CHALMERS
UNIVERSITY OF TECHNOLOGY

Evolution of a Secure Voice Communication System

A Change Pattern for Multiparty Conference Systems

Master's Thesis in Software Engineering

GUSTAV SUNDIN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016:01

Evolution of a Secure Voice Communication System

A Change Pattern for Multiparty Conference Systems

GUSTAV SUNDIN

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
Gothenburg, Sweden 2016

Evolution of a Secure Voice Communication System
A Change Pattern for Multiparty Conference Systems
GUSTAV SUNDIN

© GUSTAV SUNDIN, 2016.

Supervisor: Riccardo Scandariato, Department of Computer Science and Engineering
Examiner: Regina Hebig, Department of Computer Science and Engineering

Master's Thesis 2016:01
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: © CRYPTIFY AB

Typeset in LaTeX
Gothenburg, Sweden 2016

Evolution of a Secure Voice Communication System
A Change Pattern for Multiparty Conference Systems
GUSTAV SUNDIN
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

In order to maintain a system's security properties during evolution, it is important to follow a principled way of evolution. Doing so will decrease both the time needed to be spent on evolution as well as the number of security risks and other errors that might arise during the process. One such principled way of evolution is to use change patterns. A change pattern always covers the evolution of two closely intertwined artifacts on different levels of abstraction (such as requirements and software architecture), and gives a principled way of evolving one of the artifacts based on the changes made to the other.

In this master's thesis, a VoIP system known as Cryptify Call is evolved with the purpose of identifying at least one such change pattern. The goal of the thesis is more specifically to find a general solution on how to evolve a system from secure one-to-one into secure many-to-many communication, without violating any of the system's security requirements along the way. Two alternative solutions to this problem are identified for the Cryptify Call system, which are then expressed as abstract and context-free change patterns. This ensures that the identified solutions are applicable not only to the Cryptify Call system or other VoIP applications, but to any type of system with a suitable software architecture. Both of the identified change patterns cover how the system's software architecture has to evolve due to changes made to the system's functional requirements.

In order to identify these change patterns, the Cryptify Call system's software architecture and security requirements had to be modeled using UML and SI* notation respectively, both before and after evolution. The most important roles in the software architecture for each of the solutions were then mapped to a template showing the architectural-level transformation necessary to apply that change pattern. While two alternative change patterns are given in the thesis, only one of them was actually implemented in the Cryptify Call system.

Keywords: secure software evolution, change patterns, multiparty communication, multicall, secure communication, action research.

Acknowledgements

First of all, I would like to thank my supervisor Riccardo Scandariato for his help and patience with me. I would also like to thank everyone at Cryptify and my family for their support. Finally a big thanks to the almighty Judas Priest for eternal inspiration!

Gustav Sundin, Hisingen, January 2016

Contents

List of Figures **xii**

Abbreviations **xv**

1 Introduction **1**

1.1 Background 1

1.2 Problem Statement 2

1.3 Purpose of the Study 2

1.4 Research Methodology 3

1.5 Data Collection 5

1.6 Contributions 7

1.7 Threats to Validity 7

1.7.1 Construct Validity 8

1.7.2 Internal Validity 9

1.7.3 External Validity 9

1.7.4 Reliability 10

1.8 Structure of the Thesis 10

2 Related Work and Foundations **11**

2.1 Software Evolution 11

2.2 Secure Software Evolution 12

2.3 Model-Driven Evolution of Security Artifacts . . . 13
2.4 Thesis Foundations . . . 14
2.4.1 SI* Notation . . . 14
2.4.2 Change Patterns . . . 15
2.4.3 Identity-Based Public Key Cryptography . . . 16

3 The Existing Cryptify Call System **19**
3.1 High-Level Description . . . 19
3.2 Security Requirements . . . 20
3.2.1 The MIKEY-SAKKE Protocol . . . 21
3.2.1.1 Monthly Key Renewal . . . 22
3.2.1.2 Advantages and Disadvantages . . . 23
3.2.2 Authentication . . . 24
3.2.3 Confidentiality . . . 24
3.2.4 Trust Model . . . 27
3.3 Cryptify Management System (CMS) . . . 29
3.4 Cryptify Rendezvous Server (CRS) . . . 30
3.5 Cryptify Caller Application (CCA) . . . 31
3.5.1 Low-Level Architecture . . . 31
3.5.1.1 The MSS User Agent Layer (MUA) . . . 32
3.5.1.2 The Multi Session Stack Layer (MSS) . . . 34
3.6 Validating the Architectural Model . . . 36

4 Towards a Multiparty Calling System **37**
4.1 Evolution Plan . . . 37
4.2 Evolved Security Requirements . . . 38
4.2.1 Authentication . . . 38

4.2.2 Confidentiality 38
4.2.3 Trust Model 39
4.3 Evolved Architecture 41
4.4 Implementing the Evolved Architecture 43
4.4.1 Merging of Audio Streams 43
4.5 Performance Evaluation 45

5 Change Patterns for Multiparty Communication 49

5.1 Main Change Pattern 49
5.1.1 Solution 50
5.1.1.1 Architectural Support 50
5.1.1.2 Change Guidance 52
5.1.1.3 Automatic Transformation 54
5.1.2 Advantages and Disadvantages 57
5.2 Alternative Change Pattern 57
5.2.1 Automatic Transformation 59
5.2.2 Advantages and Disadvantages 60
5.3 Choosing and Applying a Change Pattern 61

6 Discussion and Conclusions 65

6.1 The Identified Change Patterns 65
6.2 Other Applications 66
6.3 Future Work 67
6.4 Future Research 68
6.5 Challenges 69
6.6 Conclusions 70

Bibliography **73**

A Different Merging Algorithms **I**

List of Figures

1.1 Action Research Methodology 4

1.2 Software Evolution Process 6

2.1 SI* Key 15

2.2 SI* Example 16

2.3 Change Pattern Example 17

3.1 Cryptify Call Overview 20

3.2 Authentication 1 25

3.3 Authentication 2 26

3.4 Confidentiality 27

3.5 Trust Model Before Evolution 28

3.6 Architecture of the Cryptify Caller Application 32

3.7 Internal Architecture of the MUA layer 33

3.8 Internal Architecture of the Multi Session Stack layer 35

4.1 Trust Model After Evolution 40

4.2 Internal Architecture of the MUA Layer After Evolution 42

4.3 Call Coordinator Screenshot 44

4.4 CPU Usage Graph 46

5.1 Main Change Pattern 51

5.2 Architectural Support Template for the Main Change Pattern 53

5.3 UML Profile 55

5.4 Change Guidance for the Main Change Pattern 58

5.5 Change Scenario for the Alternative Change Pattern 62

5.6 Change Guidance for the Alternative Change Pattern 63

A.1 The Clipping Function IV

A.2 Linear Attenuation IV

A.3 Viktor Toth's Method V

A.4 The Tanh Function V

Abbreviations

AES Advanced Encryption Standard, a symmetric-key encryption algorithm.

CCA Cryptify Caller Application, the system's client application.

CMS Cryptify Management System.

CRS Cryptify Rendezvous Server.

ECCSI Elliptic Curve-Based Certificateless Signatures.

GUI Graphical User Interface.

KMS Key Management Server.

MIKEY Multimedia Internet Keying Format.

MSS Multi Session Stack, the bottom layer of the client application architecture.

MUA MSS User Agent, the middle layer of the client application architecture.

PCM Pulse-Code Modulation, a way of representing audio data digitally.

QVT Operational (Query/View/Transformation), an imperative modelling language used for model-to-model transformations.

QR code Quick Response code, a type of barcode that can contain binary data.

SAKKE Sakai-Kasahara Key Encryption.

SI* A modelling language that is used throughout this thesis in order to model security features and trust relationships.

SRTP Secure Real-time Transport Protocol, a protocol used for sending real-time audio.

TLS-PSK Transport Layer Security with Pre-Shared Keys, a symmetric-key cryptography protocol.

UML Unified Modeling Language. UML is mainly used in order to visualize how a software system is designed.

1

Introduction

A large part of the budget at many software companies today is spent on evolving existing software, for example to conform to new environments or changing requirements. Any streamlining of the evolution process could therefore lead to substantial savings, both in terms of time and money. The fast pace in the field of software and the ever rising number of software systems in use further increases the need of robust methods and tools to use during evolution.

This master's thesis intends to add a piece of knowledge to the field of software evolution research by investigating the relationship between evolution and security. This is done by evolving a secure telecommunications system into supporting secure multiparty calls. This introductory chapter begins by giving some background to the thesis in section 1.1, and continues by defining the research question and purpose of the thesis in sections 1.2 and 1.3. How the thesis was conducted and the research methodology followed is described in sections 1.4. Some possible limitations to the validity of the thesis' findings are mentioned in section 1.7 and finally the structure of the rest of the thesis is described in section 1.8.

1.1 Background

Software evolution is always associated with some amount of risk, since every time the code changes; there is a probability that new bugs are introduced with the change [26]. Security bugs are no exception, and therefore security flaws have a tendency to accumulate as the software evolves unless actively counteracted [36]. While some research has been done in this area, not much of it focuses on the security aspects of evolution. How to guarantee that security requirements are maintained during software evolution is therefore still an area open to research. The motivation behind this thesis is thus to shed some light on the role of security in relation to the co-evolution of requirements, architecture and implementation.

In this thesis, a software system used for conducting secure voice and text communication is evolved into also supporting secure multiparty calls. Throughout the whole process of evolution, focus lays on finding ways to minimize the risk of introducing any security flaws. One way to do that is to follow a principled process of evolution,

for example based on change patterns, a concept introduced by Yskout, Scandariato and Joosen [51]. A change pattern (or pattern of co-evolution) is a principled way of co-evolving two related artifacts, for example requirements and software architecture. Based on the changes to one of the artifacts, the change pattern gives guidance on how to transform the second artifact. This can help streamlining the evolution process, and will guarantee a sound result as long as the change pattern itself is sound [51]. The specific purpose of this study is to come up with at least one change pattern that can generalize the evolution of any software system moving from a one-to-one communication paradigm into also supporting many-to-many communication, without jeopardizing any of the system's security properties. Such a change pattern could prove of great use to software developers evolving similar pieces of software in the future.

1.2 Problem Statement

A software system can be described on different levels of abstraction, for example on the requirements, architectural or implementation level. Any changes on one of these abstraction levels will need to propagate to all other levels in order to keep the system and its specification coherent. This is a time consuming and error-prone process that could potentially be made more efficient, for example by the use of change patterns. In theory, following such a principled way of evolution could help keeping the process on a high level of abstraction and reduce the effort needed to be spent on evolution [14]. As long as the change pattern itself is sound, the resulting software can also be guaranteed to be sound, greatly reducing the risk of introducing security issues of any kind.

In order for this approach to be useful though, a sufficiently large set of change patterns needs to be defined. And as the idea of using change patterns during evolution is still very young, not many change scenarios have been covered with precise patterns yet. A change pattern guiding the transition from one-to-one into many-to-many communication is one such pattern that still is missing. Therefore the following questions are the main research questions to be discussed in this study:

Is it possible to find one or more general solutions on how to evolve a system from secure one-to-one into secure many-to-many communication? How can such a solution be expressed in the form of a change pattern?

1.3 Purpose of the Study

The scope of the study is to investigate how security related changes on the architectural level are caused by changing requirements, and then in turn propagated to changes in the actual code. These changes can for example be changes to security

requirements, or changes to other requirements that are in turn affected by overlaying security constraints. Either way, it is desirable to be able to perform the change at such a high level of abstraction as possible. This will both prevent eroding the software architecture as well as giving the developers a better understanding of the implications of the change.

The results of this study – two separate change patterns on how to evolve a secure one-to-one communication system into also supporting secure multiparty communication – will hopefully be of use to practitioners working with software evolution in the industry. Even though it focuses mostly on the communication and security aspects of evolution, it can hopefully be combined with the findings of future studies in the same direction to form a more complete picture of the process of software evolution. The findings of the study will also act as a building block and a source of inspiration for future research in the field of software evolution.

1.4 Research Methodology

In order to answer the research questions specified in section 1.2, a study was performed at the company Cryptify AB, located in Kungsbacka, Sweden. Cryptify AB maintains and develops secure telecommunications applications for making encrypted voice calls and sending text messages. The company is specialized in software security, which meant that the security-related aspects of evolution could be studied in great detail. The specific system studied in this thesis is a smartphone application that is to be evolved from only allowing one-to-one phone calls into also supporting many-to-many communication (telephone conferences). This system, known as Cryptify Call, will be described in chapter 3.

The study was carried out using the action research methodology [13], which basically means that a real-world problem is solved by the researcher(s) and that the experience gained during this process is continuously reflected upon and analyzed in order to be able to draw general conclusions regarding the problem being investigated [18]. This flexible approach enables practical problem solving and research to be conducted simultaneously [15]. In the case of this specific study, the problem to be solved was to evolve the secure one-to-one communication system of Cryptify Call into also supporting secure multiparty communication, while at the same time identifying at least one change pattern covering this change scenario.

Action research first emerged as a research methodology in social psychology in the mid 1940s [6] and has been used when studying software and information systems since around 1985 [47]. Action research is based upon the idea that the act of performing an action leads to a deeper understanding of it, and it is for this reason not necessary to have a strict distinction between researchers and practitioners in an action research study [6]. Instead, the researchers involved in action research are supposed to both contribute practically to solving the problem being studied (i.e., perform actions) and observe it (i.e., do research) [6]. Another key aspect of action

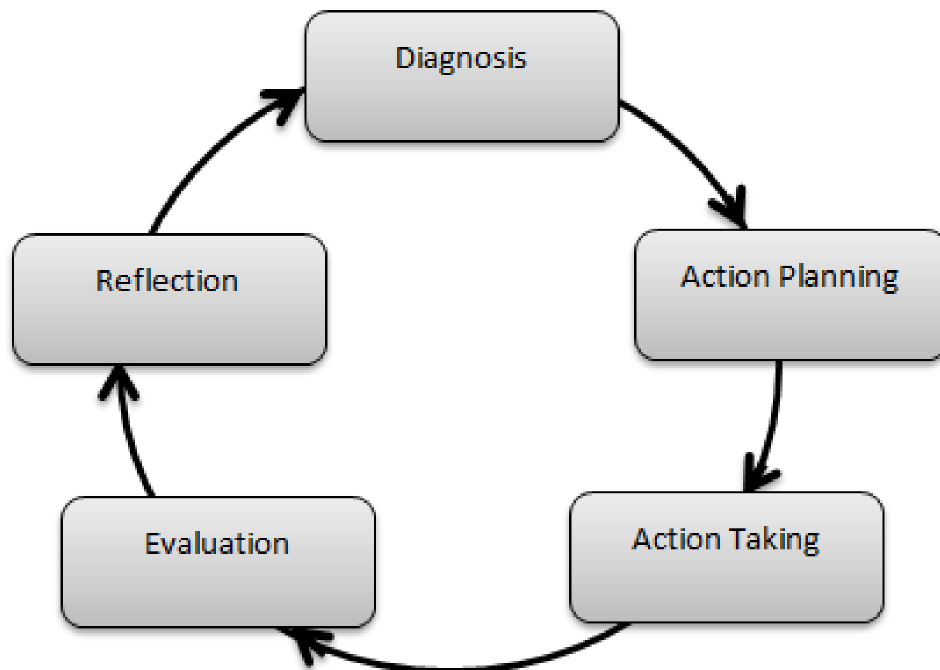


Figure 1.1: A cycle in the iterative action research methodology begins with a diagnosis in order to identify the problem that is to be solved. Next comes the action planning step, where a set of actions for how the problem can be solved is decided upon, based on the theoretical framework laying the foundation for the study. These actions are carried out in the action taking step and evaluated in the evaluation step. Finally comes the reflection step, where the knowledge gained during the other steps is analyzed in order to construct or refine some sort of theoretical results and/or to provide input for the next cycle in the iterative process. The reflection step is the most important part of the action research methodology, and it is usually an ongoing process instead of being confined to the end of each iterative cycle [6].

research is collaboration [6], which means that the researcher should strive to work together with domain experts as much as possible. In the same way, these domain experts or other participants of the study are also encouraged to contribute to the scientific and theory-building aspects of the study as well [6].

The main motivation behind all action research studies is to make a contribution of scientific knowledge to the research community, but almost as important is to help a client or organization (Cryptify AB in the case of this specific study) to solve the practical problem being investigated. An action research study can therefore be expected to give benefits to both the research community as well as to the client or organization itself, but to count as action research the study must be based upon a theoretical foundation of previous research [6]. The theoretical background of this master's thesis will be presented in chapter 2.

Action research studies commonly uses an iterative approach, as the one visualized in figure 1.1 [6] [13]. Doing so enables the researchers to continuously reflect upon

and analyze their observations in order to be able to draw conclusions and produce some sort of results. With this iterative process in mind, the study was divided into the following subtasks:

- First, the current architecture of the Cryptify Call system was visualized using UML notation (step 1 in figure 1.2). The model was then complemented by a trust model showing the security requirements that the system needs to fulfill. The relevant features in the system's trust model were also mapped to the corresponding components in the software architecture, in order to know which changes that might affect the system's security requirements. The conduction of this step and the resulting models are presented in chapter 3. This step was carried out by the author of the thesis alone, but the resulting architectural and trust models were verified by domain experts at the company.
- Next, the system's architecture was evolved in order to conform to the new requirements. The evolved architecture and trust model of the system were also visualized, again using UML notation (step 3 in figure 1.2). This step was performed by the author of the thesis in collaboration with the company and using the iterative process shown in figure 1.1. The evolved architecture and trust model is described in chapter 4. At this point, an alternative approach on how to evolve the architecture was also identified. This alternative approach was also modeled, and is presented as the alternative change pattern given in section 5.2.
- The code of the system's implementation also had to be evolved (step 2 in figure 1.2) in order to correspond to one the new architectures identified in the previous step. This step was also carried out by the author of the thesis using the iterative process shown in figure 1.1 and validated by domain experts at the company. The implementation of the evolved architecture is described in section 4.4.
- Finally, the findings of the study were summarized by the author in the form of two separate and context-free change patterns (see chapter 5). These change patterns are based upon the knowledge gained and documented in the project log during the previous steps of the study.

