisation responsible for the DVB digital television standard - and many other standards such as GSM (Global System for Mobile communications), DECT (Digital Enhanced Cordless Telecommunications) and TETRA (Terrestrial Trunked Radio).

IETF Internet Engineering Task Force, an independant open standards organisation, with no formal membership requirements. IETF working groups have published many standards regarding the **Internet Protocol**. IETF standards are known as RFCs (Request For Comments), although not every document publish as an RFC describes a standard.

IP Internet Protocol. An open standard (RFC 791) for communicating data in a packet switched network. This is the basic network for distributing data over the internet. The upper layer protocols TCP and UDP are built on top of IP.

iPAP iphion Peer Assisted Protocol, the protocol that handles the collaborative distribution of multimedia content over the network. This protocol will be implemented both on the iphion players and the broadcaster and relay services.

iphion player The set-top box that customers will buy and hook up to their internet connection and television, so that they can receive multimedia content via iphion's collaborative

network. At the moment, each player can only be tuned in to one channel at once, but later versions may include PVR (Personal Video Recorder) support.

iphiserv Old name for the token server.

IPTV Internet Protocol Television, any system where a digital television service is delivered using an internet (IP) infrastructure.

iTKP iphion Token and Key Protocol, the protocol to issue tokens and keys to authenticated clients. Tokens are needed to participate in the iPAP data exchange, keys to decrypt the obtained multimedia content.

key master or key generator, the iphion service that continuously generates keys for the encryption of multimedia content.

MPEG Moving Picture Experts Group, an international standardisation organisation that deals with the compression and transmission of audio and video data. MPEG also refers to the open standards designed by this group. MPEG standards include MP3 (MPEG-1, audio layer 3 compression format) and MPEG-4 AVC (or H.264; for video compression).

MPEG-2 An open standard (ISO/IEC 13818) for the compression and transport of broadcast-quality television. This is the chosen standard for **DVB**, but also for other systems such as ATSC, SVCD and DVD.

MPEG-TS MPEG Transport Stream, an open standard (ISO/IEC 13818-1 or H.222.0) describing a communications protocol for the distribution of audio, video and data. This is part of the MPEG-2 standards set. In MPEG-TS, the MPEG-encoded multimedia data is split into small packets (of 188 bytes) for transport over an unreliable network. In the iphion collaborative network MPEG-TS chunks are used in groups of 7 packets (1316 bytes), that fit in a single UDP packet.

NAT Network Address Translation, also called masquerading: a common technique used to connect multiple devices to the internet using a single IP address. A NAT router (e.g. ADSL or cable modem) will monitor outgoing traffic from the private network to the internet and make sure that response traffic from the internet is directed to the

correct local server. When NAT is used, unsollicited packets from the internet directly to a local server behind the NAT router is not possible. That is: any packet that isn't send in response to a previous query won't reach its destination. When two peers, both behind a NAT router, want to communicate with each other, they can use an iphion supernode to relay their communication.

NIST National Institute of Standards and Technology, the measurement standards laboratory of the U.S. Department of Commerce. NIST CSD is the Computer Security Division, which has publish several Federal Information Processing Standards (FIPS) about the use of cryptography. Although the standards are only binding for U.S. government agencies and government contractors, they are generally accepted by a wider audience.

ORC An implementation of the **iPAP** service for both servers (broadcaster, repeater) and clients (*iphion* player).

player See **iphion player**.

PVR Personal Video Recorder, a device that records video in digital format for later playback. PVR functionality is not present in the current generation *iphion* player, but it is planned for later revisions.

relay server or repeater, an iphion service that relays (encrypted) multimedia data from the broadcaster to the iphion players. Relay servers generally serve a specific part of the network and are strategically placed in the network. A relay server could be placed on a local internet exchange or in the serverpark of an ISP to serve the customers of that ISP. Each relay service is expected to handle multiple channels.

repeater See **relay server**.

RSA An algorithm for public-key cryptography, named after its inventors, Rivest, Shamir and Adleman. First published in 1978, it is still one of the most popular algorithms for digital signatures and public-key encryption [AMV96].

SSL Secure Sockets Layer, the predecessor of **TLS**.

STB Set-Top Box, also known as the **iphion player**: the box that customers will buy and hook up to their internet connection and television, so that they can receive TV content.

streaming server The iphion service that sends out the (encrypted) multimedia content to the relay services. There may be multiple streaming servers for different channels.

supernode An iphion relay service or normal iphion player node that communicates with an other peer on behalf of a requesting peer. It solves the problem when two players, both behind a NAT router, cannot communicate directly with each other.

TCP Transmission Control Protocol, an open standard (RFC 793) for transmission of two-way communication streams over the internet. TCP guarantees reliable, error-free, ordered delivery of bytes between two nodes. TCP is used in the iphion network for all communication, except the exchange of multimedia content in the collaborative network. For all secure transmissions, TCP is combined with SSL.

TLS Transport Layer Security, an open standard (RFC 5246) for establishing secure (encrypted and authenticated) point-to-point connections over a public internet network. This relies on a public/private key infrastructure where public keys (TLS certificates) are signed by a central authority (Certificate Authority). All data sent over an TLS connection will be encrypted with a temporary (symmetric) session key.

token A digital passkey that identifies an iphion player and indicates permission to exchange multimedia content for a specific channel. Tokens are issued and digitally signed by the **token server** and are only valid for a short period ($\pm$15 minutes). After this period the player will need to authenticate itself again with token server.

token server or key server, the iphion service that authenticates iphion players, determines which content can be made available to each client and distributes tokens for peer authentication in the iphion collaborative network and crypto keys for decryption of multimedia content (DVB-CSA).
The token server will also tell iphion players which peers they can contact to obtain data for the television channel they are tuned in to. The protocol to communicate with the token server is called **iTKP**.

UDP User Datagram Protocol, an open standard (RFC 768) for transmission of data in packets (datagrams) over the internet. UDP uses a simple transmission model without

guaranteeing reliability or ordering. UDP can be used both for one-to-one (unicast) communication and one-to-many transmission (multicast, broadcast). UDP unicast is used in the iphion collaborative network for the distribution of multimedia data. This allows the iphion software to use optimised algorithms for re-sending of missing or corrupted packets (or ignoring them if time is running short).

VoD Video on Demand, a system where the customers can select a movie or television program whenever they wants to view it – in contrast to a regular television broadcast where the programming is pre-determined by the broadcaster and everybody who is tuned in to a channel watches the same content simultaneously.

X.509 An ITU standard for the set up of a public key infrastructure. In particular it defines standard formats for public key certificates (X.509 certificates). A certificate contains not just a public key, but also the name/identifier of the subject and of the issuer, the validity period and other information. This data is digitally signed with the private key of the issuer.

B. Use and misuse cases

B.1 iphion player

B.1.1 Use cases

B.1.1.1 Authenticate

Name:	Authenticate
Summary:	The iphion player (client) authenticates itself to the server and receives a secure access token for further communication.
Basic path:	1. The client sets up a secure connection to the server. 2. Client and server exchange credentials (securely) and verify each other's identity. 3. Server sends a signed session token to the client that it may use to identify itself to other actors in the system.
Alternative paths:	
Exception path:	If either client or server supplies invalid credentials, the authentication is aborted and no token is given.
Trigger:	Whenever the client sends an authentication request. It will do this when coming online or switching to another channel.
Assumptions:	Authentication mechanism should protect against replay and man-in-the-middle attacks.
Precondition:	Clients and servers can check credentials of other actors.
Postcondition:	The client is now registered and is (the only one) in possession of a token that grants this specific client access to exchange information with others for a limited period.

Threats:	1. An attacker can try to steal credentials in order to obtain access for himself (B.1.2.3). Note that obtaining a token isn't enough, because all requests must also be signed with the client's private key. 2. An attacker can try to flood the system so that it cannot obtain access (B.1.2.4).