1.5 Data Collection

As the main intent of this thesis is to extend the currently existing literature on software evolution with at least one new change pattern, the thesis can be described as theory-driven action research, using the terminology of Dick [15]. But in order to be able to express the general change patterns in chapter 5, some data of course first had to be gathered and analyzed. In this section, the data collection strategy used throughout this thesis project will be described, as it is important to clearly

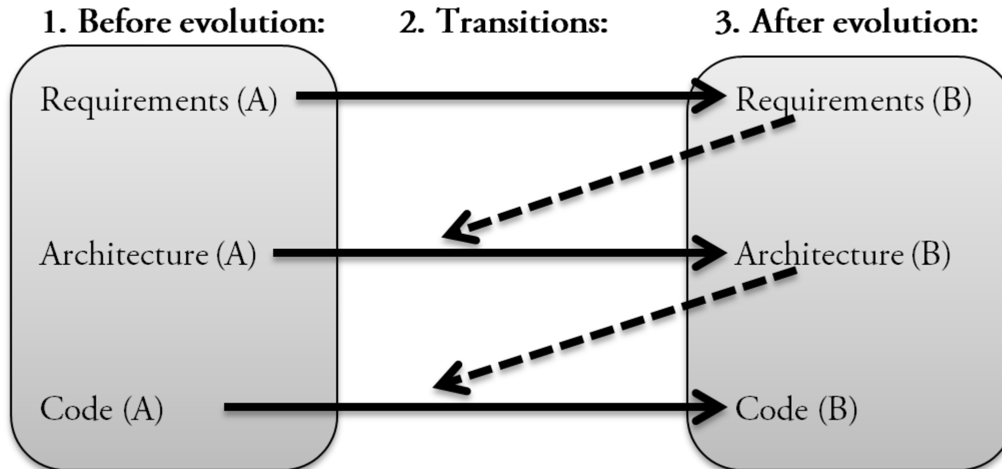


Figure 1.2: This figure shows the process of software evolution. A set of software artifacts are evolved through a set of transitions. The dashed arrows show that a change in the code is usually caused by the software architecture being evolved, which is in turn usually caused by a change in the system's requirements.

specify the techniques that will be used to gather empirical data when following the action research methodology [6].

The data which was collected during the study consists of the artifacts shown in figure 1.2 (requirements, architecture and code), both before and after evolution, as well as the transitions used to evolve these artifacts. By analyzing these transitions, it has been possible to identify two separate solutions for how to evolve a software architecture into supporting many-to-many communication between the system's end users, and to express these two solutions as two context-free change patterns. Both of these change patterns were identified by observing a change in the system's trust model. These changes in turn led to some changes in the software architecture of the system, which are captured in the form of an architectural-level transformation for each of the identified change patterns.

The necessary data about the transitions that the different artifacts went through was gathered by the author of the thesis through a detailed log of the actions performed during the evolution process. The actual artifacts themselves were recorded and stored in a Git repository, making the initial and the evolved versions accessible, as well as all intermediate steps. The combined use of a Git repository and a detailed log of all changes performed while evolving the software made it possible not only to keep track of which components that had changed and in what way, but also why. This ensured that the rationale behind each design decision was not lost. By recording all data in this way, it was also possible to analyze and interpret it continuously throughout the thesis project, and finally summarize the knowledge gained during the project in the form of two context-free and generally applicable change patterns.

1.6 Contributions

The main contribution of this master's thesis consists of the two identified change patterns that are presented in chapter 5. One of these patterns was also applied to the Cryptify Call system during the case study, both on an architectural level and implemented by the author of this thesis in the actual code. This implementation later acted as the foundation to the code for supporting conference calls that has now been put into production by the company, and is about to be released to the end users of the Cryptify Call system. The second identified change pattern could in theory also been have implemented in the Cryptify Call system, but the performance restrictions in many of the smartphone devices used by the end users of the Cryptify Call system today meant that this solution would not work very well in practice. Therefore, in combination with the thesis' time constraints, a decision was made by all the involved stakeholders not to implement this solution. This decision will be discussed in further detail in sections 4.1 and 6.1. The change pattern that was not implemented is still included in this thesis for the reason that the pattern is still applicable to other systems with less strict performance requirements.

1.7 Threats to Validity

In this section, some threats to the validity of the findings from this study will be discussed. These threats have been classified according to the scheme used by Runeson and Höst [42], namely by sorting them into the following categories; construct validity, internal validity, external validity and reliability. Construct validity refers to the validity of the research questions and methodology used in the study, internal validity covers concerns about causal relations in the study (in other words, what is the cause and what is the effect of various observations), external validity specifies whether the findings of the study can be generalized or not, and the reliability of the study takes into account how much the individual researcher(s) behind the study have influenced its outcome [42].

One thing that is important to keep in mind is that neither of the two identified change patterns can ever be used in order to improve the security of the system to which they are applied. Instead, both of the change patterns only strive to guarantee that no new security risks of any kind will be introduced to the system when it is evolved according to that change pattern. This means that it will not be possible to detect or remove security risks that already exist in the system before evolution by using any of these patterns. That both of the two identified change patterns guarantee that no new security risks will be introduced during the evolution process, at least when applied to the Cryptify Call system, has been verified through a detailed threat model analysis carried out by the author in collaboration with domain experts at Cryptify AB (see section 1.7.1). However, this does not necessarily mean that the soundness of the change patterns can be guaranteed when they are

applied to other systems, as will be discussed in section 1.7.3 below.

1.7.1 Construct Validity

Since the author of this thesis only had limited control over the environment in which the study was conducted, it would have been very problematic to conduct the study as a traditional experiment with the aim of proving or rejecting a scientific hypothesis, for example by singling out and investigating just a single variable. Instead, a decision was made to follow the action research methodology, in which a problem is studied in its natural real-world environment in all its complexity [15]. This type of research is well-suited in many cases when the goal is to draw general conclusions from a concrete real-world scenario, as it in a real-life setting might be so many factors and variables affecting each other that singling out just a single one of them would become more or less meaningless [15]. However, it is important to point out that all action research is subject to some inherent characteristics. According to Checkland and Holwell, an action research study can never claim an as strong truth claim as for example a controlled experiment [11]. Nevertheless, some claim of truth can be made, as long as the study is properly structured, and it is usually a claim that is stronger than a mere plausibility, according to the same authors [11]. Another characteristic of action research, that actually strengthens the construct validity of this thesis, is the fact that each iteration of the process visualized in figure 1.1 provides the opportunity of trying out the results that have been achieved during the previous iterations in practice [15].

Instead of proving or rejecting a scientific theory, an action research study aims at identifying more general conclusions or themes of knowledge [11]. Therefore this study and its research questions were designed with the aim of achieving results that are as generally applicable as possible, while at the same time making sure that the intermediate problem of adding secure multiparty support to the Cryptify Call system was solved during the process. That the author's university supervisor is a leading expert within the field of change pattern research strengthens the claim that the thesis is properly constructed and that the two resulting change patterns are sound.

When it comes to the validity of the two change patterns themselves, they have both been shown to fulfill the goal of adding support for secure multiparty communication without introducing any new security risks of any kind, at least for the Cryptify Call system. This was shown by a threat model analysis [30] [1] that was conducted on both of the identified architectural solutions by the author of the thesis in collaboration with several experts in software security at Cryptify AB. The actual change patterns themselves have also been verified by the author's university supervisor. However, there is still a possibility that these patterns are only applicable to the specific context of the Cryptify Call system, and not to any software system in general as is desired. This will discussed further in section 1.7.3.

One additional risk to the validity of the alternative change pattern (see section 5.2) is the fact that this pattern was only described theoretically and never implemented in the Cryptify Call system. This of course leads to a small risk of the architectural design being flawed in some way, but as no actual implementation is necessary in order to perform a threat model analysis, this risk should be minimal.

1.7.2 Internal Validity

A performance evaluation (see section 4.5) was carried out in order to test the performance of the implemented solution in the Cryptify Call system. However, due to the high variance in the gathered data samples, it was not possible show any statistical significance of the positive results. This means that these results could as well be the result of chance. Unfortunately, it was not possible to gather more test data due to the time constraints of the thesis project. However, this fact does not have any implications for the validity of the implemented change pattern for two reasons. First, performance is to a large extent a question of implementation-level details rather than architectural-level design decisions. More importantly however, is that neither of two change patterns described in this thesis strives to achieve the best possible performance whatsoever. Instead, both of the two described change patterns only set out to guarantee that the security level before evolution is maintained also after the evolutionary process is done.

1.7.3 External Validity

The two change patterns identified by the author of this thesis are both shown to be applicable to at least the Cryptify Call system[1], which has been verified by both experts at Cryptify AB and by the author's university supervisor. However, this does not mean that these patterns can be guaranteed to work for just any arbitrary system. In order to do so, further studies would have to be conducted where the change patterns are applied to other systems as well.

As already stated, most action research aims at identifying general lessons by studying a practical problem in a complex real-world scenario, as opposed to proving or rejecting a scientific hypothesis by for instance studying some single variable [11]. This means that most action research studies carry low external validity, which simply is an inherent characteristic of action research that is difficult to tackle [11] [29]. However, measures were taken in order to express both of the change pattern in an as general and context-free way as possible, but a change pattern obviously still have to keep some level of concreteness in order to be of any use at all when it is to be applied. The belief of the author of this thesis is therefore that both of the change

[1] However, as already stated, only one of the two change patterns were actually implemented in practice in the Cryptify Call system. The reasons behind this decision will be discussed further in sections 4.1 and 6.1

patterns described in chapter 5 should be applicable to at least other systems with a similar software architecture and a similar trust model to the Cryptify Call system. Regardless of how generally applicable the results of this thesis will prove to be, the findings should at least be able to use as part of the foundations for future research within the field of change patterns or secure software evolution in general.

1.7.4 Reliability

The greatest threat to the reliability of this thesis is the fact that the results are based on data that to a large extent is qualitative and interpretive. This means that some other researcher or research team that carried out an identical study could have drawn different conclusions and achieved slightly different results than those presented in this thesis. However, this threat is to some extent mitigated by the collaborative aspects of action research [6]. This is because all design choices and other important decisions were taken by the author, the university supervisor and the managers and domain experts at the company in collaboration. In the same way were all results of the thesis continuously reviewed and confirmed by all of these stakeholders. This means that all actions performed during the thesis project are well-grounded in both theory and in practical domain knowledge, and should therefore strengthen the reliability of this study [6].

Another reliability threat is the bias introduced by the fact that the author has acted as both researcher and participant of the study, but this dual researcher/participant role is of course inherent in all action research, and is also to some extent mitigated by the collaborative aspects of action research described above [6].

Finally it should be said that the managers at Cryptify could sometimes have conflicting goals to those of the supervisor and the examiner at the university. In order to cope with this problem, a senior programmer/team leader at Cryptify was appointed to act as a dedicated company supervisor for the thesis project, which greatly helped when deciding how to best meet the goals of the different stakeholders.

1.8 Structure of the Thesis

The rest of this thesis is structured in the following way. Chapter 2 gives the background and theoretical framework of the thesis by summarizing the current state of research in the field of software evolution, as well as introducing a couple of important concepts. Chapter 3 gives a description and an architectural model of how the Cryptify Call system looked before evolution, and chapter 4 describes how the system had to change in order to conform to the new requirements. The findings of the study are summarized in the form of a change pattern in chapter 5. The thesis ends with a discussion on these findings in chapter 6.

2

Related Work and Foundations

This chapter covers the state of current research in the field of software evolution, including the sub-fields of secure software evolution (section 2.2) and model-driven software evolution (section 2.3). Section 2.4 briefly introduces a couple of important concepts in the scope of this specific study, namely SI* notation (section 2.4.1), change patterns (section 2.4.2) and identity-based public key cryptography (section 2.4.3).

2.1 Software Evolution

Since the famous laws of evolution was first formulated by Manny Lehman back in 1980 [31], quite a lot of research has been done in the field of software evolution. Nowadays, software evolution is considered crucial in preventing software aging, often referred to as software decay, a phenomenon that is (usually) caused by changes in the software's environment [34]. A foundation for a theoretical framework for helping software architects planning for software evolution has been proposed by Garlan et al. [23] and further elaborated by Barnes [3]. Yet many questions still remain unanswered, and very little has been written about concrete tools and techniques to use during software evolution [9]. Much of the research that do exist on software evolution focuses on topics such as how to prepare an architecture for evolution, management-related issues, reverse engineering and modelling techniques [34] [9]. Unfortunately, a majority of these studies are case studies, which means that their findings might not be as generally applicable as desired [9].

In a paper from 2009, Ernst, Mylopoulos and Wang [21] state that requirements evolution is a field which will require much research in the near future. The root cause of change is still an area open to research, but it can generally be assumed that changing requirements - either due to new business needs or to changes in the system environment - is the main motivation for most software evolution [21]. But even if most evolution is caused by changing requirements, most existing research instead focuses on other aspects such as design and code [21][38]. For example, Bass, Clements and Kazman [7] have come up with a set of design principles on how to, already from the design phase, make a software architecture as easy to evolve as

possible. Some other key research topics that are identified in the study by Ernst et al. are inventing tools for evolving requirements and implementing requiremental changes, as well as identifying what characterizes high variability in a design – in other words; a design that is well suited for evolution [21].

In a paper from 2005, Mens et al. summarize a set of research challenges concerning software evolution [33]. In the paper, they state that there is a huge need for finding tools and techniques that support co-evolution between different types or representations of software artifacts. Other research challenges noted in the same paper are the need for tools and techniques to preserve or improve software quality over time as well as a theoretical model of software evolution. Some advances have been made since then, for example by Ernst, Borgida and Mylopoulos [20]. In their paper, they investigate the relationship between requirements and evolution, but at the same time point out that the co-evolution challenge mentioned by Mens et al. [33] is yet to be solved. This thesis aims to contribute with a piece of knowledge, in the form of two change patterns, that will help in partly solving this co-evolution problem.

Much current research also focuses on making software systems self-adaptive by planning for future changes already in the system's design phase [21], an area which is outside of the scope of this master's thesis.

2.2 Secure Software Evolution

Security requirements engineering is widely recognized as an important aspect of software engineering, and its importance has come to grow during later years based on the idea that security issues need to be addressed early during the design process [22]. As a result, a variety of tools and techniques to help managing security requirements have been developed during the last 10-15 years [38]. However, only a few of these take software evolution into account [38]. To address this problem, Nhlabatsi, Nuseibeh and Yu [38] have suggested that the field of security requirements engineering should be combined with the field of software evolution, resulting in what they choose to call security requirements engineering for evolving systems. Regardless of the lack of research in this field, it is an important aspect of software evolution since security vulnerabilities in a system can arise both from changes in the external environment and from internal evolutionary changes [22]. Therefore the goals of secure software evolution research can be summarized as finding out how to maintain security properties during software evolution and how software evolution impacts security requirements.

One tool that actually puts security requirements engineering into a evolutionary context is SeCMER, a tool for managing evolving requirements and automatically detecting any violations of the systems security properties, developed by Bergmann et al. [8]. Another method for detecting changes to security properties is the OpenArgue tool developed by Yu et al. [53]. OpenArgue uses the argumentation tech-

nique introduced by Haley et al. [25] in order to validate that certain security properties are fulfilled in an evolved software architecture.

The goals of this thesis correspond to the goals of secure software evolution research proposed by Nhlabatsi, Nuseibeh and Yu [38]; namely to find a way of guaranteeing that the security properties of a system are maintained during the evolution of that system. It is important to once more point out that this thesis does not aim to find a way of increasing the level of security a system has, but only preventing the system's security from weakening during evolution.

2.3 Model-Driven Evolution of Security Artifacts

Model-driven engineering holds great potential for improving software evolution processes [14]. The main idea is to work with different types of models instead of the actual code when performing both development and evolution. By raising the abstraction level in this way, many advantages can be gained; the effort needed to be spent on developing or evolving software will be reduced, the developers will get a better overview of the software and the risk of programmer errors will be reduced [14]. A problem with many current model-driven techniques, however, is that the models become obsolete very quickly, as the actual code often evolves independently of the corresponding models [21]. One way of solving this problem could be to improve the possibilities of adding application specific code to the models (as opposed to adding custom code to code generated from the models), and this is a key research topic within the field of model-driven engineering [14].

When it comes to conducting secure software evolution in a model-driven way, additional problems arise. Jürjens [27] states that security requirements are not only hard to come up with, but also difficult to enforce in the software. The security requirements also have to be re-verified, often manually, after each modification of the software [28]. This means that performing model-driven evolution in a secure context will be challenging, but possibly also very beneficial once these challenges have been solved [27].

Most existing model-driven techniques that aim to resolve these issues are based on the Unified Modeling Language (UML) [22]. Examples of such techniques include the UMLchange framework developed by J. Jürjens [46] (which in turn is based on an older framework called UMLsec [27]) and SecureUML, proposed by Basin, Doser and Lodderstedt [5]. SecureUML also uses an approach called Role-Based Access Control, which is a technique currently getting attention from researchers within the field. This technique is intended to help managing complex access control rules, but most other models (such as PRBAC [39]) does not handle evolution very well, according to Montrieux, Wermelinger and Yu [37]. Basin et al. have however later developed a tool called SecureMOVA that can be used to automatically verify the security requirements of a SecureUML model [4].

Another technique which has received researchers' attention recently is model-based argument analysis. This technique is used to evaluate the impact on security caused by evolutionary changes, for example in tools such as OpenPF [45] and OpenArgue [53].

In this thesis, modelling is mainly used in order to describe the changes in a system's trust model that arise from changing requirements. For each of the resulting change patterns that are presented in chapter 5, a model-to-model transformation is also given using the QVT Operational transformation language [17].

2.4 Thesis Foundations

This section introduces some important concepts that will be used throughout this thesis. Section 2.4.1 describes the SI* modelling language, which will later be used in order to describe the security features of the Cryptify Call system. Section 2.4.2 gives a description of what a change pattern is, including an example, and section 2.4.3 introduces the concept of identity-based public key cryptography.

2.4.1 SI* Notation

The SI* notation is a modelling language [32] that will be used throughout this thesis when describing the security requirements and other properties of the Cryptify Call system. Amongst the core concepts of SI* are the modelling of *actors* and the *goals* that these actors strive to fulfill. An actor can either be an *agent* (a human or software entity) or a *role*, which can in turn be played by some agent. It is also possible to model the *resources* that actors need in order to fulfill their goals. Furthermore, the SI* notation describes the different relationships and dependencies that might exist between these different entities, for example that a goal might be divided into subgoals and that some actor might depend on some other actor to provide a certain resource. SI* is based upon an older framework known as i* [52], but is evolved in order to better encompass security properties such as trust and permissions [32]. The graphical representations of the the different SI* entities and relationships used in this thesis are shown in figure 2.1. The different relationships are also described in the list below for reference. Figure 2.2 shows a simple example of SI* notation, taken from [32].

De – *Execution dependency*, means that one actor appoints another actor to fulfill a goal or provide a resource.

P – *Provide*, indicates that an actor provides a certain resource.

R – *Request*, indicates that an actor requests a certain resource or the fulfilment of a certain goal.

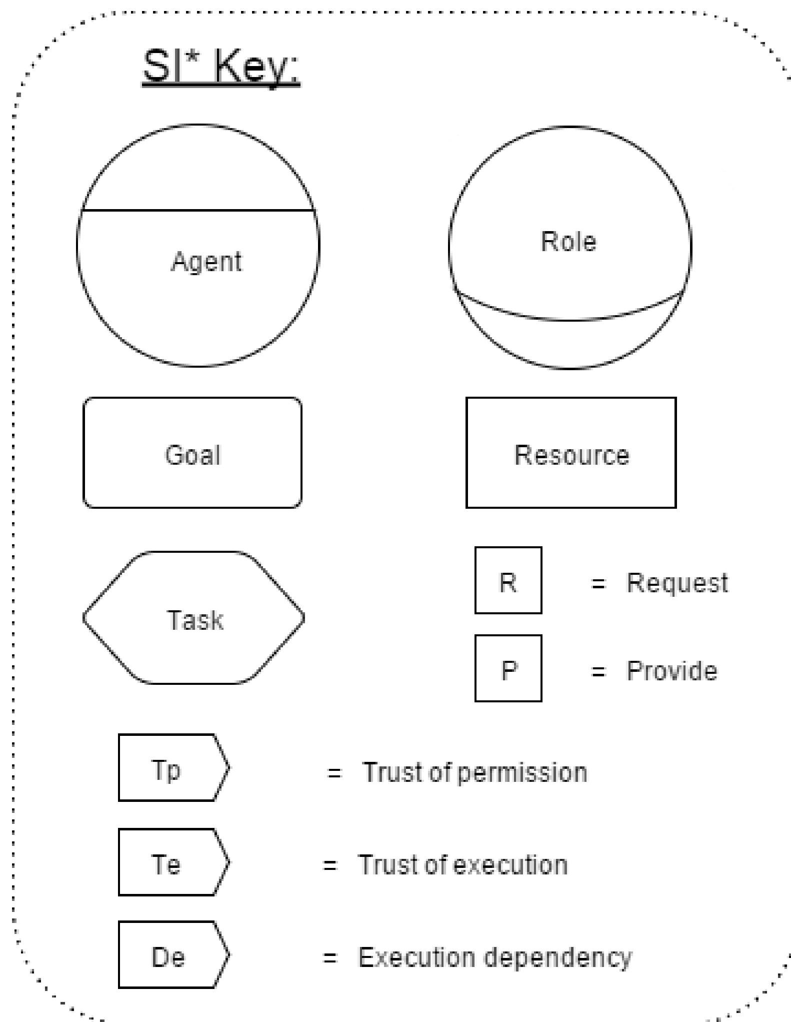


Figure 2.1: Graphical representations of the different SI* entities and relationships used in this thesis.

Te – *Trust of execution*, shows that one actor trusts another actor to fulfill a goal or provide a resource. Trust of execution is therefore a stronger relationship than the execution dependency described above.

Tp – *Trust of permission*, shows that one actor trusts another actor not to misuse a goal or a resource.

2.4.2 Change Patterns

An area that has many times been pointed out as an area in need of research is how to properly handle co-evolution between different artifacts (such as requirements, architecture, code et cetera). One solution to this problem is to use change patterns, a concept introduced by Yskout, Scandariato and Joosen [51]. A change pattern always covers the evolution of two closely intertwined artifacts, and gives a principled

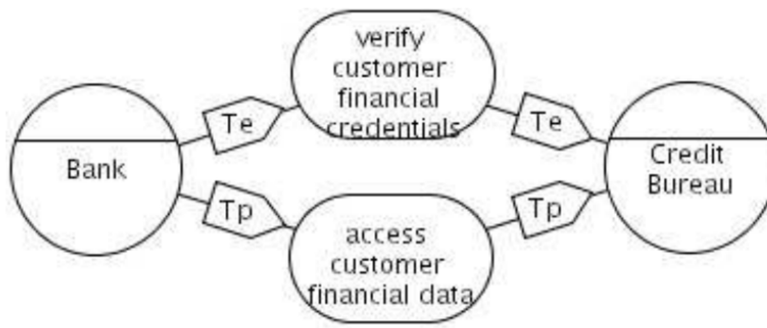


Figure 2.2: Example of SI* notation (taken from [32]). The Bank agent in this scenario trusts the Credit Bureau to fulfill the goal of verifying a customer's financial credentials (shown by the *Te* relationship in the model). The Bank also trusts the Credit Bureau not to misuse the customer's financial data in any way (shown by the *Tp* relationship).

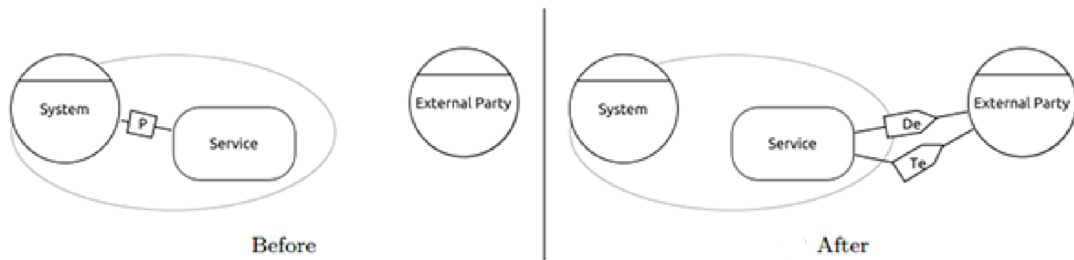
way of evolving one of the artifacts based on the changes made to the other artifact. A change pattern usually consists of a change scenario that describes the changes to one of the artifacts, and some kind of transformation on how to change the other artifact to cope with these changes.

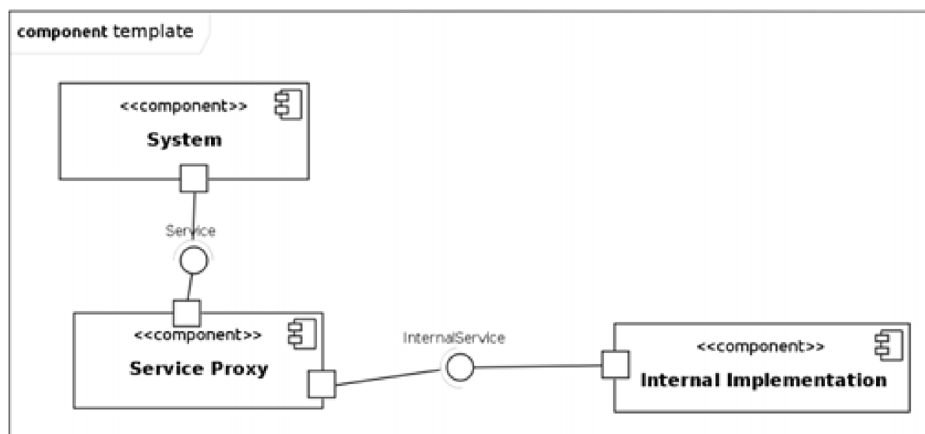
An example change pattern, taken from [50], is shown in figure 2.3. This change pattern covers a change scenario (shown in figure 2.3a) where some service used by the system is instead to be delegated to a trusted external party. The architectural support template shown in figure 2.3b shows that a new proxy component should be introduced in the architecture, while the change guidance shown in figure 2.3c shows the transformation necessary to make the architecture comply with the after situation in the given change scenario – namely to let the service proxy use the external service instead of the internal implementation.

2.4.3 Identity-Based Public Key Cryptography

In order to enable secure communication between end users, the Cryptify Call system uses a type of cryptography known as identity-based public key cryptography for key exchange and authentication. Instead of having a traditional public key infrastructure in which a certification authority issues and verifies certificates for each user, identity-based public key cryptography works by letting a user's public key (i.e., identity) be derivable from some publicly available property, such as that user's email address or phone number. A trusted third party (in this case a Key Management Server is used) then generates and distributes private keys corresponding to each identity [49] [41]. Identity-based cryptography was first proposed by Adi Shamir in 1984 [43].

The most striking advantage of identity-based cryptography over a traditional public key infrastructure is obviously that there is no need for issuing, managing and





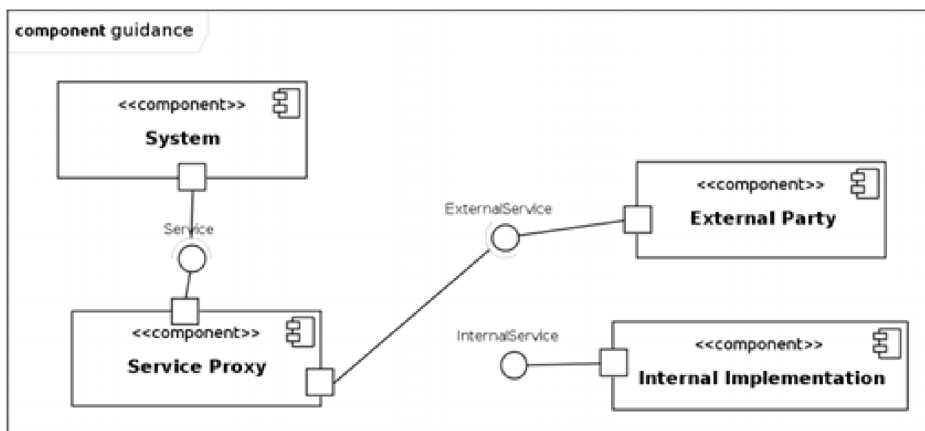


Figure 2.3: Example change pattern (taken from [50]). The change pattern consists of a change scenario, an architectural support template and a change guidance on how to transform the software architecture to comply with the changes caused by the change scenario.

validating digital certificates. This results in a simpler software architecture, better performance and increased user-friendliness when compared to a system using a public key infrastructure [49] [41].

One of the potential downsides with identity-based public key cryptography is that all the keys are stored in a single location, namely the Key Management Server. This is known as the key escrow problem [40]. Key escrow might be an advantage as well as a disadvantage under certain circumstances, but the problem is that if this location gets compromised, the attacker can decrypt all communication encrypted using the compromised keys, including past communication [49]. In the Cryptify Call system, this risk is mitigated by having the Key Management Server operate entirely offline.

3

The Existing Cryptify Call System

This chapter contains an architectural description as well as the most significant security requirements of the Cryptify Call system. The chapter begins with a high-level description of the system in section 3.1, followed by a description of the security requirements and how these are fulfilled in section 3.2. Finally the roles and internal architecture of the different components the system consists of are explained in sections 3.3 through 3.5. For the sake of clarity, focus lays on architectural details that are related to fulfilling the security requirements.

The requirements and the high-level description of the system were elicited by interviewing several employees at Cryptify. The low-level internal software architecture of the client application (see section 3.5) was elicited through reverse engineering. A first draft of this architectural model was created with the help of the Doxygen tool [16]. The model was then completed by manually stepping through the code, following all different conceivable use cases, and finally validated by once more interviewing the software developers with deeper insight into the architecture. All these steps except the final validition was carried out by the author of this thesis alone.

3.1 High-Level Description

The system that is investigated in this case study is known as Cryptify Call. The system is used in order to encrypt two-party voice and text messaging communication between end users. This system is to be evolved into also supporting many-to-many calls during the course of this thesis. The end user application is known as the Cryptify Caller Application (CCA) and can be run on iOS or Android smartphones. Other than this, two more components exist in the system; the Cryptify Rendezvous Server (CRS) and the Cryptify Management System (CMS). An overview of how these components are related to each other is shown in figure 3.1. Each of these components will be described in more depth in the sections 3.3–3.5.

The CMS is the management system which is used to generate crypto credentials for the end users, which are then distributed as printed QR codes that are scanned with the end user's smartphone. A single CMS and the end users using crypto credentials generated by that CMS together forms what is known as a security domain. The

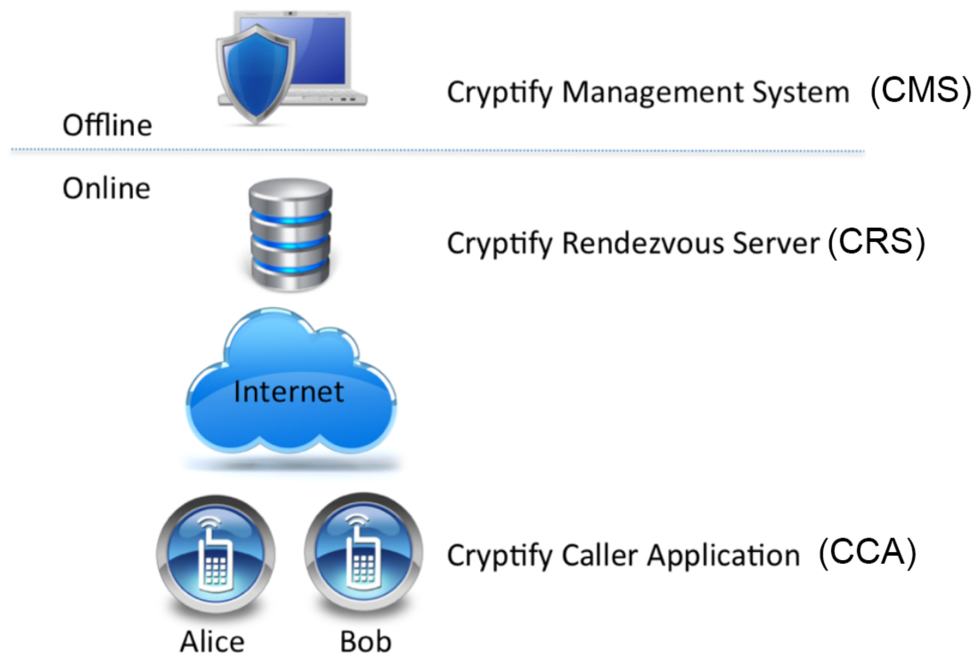


Figure 3.1: Overview of a single security domain within in the Cryptify Call system. In this example, Alice and Bob connects to the CRS and uses the cryptographic keys generated by the CMS.

CMS is run as a desktop application on an offline computer, in order to reduce the risk of it getting compromised.

The CRS is the system's central node whose main role is to transmit communication between end devices. The CRS is considered a non-critical node from a security perspective, as it merely acts as a relay of already protected information and never exposes any unencrypted data. It might therefore be deployed as any standard Internet server or on some cloud-based platform.

The Cryptify Call system is sold as an enterprise product and not target directly towards end users. The implications of this is that every organization that wishes to use Cryptify Call is expected to set up and manage their own security domain, i.e. deploy their own CMS (note that a single CRS can in theory be shared by different security domains, as the CRS does not contain any security-critical data).

3.2 Security Requirements

Apart from the purely technical requirements of providing the functionality of voice and text communication between end users, there are some security requirements that need to be fulfilled in order for the end users to be able to use and trust the system.

Out of these requirements, which are derived from a specifications document issued by CESG[1] entitled Secure VoIP Client version 2.0 [10], there are two requirements that stand out as the by far most important security characteristics of the system. These are to be able to *authenticate* system users and to guarantee *confidentiality* of their intercommunication. Authentication is necessary in order for a client to know that the other party of a call really is who he or she claims to be, and confidentiality is necessary in order to guarantee that no external party can eavesdrop on a private conversation. In short, the authentication requirement is fulfilled by letting one of the parties of a call create a digital signature which is then validated by the other party. Confidentiality is achieved by exchanging a common Session Key between the parties, which is then used in order to protect the contents of their conversation.

3.2.1 The MIKEY-SAKKE Protocol

In order to fulfill these security requirements, a cryptographic technique known as identity-based cryptography is used. This technique combines aspects of both asymmetric and symmetric cryptography. The authentication requirement and the key exchange between clients (which in turn is necessary to later achieve the confidentiality requirement) are both fulfilled with an asymmetric cryptographic method called MIKEY-SAKKE. Once a call has been initiated and keys exchanged with the help of the MIKEY-SAKKE protocol, a symmetric key algorithm is used to protect the contents of the call.

The MIKEY-SAKKE protocol actually consists of several different techniques, which are combined in order to provide secure end-to-end communication between end users; the SAKKE algorithm is used to exchange a common Session Key (the key later used to ensure confidentiality of the call) between the participants, the ECCSI algorithm is used for authentication, and the communications protocol used between the clients is known as MIKEY. The MIKEY-SAKKE protocol is described in detail in RFC 6509 [24], but a brief summary is given below and the protocol will be described further in the following sections.

A group of clients that are able to make secure calls to each other forms a so called security domain, as already stated. Each client in the security domain is identified by its unique public identity, which simply consists of a normal phone number appended with the current year and month (meaning that these identities change every month). For each security domain there also exists a Key Management Server (KMS), which is responsible for generating all the keys that are necessary to fulfill the system's security requirements. The KMS is therefore also known as the security domain's root of trust. The KMS generates a set of unique personal keys for each client, which are used for signing and decrypting messages. The KMS also generates a single public key that is used for authenticating signatures and another public key that is used when encrypting and decrypting messages. In the case of the Cryptify Call system, the CMS – described in more detail in section 3.3 – assumes

[1]The UK national technical authority for information assurance.

the KMS role.

The following two keys are the public keys generated by the KMS and distributed to all of the clients in the security domain:

- **KMS Public Key** – used for encrypting and decrypting SAKKE messages.
- **KMS Public Authentication Key (KPAK)** – used by clients when authenticating the signature sent by some other client.

In addition to this, the following set of keys (hereafter also referred to as a client's personal keys) are generated for all of the clients in the security domain (so that each client receives a unique set of keys):

- **Receiver Secret Key** – used by the client together with the KMS Public Key to decrypt SAKKE messages it receives.
- **Secret Signing Key** – used for signing messages with a digital signature.
- **Public Validation Token** – sent by the Caller of a call as part of the MIKEY-SAKKE message and used by the receiving party to verify its counterparty's digital signature.

These keys come into use when some client, called the Caller, wants to set up a secure phone call to another client, referred to as the Callee. The KPAK is used together with the Caller's Secret Signing Key and Public Validation Token for authentication, while the KMS Public Key and the Receiver Secret Key are used together with the Callee's public identity in order to exchange a common Session Key between the two clients. This Session Key will be used during the actual call in order to provide confidentiality to the conversation. In other words, the secure call is set up using an asymmetric key algorithm, while the actual voice data is protected using a symmetric key algorithm.

3.2.1.1 Monthly Key Renewal

As the clients' identities are changed every month (due to the fact that the current year and month is part of the identity), it means that the personal keys described above have to change as well. Normally, the KMS Public Key and the KMS Public Authentication Key remain unchanged, while the other keys are re-generated by the KMS each month and then distributed to the clients.

A client gets its first set of personal keys by scanning a QR code distributed to the user by the CMS manager. When the keys need to be renewed, however, the new keys are distributed wirelessly and invisible to the end users. In short, this process

is implemented in the Cryptify Call system by letting the CMS generate the new keys, which are then burned to a DVD. The CRS then reads the data from the DVD and wirelessly distributes the proper set of keys to each respective client with the help of AES encryption using a pre-shared key. The pre-shared key (referred to as the Update Key) is originally distributed to the clients via the same QR code that contains their personal keys, and then changed each month by including the next month's Update Key in each update message. This whole process will be described in more detail in sections 3.3 and 3.4.

3.2.1.2 Advantages and Disadvantages

An identity-based cryptography scheme such as MIKEY-SAKKE obviously comes with its own benefits and drawbacks when compared to other cryptographic methods. The main benefit of identity-based cryptography is that the cumbersome mechanism of individual certificates in certificate-based encryption is avoided. The Caller doesn't need to know anything about the Callee except its public identity – which as previously stated consists only of the Callee's phone number and the current year and month – in order to compose a MIKEY-SAKKE message to him or her. This indirectly leads to another advantage, namely that there is no need for the Callee to be online or even created yet in order to create and encrypt a MIKEY-SAKKE to him or her. This in turn means that it is possible to send for example text messages to a client that has never been online, store these messages (still in encrypted form) somewhere along the way and deliver them as soon as the receiving client comes online.

The main drawback of identity-based encryption is the vulnerability of the KMS, since it contains all the cryptographic keys used within the security domain. If an attacker would gain access to the KMS, it would not only give the attacker the opportunity to pose as any client in that security domain, it would also mean that the attacker could decrypt any message sent within the security domain that is has been able to intercept. In order to minimize this threat, the Cryptify Management Server (CMS) which acts as the KMS in the Cryptify Call system always operates completely offline.