B.1.1.2 Update set-top box

Name:	Update stb
Summary:	When new features or bug fixes are available for the iphion player, the software on the stb should be updated to the newer version.
Basic path:	1. The player (client) asks the update server if there are updates available. 2. The server validates the request and determines which updates are required. 3. The server send a batch of updates to the client. 4. The client installs all patches and reboots to use the new software.
Alternative paths:	
Exception path:	1. If no updates are available, the other steps are ignored and the client will proceed with its normal service. 2. If the update process is aborted while writing the data, the client will request a full update (reinstall) next.
Trigger:	Whenever the client requests a software update. Each client must do this periodically.
Assumptions:	-
Precondition:	
Postcondition:	If completed successfully, the client is running the latest software version.
Threats:	An attacker may attempt to install unauthorised code instead of the expected updates (B.1.2.1).

B.1.1.3 Fetch content

Name:	Fetch content

Summary: Video data is obtained via a collaborative network from multiple sources: iphion relay servers and other (end-point) peers in the network.

Basic path:

1. The customer selects a TV channel on the iphion player.
2. The player asks the iphion server where to find peers who offer this data.
3. The player securely exchanges credentials with the selected peer(s).
4. The player asks these peers for the video data.
5. Content is delivered to the player.

Exception path:

Trigger: Whenever a client fetches content. Often clients will be fetching data continuously while there are turned on.

Assumptions: -

Precondition:

Postcondition: Iphion video data is sent to the player.

Threats:

1. Attackers may try to make it impossible to fetch any data (flood system, B.1.2.4),
2. send bad content instead of the expected video data (B.1.2.6), or
3. liberate a copy of the data (B.1.2.5).

B.1.1.4 Display content

Name: Display content

Summary: The main objective of the iphion player is to display live video of the TV channel that the user has selected.

Basic path:

1. Decrypt the video/audio stream.
2. Decode the video/audio stream to TV format.
3. Output data to the TV.

Exception path: When data is not available in a timely fashion, an error message with some suggestions may be shown (e.g. "plug in network cable").

Trigger: Whenever the player has obtained video content. Often clients will be displaying content continuously while there are turned on.

Assumptions: -

Precondition: The content and the required decryption keys are available to the player.

Postcondition:	Video output will continue to be displayed (and optionally redistributed to others) as long as content data and keys are available.
Threats:	1. Malicious hackers may try to hack into the box and break functionality, obtain keys or decrypted stream (B.1.2.2). 2. The TV data is send out in the clear, but recoding is not trivial and this isn't a threat that iphion cares about.

B.1.2 Misuse cases

B.1.2.1 Spread malicious code

Name:	Spread malicious code
Summary:	An attacker may send malicious code to a player masquerading as a regular software update.
Basic path:	1. A crook intercepts an update request from a client (e.g. by DNS poisoning or network sniffing). 2. The crook sends a software update in the expected format. 3. The client installs this software and reboots. 4. The crook now has full control over the player and its private data (crypto keys).
Alternative paths:	
Exception path:	
Trigger:	Whenever the client requests software updates (B.1.1.2).
Assumptions:	The attacker can intercept or relay queries to its own system. This is trivial when attacker and player-owner collaborate.
Precondition:	-
Postcondition:	The attacker obtains full control over a player.
Mitigation points:	1. When fetching updates (B.1.1.2), the client should check if it has a connection with a trusted server (SSL certificiate check). 2. Before installing updates the client must validate that the software originates from iphion (by checking a digital signature; ??).
Mitigation guarantee:	A player won't install untrusted software.
Related business rule:	The players will operate as specified.

Potential misuser profile:	Skilled: The attacker must know the protocol and format of software updates and must be able to intercept and/or reroute connections.
Stakeholders and risks:	1. Customers: Malicious code may disrupt operation of the protocol and cause problems for other players. 2. Iphion: Malicious code may be used to facilitate other types of misuse such as hacking into the player.

B.1.2.2 Hack set-top box

Name:	Hack stb
Summary:	Malicious hackers may try to hack into a player system and disrupt normal operation, such as content displaying.
Basic path:	1. A crook sends malicious data to a player to try and trick the player into doing what he wants (e.g. exploiting buffer overflow). 2. If successful the attacker may be able to execute his own code and/or disrupt normal operation of the player.
Alternative paths:	
Trigger:	Whenever the client is switched on and connected to the network.
Assumptions:	The attacker can connect to the iphion player. This may be tricky when it is behind a masquerading firewall, but trivial when attacker and owner collaborate.
Precondition:	-
Postcondition:	The attacker may obtain full control over a player.
Mitigation points:	1. The player should run up-to-date code with the latest bugfixes (B.1.1.2). This doesn't fully prevent attacks, but makes them a lot harder. 2. The player should discard data packets that aren't relevant to its operation with as little fuss as possible.
Related business rule:	The players will operate as specified.
Potential misuser profile:	Varies. Script-kiddies may try to use standard hacking tools against the player. Tracking regular security updates should mitigate this. Advanced hackers may try to exploit bugs in iphion's own software.

Stakeholders and risks:	• Customers: A hacked box may cease to function completely. • Iphion: A larger scale hack may bring down the complete network.

B.1.2.3 Get privileges

Name:	Get privileges
Summary:	Access should only be granted to trusted clients.
Basic path:	1. A normal authentication setup is observed by an attacker. 2. The keys exchanged give the attacker full access to the protocol data and to the (protected) content.
Trigger:	Whenever the client authenticate itself to the servers (B.1.1.1). This happens at regular intervals and whenever it switches channels.
Assumptions:	The attacker can intercept communication between a player and the iphion server. This is trivial when attacker and player-owner collaborate.
Precondition:	-
Postcondition:	An attacker has complete access to participate in the protocol and can decrypt all data that is exchanged. The attacker gets full access even though he never paid for it.
Mitigation points:	Authentication procedure should be set up so that even with full access to the data exchange, no information is leaked that others can use to obtain access to the network or the data (B.1.1.1).
Related business rule:	The network and content are accessible to customers only.
Potential misuser profile:	
Stakeholders and risks:	Iphion: Full access may be granted to people who never paid for it.

B.1.2.4 Flood system

Name:	Flood system
Summary:	Flooding a player with lots of data packets causes a denial-of-service attack.

Basic path:	1. The client requests meta data or multimedia content. 2. Attackers send lots of irrelevant data to the client. 3. Network download gets congested and the client is unable to receive the requested data (in a timely fashion).
Alternative path:	By sending specially crafted packets (e.g. crypto setup) the player will perform expensive calculations for each packet before it can decide to discard it as invalid.
Trigger:	Whenever a client is turned on and requests data from the network.
Assumptions:	-
Precondition:	-
Postcondition:	The player will be so busy handling invalid packets that normal operation comes to a halt.
Mitigation points:	When receiving packets it should be simple and efficient to decide which packets can be quickly discarded (B.1.1.3). This helps, but may still not be enough to counter a flood attacked.
Related business rule:	The players will operate as specified - even under extraordinary circumstances.
Potential misuser profile:	Unskilled: Bandwidth flooding doesn't require technical know-how and cannot be prevented by the system. More sophisticated attacks do involve knowledge of the protocol and system.
Stakeholders and risks:	• Customer: The player may stop working without a clear indication of the cause. • Iphion: Servers may also be flooded, causing an outage for many players at once.

B.1.2.5 Obtain content

Name:	Obtain content
Summary:	Content is distributed over the network. An attacker may intercept this and decode it.
Basic path:	1. The attacker participates in the data exchange and receives the content directly from peers. 2. The attacker intercepts regular data exchanges and obtains the content this way.
Trigger:	Whenever a client is fetching content.
Assumptions:	-

Precondition:	-
Postcondition:	Content is available to non-customers and may be distributed to more people.
Mitigation points:	1. Non-customers shouldn't be able to participate in the content distribution network (B.1.1.1). 2. Content data should be encrypted so that even if it is intercepted, it cannot be decoded by an attacker (B.1.1.4). 3. Content decryption keys shouldn't be available directly to customers, so that they cannot share these keys with others.
Related business rule:	The network and content are accessible to customers only.
Potential misuser profile:	Skilled, providing content is properly protected.
Stakeholders and risks:	Iphion: Full access may be granted to people who never paid for it.

B.1.2.6 Insert bad content

Name:	Insert bad content
Summary:	People may distribute their own data using the iphion distribution network. This isn't limited to video content.
Basic path:	1. The attacker participates in the data exchange and submits alternative content rather than the content from the iphion servers. 2. This content may be distributed further by the peers receiving it.
Trigger:	Whenever a client is fetching content.
Assumptions:	The attacker can generate content in the appropriate format.
Precondition:	-
Postcondition:	1. Iphion players and network are used for the distribution of unauthorised data. 2. Iphion players may display inappropriate video streams.
Mitigation points:	All content data received should be validated -to make sure it originates from the iphion servers- before it is relayed or displayed by the player (B.1.2.5, B.1.1.3).
Related business rule:	The iphion network should deliver the content from the providers to the customers.

Potential misuser profile:	Skilled: content would have to be generated in exactly the right format for distribution.
Stakeholders and risks:	• Customers: May receive unwanted data - or may not be able to receive the data they want. • Iphion: The network won't deliver the data to the customers.

B.1.3 Mitigation cases

B.1.3.1 Validate software

Name:	Validate software
Summary:	Before running software on the iphion client, the client should verify that it is indeed software created by iphion for this client.
Basic path:	1. Software is read from disk or via the network. 2. The software is validated by the iphion player (e.g. via digital signatures). 3. Iphion player runs the validated software.
Exception path:	If the software cannot be validated, or doesn't validate correctly, then it must not be used. The iphion player should then try to obtain other software via an update (B.1.1.2).
Trigger:	Whenever software is loaded from a local or remote source.
Mitigation:	Prevents crooks from effectively spreading malicious code (B.1.2.1).

B.1.3.2 Validate content

Name:	Validate content
Summary:	Before distributing or displaying multimedia content via the iphion client, the client should verify that it is indeed content distributed by iphion for this client.
Basic path:	1. Receive multimedia data from peers via the iphion network. 2. Validate integrity and origin of the content (e.g. via digital signatures). 3. Display and optionally relay the content data.

Exception path:	If the multimedia content cannot be validated, or doesn't validate correctly, then it must not be used. The iphion player should then re-try to obtain content (B.1.1.3), possibly from other sources.
Trigger:	Whenever multimedia data is obtained.
Mitigation:	Prevents crooks from effectively spreading bad content (B.1.2.6).

B.1.3.3 Encrypt content

Name:	Encrypt content
Summary:	The multimedia content distributed by iphion will be encrypted to prevent unauthorised parties from obtaining and using it.
Basic path:	1. iphion server encrypts the multimedia content. 2. iphion servers distribute the content via the iphion collaborative network. 3. iphion clients decrypt the content before displaying it.
Exception path:	If an iphion client receives content that it cannot decrypt, then it should discard this data and re-try to obtain the content (B.1.1.3).
Trigger:	Whenever multimedia data is distributed.
Mitigation:	Prevents crooks from obtaining content (B.1.2.5).

B.2 iphion servers

B.2.1 Use cases

B.2.1.1 Obtain content

Name:	Obtain content
Summary:	Multimedia data is obtained from content providers. For each channel a continuous stream will be provided. The method of data provision may differ per provider/channel.
Basic path:	1. Data is sent from the content provider to the iphion network. 2. All streams are processed individually by iphion servers. 3. Encoded data is sent out into the collaborative iphion network by the broadcaster(s).

Exception path:	
Trigger:	Whenever a client is fetching content.
Assumptions:	A reliable (internet) connection between the content providers and iphion is available.
Precondition:	
Postcondition:	Multimedia streams are received and processed.
Threats:	1. Attackers may try to make it interrupt the reception (e.g. flood the system, ??), 2. send bad content instead of the expected video data (B.1.2.6), or 3. liberate a copy of the multimedia data (B.1.2.5).

B.2.1.2 Distribute content

Name:	Distribute content
Summary:	Encrypted multimedia data is distributed by the broadcaster(s) and relay servers to the iphion players.
Basic path:	1.
Exception path:	
Trigger:	Whenever a client is fetching content.
Assumptions:	A reliable (internet) connection between the content providers and iphion is available.
Precondition:	
Postcondition:	Multimedia streams are received and processed.
Threats:	1. Attackers may try to make it interrupt the reception (e.g. flood the system, B.1.2.4), 2. send bad content instead of the expected video data (B.1.2.6), or 3. liberate a copy of the multimedia data (B.1.2.5).

B.2.1.3 Authenticate clients

Name:	Authenticate clients
Summary:	All iphion servers should verify client credentials before granting access to data or resources.

Basic path:	1. An iphion player connects to one of the iphion servers. 2. Both player and iphion server exchange credentials (in a secure way). 3. The client requests specific data. 4. The server only returns the requested data if the authenticated client is authorised to access this information.
Exception path:	• If the client credentials cannot be verified, the connection should be aborted by the server. • If the server credentials cannot be verified, the connection should be aborted by the client, even if there is no alternative to obtain this data (retry later). • If the client is authenticated, but not authorised to have the requested data, the server should refuse access.
Trigger:	Whenever a client requests data from one of the iphion servers.
Assumptions:	A secure mechanism must be in place for clients and servers to authenticate each other.
Precondition:	
Postcondition:	
Threats:	Attackers may try to obtain authorization by supplying false credentials. Proper checks will prevent this.

B.2.1.4 Provide meta data

Name:	Provide meta data
Summary:	The iphion players need more than just content data: electronic program guide, software application updates, account information, etc. Furthermore other system data needs to be made available to other parties, such as statistics and performance reports.
Basic path:	1. An iphion player requests a certain type of meta data, directly from a specific server. 2. The server checks if the client may access this data and responds with the requested information.

Exception path:	1. An iphion employee or (content) partner requests a data directly from a specific server. 2. The server checks if the client may access this data and responds with the requested information.
Trigger:	Meta data will be fetched periodically and whenever a user switches channels.
Assumptions:	Clients are properly connected to the network and servers offer the relevant services.
Precondition:	
Postcondition:	Data is only made available to those who are authorised to access it.
Threats:	1. The main threat is crooks trying to obtain data that they shouldn't have access to (B.1.2.5). 2. The other threats that apply to public data services (B.2.1.3) apply here as well.

B.2.1.5 Provide public data

Name:	Provide public data
Summary:	Iphion offers services to third-parties, such as a general website for potential customers and content partners.
Basic path:	1. A visitor connects to a public iphion service. 2. The visitor checks the authentication offered by the iphion service with a trusted third party. 3. The visitor requests data. 4. The iphion service returns the requested public data.
Alternate path:	The authentication step may be omitted (e.g. by using HTTP rather than HTTPS).
Exception path:	1. If authentication, the visitor may abort the connection. 2. If the requested data isn't public, the iphion server must refuse access.
Trigger:	Whenever a request is sent to a public iphion service.
Assumptions:	
Precondition:	
Postcondition:	
Threats:	If public and restricted data are distributed using the same service, caution should be taken to prevent leaking restricted data to unauthorised visitors.

B.2.2 Misuse cases

B.2.2.1 Obtain content

Name:	Obtain content
Summary:	Multimedia content is send from content providers to the iphion servers and then further distributed via iphion's collaborative network. At either stage an attacker may try to obtain this content.
Basic path:	1. A content provider sends data to the iphion server. 2. An attacker manages to intercept this information.
Alternative paths:	1. An iphion server sends out multimedia data to the iphion network. 2. An attacker manages to intercept this information.
Exception path:	
Trigger:	Whenever multimedia data is send by the content providers (B.2.1.1) or iphion servers (B.2.1.2). This happens continuously.
Assumptions:	The attacker can intercept or relay data streams on its own system.
Precondition:	-
Postcondition:	Restricted multimedia content data ends up with an unauthorised party.
Mitigation points:	1. When obtaining media from content providers (B.2.1.1), a secure channel should be used for the media transport (B.2.3.1). 2. When distributing media from the iphion servers (B.2.1.2), all the media should be encrypted (B.2.3.4).
Mitigation guarantee:	Multimedia content is only available to authorised parties.
Related business rule:	Content is only accessible for iphion customers.
Potential misuser profile:	Skilled: The attacker must be able to intercept and/or reroute internet data steams.
Stakeholders and risks:	1. Iphion: Full content access might be obtained by people who never paid for it.