Another drawback with identity-based encryption is that once an identity has been issued by the KMS, it cannot be revoked. The impact of this threat is limited in the MIKEY-SAKKE method by including the time component (current year and month) in each identity. To remain valid, the set of personal keys associated with an identity has to be renewed by the KMS once per month, meaning that a lost or compromised identity will only be valid for the remainder of the month it was compromised. However, another implication of this is that since the user's phone number is used as identity, it will not be possible for a user that loses his or her cryptographic identity to get a new one using the same phone number until the following month.

3.2.2 Authentication

When a client, called the Caller, wants to set up a secure phone call to another client, referred to as the Callee, the following procedure takes place. First of all, the Caller must generate a random Session Key to use for this particular session. The Session Key is then encrypted in a way that only the Callee can decrypt by using the Sakai-Kasahara Key Encryption (SAKKE) algorithm, which utilizes the KMS Public Key and the Callee's public identity. This public identity consists of the Callee's normal phone number appended with the current year and month, and is equivalent to the public key in a traditional asymmetric cryptographic algorithm. The Caller then signs the SAKKE message with the use of the Elliptic Curve-Based Certificateless Signatures (ECCSI) algorithm and his or her own Secret Signing Key. The SAKKE message, the ECCSI signature and the Caller's Public Validation Token are thereafter encoded as a message in the Multimedia Internet Keying format (MIKEY) and sent to the Callee.

Once the MIKEY message has been received, the Callee begins by validating the authenticity of the Caller's ECCSI signature, using the KMS Public Authentication Key (KPAK) provided by the CMS and the Caller's Public Validation Token. If this check passes, the Callee can proceed with decrypting the Session Key encapsulated within the SAKKE message, using his or her Receiver Secret Key and the KMS Public Key. The authentication requirement is now fulfilled, and both of the parties now also have access to the same Session Key, which will be used to encrypt further communication (see section 3.2.3) and thus fulfill the confidentiality requirement as well. Note that the Callee does not need to authenticate itself to the Caller, as the Caller knows that the holder of the Receiver Secret Key corresponding to the intended Callee's public identity will be the only one able to decrypt the SAKKE message.

The authentication requirement has been captured in figures 3.2 and 3.3 using the SI* notation, which is a modelling language used to model security requirements in a precise way [32]. Figure 3.2 covers the goal of the client initiating the call, which is to establish a Session Key and share it with the other party of the call. Figure 3.3 complements the first figure by capturing the goal of the Callee of the call, which is to authenticate the Caller's identity and get access to the Session Key. In order for this whole process to work, both parties must put trust in that the crypto keys provided by the offline CMS are valid (see figure 3.5).

Implementation-wise, the process of creating and validating the digital signatures is implemented within the crypto module of the client application (see section 3.5.1.2).

3.2.3 Confidentiality

Once the Caller has been authenticated by the Callee and the session has been initiated, further communication (including the actual media payload) is encrypted

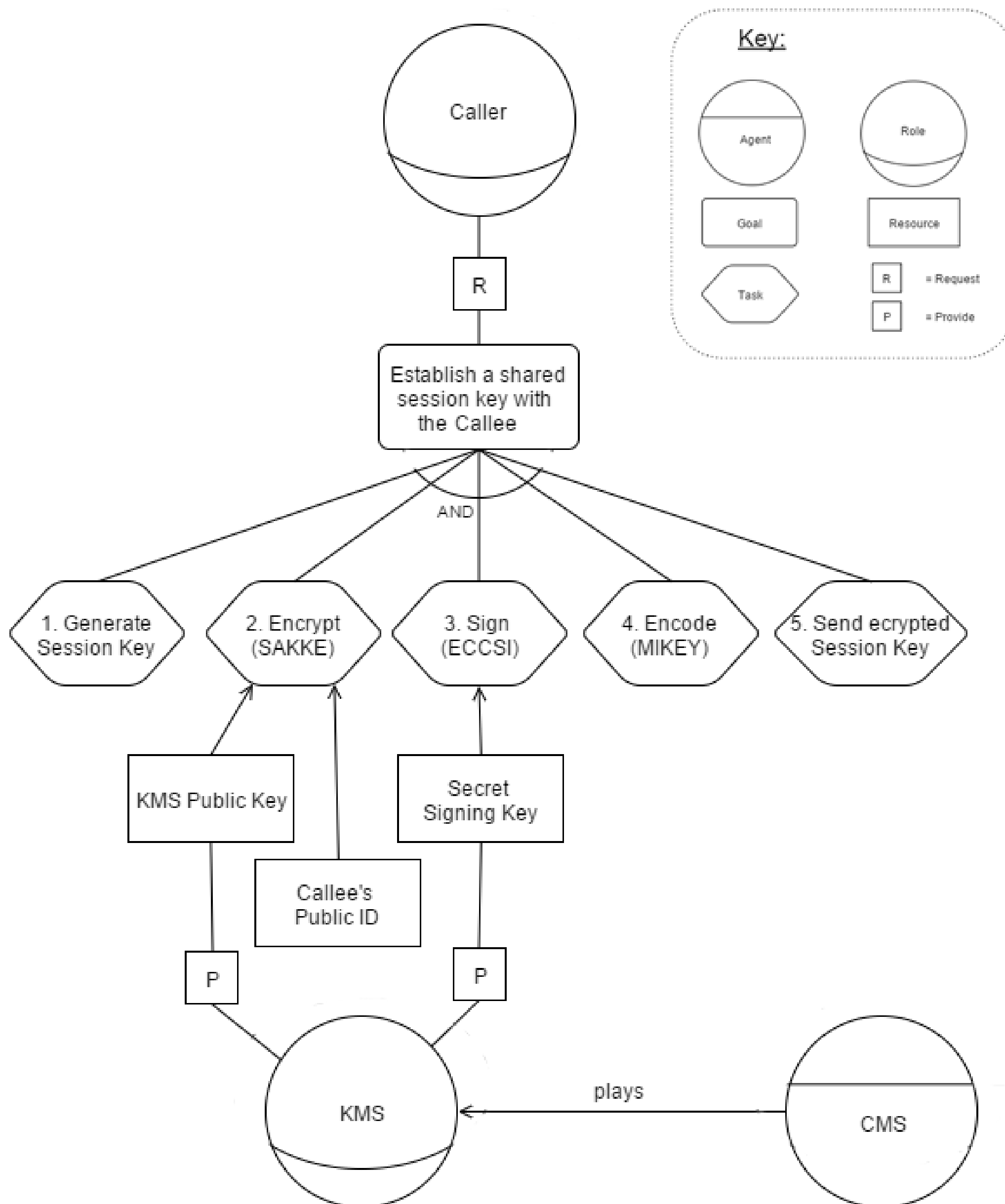


Figure 3.2: The goal of the Caller of a call is to establish a shared and secret Session Key with the other party. This Session Key is encrypted using the SAKKE algorithm and signed using the ECCSI algorithm. The encrypted message and the ECCSI signature is then encoded in the MIKEY format together with the Caller's Public Validation Token and sent to the Callee. The Callee is now ready to authenticate the identity of the Caller, so that the encrypted message can finally be decrypted and give the Callee access to the secret Session Key.

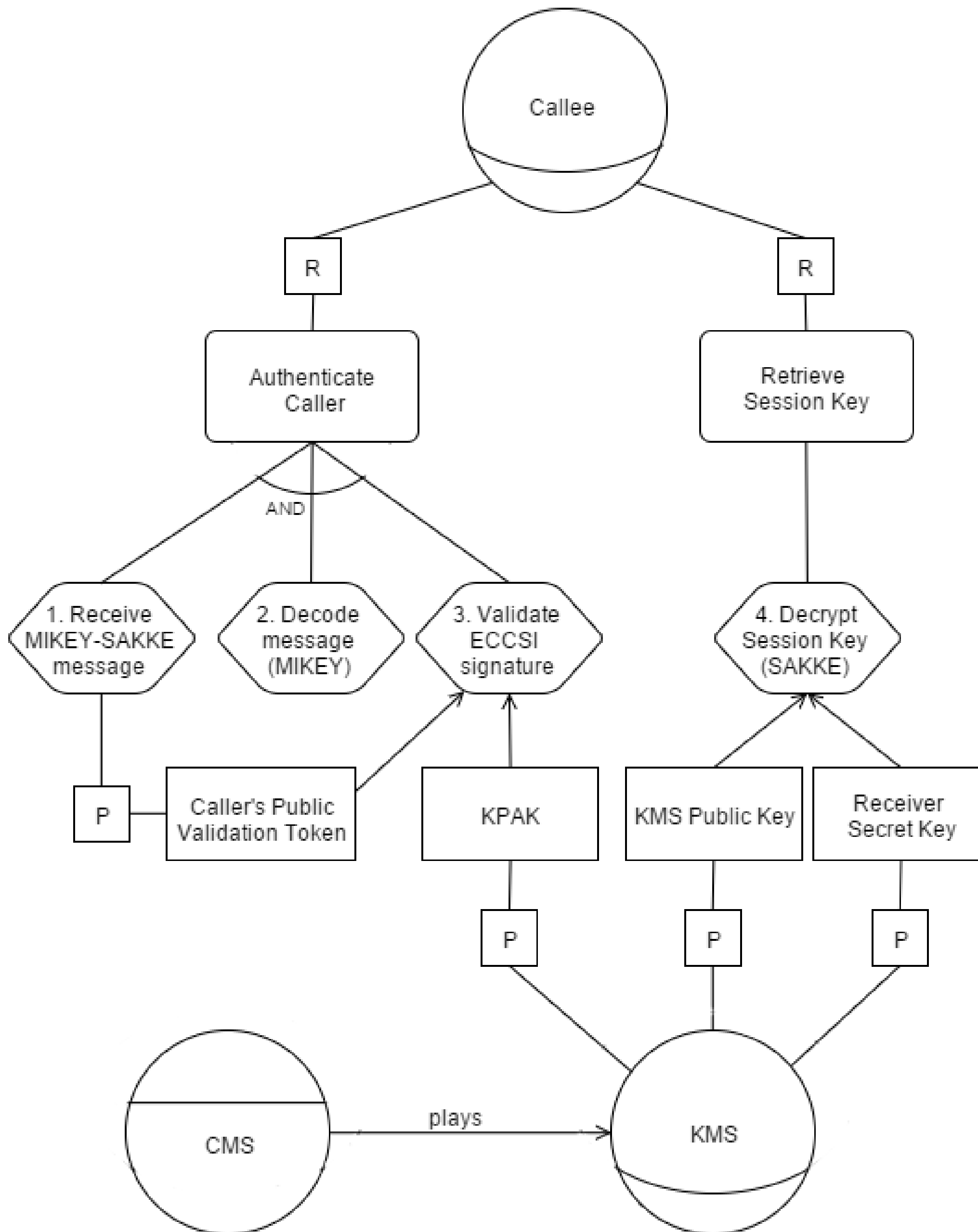


Figure 3.3: The goal of the Callee of a call is to authenticate the other party and get access to the secret Session Key that will be used to encrypt further communication between the two parties. The Callee begins by decoding the MIKEY message in order to get access to the encrypted Session Key as well as the Caller's Public Validation Token and ECCSI signature. Thereafter, the Callee can validate the authenticity of the ECCSI signature and finally decrypt the SAKKE message containing the Session Key. Both parties now have access to the same Session Key, which will be used in order to provide confidentiality to their subsequent communication.

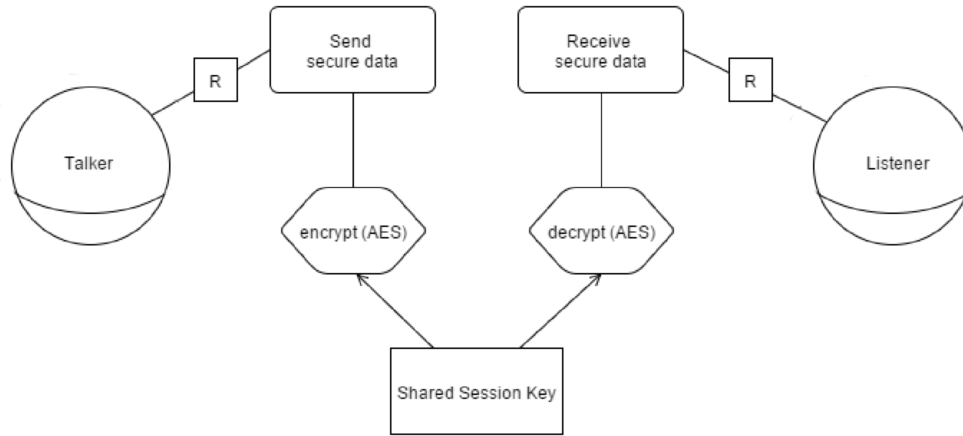


Figure 3.4: The media data sent between two clients during a secure call must be protected end-to-end in order to guarantee confidentiality of the contents of their conversation. The shared Session Key is used together with the Advanced Encryption Standard (AES) algorithm in order to achieve this.

using the Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM), which is a symmetric-key algorithm; meaning that the same key is used for both encryption and decryption at both ends. This is done in order to ensure end-to-end confidentiality of the contents of the conversation, as well as to provide data integrity by protecting against tampering with the media data. In this case, the key being used is the Session Key that was exchanged during the authentication process described in the previous section. Since anyone wiretapping the conversation will not have access to the Session Key, the requirement of confidentiality is fulfilled. The confidentiality requirement is captured using SI* notation in figure 3.4. The actual functionality is implemented within the crypto module of the client application (see section 3.5.1.2).

Note that the data only needs to be protected until it reaches the receiving client's phone in order for the confidentiality requirement to be considered fulfilled. After this point, it is impossible to control what the end user chooses to do with the information from a technological perspective, and therefore considered to be out of scope of the security requirements.

3.2.4 Trust Model

To recapitulate, the requirement of authentication is fulfilled by having an offline KMS acting as the root of trust for each security domain. Based on this trust, a client can authenticate the digital signatures sent to it by other clients. Once a client has authenticated its counterparty, the process of ensuring confidentiality is carried out between the two clients alone. Apart from trusting the KMS for authenticating other clients, the clients also trust each other for keeping their respective secret keys (private crypto keys as well as the random Session Keys used for each call session) private. These trust relationships have been modelled in figure 3.5 using

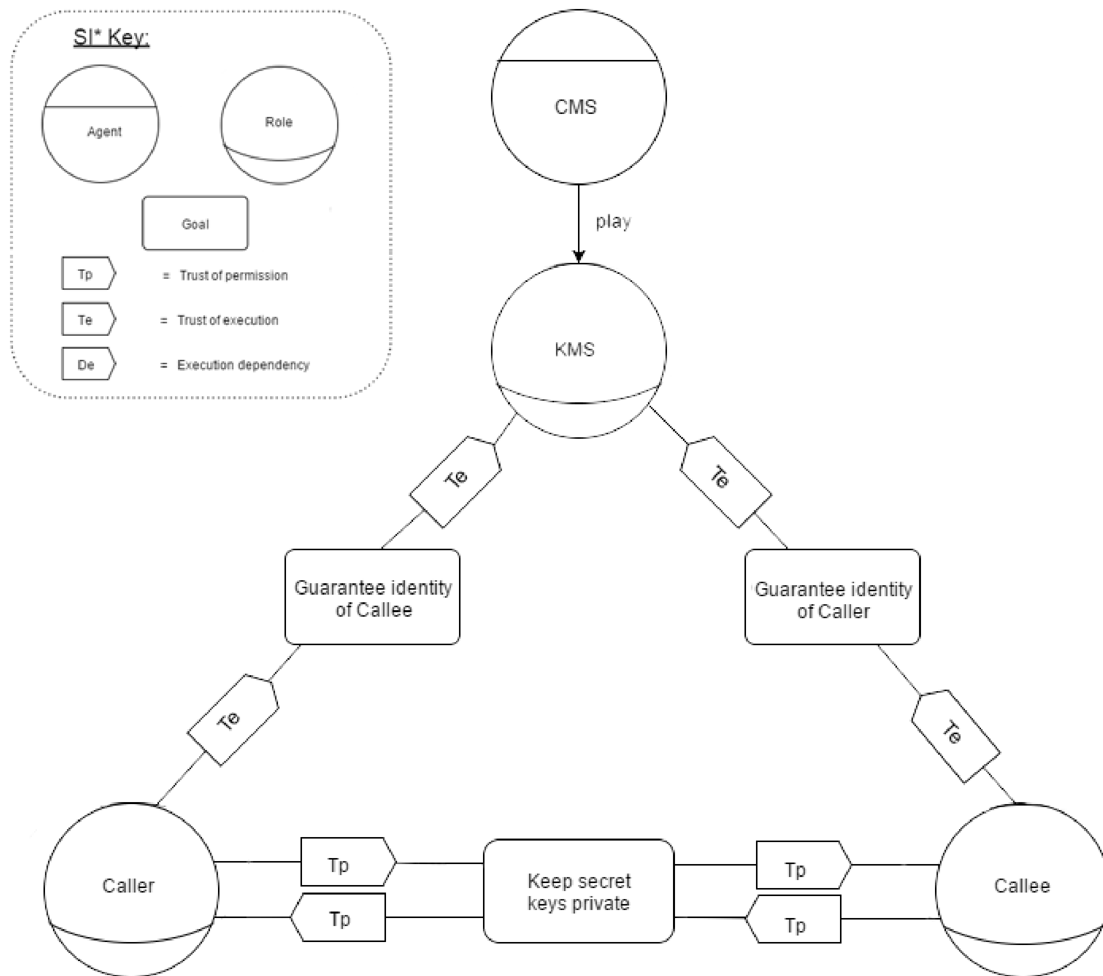


Figure 3.5: Trust model of the Cryptify Call system. The clients trust that the keys generated by the CMS are valid in order to authenticate the identity of other clients. In addition to this, the clients also trust each other not to leak any of their secret MIKEY-SAKKE or Session Keys.

SI* notation.

3.3 Cryptify Management System (CMS)

The Cryptify Management System (CMS) acts as the system's Key Management Server (KMS) and desktop administration tool. From here, user management is handled (adding and deleting users to and from the security domain). Due to the inherent key escrow property of identity-based encryption (see section 1.7), the CMS must store all keys being used within the security domain. This means that if the CMS should get compromised, then all past and present communication secured using the keys generated by that CMS, as well as the identities of all the users in the security domain, will also be compromised. This must of course be prevented at all costs, so in order to mitigate this risk, the CMS operates entirely on an offline computer. This means that the CMS will be completely isolated from Internet threats. Output data is instead printed in the form of QR codes or burned in encrypted form onto a blank CD or DVD.

When the CMS manager adds a new user to the security domain, the CMS generates a set of personal keys for that user (see section 3.2.1). To provide the user with his or her set of personal keys the CMS prints them together with the KMS Public Key and the KMS Public Authentication Key in the form of a QR code. Contained in this QR code is also the Update Key that will be used for the next month's key renewal and a pre-shared key that the client will use to communicate securely with the CRS. After the QR code has been scanned by a client application, the client has access to all the necessary keys it needs and is ready to make secure calls. Since a client's set of personal keys have to be updated every month (due to the fact that its identity also change every month), deleting a user from the security domain is handled by preventing the CMS from generating new keys for that user. The user will be able to use the system for the remainder of the current month, but will not receive any new set of personal keys the following month, and thus will be unable to communicate securely from that on.