B.2.2.2 Get privileges

Name:	Get privileges

Summary:	Access privileges should only be granted to trusted clients.
Basic path:	1. An attacker connects to the authentication server, pretending to be an iphion client. 2. The server grants the attacker access (keys) to the multimedia content and other restricted data.
Trigger:	Whenever an attacker connects to the authentication service.
Assumptions:	
Precondition:	-
Postcondition:	The attacker gets full access to iphion data even though he never paid for it.
Mitigation points:	Authentication procedure should be secured so that only real iphion customers can obtain authorised data access (B.1.1.1).
Related business rule:	The content is accessible to customers only.
Potential misuser profile:	
Stakeholders and risks:	Iphion: Full access may be granted to people who never paid for it.

B.2.2.3 Obtain data

Name:	Obtain data
Summary:	Apart from multimedia content, the iphion servers provide a lot of additional restricted data. An attacker may intercept this data while authorised clients access it.
Basic path:	1. An iphion client or partner requests information from an iphion server. 2. The server sends out the requested data. 3. An eavesdropping attacker obtains a copy of the data.
Alternative paths:	
Exception path:	
Trigger:	Whenever restricted data is send from an iphion server to an authenticated client.
Assumptions:	The attacker can intercept or relay data streams on its own system.
Precondition:	-
Postcondition:	Restricted iphion data ends up with an unauthorised party.

Mitigation points:	When providing access to restricted data (B.2.1.4), the data itself should always be send out encrypted (B.2.3.3).
Mitigation guarantee:	Restricted meta data is only available to authorised parties.
Related business rule:	Corporate data is only accessible for iphion customers.
Potential misuser profile:	Skilled: The attacker must be able to intercept and/or reroute internet data steams.
Stakeholders and risks:	1. Iphion: Full data access might be obtained by people who never paid for it.

B.2.2.4 Hack server

Name:	Hack server
Summary:	Malicious hackers may try to hack into an iphion server disrupt its normal operation or use this as a stepping stone to obtain privileged data.
Basic path:	1. A crook sends malicious data to a server to try and trick the player into doing what he wants (e.g. exploiting buffer overflow). 2. If successful the attacker may be able to execute his own code and/or disrupt operation of the server.
Alternative paths:	
Trigger:	True: iphion servers are always online (B.2.1.5).
Assumptions:	The attacker can connect to the iphion server. Servers that cannot be reached from the internet are not directly affected by this attack (but they may be attacked via other servers).
Precondition:	-
Postcondition:	The attacker may obtain full control over an iphion server. This can be used to obtain access to data and clients as well.
Mitigation points:	• The servers should recognise and discard invalid input. • Server software should be developed and installed with security in mind. • Server software should be updated as new versions become available (B.2.3.2). • Access to the servers should be limited to what's absolutely necessary. • The servers should be monitored for abnormal behaviour.

Related business rule:	The servers will operate as specified.
Potential misuser profile:	Varies. Script-kiddies may try to use standard hacking tools against the player. Tracking regular security updates should mitigate this. Advanced hackers may try to exploit bugs in iphion's own software.
Stakeholders and risks:	• Customers: A hacked server may disrupt services for all clients. • Iphion: A successful hack may bring down the complete network and cause serious reputation challenges.

B.2.2.5 Denial-of-Service

Name:	Denial-of-Service
Summary:	Sending invalid or too many packets may cause a server to become unresponsive to regular requests, this constitutes a denial-of-service attack.
Basic path:	1. A client requests meta data or multimedia content. 2. Attackers send lots of bogus requests to the server. 3. The server is too busy handling all bogus requests, so that it is unable to respond to valid requests in a timely fashion.
Alternative path:	By sending specially crafted packets (e.g. crypto setup) the server will perform expensive calculations for each packet before it can decide to discard it as invalid.
Trigger:	True: iphion servers are always online to handle requests (B.2.1.5).
Assumptions:	-
Precondition:	-
Postcondition:	The server will become unresponsive and clients relying on this server will not operate as they should.
Mitigation points:	• When receiving packets it should be simple and efficient to decide which packets can be quickly discarded. • Simple requests shouldn't cause a disproportional workload for the server. • The relay network should be set up so that disruption of a part of the network should not affect the rest of the network. • Enough bandwidth should be available to avoid congestion.

Related business rule:	The servers will operate as specified - even under extraordinary circumstances.
Potential misuser profile:	Unskilled: Bandwidth flooding doesn't require technical know-how and cannot be prevented by the system. More sophisticated attacks do involve knowledge of the protocol and system.
Stakeholders and risks:	• Customer: Players may stop working without a clear indication of the cause. • Iphion: A successful hack may bring down the complete network.

B.2.3 Mitigation cases

B.2.3.1 Secure channel

Name:	Secure channel
Summary:	Multimedia content from the content providers may be sent to the iphion servers by means of a secure channel.
Basic path:	1. A secure channel is set up between a content provider and iphion. 2. Multimedia content is sent to the iphion servers via this channel.
Exception path:	If a secure channel can not be set up, then content may not be sent from the content provider.
Trigger:	Whenever a connection with a new content partner is established.
Mitigation:	Prevents crooks from obtaining unencrypted content that is exchanged between providers and iphion servers (B.2.2.1).

B.2.3.2 Update server software

Name:	Update server software

Summary:	New features and bug fixes must be installed on the servers as they become available. These can be either in external software packages or in *iphion*'s own code.
	1. New software becomes available. 2. A migration path is worked out (how to shut down, install, validate, restart) the software updates. 3. The software is first tested in a non-production environment. 4. Software is installed on the production platform. 5. Software is activated on the production platform.
Exception path:	In some cases it may be necessary to temporarily run servers with both the old and new version of specific software to ease migration of the clients.
Trigger:	New software versions may be released by third parties, or by the development team at *iphion*.
Mitigation:	This counters attacks based on weaknesses found in software and protocol; specifically those targeted at the server (B.2.2.4).

B.2.3.3 Encrypt data

Name:	Encrypt data
Summary:	Information that is provided to authenticated visitors must be encrypted to prevent other from obtaining the data.
Basic path:	1. A visitor connects to an iphion server. 2. The visitor is authenticated by the server (and vice versa). 3. A means of data encryption is agreed between the parties. 4. Data is requested (encrypted). 5. The requested data is sent encrypted.
Exception path:	1. Either party may abort the session if the other cannot be authenticated. 2. The server won't send any data if the requesting party isn't authorised.
Trigger:	Whenever somebody requests restricted information.
Mitigation:	Prevents crooks from obtaining restricted information that is offered by the iphion servers (B.2.2.3).

B.2.3.4 Encrypt content

Name:	Encrypt data
Summary:	Multimedia content distributed by the iphion servers must be encrypted to prevent unauthorised access.
Basic path:	1. An iphion server obtains multimedia data from a content provider. 2. The multimedia data is encrypted. 3. The encrypted data is distributed via the iphion collaborative network.
Exception path:	
Trigger:	Whenever multimedia data is relayed (that is: continuously).
Mitigation:	Prevents unauthorised parties from obtaining multimedia content data that are distributed by the iphion servers (B.2.2.1).

C. Key creation manual

The secure certificate authority is a secure computer and peripherals, without any hard disks, network connections or any other means by which information can be retrieved from it by accident or by malicious users.

The computer itself stores no information at all; therefore, it does not need to be the same computer that acts as a secure certificate authority each time. All that is required is that is has a correct time set in the BIOS, and that no peripherals are connected to it except those that are described in this document.

C.1 Setting up a secure certificate authority

The process of setting up a secure certificate authority is quite involved. The steps that need to be taken are described below. To complete these steps successfully, you will require the following items:

- A secure computer. This computer must only contain the following pieces of equipment:
  - Motherboard, processor, graphics card, memory: the bare necessities of what make a computer a computer.
  - At least four USB ports, or, if the computer does not have that many USB ports, create more ports using a USB hub.
  - A CD-RW or DVD-RW drive must be present.
- A keyboard.
- A Canon MP150 printer, connected via USB. Other printers cannot be used unless the software is patched to support other makes and models.
- A monitor needs to be attached to the computer. The computer will display only text modes.
- Three USB stick of at least 512MB in size each.

- At least three (but probably twelve) CDs or DVDs.

Specifically, the computer **MUST NOT** contain a harddisk, and no wired or wireless networks may be attached to the computer.

Build an Ubuntu live system image

This image is require to be able to boot the secure machine in each of the steps that are required to build a certificate chain.

To create this image, go to the `support/ca-boot-systems` directory in an iphion GIT repository checkout on your normal work station, and type: `./gen-chroot.sh`

Build an image of the live system

This builds a *squashfs* image of the Ubuntu live system. It also builds a ISO file that contains this *squashfs* image and some small things to make the system bootable.

Create the necessary images by typing: `./gen-live-cd.sh`

Build a USB stick which will create the root certificate CD

You will now need to build a USB stick, that you will need to boot on a secure computer as described earlier in this document. The procedure that the USB stick executes is described in detail below.

First, build the USB stick by typing: `./install-root-generator-on-stick.sh`

The script will ask you to insert a USB stick into the computer once it is ready to deploy the bootable image into a stick. Insert a USB stick that is at least 512MB in size into your work station. **WARNING**: this stick will be completely wiped of all data! Make sure that nothing that you want to keep is on the stick!

Once the stick is inserted, the program will ask you for the device name (it will give a menu with suggestions). Type the name of the device, and the program will install the image onto the stick.

Boot the USB stick on a secure computer

Now, turn off the secure computer if it was on. Make sure that there are no CDs and DVDs in the computer. Unplug all USB devices except for the optional USB hubs, the keyboard and the printer.

Plug in the USB stick that you have created. Now, turn on the computer, **verify that the time and date in the BIOS are set correctly** (to UTC time), and make it boot from the USB stick.

At the USB stick's boot prompt with the iphion logo, press ENTER, or wait 30 seconds for an automatic boot.

The USB stick will now create the root RSA key (this takes some time).

Once the stick is done with the certificate generation, you are ready to burn CD sets for the root certificate. Each set of CDs belongs to a single person. Each person gets a set of CDs (at least one, more for safety purposes, in case a CD gets damaged).

To start the procedure, make the recipient of the CD set to type in a passphrase, which he or she can remember easily, but is not easy to guess. The system will now generate an ISO image with an encrypted private root key on it. It will now ask you whether you want to write the result to a CD or DVD. You will need a blank CD or DVD to store the root certificate and its boot system on. Insert the medium into the computer and type `y` to confirm.

When the burning of the CD is finished (open the drive manually if the CD drive is closed), take out the CD and write **Root certificate** on it, along with the text **original CD** or **CD copy** depending on whether this is the first CD that you have built or a copy. Also write on it **For production** or **For development**, depending on what the CA will be used for. Finally, write the current date on the CD.

When you have finished writing, the system will ask you whether you want to burn this specific CD again. Burn the CD as many times as you see fit, for this specific person. When the person has sufficient copies, type `n` at the prompt.

The system will now ask you whether you want to create another set of CDs. If there are more people that need to receive the root certificate, repeat the procedure in the paragraph above after typing `y` at the prompt.

Be sure to burn backup copies of the CD or DVD: this type of storage medium will not last forever and scratches easily. You can burn multiple CDs or DVDs by repeatedly typing `y` at the burn prompt.

Once you are done and you have created enough copies of the root CD (be sure to remove the last CD before you do the next step), type `n` at the prompt that asks you whether you want to create another CD set. Now, press CTRL-ALT-DEL to reboot the computer. Once the computer has rebooted, remove the USB stick. The USB stick is not required any longer and can be used in other steps of this procedure.

Boot a root certificate CD on the secure computer

Insert one of the root certificate CDs into the secure computer and boot the computer from it. At the CDs boot prompt, press enter or wait 30 seconds for an automatic boot.

Once the CD has booted, it will wait until you insert a USB script stick into it. You will need to create such a stick in the next step.

Create a root certificate script stick

On your work station, you will need to create a USB script stick which will work with the root CD to build the intermediate certificates.

To build such a stick, type: `./install-root-ca-menu-on-stick.sh`

The script will ask you to insert a USB stick into the computer once it is ready to deploy the script image onto a stick. Insert a USB stick that is at least 512MB in size into your work station. **WARNING**: this stick will be completely wiped of all data! Make sure that nothing that you want to keep is on the stick!

Once the stick is inserted, the program will ask you for the device name (it will give a menu with suggestions). Type the name of the device, and the program will install the image onto the stick.

Print the root certificate private data on paper

Insert the USB script stick that you have just built into the secure computer. The computer will now ask which USB stick you want to use. Enter a device name from the suggestions menu.

The script stick will be started, and it will ask for the passphrase that unlocks the root private key that is stored on the CD. Enter it. If the passphrase is entered incorrectly, you will get lots of errors, and the stick will be rejected. Enter the device name again to retry.

A menu will now be dispput. Option a of this menu is the print option. Be sure that, before you type a, you turn on the printer, that it is connected to the secure computer, and that there is sufficient paper in it (you will need about ten pages).

Now, type a. The printer will print a dump of the root certificate's private data in a hexadecimal format with checksums. This dump is the root certificate's private key in an **unencrypted** form: if you type in the private key and compile it back to binary format, you will have access to the root private key without a passphrase. **Take great care where you store this**

paper copy! Beware: the printer prints the first page first, of course, but it stacks the papers in reverse order on its collection tray. The pages are numbered, so this is not that much of an issue. Stapling the pages together preserves their order nicely.

Then, type `e`. The printer will print a dump of the root certificate's private data in a 2D barcode format. This dump is also **not** encrypted. **Take great care where you store this paper copy!** Beware: the printer prints the first page first, of course, but it stacks the papers in reverse order on its collection tray. The pages are **NOT numbered**, so you'll need to number the pages yourself. Stapling the pages together preserves their order nicely.

Now, type `d` to store the root certificate's public information on the USB stick.

Create the client and server intermediate certificate USB sticks

Now, choose option `b` from the menu. The script will ask you on which USB stick you want to install the CD creator for the client intermediate certificate. Insert a second USB stick, wait a while until the kernel detects the stick, and press ENTER to reprobe the list of the USB sticks. Now, choose the correct device from the menu.

The stick will now be filled with the correct data. When the root certificate menu reappears, the stick can be removed from the secure computer. Lay this stick aside for a moment.

Now, choose option `c` from the menu. The script will ask you on which USB stick you want to install the CD creator for the server intermediate certificate. Insert a third USB stick, wait a while until the kernel detects the stick, and press ENTER to reprobe the list of the USB sticks. Now, choose the correct device from the menu.

The stick will now be filled with the correct data. When the root certificate menu reappears, the stick can be removed from the secure computer. Lay this stick aside for a moment, with the client CD creator stick that you already put aside.

Now, choose `q` in the menu. Remove the USB script stick from the secure computer. Then, press CTRL-ALT-DEL to reboot the computer. Take out the root CD when it is ejected from the system and close the tray.

Also take out the USB script stick. Insert the USB script stick into your desktop computer and copy the root public key information (`cacert.pem`, `cacert.der` and `index.txt`) in the `rootCA` directory to the appropriate directory in the Git repository. For production usage, copy the information

to `support/certificates/production/root/`. For development usage, copy the information to `support/certificates/development/root/`.

Create the client intermediate certificate CD

Now, reboot the computer and boot from the client certificate authority CD creator USB stick (the first one that you put aside).

Once the USB stick has booted, it will allow you to burn one or more CDs or DVDs sets with the client intermediate certificate authority on it. It will ask you for a passphrase to protect the private key data with. Type a passphrase that you can easily remember but is difficult to guess. Press `y` at the burn prompt each time you want to burn a CD or DVD.

When the burning of a CD is finished (open the drive manually if the CD drive is closed), take out the CD and write **Client intermediate certificate** on it, along with the text **original CD** or **CD copy** depending on whether this is the first CD that you have built or a copy. Also write on it **For production** or **For development**, depending on what the CA will be used for. Finally, write the current date on the CD.