When it is time for the monthly renewal of the clients' personal keys, the CMS first generates a new set of personal keys for each client. This update data is then burned in encrypted form, using the AES algorithm with a pre-shared secret (manually configured on the CMS as well as the CRS), onto a CD or DVD. The CD/DVD is then read and decrypted by the CRS, which in turn distributes the renewed keys wirelessly to each client, using the AES algorithm with a pre-shared Update Key. The Update Key, as already mentioned, is originally distributed to the clients with the QR code that also contains their MIKEY-SAKKE keys, and in each monthly update a new Update Key is included to be used for next month's update.

3.4 Cryptify Rendezvous Server (CRS)

The Cryptify Rendezvous Server (CRS) is the system's central node and is responsible for handling communication between the clients within a single security domain. Regardless of this, the CRS is not considered a security-critical component of the system, as it only forwards already protected data between clients and does not play any role in the actual encryption or decryption of any of this data. The most important responsibility of the CRS is to maintain a connection to all clients in the security domain that are currently online. Via these connections, the CRS can set up call sessions and deliver media data and text messages between the clients. Since the CRS does not play any important role from a security requirements point of view, it is not included in the trust model of the system (figure 3.5).

When a client wants to set up a secure call with some other client, the initiating client sends a message to the CRS, containing the identity of the client it wants to reach as part of the message's metadata header and the encrypted MIKEY-SAKKE initiation message (described in section 3.2.2) as payload. As the CRS maintains a connection to all clients currently online, it immediately forwards the MIKEY-SAKKE message to the right recipient, assuming that client is currently available. Once the call has been established, all communication continues to pass through the CRS. The only difference is that the payload now consists of AES-encrypted voice data (see section 3.2.3) instead of MIKEY-SAKKE messages, but just as before can the payload only be decrypted by the receiving client and never by the CRS itself.

All messages sent between a client and the CRS (encrypted messages bound for another client as well as other signaling messages) are encrypted using the Transport Layer Security with Pre-Shared Keys (TLS-PSK) protocol, in addition to the inner MIKEY-SAKKE or AES encryption used by the clients. This makes it difficult or impossible for an eavesdropper to analyze intercepted signaling messages and traffic patterns. The pre-shared key used by the TLS-PSK protocol was initially shared to the clients via the QR code they scanned to get access to their MIKEY-SAKKE keys (see section 3.3).

The CRS is also responsible for distributing the renewed set of personal keys to all the clients once per month. After the new keys have been generated by the CMS, they are burned in encrypted form using the AES algorithm onto a CD or DVD. The data on the CD/DVD is then read and decrypted by the CRS, using a pre-shared secret manually configured onto the CMS and the CRS. Thereafter, the CRS wirelessly distributes the right set of personal keys to each client. This distribution uses a pre-shared key, referred to as the Update Key, and 256-bit AES in Cipher Block Chaining (CBC) mode with SHA512-HMAC hashing in order to provide both encryption and message authentication. Each Update Key is only used once, so the update message also contains the key that will used to encrypt the following month's update message. Each client gets its first Update Key when scanning its QR code.

As the CRS implements none of the security functions it may be deployed as a cloud

service on the Internet. Loss or compromise of the CRS will not compromise the security of the system, but can of course impact the availability of the service. In order to increase the system's redundancy, multiple CRS servers might therefore be used in parallel as a cluster. As the CRS does not handle any sensitive data, it is also possible for several organizations (using separate security domains) to share a single CRS or CRS cluster.

3.5 Cryptify Caller Application (CCA)

The Cryptify Caller Application is the system's client application, and allows a user to establish secure calls and to send encrypted text messages to other users connected to the same security domain. Every user's normal telephone number is used to identify that user, together with the current year and month. In order to get access to the KMS Public Key, the KMS Public Authentication Key and the client's set of personal keys (Receiver Secret Key, Secret Signing Key and Public Validation Token), the user initially scans a QR code containing these keys, which is distributed by the CMS manager. The QR code also contains the pre-shared key for communicating with the CRS and the Update Key that will be used to protect the following month's update message. Thereafter, the personal keys and the Update Key are renewed once per month via the update messages distributed wirelessly by the CRS.

The client application connects to the CRS via the mobile network or WLAN. Voice data is sent between two clients using the SRTP protocol protected with 128-bit AES in Galois/Counter Mode (GCM) mode, after first having authenticated each other and exchange a common Session Key with the help of the MIKEY-SAKKE protocol. Loss or compromise of a single client does not compromise the complete system, and a compromised client can also be blacklisted and thus be prevented from establishing a connection with its CRS.

3.5.1 Low-Level Architecture

The actual evolution to be performed (described in chapter 4) during the thesis was early on determined to be localized to the internal architecture of the client application. Therefore, in the following sections, the client application is described in more detail than the architecture of the CMS and the CRS. No complete architectural description of the client application existed at the beginning of the project, so the following architectural description had to be reverse engineered from the application's source code. The resulting architectural model was then validated by the domain experts at the company.

The client application is implemented as a three-layered application (see figure 3.6). The user interacts with the application through a Graphical User Interface (GUI),

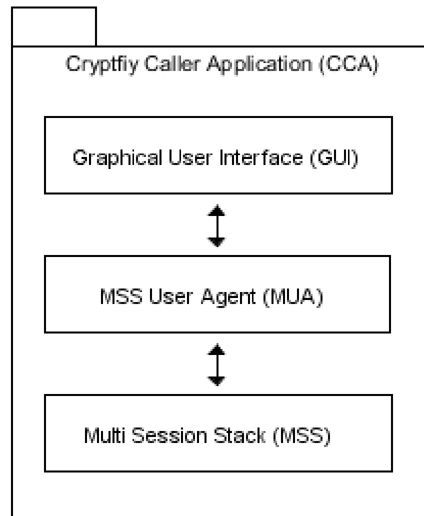


Figure 3.6: Architecture of the Cryptify Caller Application, which consists of three layers; a graphical user interface, a middle layer called the MSS User Agent and a platform-independent bottom layer called the Multi Session Stack.

the application's top layer, while the platform-independent bottom layer known as the Multi Session Stack (MSS) contains most of the actual logic. The middle layer, the MSS User Agent (MUA), acts as an interface between the other two layers, and also contains some virtual functions whose implementations differ between different platforms.

The code necessary to fulfill the security requirements from section 3.2 is implemented inside a special crypto module in the Multi Session Stack layer; the authentication requirement with methods for creating and validating digital ECCSI signatures and the confidentiality requirement with methods for encrypting and decrypting data messages.

3.5.1.1 The MSS User Agent Layer (MUA)

The middle layer of the client application is known as the MUA, which stands for MSS User Agent. The MSS User Agent acts as the interface between the GUI and the Multi Session Stack layer, the application's bottom layer. The implementation of the MSS User Agent layer for the OSX platform is visualized in figure 3.7. On other platforms, the implementation of this layer differs slightly from the architecture shown in figure 3.7. The OSX implementation of the MSS User Agent layer is implemented in the Objective C++ language.

The most important component shown in figure 3.7 is the OsxMua. This component is responsible for keeping track of an object called ActiveCall, representing the current running phone session. Through the ActiveCall object, the OsxMua makes various function calls down to the Multi Session Stack layer (described in section 3.5.1.2), such as sending and fetching audio data.

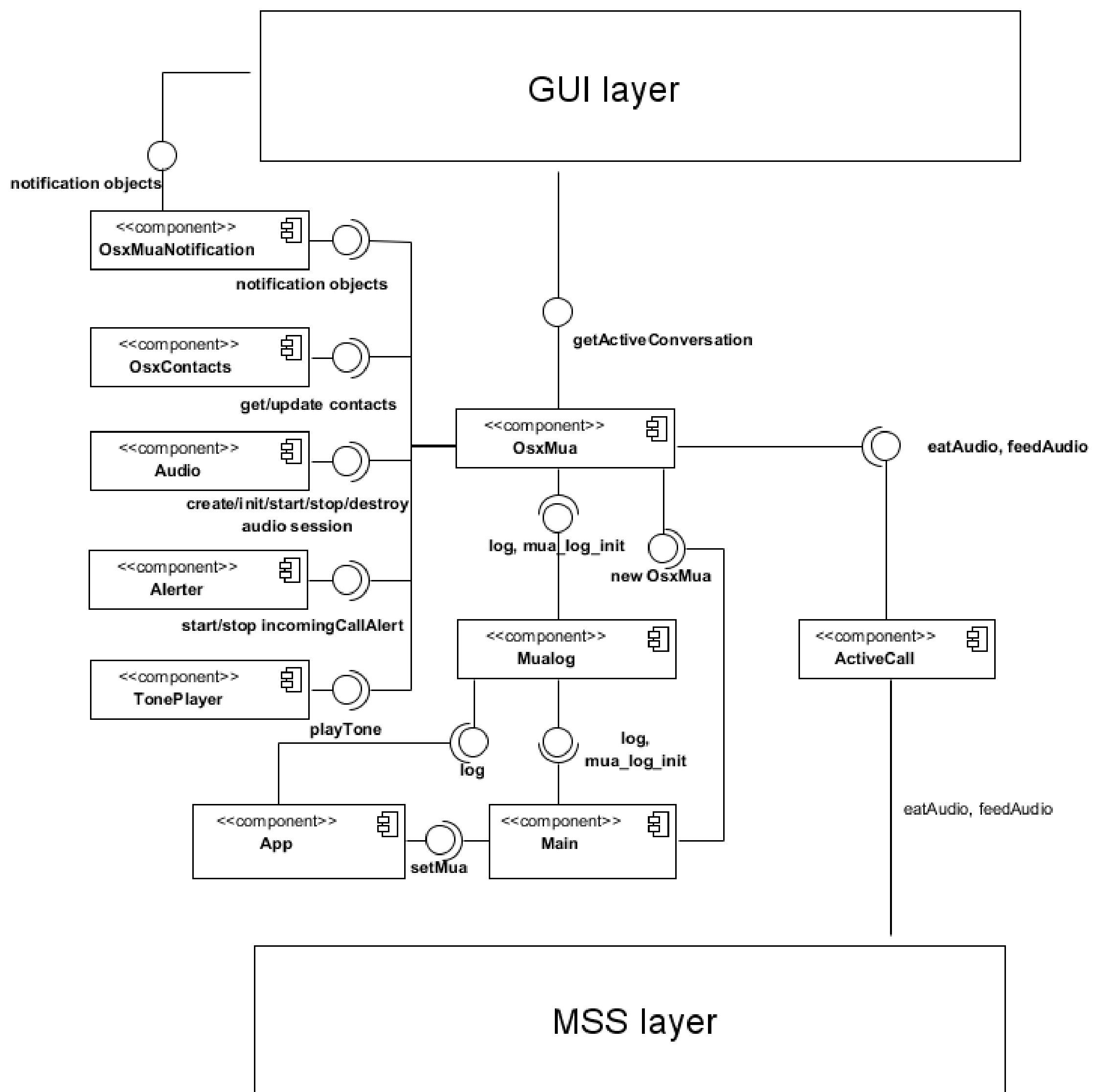


Figure 3.7: Internal architecture of the MSS User Agent layer for the OSX platform.

3.5.1.2 The Multi Session Stack Layer (MSS)

The bottom layer of the client application is called the Multi Session Stack (MSS). This layer is largely platform independent and contains most of the actual logic contained within the client application. It is structured into different modules, which communicates internally through a routing module. The internal architecture of the Multi Session Stack is visualized in figure 3.8 and the purposes of the different modules are described below. Most of the Multi Session Stack layer is implemented in C, but some parts are written in C++.

mssapi – the interface for communicating with the upper layers.

routing – takes care of internal (asynchronous) communication within the Multi Session Stack layer.

session – the Multi Session Stack layer's internal representation of a voice call.

letter – the module used for sending text messages.

object – the module responsible for encoding and decoding create, read, update and delete requests sent to and from the CRS.

crypto – the module responsible for encrypting and decrypting MIKEY-SAKKE and AES messages. This module is used to fulfill both the authentication and the confidentiality requirements described in section 3.2.

umsg – the module used for communicating with the CRS.

media – contains a codec and buffers for handling audio, as well as an implementation of the SRTP protocol.

platform – contains common functionality that requires platform-specific implementations, such as creating UDP sockets and getting the current time in nanoseconds.

utils – contains common utilities used by most of the other modules, such as global definitions and tools for generating the JSON messages used for internal communication.

When it comes to the security requirements of authentication and confidentiality, these are both implemented within the crypto module. The authentication is handled by the Caller creating and sending a digital ECCSI signature which is then validated in the Callee's crypto module. In order for this to work, both the clients need to trust that the identities generated by the CMS are all valid, as discussed in section 3.2.4. Once a session has been established and a shared Session Key has been exchanged, the requirement of confidentiality can also be fulfilled by encrypting and decrypting all further communication, using that Session Key. Just like the authentication, the process of encrypting and decrypting messages is also taken care of by the crypto module.

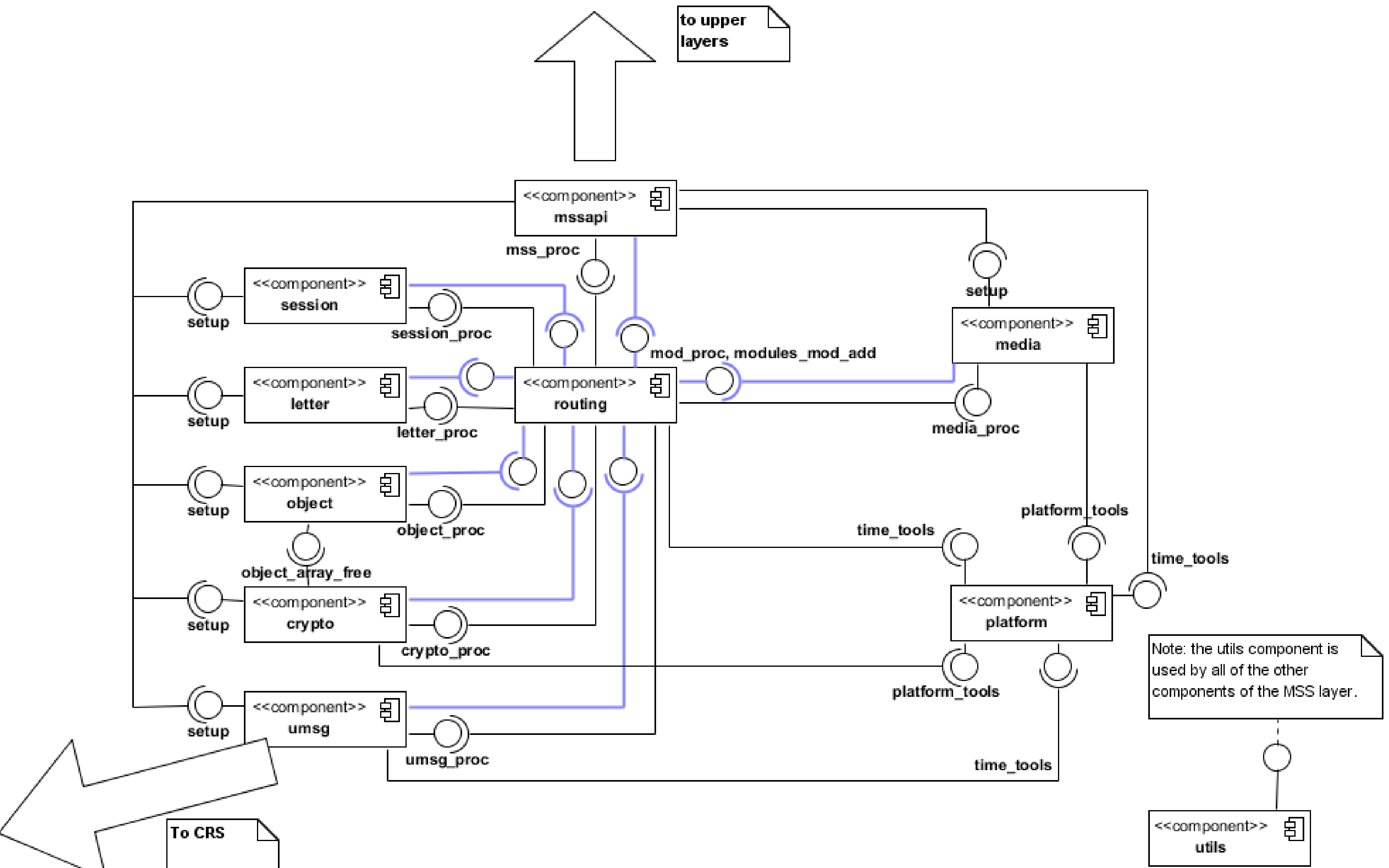


Figure 3.8: Internal Architecture of the Multi Session Stack layer

3.6 Validating the Architectural Model

As no complete documentation of the Cryptify Call system's software architecture existed before the start of this thesis, the whole architectural model given in this chapter had to be manually elicited by the author of this thesis through reverse engineering. A first draft of this model was created with the help of the Doxygen tool [16] and by interviewing several developers with insight into different parts of the code. This model was then completed by manually stepping through the code of the Cryptify Caller Application, the system's client application. The resulting architectural model was then finally validated by several Cryptify developers in order to guarantee its correctness.

4

Towards a Multiparty Calling System

This chapter concerns the actual evolution from the initial architecture into an architecture supporting conference calls in addition to normal 1-to-1 calls. Section 4.1 describes the rationale behind the technical solution chosen, section 4.2 describes how the system's security requirements had to change and section 4.3 contains the architectural model of the evolved system. The process of actually implementing the evolved software architecture is described in section 4.4, and finally will the results of a performance evaluation of the implemented solution be shown in section 4.5

4.1 Evolution Plan

One naive solution of implementing a conference call would be to set up a separate session between each pair of participants in the conversation. This would automatically solve the authentication and confidentiality requirements by letting each pair of clients in the conference session take care of that separately of each other.

However, there are some technical limitations that first must be considered before deciding upon such a solution. To be precise, to provide end-to-end encryption between all participants of a conference call would require each client to set up a unique SRTP session to each of the other participants. According to a prestudy conducted at Cryptify before the start of this thesis, such a solution might just be too performance heavy for a normal smartphone to handle [12]. The bottlenecks seem to be to encode multiple audio streams simultaneously and the network load of maintaining multiple SRTP sessions at the same time [12]. Measurements in said prestudy indicates that an iPhone 5 with a dual-core CPU should be able to handle a three-party call without problems, but that calls with more than three participants would stress the phone's CPU too much [12]. Based on this prestudy, some additional constraints therefore had to be introduced. These are that regardless of the number of participants in a conversation, each end user client participating in the conversation is only allowed to maintain a single upstream and a maximum of five concurrent downstreams of audio.

The hardware constraints in most smartphones on today's market therefore led to the decision (made by the author of this thesis and the managers at Cryptify AB) of introducing a new node into the system. All the participants of a conference call will connect to this new node, hereafter referred to as the Call Coordinator, instead of connecting directly to each other. This Call Coordinator will be implemented as an evolved client application, that instead of being run on a smartphone will be run on a desktop or laptop computer (the initial target will be the OSX operating system). The usage scenario is that the organizer of the conference call uses this computer to set up, manage and participate in the conference call. The Call Coordinator will be able to draw advantage of the computer's better performance and will be responsible for coordinating the conference call as well as fulfilling the evolved security requirements described in section 4.2.