Once you are done and you have created enough copies of the client CD set (be sure to remove the last CD before you do the next step), type `n` at the burn prompt. If you want to create another CD set for another person, type `y` at the prompt, otherwise type `n`.

Now, press CTRL-ALT-DEL to reboot the computer. Once the computer has rebooted, remove the USB stick. The USB stick is not required any longer and can be used in other steps of this procedure.

If more CD sets are required, one of the client CDs can be booted to create another CD set.

Create the server intermediate certificate CD

Now, reboot the computer and boot from the server certificate authority CD creator USB stick (the second one that you put aside).

Once the USB stick has booted, it will allow you to burn one or more CDs or DVDs sets with the server intermediate certificate authority on it. It will ask you for a passphrase to protect the private key data with. Type a passphrase that you can easily remember but is difficult to guess. Press `y` at the burn prompt each time you want to burn a CD or DVD.

When the burning of a CD is finished (open the drive manually if the CD drive is closed), take out the CD and write **Server intermediate certificate** on it, along with the text **original CD** or **CD copy** depending

on whether this is the first CD that you have built or a copy. Also write on it **For production** or **For development**, depending on what the CA will be used for. Finally, write the current date on the CD.

Once you are done and you have created enough copies of the server CD set (be sure to remove the last CD before you do the next step), type `n` at the burn prompt. If you want to create another CD set for another person, type `y` at the prompt, otherwise type `n`.

Now, press CTRL-ALT-DEL to reboot the computer. Once the computer has rebooted, remove the USB stick. The USB stick is not required any longer and can be used in other steps of this procedure.

If more CD sets are required, one of the server CDs can be booted to create another CD set.

Boot from the client intermediate certificate CD

Boot the secure computer from the client intermediate certificate CD. Once the CD has finished booting, it will ask for a USB script stick to be inserted.

On your work station, type: `./install-client-ca-menu-on-stick.sh` to create such a stick. The script will issue a warning that no client certificate requests have been placed on the stick image. This warning can be ignored.

Insert the generated stick into the secure computer. Choose the correct device name from the menu that appears.

The stick will ask for the passphrase of the private key data that is on the stick. Enter this.

Now, type `a`. The printer will print a dump of the client intermediate certificate's private data in a hexadecimal format with checksums. This dump is the client intermediate certificate's private key in an **unencrypted** form: if you type in the private key and compile it back to binary format, you will have access to the client intermediate private key without a passphrase. **Take great care where you store this paper copy!** Beware: the printer prints the first page first, of course, but it stacks the papers in reverse order on its collection tray. The pages are numbered, so this is not that much of an issue. Stapling the pages together preserves their order nicely.

Then, type `d`. The printer will print a dump of the client intermediate certificate's private data in a 2D barcode format. This dump is also **not** encrypted. **Take great care where you store this paper copy!** Beware: the printer prints the first page first, of course, but it stacks the papers in reverse order on its collection tray. The pages are **NOT**

numbered, so you'll need to number the pages yourself. Stapling the pages together preserves their order nicely.

From the menu, choose option `c` to copy the client intermediate certificate public information to the USB script stick.

Once the printing has finished, choose `q` from the menu. Now, reboot the computer.

Take out the USB script stick. Insert the USB script stick into your desktop computer and copy the client intermediate CA public key information (`cacert.pem` and `cacert.der`) in the `clientCA` directory to the appropriate directory in the Git repository. For production usage, copy the information to `support/certificates/production/client/`. For development usage, copy the information to `support/certificates/development/client/`.

Boot from the server intermediate certificate CD

Boot the secure computer from the server intermediate certificate CD. Once the CD has finished booting, it will ask for a USB script stick to be inserted.

On your work station, type: `./install-server-ca-menu-on-stick.sh` to create such a stick. The script will issue a warning that no server certificate requests have been placed on the stick image. This warning can be ignored.

Insert the generated stick into the secure computer. Choose the correct device name from the menu that appears.

The stick will ask for the passphrase of the private key data that is on the stick. Enter this.

Now, type `a`. The printer will print a dump of the server intermediate certificate's private data in a hexadecimal format with checksums. This dump is the server intermediate certificate's private key in an **unencrypted** form: if you type in the private key and compile it back to binary format, you will have access to the server intermediate private key without a passphrase. **Take great care where you store this paper copy!** Beware: the printer prints the first page first, of course, but it stacks the papers in reverse order on its collection tray. The pages are numbered, so this is not that much of an issue. Stapling the pages together preserves their order nicely.

Then, type `d`. The printer will print a dump of the server intermediate certificate's private data in a 2D barcode format. This dump is also **not** encrypted. **Take great care where you store this paper copy!** Beware: the printer prints the first page first, of course, but it stacks the papers in reverse order on its collection tray. The pages are **NOT**

numbered, so you'll need to number the pages yourself. Stapling the pages together preserves their order nicely.

From the menu, choose option `c` to copy the server intermediate certificate public information to the USB script stick.

Once the printing has finished, choose `q` from the menu. Now, reboot the computer.

Take out the USB script stick. Insert the USB script stick into your desktop computer and copy the server intermediate CA public key information (`cacert.pem` and `cacert.der`) in the `serverCA` directory to the appropriate directory in the Git repository. For production usage, copy the information to `support/certificates/production/server/`. For development usage, copy the information to `support/certificates/development/server/`.

C.2 Set up the U-boot certificate

We use U-boot on the player set-top-boxes. The system allows the signing of images that are booted, and also of other files that are loaded by U-boot. For this purpose, we require a CD that contains the necessary components to be able to sign files.

Build a USB stick which will create the U-boot certificate CD

You will now need to build a USB stick, that you will need to boot on a secure computer as described earlier in this document. The procedure that the USB stick executes is described in detail below.

First, build the USB stick by typing: `./install-uboot-generator-on-stick.sh`

The script will ask you to insert a USB stick into the computer once it is ready to deploy the bootable image into a stick. Insert a USB stick that is at least 512MB in size into your work station. **WARNING**: this stick will be completely wiped of all data! Make sure that nothing that you want to keep is on the stick!

Once the stick is inserted, the program will ask you for the device name (it will give a menu with suggestions). Type the name of the device, and the program will install the image onto the stick.

Boot the USB stick on a secure computer

Now, turn off the secure computer if it was on. Make sure that there are no CDs and DVDs in the computer. Unplug all USB devices except for the optional USB hubs, the keyboard and the printer.

Plug in the USB stick that you have created. Now, turn on the computer and make it boot from the USB stick.

At the USB stick's boot prompt with the iphion logo, press ENTER, or wait 30 seconds for an automatic boot.

The USB stick will now create the U-boot certificate files. It will produce four files, and each of them will have to be encrypted. Therefore, the procedure will ask you to enter your passphrase **eight** times!

Once the stick is done with the certificate generation, you are ready to burn CDs that contain the U-boot certificate.

The system will now generate an ISO image. It will then ask you whether you want to write the result to a CD or DVD. You will need a blank CD or DVD to store the certificate and its boot system on. Insert the medium into the computer and type `y` to confirm.

When the burning of a CD is finished (open the drive manually if the CD drive is closed), take out the CD and write **U-boot certificate and AES keys** on it, along with the text **original CD** or **CD copy** depending on whether this is the first CD that you have built or a copy. Also write on it **For production** or **For development**, depending on what the certificate will be used for. Finally, write the current date on the CD.

When you are finished writing, the system will ask you whether you want to burn this specific CD again. Burn the CD as many times as you see fit, for each person that needs copies. When you have made sufficient copies, type `n` at the prompt.

Be sure to burn backup copies of the CD or DVD: this type of storage medium will not last forever and scratches easily. You can burn multiple CDs or DVDs by repeatedly typing `y` at the burn prompt.

Once you are done and you have created enough copies of the root CD (be sure to remove the last CD before you do the next step), type `n` at the prompt that asks you whether you want to create another CD set. Now, press CTRL-ALT-DEL to reboot the computer. Once the computer has rebooted, remove the USB stick. The USB stick is not required any longer and can be used in other steps of this procedure.

Create a U-boot certificate script stick

On your work station, you will need to create a USB script stick which will work with the U-boot CD.