4.2 Evolved Security Requirements

In order to give room for conference calls, the security requirements described in section 3.2 will naturally have to change slightly, as each session now can contain any number of participants instead of just two. In the architectural solution decided upon in this case, however, it was possible to keep the authentication and confidentiality requirements basically unchanged. The real change instead took place within the system's trust model, as will be described in section 4.2.3.

4.2.1 Authentication

Authentication remains just as important in the evolved as in the initial architecture. However, it is now not only one other participant that needs to be authenticated, but all the participants of the call. The chosen solution to this problem was to let the authentication requirement remain unchanged, and instead let the change take place in the system's trust model (see section 4.2.3) by letting one of the participants act as a trusted Call Coordinator. The authentication process between a single client and the Call Coordinator stays the same as in the initial architecture, but the client must now also trust the Call Coordinator to authenticate the other participants of the conference call.

4.2.2 Confidentiality

The ideal solution to ensure data confidentiality would be to maintain end-to-end encryption between all the clients participating in a conference call session. However, as already mentioned in section 4.1, such a solution would simply not be possible in the case of the Cryptify Call system when taking performance limitations into account. Therefore, the same approach as for the authentication requirement was

chosen and let the confidentiality requirement stay between the Call Coordinator and each of the participants. In addition to this, an addition to the trust model had to be made, which is that all of the participants have to trust that the Call Coordinator forwards all media data in encrypted form to the other participants of the conference session, without manipulating or censoring any of it along the way. Finally, the regular participants will also have to trust the Call Coordinator to only forward the data they send to the other authenticated participants of the conversation, without leaking it to some untrusted external party.

4.2.3 Trust Model

As already mentioned, the key evolution of the security requirements took place at the level of the system's trust model. The evolved trust model is shown in figure 4.1, again using SI* notation. In the figure, only two attached clients are shown for simplicity, one Call Coordinator and one regular participant. Each additional participant added to the call would have the same trust relationships to the Call Coordinator and the KMS as the regular participant in the figure.

The participants that attach to a conference session put trust in the Call Coordinator that it will not leak any of its session or private keys and trust the system's KMS for authenticating the Call Coordinator. In addition to this, all of the connected participants also trust that the Call Coordinator will authenticate all of the other participants, and that no voice data that they send will get manipulated or censored along the way. The fact that the Call Coordinator gets access to the unprotected voice data sent during the conversation is not a problem, since it also counts as a participant of the conference call and therefore would do so anyway. In theory, however, the Call Coordinator could discard or manipulate some voice data before forwarding it to the other participants, and the participants therefore must put trust in the Call Coordinator that it will not do so. The participants will also have to trust the Call Coordinator to only forward the contents of their conversation to the other participants, and not to some external party that is not part of the conference call — in other words, the participants must trust that the Call Coordinator do not keep any of the participants hidden from the rest.

The Call Coordinator will put the same trust in each of its counterparties as if they would participate in a normal 1-to-1 call with each other. In other words, the Call Coordinator trusts each of its counterparties to keep their respective secret keys private, and it trusts the keys generated by the system's KMS in order to authenticate their identities.

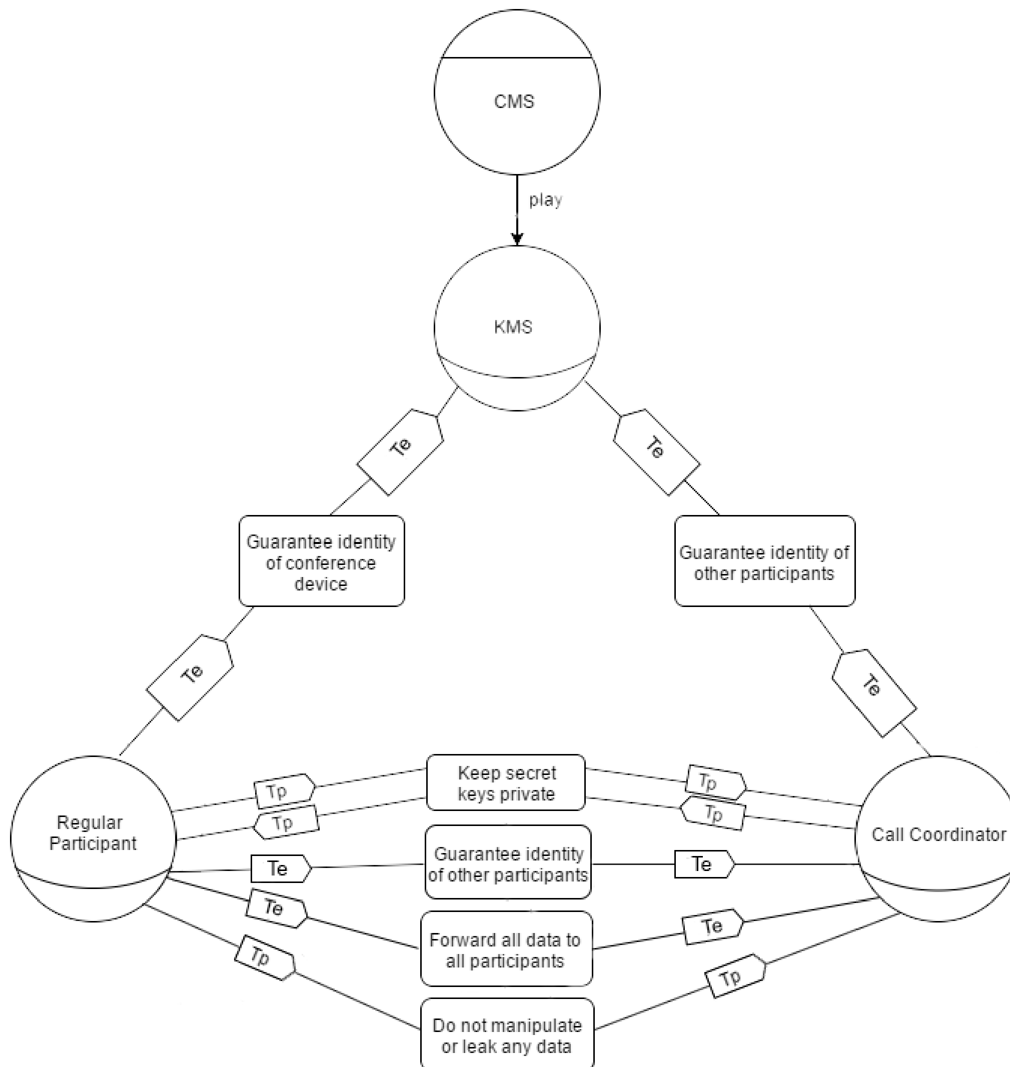


Figure 4.1: Trust model after evolution. The trust model stays largely the same as it did before evolution (see figure 3.5). The only difference is the addition of the Call Coordinator, which now acts as the counterparty to each of the participants of a single conference call session. In addition to the normal trust relationships a participant needs to have with its counterparty, the regular participants must also trust the Call Coordinator to authenticate the other participants as well as to trust that it will not manipulate, censor or leak any data sent to or from any of the participants. The model only shows one regular participant connected to the Call Coordinator for the sake of clarity. Additional participants would have the same trust relationships as the one already modeled in the figure.

4.3 Evolved Architecture

In order to evolve the Cryptify Call system with the secure conference call functionality, the author and the managers at Cryptify AB decided upon a solution that would require only limited architectural changes to the system. In fact, architectural changes were only required in the Call Coordinator, the desktop client that acts as moderator for the call and to which the other participants connect. No architectural changes had to be made to the CRS, CMS or to the other client applications, to which a conference call works just like a normal 1-to-1 call, with the Call Coordinator being their counterparty.

In addition to localizing most of the necessary architectural changes to the Call Coordinator, this solution also has the advantage of keeping the evolved security requirements simple. The only changes necessary to the system's security requirements was to add the trust relationships shown in figure 4.1 between the Call Coordinator and the regular participants. More importantly, however, is that this solution is compatible with the hardware constraints introduced in section 4.1, since most of the performance-heavy computations will be carried out by the Call Coordinator, which can easily be given enough hardware resources for the task.

In section 3.5, the architecture of a client application running on the OSX operating system was described. In order to evolve this client into a Call Coordinator for the Cryptify system, some architectural changes had to be made. All of these changes have been located to the MSS User Agent layer, and the evolved architecture of this layer is shown in figure 4.2. Of course the GUI and some other details had to be altered as well, but these changes will not be described here as they are not related to the software architecture of the system.

The first change that had to be made to the architecture was to let the MSS User Agent allow for multiple ActiveCall instances running concurrently, each one representing one participant connected to the conference call. Next, a new component (labeled Merger in figure 4.2) was added between the OsxMua and the ActiveCall component. The Merger component is responsible for merging all the incoming audio streams into a single outgoing audio stream for each of the connected participants of the call, as opposed to just forwarding all audio streams separately to all of the participants without doing any kind of merging. This step was necessary due to the regular clients of the system being constrained to a maximum of five concurrent downstreams of audio, as described in section 4.1. By performing this merging before sending any data from the Call Coordinator, each regular participant connected to the Call Coordinator will only need to maintain a single downstream of audio each, regardless of the number of participants in the conversation. In order to also let the Call Coordinator itself participate in the conference call, a CircularBuffer component was finally added for outgoing respectively incoming audio between the OsxMua and the Merger.

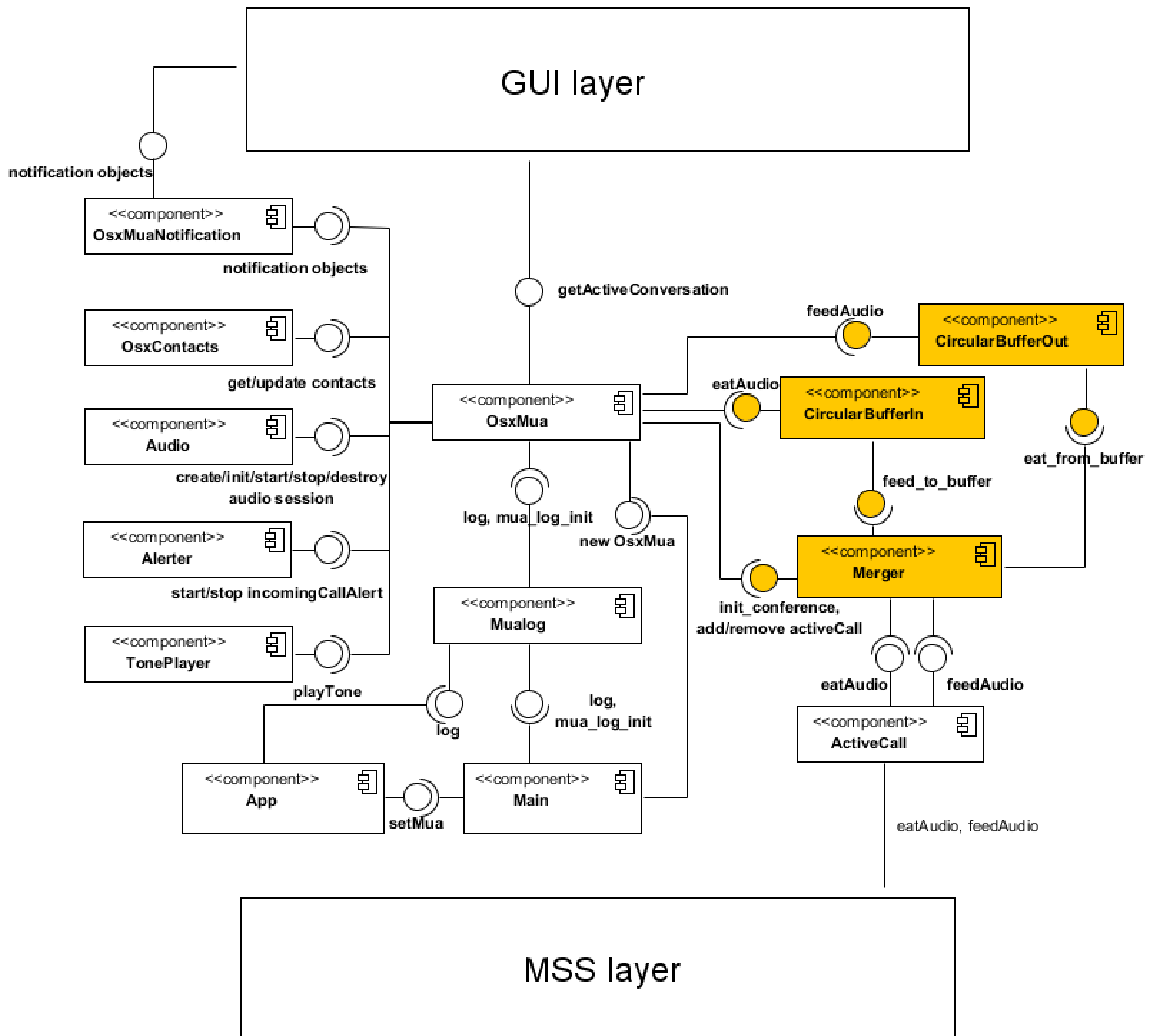


Figure 4.2: Internal architecture of the Call Coordinator's MSS User Agent layer after evolution. The new components that have been introduced are highlighted in the model. The OsxMua component now supports multiple simultaneous ActiveCall instances instead of being limited to just one. The audio sent to and from each one of them is being coordinated via the Merger component, and the audio sent to and from the Call Coordinator itself is being passed through one of the CircularBuffer components first. Apart from these changes, the architecture is the same as before evolution (see figure 3.7).

4.4 Implementing the Evolved Architecture

Once a solution for how to implement the evolved functionality of supporting secure conference calls had been decided upon in collaboration with the managers at Cryptify AB, the actual implementation was carried out by the author of this thesis alone. As most of the client-side logic is contained within the Multi Session Stack layer and platform-independent, creating the desktop application that would act as the Call Coordinator proved a relatively simple task once the evolved software architecture had been designed. The components that needed to be implemented in order to get a functional prototype running were just a basic GUI and a platform-specific implementation of the MSS User Agent layer. This client application then had to be evolved with the capability of setting up telephone conferences, according to the architectural changes specified in section 4.3 and visualized in figure 4.2. The main difficulty in this turned out to be the merging of the different audio streams and making sure that each participant receives the right audio data. The method used in order to achieve this merging is described in section 4.4.1.

In order to evolve the MSS User Agent layer of the Call Coordinator device to comply with the evolved architecture shown in figure 4.2, around 500 lines of code had to be added. Most of these lines are localized to the new Merger and CircularBuffer components, with only a few minor changes made to the OsxMua component, meaning that the risk of having introduced any new security-related bugs to the code is relatively low. The implementation also proved satisfying to the Cryptify management in other aspects as well, as will be shown in the performance evaluation conducted in section 4.5. This led to the decision of using the implemented code as the foundation for the Call Coordinator code that has now been put into production and is about to be released on the market. A screenshot of this application is shown in figure 4.3.

4.4.1 Merging of Audio Streams

Due to the performance limitations of the smartphone clients described in section 4.1, measures had to be taken in order to reduce the number of audio streams being sent to each of the participants. This was achieved by merging together all the audio streams originating from each of the participants of the call, including any voice data sent by the Call Coordinator but excluding the voice data sent by the receiving client itself (as the end user would otherwise hear an echo of his or her own voice), before sending the merged audio voice data out to each of the participants. The method used by the Call Coordinator in order to merge multiple incoming audio streams into a single outgoing stream for each of its counterparties is the following.

First, the Merger component tries to fetch a new audio packet from each of the participants using a method named eatAudio. Next, the Merger loops through all the participants of the conference call, including the Call Coordinator itself (represented

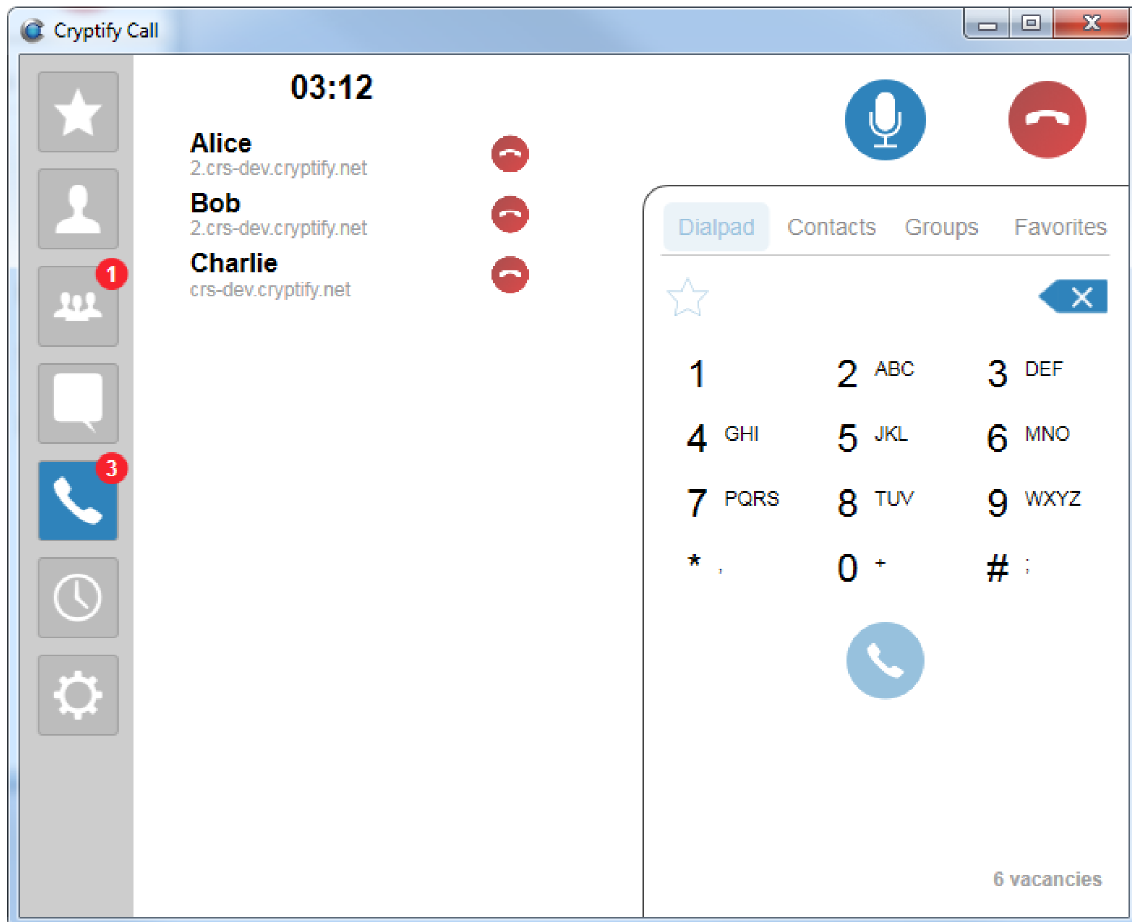


Figure 4.3: A screenshot of the implemented Call Coordinator application, showing a running conference call session with three participants connected to the Call Coordinator.

by the OsxMua component), in order to feed a merged audio stream to each of them. As the audio is encoded using the PCM audio data format[1], the merging can be done by just adding together the data contained in each audio packet retrieved by the Merger in the first step. The packet sent by the current participant itself is omitted from the merging though, in order to remove that participants own echo from the output stream. If the addition would result in a value that is outside of the range allowed by the PCM data format, that value is instead changed to the maximum or minimum value allowed respectively, a procedure known as clipping. Finally, the merged audio packet is sent to the ActiveCall instance representing the intended receiver of the packet, using the feedAudio method.

A couple of more complicated functions for the actual merging of the audio streams were considered to be used instead of the clipping function described above. However, due to the large amplitude overhead available in the PCM format, clipping seems to occur very rarely as long as only one or a few participants are speaking at the same time. As it in any case would be hard to discern anything if too many participants are speaking simultaneously, regardless of which merging function is being used, it was decided to stick to the clipping function in the final implementation. Some of the other merging algorithms that were considered to be used instead of the simple clipping function are described in Appendix A.

4.5 Performance Evaluation

In order to give some indication of the effectiveness of the implemented solution, a performance evaluation was carried out by the author of the thesis. During this performance evaluation, a MacBook Pro (early 2015 model) laptop computer was used as Call Coordinator. This device managed to maintain a multiparty call with 50 simultaneously connected participants – which should be more than enough for the intended use cases of the Cryptify Call system – without ever exceeding 100% CPU load. The results of these measurements, which are shown in the graph in figure 4.4, indicate that the CPU usage is roughly linear to the number of participants connected to the Call Coordinator. The implications of this is that if it should be necessary to support an even larger number of participants, this can easily be achieved by adding more processing the power to the Call Coordinator (or, alternatively, by improving the resource usage of the algorithms implemented in the Call Coordinator).

The roundtrip time between a pair of participants – that is, the time it takes for a data packet to travel from one client to another and then back again – was measured as well to see if any additional delay had been added to the system after the evolved

[1]Pulse-code modulation (PCM) is a common way of converting a analog audio signal into a digital format. At regular intervals, the amplitude of the analog audio signal is sampled and converted to an integer value. In the Cryptify Call system, a sample rate of 48 000 samples per second is used; meaning that each second of voice audio is represented digitally in the system by 48 000 integer values.

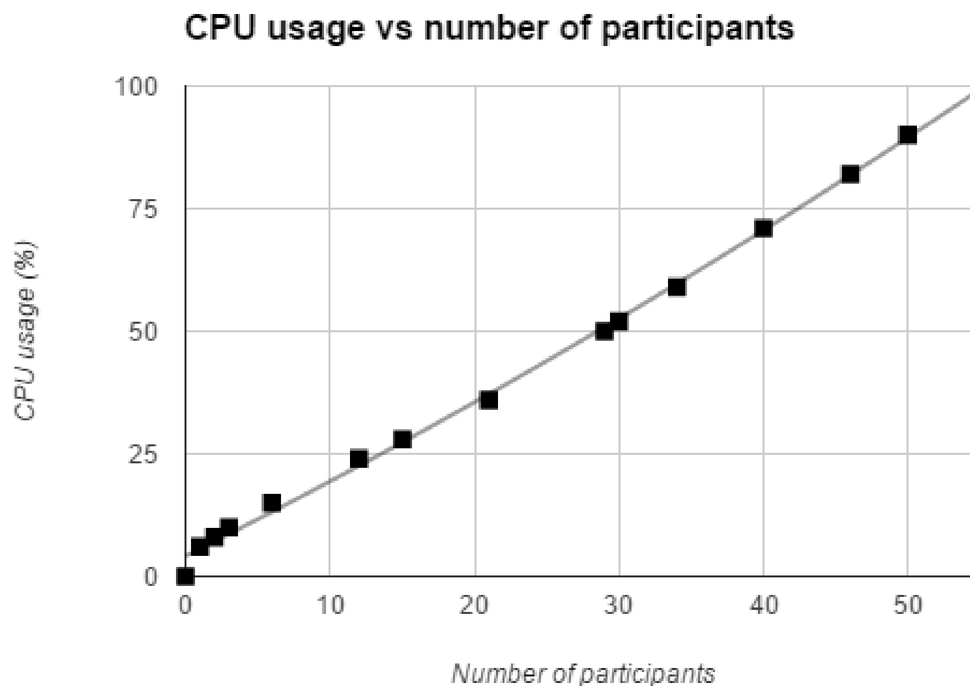


Figure 4.4: CPU usage vs. number of participants.

architecture had been implemented. A low delay is necessary in order to make the end users satisfied with the quality of the service, and it is therefore important to keep the total roundtrip time as low as possible. The roundtrip time was measured by sending data packets originating from both the Call Coordinator and from a normal client, with the target client also varying between the Call Coordinator and a normal client respectively. The measurements were performed both before and after evolution, and the results are summarized in table 4.1. For these measurements, the MacBook Pro was again used as Call Coordinator, and an iPhone 4 was used as the regular client. For the measurements before evolution, the normal OSX client application was run on the MacBook Pro (in other words, before it was evolved to act as Call Coordinator). For the measurements after evolution, a varying amount of simultaneously connected participants was used to make sure that this factor did not affect the results. Since the roundtrip time is affected by random network fluctuations, a relatively large number of measurements had to be made in order to determine an average value. Each of the average values specified in table 4.1 is therefore based upon between 100 and 300 individual measurements. It was not possible to do more measurements than this due to the time constraints of the thesis.

Interesting to note is that when measuring the time for a packet to travel from the MacBook to the iPhone and back again, the average roundtrip time proved to be exactly the same whether using a normal two-party call (before evolution) or a multiparty call with the MacBook as Call Coordinator (after evolution). This alone is a clear indicator that no additional delay seems to have been added to the system by the evolution performed during the thesis project. The added delay of 58

	Before evolution	After evolution	Difference
MacBook –> iPhone	196 ms	196 ms	0
iPhone –> MacBook	299 ms	357 ms	+ 58 ms
iPhone –> iPhone	239 ms	208 ms	-31 ms

Table 4.1: Roundtrip time in milliseconds, before and after evolution.

milliseconds in roundtrip time when transferring a packet from the iPhone to the MacBook and back again is explained by the implementation of the circular buffer within the Call Coordinator. As the length of the audio packets that is being put into this buffer is 60 milliseconds, the Call Coordinator correspondingly tries to eat from the buffer every 60 milliseconds. Theoretically there should therefore be a 60 milliseconds long delay before it is possible to consume the first packet, which is close enough to the measured value of 58 milliseconds. As the audio packets thereafter should continue to arrive at roughly regular intervals, the delay should continue to be constant throughout the duration of the call. The measurements indicate that this really is the case, and the added delay is small enough to be acceptable under the current circumstances.

When measuring the roundtrip time between two normal iPhone client applications, a 31 milliseconds faster roundtrip time was measured when the clients where connected through a Call Coordinator. However, this does not necessarily mean that the evolved system provides a shorter overall roundtrip time, as this difference is not statistically significant due to the large variance observed in the gathered measurement data (the standard deviations for the measurements before and after evolution are 24 ms and 21 ms respectively). The observed difference in roundtrip time could therefore be caused by temporary network fluctuations or measurement errors. In order to be able to show a statistically significant difference, more data would have to be gathered, perhaps as many as 1000 measurements both before and after evolution. As each measurement had to be done manually in a rather time-consuming manner, the time constraints of this thesis unfortunately prevented further testing to be done.

5

Change Patterns for Multiparty Communication

In this chapter, the findings of this master's thesis will be presented. Most importantly, section 5.1 summarizes the evolution performed in chapter 4 in the form of an abstract and context-free change pattern. This change pattern can hopefully be reused by software engineers encountering the same or a similar change scenario as described here in the future. An alternative change pattern is also proposed in section 5.2.

5.1 Main Change Pattern

In this section, a precise change scenario for a system evolving from secure one-to-one into secure multiparty communication will be described, together with an architectural-level description of how to implement the proposed solution. The change scenario is shown using SI* notation in figure 5.1 and depicts a change in the system's trust model. As long as a system has a trust model that correspond to the trust model shown in figure 5.1a, then the change pattern can be applied by following the architectural transformation given in section 5.1.1. Doing so will result in the trust model shown in figure 5.1b.

Figure 5.1a, representing the system's trust model before evolution, shows the relationship between the system, the service it provides and two clients (denoted as Caller and Callee) participating in a secure conversation with each other. The two clients trust that the keys and identities generated by the system's Root of Trust[1] are valid and are dependent on the Call Service (owned by the system) to provide the actual functionality of their communication. The clients also trust each other not to leak any sensitive information, such as their respective private keys or their shared Session Key.

Figure 5.1b shows the situation after the Call Service has been evolved to support

[1] Which is the Key Management Server – a role played by the CMS – in the case of the Cryptify Call System.

multiple participants of the same call session. The participants still trust the keys and identities generated by the system's Root of Trust and depend on the functional Call Service provided by the system, just as before. In addition to this, however, one of the participants now has to act as the coordinator of the call. This special client, referred to as the Call Coordinator, sets up a normal call session with each of the other participants, with each of these individual call sessions having the same trust relationships as between the Caller and the Callee in figure 5.1a. In addition to this, however, the regular participants also have to trust the Call Coordinator to forward all data sent by a participant during the conversation to all of the other participants. Finally, the regular participants also have to trust the Call Coordinator not to misuse its role, for example by modifying or censoring the data sent by some of the clients, or by leaking data to some unauthorized party. Figure 5.1b shows a conference call with a total of two participants (including the Call Coordinator), but any number of additional participants could be added as well. All additional participants would have the same relationships as the the regular participant already included in the figure.

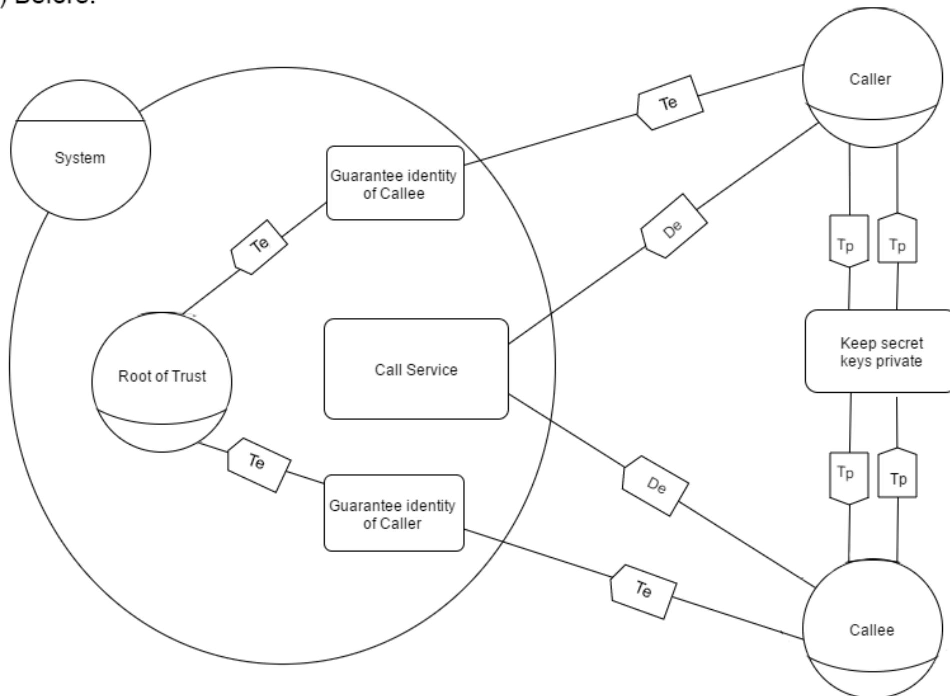
5.1.1 Solution

As the Call Coordinator introduced in figure 5.1b will also act as a participant of the multiparty conversation, the natural way to implement this change pattern is to evolve a normal client into also assuming the Call Coordinator role. This way, the Call Coordinator can still act as a participant of the call, while at the same time take care of its added responsibilities of coordinating the call. If this is possible, it will be an easy process to implement the change pattern, as will be shown in the following sections. Section 5.1.1.1 describes some preparatory work that has to be done in order to prepare the system to be evolved according to the architectural transformation that is given in section 5.1.1.2.

5.1.1.1 Architectural Support

In order to evolve the architecture of a software system according to the change scenario described in section 5.1, some important roles first need to be identified according to the architectural support template shown in figure 5.2. More specifically, the roles mentioned in the deployment diagram (figure 5.2a) should be mapped to the corresponding components in the initial architecture of the system being evolved.

Figure 5.2a shows a Client device, which consists of a Participant module (in turn containing a Cryptographic Module) and a Call Participant Proxy. The latter is the interface used by the client when exchanging data with its counterparty during a secure conversation. The actual data exchange between two clients is facilitated by some Call Service deployed on a separate device, as also shown in figure 5.2a. The interfaces between these different components are shown in the component diagram in figure 5.2b. Note that it is assumed in the component diagram that each



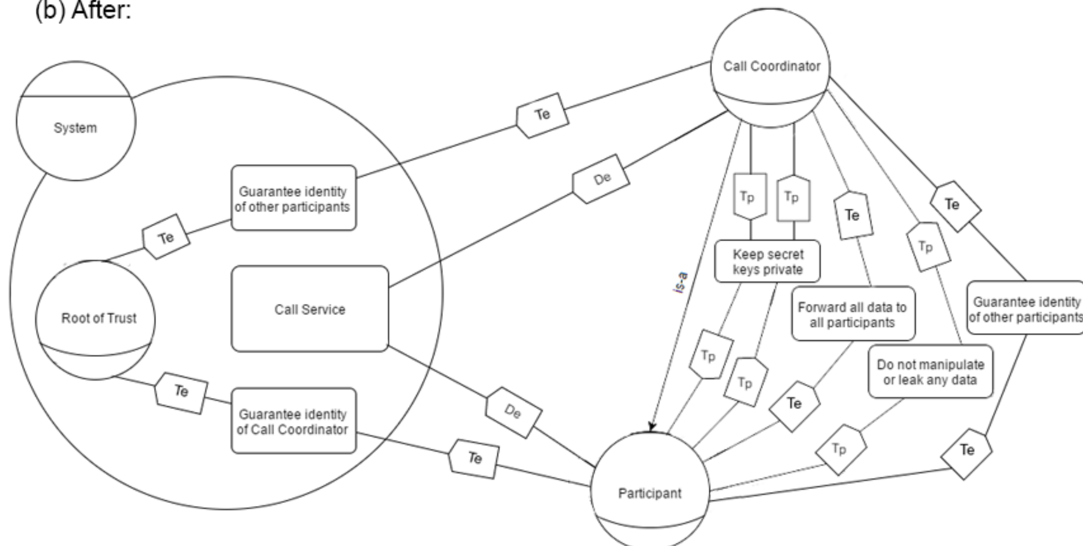


Figure 5.1: Change scenario for the main change pattern. The figure shows the trust model of the system, both before and after evolution. A new role, the Call Coordinator, has been added to the system in the situation after evolution. The other participants trust and depend on the Call Coordinator to manage the conference call and to forward all data (without modifying it in any way) sent during the conversation to all of the participants, and to no one else. The Call Coordinator, which is also a participant of the call in addition to its other responsibilities, in turn trusts the regular participants not to leak any sensitive data or cryptographic keys regarding their conversation. Note that while all of the participating clients must put trust in the Call Coordinator, there is no direct trust relationship between any of the regular participants themselves.

participant can act as either Caller or Callee, depending on who is calling who. In some implementations however, there might be certain client types that will always assume only one of these roles. If that is the case, either the CallerPort or the CalleePort in figure 5.2b can be ignored for those clients.

5.1.1.2 Change Guidance

When applying the change pattern to the software architecture of the target system, the deployment diagram in figure 5.2a will have to be evolved by adding a Call Coordinator device, as shown in figure 5.4a. This Call Coordinator, which still has all the capabilities of a normal client, must also be given the capability of maintaining multiple simultaneous counterparties, each of them represented by a separate Call Participant Proxy. The Call Coordinator also needs access to some merging module that can merge all of the incoming data streams into a single outgoing data stream for each of the Call Coordinator's individual counterparties. Note that the Client and Service Provider devices remain unchanged in the evolved deployment diagram shown in figure 5.4a.

The Multiparty Participant component in figure 5.4 encapsulates two components, a Merger and a normal Participant component. The interfaces and internal architecture of the Multiparty Participant component are shown in the component diagram in figure 5.4b. In this solution, the Call Coordinator can dynamically choose between the roles of Caller and Callee, and it can even assume different roles for different of its counterparties in the same conversation. If an implementation where the Call Coordinator only uses one of these roles is desired, then only one of the CallerPort and CalleePort needs to be used.

As already stated, the best way to move from the architecture shown in figure 5.2a into the evolved architecture shown in figure 5.4a is to evolve a normal client into also assuming the Call Coordinator role. The transformation described below gives a more detailed guidance of how this is done, and can be used as a template for how to evolve a system according to the change scenario described in section 5.1. Once the relevant roles in the initial software architecture have been identified, as described in section 5.1.1.1, applying the transformation should in most cases be a straightforward and relatively easy process. The following change guidance is also expressed in the form of an automatic model-to-model transformation in section 5.1.1.3.