To build such a stick, type: `./install-uboot-menu-on-stick.sh`

The script will ask you to insert a USB stick into the computer once it is ready to deploy the script image onto a stick. Insert a USB stick that is at

least 512MB in size into your work station. **WARNING**: this stick will be completely wiped of all data! Make sure that nothing that you want to keep is on the stick!

Once the stick is inserted, the program will ask you for the device name (it will give a menu with suggestions). Type the name of the device, and the program will install the image onto the stick.

Print the U-boot files on paper

Insert the USB script stick that you have just built into the secure computer. Boot from this USB stick.

The stick will ask for the U-boot certificate and AES key CD. Insert this CD into the computer and press ENTER. Once the CD has been mounted, the stick will need to decrypt the files that it requires. For this purpose, you will need to enter the passphrase to unlock the files **four** times.

A menu will now be dispput. Options `a` and `b` of this menu are the printing options. Be sure that, before you type `a` or `b`, you turn on the printer, that it is connected to the secure computer, and that there is sufficient paper in it (you will need about three pages per print option).

Now, type `a`. The printer will print a dump of the U-boot files. The printed files are not encrypted in any way. **Take great care where you store this paper copy!** The pages are not related to each other in any way, so their order is not important.

Then, type `b`. The printer will print a dump of the U-boot files in a 2D barcode format. This dump is also **not** encrypted. **Take great care where you store this paper copy!** The pages are not related to each other in any way, so their order is not important.

Now, type `c` to store the public and private U-boot files information on the USB stick. If the CD was not already mounted, the script will you to do so.

Choose q from the menu. Now, reboot the computer.

Take out the USB script stick. Insert the USB script stick into your desktop computer and copy the U-boot public information in the `public` directory (all the files that are present there) to the appropriate directory in the Git repository. For production usage, copy the information to `support/certificates/production/u-boot/`. For development usage, copy the information to `support/certificates/development/u-boot/`.

The `private` directory contains files that are **not** suitable for storage in the Git repository, but may be required by the party that produces the STB.

The setup is now complete

You have now created five types of CDs: a root certificate authority CD, a client certificate authority CD, a server certificate authority CD, a U-boot certificate CD and a U-boot image signing CD. Futhermore, hardcopies have been printed on paper for the most crucial data on the CDs.

C.3 Generating client certificates

To generate client certificates for Iphion player, you must boot the secure computer with the client intermediate certificate authority CD.

The client private keys are generated on the secure computer, however the client IDs that will be used in the *common name* and as certificate ID number, need to be generated first on a network-connected computer (to be sure these are unique and to directly store them in the network database). The common name (client identification) will be of the form *number*`@users.iphion.nl`. These IDs are generated with the `prepare_peer_ids` script, which use is outside the scope of this document.

Next, on your workstation, make sure your GIT checkout is up-to-date and then build a USB stick with the `./install-client-ca-menu-on-stick.sh` program, which requires the `peers_ids_file` with a list of fresh unique IDs as its arguments.

Insert the USB stick into the secure computer, choose the correct USB stick device name, enter the passphrase to unlock to private key and choose option `b` from the menu. The certificates will now be signed by the intermediate authority.

Next, choose `q` from the menu. Remove the USB stick and plug it into your work station. There, run the `./read-back-client-usb-stick.sh /dir/to/place/certificate/files` to copy the certificates from the USB stick to the specified directory and update the serial information in the GIT repository.

C.4 Generating server certificates

Server certificates don't use a unique number as ID, but rather use the internet hostname. This should be a combination of the service and the software version for which the certificate will be used. For example: `epg.1-1-3.iphion.nl`.

Server certificates are **not** generated on the secure computer, but rather on the server where they will be used (or equivalent environment). Only

the server certificate requests will be copied. Generate a server key and certificate request with the ./generate_server_keyreq.sh script, adding the hostname with the -i option. Copy the certificate requests to a local directory on your workstation.

Make sure your GIT checkout on your workstation is up-to-date and then build a USB stick with the ./install-client-ca-menu-on-stick.sh program, which requires the list of all request files as its argument.

You must boot the secure computer with the server intermediate certificate authority CD. Insert the USB stick into the secure computer, choose the correct USB stick device name, enter the passphrase to unlock to private key and choose option b from the menu. The certificates will now be signed by the intermediate authority.

Next, choose q from the menu. Remove the USB stick and plug it into your work station. There, run the ./read-back-server-usb-stick.sh /dir/to/place/certificate/files to copy the certificates from the USB stick to the specified directory and update the serial information in the GIT repository. Now copy the signed server certificate to the correct location on the server.

C.5 Generating signed image files for U-boot

To generate signed image files for U-boot, you must boot the secure computer with the U-boot menu stick, as described below.

On your work station, make sure your Git checkout is up-to-date and then build a USB stick with the ./install-uboot-menu-on-stick.sh program, now with the files to sign as its arguments. The extensions of the files must be correct. Currently, only files that end in .uImage, .bin and .mipsel_boot are supported.

Insert the USB stick into the secure computer and boot from it. The CD will require the U-boot certificate and AES key CD to be inserted into the secure computer once it has booted. Do so and press enter. The encrypted files now need to be decrypted, which requires you to enter your passphrase **four** times.

From the menu that appears, choose d. The files on the stick will now be signed. Two versions are generated per input file, one encrypted, one not encrypted.

Now, choose q from the menu and reboot the computer. Take out the script stick and insert it into your computer. The signed files will have been stored on the stick, probably under /media/casper-rw/signed_files.

C.6 Making copies of CDs or sets of CDs for other people (not U-boot)

Whatever you do: do **NOT** copy CDs in your desktop computer! The CDs contain sensitive information (albeit encrypted with a passphrase) that are not allowed to leave the secure computer environment.

To make a copy of a CD, boot the secure computer from that CD. Then, on your desktop PC, produce the corresponding script stick for the USB. When the CD asks for the script stick, insert the stick into the secure computer. In the menu, choose the option to create a copy of the CD (the option letter differs depending on which CD you are trying to copy).

The script will now prompt you for a stick device name. Insert a **second** USB stick (the contents will be wiped, be careful), wait until the kernel detects the stick, and then press ENTER to reprobe the USB sticks. Now, enter the correct device name at the prompt.

When the menu reappears, type `q` to stop the USB script stick, and press CTRL-ALT-DEL to reboot the computer. Take out the CD when you are prompted to do so. Press enter afterwards. When the computer is rebooting, remove the USB script stick from the computer, but leave in the other USB stick and boot from that.

The USB stick, once booted, will ask you for the passphrase that will unlock the private data on the stick. This is the same passphrase as is needed for the CD that you are trying to copy. The stick will now ask you whether you want to make a copy of the CD. Answer `y`. The stick will now ask for a new passphrase. If you are making a backup for yourself, enter the same passphrase as above. If you making a copy for somebody else, have that other person enter their passphrase.

You can now burn as many copies as you think is necessary. Enter `n` at the prompt when you are done burning the last required CD. You now have the option of burning another CD set for somebody else with a different passphrase. Enter `y` if you want to burn another set. Enter `n` to stop the copying operation; reboot the computer with CTRL-ALT-DEL now.

C.7 Making copies of the U-boot CD

Whatever you do: do **NOT** copy CDs in your desktop computer! The CDs contain sensitive information (albeit encrypted with a passphrase) that are not allowed to leave the secure computer environment.

To make a copy of a CD, boot the secure computer with the U-boot script stick, which can be created using the `./install-u-boot-menu-on-stick.sh`

script. Once the menu appears (after inserting the original CD and unlocking the files on it), choose option `e`. You will be asked for a new passphrase. You will need to enter this **eight** times. A new CD image will now be created. When the prompt appears that asks you whether you want to burn the image, take out the original CD and replace it with a blank one. Press `y` now, and subsequently, type either `cd` or `dvd` depending on the type of the blank medium.

You can now burn as many copies as you think is necessary. Enter `n` at the prompt when you are done burning the last required CD. You now have the option of burning another CD set for somebody else with a different passphrase. Enter `y` if you want to burn another set. Enter `n` to stop the copying operation; you will be returned to the menu, after you are asked to replace the original CD in the drive.

Bibliography

[ALF93] Anne Dardenne Axel, Axel Van Lamsweerde, and Stephen Fickas, *Goal-directed requirements acquisition*, Science of Computer Programming, 1993, pp. 3–50.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, *Basic concepts and taxonomy of dependable and secure computing*, IEEE Transactions on Dependable and Secure Computing **1** (2004), no. 1, 11–33.

[AMV96] Paul van Oorschot Alfred Menezes and Scott Vanstone, *Handbook of applied cryptography*, ch. 12 - Key Establishment Protocols, CRC Press, 1996.

[Asn08] Yudis Asnar, *Si* goal risk framework tool*, 2008.

[BBB+06] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid, *Recommendation for key management*, Tech. Report 800–57, National Institute of Standards and Technology (NIST), 2006.

[Bon07] Alessio Bonetti, *Si* user's guide*, Tech. report, 2007.

[Can05] Ericsson Research Canada, *Distributed security infrastructure and digital signatures in the kernel*, 2005, http://disec.sourceforge.net/.

[Com96] EBU/CENELEC/ETSI Joint Technical Commitee, *Support for use of scrambling and conditional access (CA) within digital broadcasting systems*, Tech. Report 289, European Telecommunications Standards Institute (ETSI), 1996.

[Cus07] ETSI DVB Custodian, *DVB scrambling technology licence and non-disclosure agreement*, 2007, http://www.etsi.org/WebSite/document/Algorithms/Licence.SCRAM.doc.

[DA99] T. Dierks and C. Allen, *The TLS protocol: version 1.0*, Tech. Report RFC 2246, IETF Network Working Group, 1999.

[DR08] T. Dierks and E. Rescorla, *The TLS protocol: version 1.2*, Tech. Report RFC 5246, IETF Network Working Group, 2008.

[FIS02] *Federal information security management act*, no. 44 U.S.C., Sec 354, United States Code, 2002.

[FSF07] Inc Free Software Foundation, *Gnu general public license, version 3*.

[GKMP04] Paolo Giorgini, Manuel Kolp, John Mylopoulos, and Marco Pistore, *Methodologies and software engineering for agent systems*, ch. The Tropos Methodology: an overview, Kluwer Academic Publishing, 2004.

[GMNS03] Paolo Giorgini, John Mylopoulos, Eleonora Nicchiarelli, and Roberto Sebastiani, *Formal reasoning techniques for goal models*, Journal of Data Semantics **1** (2003), 1–20.

[GMS05] Paolo Giorgini, John Mylopoulos, and Roberto Sebastiani, *Goal-oriented requirements analysis and reasoning in the tropos methodology*, Engineering Applications of Artificial Intelligence **18** (2005), 159–171.

[How07] David Howells, *Kernel module signing (modsign)*, 2007, http://lwn.net/Articles/222162/, http://cvs.fedora.redhat.com/viewvc/rpms/kernel/F-8/?root=extras.

[idct06] ISO/IEC Joint Technical Committee 1 'Information Technology' / Subcommittee 31 'Automatic identification and data capture techniques', *Data matrix bar code symbology specification*, no. 16022, ISO/IEC, 2006.

[Inc02] RSA Security Inc., *PKCS 1: RSA cryptography standard v2.1*, Tech. report, 2002.

[IT04] Telecommunication Standardization Sector ITU-T, *Message sequence chart (msc)*, no. Z.120, International Telecommunication Union ITU, 2004.

[IT08] ———, *Public-key and attribute certificate framework*, no. X.509, International Telecommunication Union ITU, 1988–2008.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-hashing for message authentication*, Tech. Report RFC 2104, IETF Network Working Group, 1997.

[KL07] J. Katz and Y. Lindell, *Introduction to modern cryptography*, Chapman and Hall/CRC Press, 2007.

[Koc09] Werner Koch, *GNU privacy guard (GnuPG)*, 2009, http://www.gnupg.org/.

[MAM+99] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, *X.509 internet public key infrastructure online certificate status protocol (ocsp)*, Tech. Report RFC 2560, IETF Network Working Group, 1999.

[Min09] Peter Minten, *iphion peer assisted protocol (iPAP) v3*.

[MMZ07] Fabio Massacci, John Mylopoulos, and Nicola Zannone, *An ontology for secure socio-technical systems*, Handbook of Ontologies for Business Interaction, The IDEA Group, 2007.

[NI01a] Information Technology Laboratory (NIST-ITL), *Security requirements for cryptographic modules*, Tech. Report 140–2, National Institute of Standards and Technology (NIST), 2001.

[NI01b] ______, *Specification for the advanced encryption standard (AES)*, Tech. Report 197, National Institute of Standards and Technology (NIST), 2001.

[NI09] ______, *Digital signature standard (dss)*, Tech. Report 186–3, National Institute of Standards and Technology (NIST), 2009.

[oapmhi00] ISO/IEC Joint Technical Committee 1 'Information Technology' / Subcommittee SC 29 'Coding of audio picture multimedia and hypermedia information', *Generic coding of moving pictures and associated audio information*, no. 13818, ISO/IEC, 2000.

[ohsi98] ISO Technical Committee 159 'Ergonomics' / Subcommittee 4 'Ergonomics of human-system interaction', *Ergonomic requirements for office work with visual display terminals (vdts)*, no. 9241, ch. 11: Guidance on Usability, ISO, 1998.

[ohsi99] ______, *Human-centred design processes for interactive systems*, no. 13407, ISO, 1999.

[Poe08] M.L. Poelstra, *Analysing and improving iphion collaborative IPTV*, Master's thesis, Eindhoven University of Technology, 2008.

[Pro09] The Mozilla Project, *Network security services (NSS)*, 2009, http://www.mozilla.org/projects/security/pki/nss/.

[Res99] E. Rescorla, *Diffie-Hellman key agreement method*, Tech. Report RFC 2631, IETF Network Working Group, 1999.

[Res00] ———, *HTTP over TLS*, Tech. Report RFC 2818, IETF Network Working Group, 2000.

[RHCF05] Manny Rayner, Beth Ann Hockey, Nikos Chatzichrisafis, and Kim Farrell, *OMG unified modeling language specification*, Version 1.3, 1999 Object Management Group, Inc, 2005.

[Riv92] R. Rivest, *The MD5 message-digest algorithm*, Tech. Report RFC 1321, IETF Network Working Group, 1992.

[SC02] Victor F.A. Santander and Jaelson F. B. Castro, *Deriving use cases from organizational modeling*, Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE'02), 2002.

[Sim96] W. Simpson, *PPP challenge-handshake authentication protocol*, Tech. Report RFC 1994, IETF Network Working Group, 1996.

[SLdW08] Marc Stevens, Arjen Lenstra, and Benne de Weger, *Vulnerability of software integrity and code signing applications to chosen-prefix collisions for MD5.*

[SO05] Guttorm Sindre and Andreas L. Opdahl, *Eliciting security requirements with misuse cases*, Requirements Engineering **10** (2005), no. 1, 34–44.

[Sol92] K. Sollins, *The TFTP protocol (revision 2)*, Tech. Report RFC 1350, IETF Network Working Group, 1992.

[vS09] Johan van Selst, *iphion crypto key policy.*

[WAH06] Carl Worth, Steve Ayer, and Jamey Hicks, *Itsy package management system*, 2006, http://www.handhelds.org/moin/moin.cgi/Ipkg.

[Wei01] Joel Weise, *Public key infrastructure overview.*

[WW04] Ralf-Philipp Weinmann and Kai Wirt, *Analysis of the DVB common scrambling algorithm.*

[X905] Accredited Standards Committee X9, *The elliptic curve digital signature algorithm (ECDSA)*, Tech. Report X9.62, American National Standards Institute, 2005.

[YM94] Eric S. K. Yu and John Mylopoulos, *From E-R to 'A-R' - modelling strategic actor relationships for business process*

reengineering, Proceedings of 13th Int. Conf. on the Entity-Relationship Approach (ER'94), number 881 in Lecture Notes in Computer Science, Springer-Verlag, 1994, pp. 548–565.