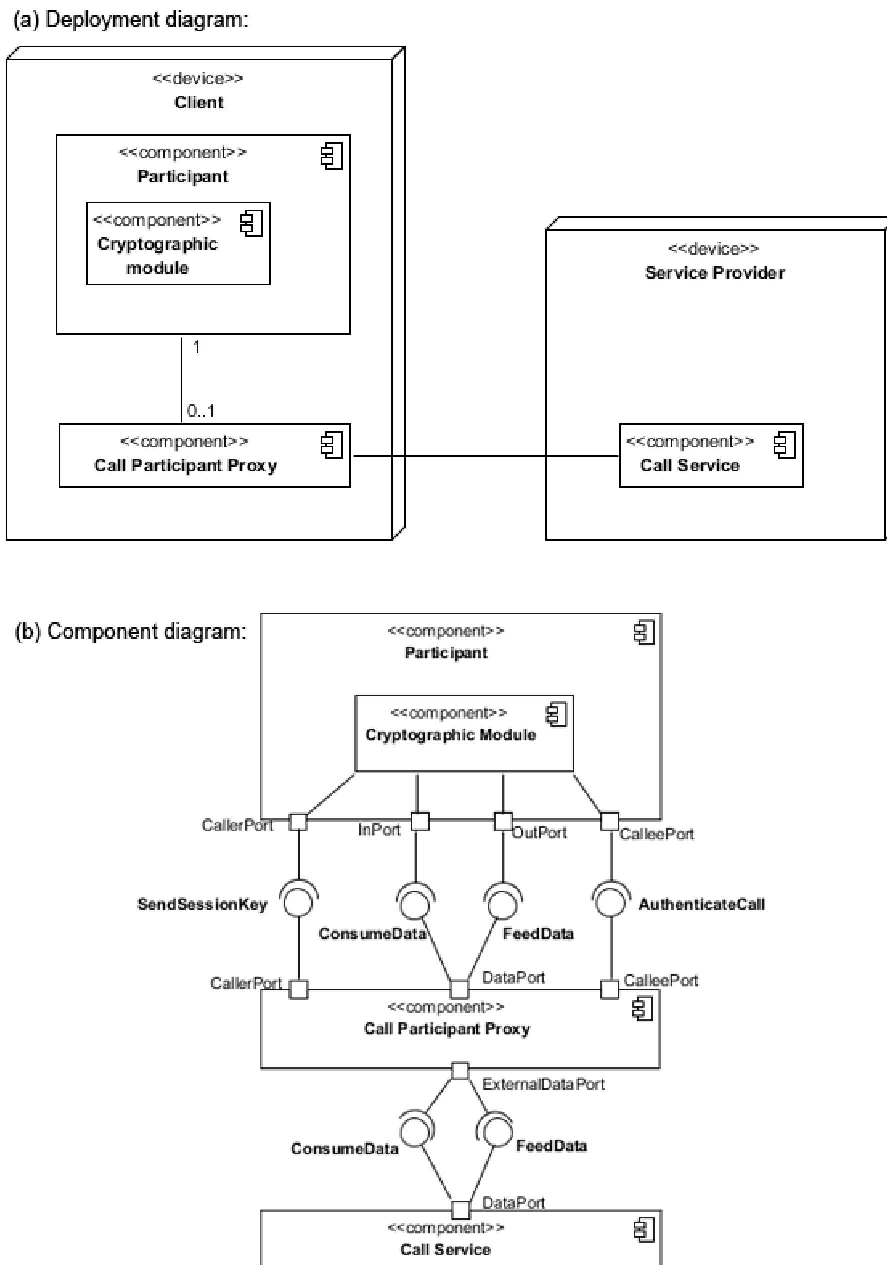


Figure 5.2: Architectural support template for the main change pattern. The deployment diagram shows the client device, consisting of a Participant component and a Call Participant Proxy, and some other device providing the system's Call Service. The Call Participant Proxy is used by the client to exchange data with its counterparty, via the Call Service provided by the Service Provider device. The interfaces between these different components are detailed in the component diagram.

1. In order to create the Call Coordinator shown in the deployment diagram in figure 5.4a, a Multiparty Participant component (as shown in figure 5.4b) first needs to be developed. This is done in the following way:
   (a) Start by encapsulating a Participant component, taken from the implementation of a normal client, into the Multiparty Participant.
   (b) Also add a Merger component to the Multiparty Participant. The Merger component is used to merge the incoming data streams from each of the Call Coordinator's counterparties (plus any data sent by the Call Coordinator itself) into a single outgoing data stream for each of its counterparties.
   (c) Move the CallerPort, CalleePort, InPort and OutPort from the Participant component to the Multiparty Participant component. Under certain circumstances, the software architect can choose to ignore either the CallerPort or the CalleePort, as described above.
   (d) Let the Cryptographic Module feed its outgoing data through the Merger component, which will merge all the different data streams and then in turn feed the merged data out through the Multiparty Participant's OutPort.
   (e) Connect the rest of the ports directly to the Cryptographic Module.
2. The Call Coordinator needs to be given the capability of handling multiple simultaneous connections to other clients, instead of just one. Therefore, the Multiparty Participant component should be able to have any number of simultaneous Call Participant Proxies, as shown in figure 5.4a.
3. The software architecture has now been evolved according to the change pattern. When the architecture has been implemented, the system is therefore guaranteed to support secure multiparty calls using the trust model shown in figure 5.1b. Note that the Client and Service Provider devices do not have to change architecturally in any way, as a multiparty call using this solution will technically look exactly the same as a two-party call from their point of view.

5.1.1.3 Automatic Transformation

The transformation for how to evolve a regular client into a Call Coordinator is also expressed as a model-to-model transformation in code snippet 5.1 below, using the QVT Operational transformation language [17] [35]. The transformation is designed in order to be used on an architectural model expressed as a UML component diagram.

In order to use the transformation, the UML profile presented in figure 5.3 first needs to be applied to the UML model of the initial architecture. This is done by applying the stereotype *ComponentType* to each of the elements corresponding to one of the roles that were identified in section 5.1.1.2. The *type* attribute of each

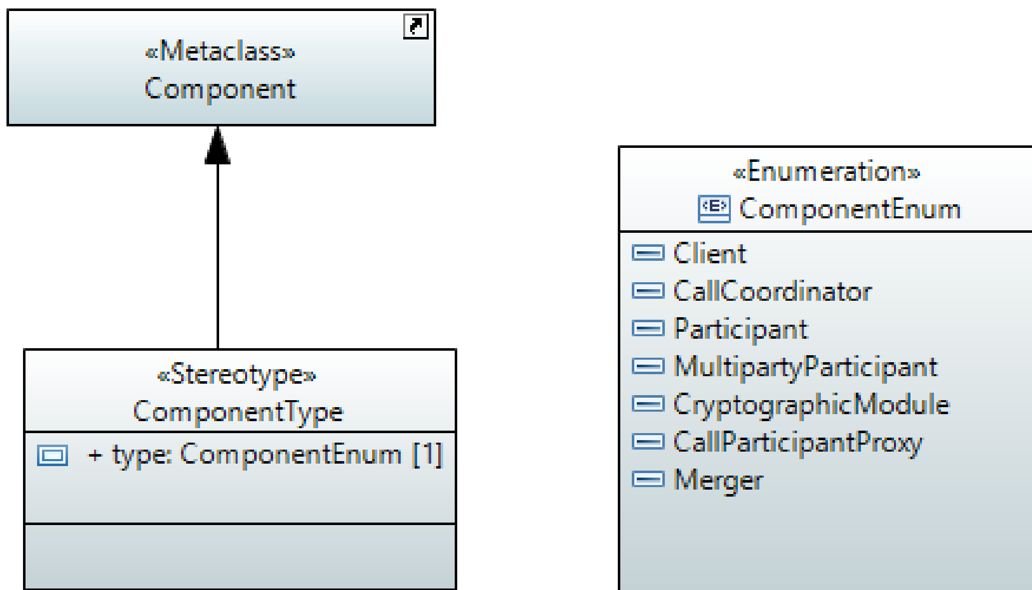


Figure 5.3: Graphical representation of the UML profile that needs to be applied to the system's architectural model before it can be transformed using one of the automatic transformations. The *ComponentType* stereotype is applied to each component that has an architecturally significant role in the sense of the transformation. This way, the component can be assigned one of the *ComponentEnum* types that corresponds to its architectural role.

of these components is consequently set to the *ComponentEnum* that corresponds to each component's architectural role. When this is done, the transformation will be able to identify the components shown in figure 5.2 and automatically transform the architecture according to the change guidance shown in figure 5.4. Note that the actual names of the components does not matter, which simplifies the process of transforming the architecture.

Code snippet 5.1: Main pattern transformation

```
modeltype UML uses 'http://www.eclipse.org/uml2/5.0.0/UML';

transformation MainPatternTransformation(inout model : UML, in profile :
    UML);

property componentType : Stereotype = profile.objects()[Stereotype]![name
    = 'ComponentType'];

// Global variables used when comparing enum values
property clientEnum : EnumerationLiteral;
property participantEnum : EnumerationLiteral;

main() {
   // Assign values to the global enum variables
```

```
   profile.objects()[EnumerationLiteral]->enum_helper();

   // Perform actual transformation
   model.objects()[Component]->client2callCoordinator();
}

mapping inout Component::client2callCoordinator()
when {
   // Only perform this mapping for any component of the Client type
   not
      self.getAppliedStereotype('Profile::ComponentType').oclIsUndefined()
   and self.getValue(componentType, 'type') = clientEnum
}
{
   // Transform the Client into a Call Coordinator
   self.name := "Call Coordinator";
   self.setValue(componentType, 'type', 'CallCoordinator');

   // Create a Multiparty Participant component and add it to the Call
      Coordinator
   var multipartyComponent = object Component { name:='Multiparty
      Participant' };
   self.packagedElement += multipartyComponent;
   multipartyComponent.applyStereotype(componentType);
   multipartyComponent.setValue(componentType, 'type',
      'MultipartyParticipant');

   // Create a Merger component and add it to the Call Coordinator
   var mergerComponent = object Component { name := 'Merger' };
   multipartyComponent.packagedElement += mergerComponent;
   mergerComponent.applyStereotype(componentType);
   mergerComponent.setValue(componentType, 'type', 'Merger');

   // Locate the Participant component(s) and move it to the Multiparty
      Participant component
   var participantComponents = model.objects()[Component]
     ->select(c | not
        c.getAppliedStereotype('Profile::ComponentType').oclIsUndefined()
             and c.getValue(componentType, 'type') = participantEnum);
   multipartyComponent.packagedElement += participantComponents;
}

helper EnumerationLiteral::enum_helper() {
   // These two EnumerationLiterals will be needed to make enum
      comparisons in the client2callCoordinator mapping
   if (self.name = 'Client') then clientEnum :=
      self.oclAsType(UML::EnumerationLiteral) endif;
   if (self.name = 'Participant') then participantEnum :=
      self.oclAsType(UML::EnumerationLiteral) endif;
```

```
}
```

5.1.2 Advantages and Disadvantages

The main benefit with the solution described in section 5.1.1 is that it is simple to implement. Architectural changes are only necessary to the one client that is to assume the Call Coordinator role. No changes are necessary to any other part of the system's software architecture, as made clear in figure 5.4a (assuming that the clients already before the evolution had the capability of dynamically choosing between the Caller and Callee roles). The performance requirements also remain the same for all of the system's hardware nodes except for the Call Coordinator, which can easily be upgraded with enough hardware resources to maintain a multiparty call with a large number of participants.

The downside of this solution is that the system's trust model have to be somewhat modified, since the participants of a multiparty call no longer can maintain complete end-to-end encryption between each other. Instead they have to put some amount of trust in the Call Coordinator, as shown in figure 5.1b.

5.2 Alternative Change Pattern

In some systems it might not be possible or desirable to put trust into a Call Coordinator as was done in in the main change pattern (see figure 5.1). Therefore, an alternative change pattern which takes another approach is described in this section, namely to rely on peer-to-peer security instead of using a centralized Call Coordinator. The change scenario for this alternative change pattern is shown in figure 5.5. The situation before evolution, shown in figure 5.5a, is the same as for the main change pattern (see figure 5.1a), but the situation after evolution instead uses the trust model shown in figure 5.5b. With this solution, each participant of a multiparty call have the same trust relationships to all of their counterparties as they would have to their lone counterparty in a normal two-party call.

The roles that need to be identified and mapped to the corresponding components in the software architecture of the system being evolved are the same as for the main change pattern (see section 5.1.1.1 and the architectural support template in figure 5.2). The deployment diagram in figure 5.6a shows the relevant roles in the evolved architecture. The internal structure of every client is now similar to that of the Call Coordinator described in section 5.1.1.2, except that there is no need for a Merger component in this solution, as each client sends its outgoing data directly to each of its counterparties instead of letting a Call Coordinator forward it. The transformation necessary in order to evolve a system according to this change pattern is therefore similar to the transformation described in section 5.1.1.2. The

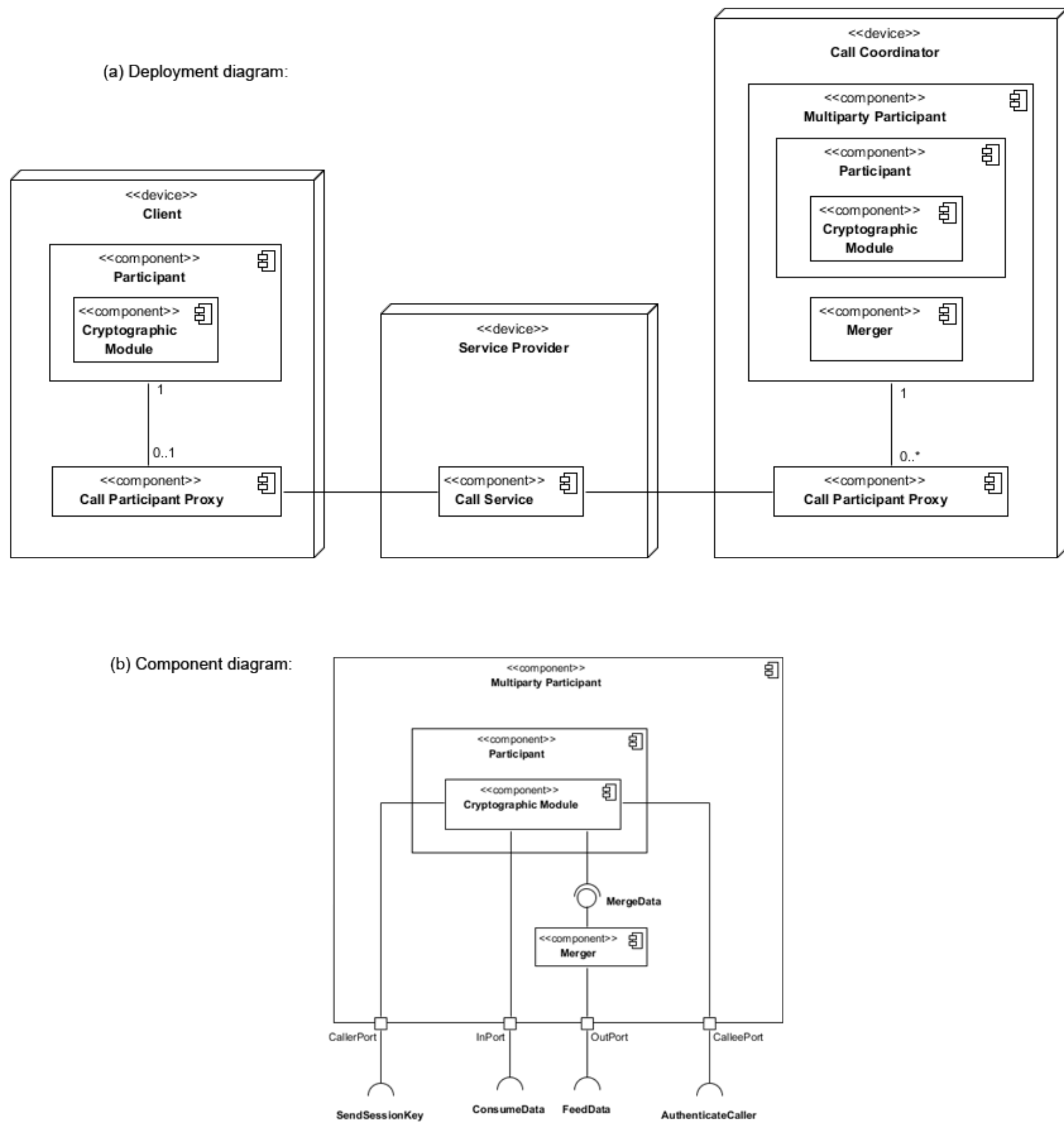


Figure 5.4: Change guidance for the main change pattern. One of the clients is evolved into a Call Coordinator, as shown in the deployment diagram, which can have any number of simultaneous counterparties. The component diagram shows how the Multiparty Participant component encapsulates a normal Participant component and a Merger component, which is used to assemble all the different signals sent to the Call Coordinator into a single outgoing signal for each of the Call Coordinator's counterparties. Both the Call Coordinator and the regular participants each have the capability of acting as either Caller or Callee in this solution.

transformation for the alternative change pattern is given below, and is to be applied to all the clients of the system (instead of just one, as were the case with the main change pattern).

1. Each client in the system is to be evolved with a Multiparty Participant component (see figure 5.5). This component is created in the following way:
   (a) Start by encapsulating a Participant component, taken from the implementation of a normal client, into the Multiparty Participant.
   (b) Move the CallerPort, CalleePort, InPort and OutPort from the Participant component to the Multiparty Participant component.
   (c) Connect these ports directly to the Cryptographic Module encapsulated in the Participant component. The result should now look like the Multiparty Participant component depicted in figure 5.5b.
2. Each client also needs to be given the capability of handling multiple simultaneous connections to other clients, instead of just one. Therefore, the Multiparty Participant component should be able to have any number of simultaneous Call Participant Proxies, as shown in figure 5.5a.

5.2.1 Automatic Transformation

Just as for the main change pattern, the change guidance for the alternative change pattern is also given in the form of an automatic transformation, expressed in the QVT Operational language, in code snippet 5.2 below. Before the transformation can be executed, the UML profile shown in figure 5.3 needs to be applied to the UML model of the initial architecture, just as in the case of the main transformation.

Code snippet 5.2: Alternative pattern transformation

```
modeltype UML uses 'http://www.eclipse.org/uml2/5.0.0/UML';

transformation AlternativePatternTransformation(inout model : UML, in
   profile : UML);

property componentType : Stereotype = profile.objects()[Stereotype]![name
   = 'ComponentType'];

// Global variables used when comparing enum values
property clientEnum : EnumerationLiteral;
property participantEnum : EnumerationLiteral;

main() {
  // Assign values to the global enum variables
  profile.objects()[EnumerationLiteral]->enum_helper();
```

```
   // Perform actual transformation
   model.objects()[Component]->evolveClient();
}

mapping inout Component::evolveClient()
when {
   // Only perform this mapping for any component of the Client type
   not
      self.getAppliedStereotype('Profile::ComponentType').oclIsUndefined()
   and self.getValue(componentType, 'type') = clientEnum
}
{
   // Create a Multiparty Participant component and add it to the Client
   var multipartyComponent = object Component { name:='Multiparty
      Participant' };
   self.packagedElement += multipartyComponent;
   multipartyComponent.applyStereotype(componentType);
   multipartyComponent.setValue(componentType, 'type',
      'MultipartyParticipant');

   // Locate the Participant component(s) and move it to the Multiparty
      Participant component
   var participantComponents = model.objects()[Component]
      ->select(c | not
         c.getAppliedStereotype('Profile::ComponentType').oclIsUndefined()
               and c.getValue(componentType, 'type') = participantEnum);
   multipartyComponent.packagedElement += participantComponents;
}

helper EnumerationLiteral::enum_helper() {
   // These two EnumerationLiterals will be needed to make enum
      comparisons in the client2callCoordinator mapping
   if (self.name = 'Client') then clientEnum :=
      self.oclAsType(UML::EnumerationLiteral) endif;
   if (self.name = 'Participant') then participantEnum :=
      self.oclAsType(UML::EnumerationLiteral) endif;
}
```

5.2.2 Advantages and Disadvantages

The advantage of this solution is that end-to-end encryption and authentication can be maintained between all pairs of participants in the multiparty call. No additional trust – apart from the fact that each call session now can consist of any number of participants – has to be introduced to the system's trust model. On the downside, however, this solution requires every client application to be modified in order to support multiple simultaneous call sessions. As each participant will need to

encrypt and decrypt a separate data stream to and from each of its counterparties, this solution will also be extremely performance-heavy for the clients if the number of participants in a conversation is large. Another potential flaw with this design arises if the communication links between some of the participants should break down while others remain intact. If this would happen, some of the participants would not be able to communicate with each other any longer, and thus giving the different participants differing views of who really is part of the conversation.

Some caution should also be advised in order to avoid introducing unnecessary audio feedback to a multiparty conversation, as the risk of doing so is increased in a solution such as this where all of the participants continuously send data to each other. This should not be an issue in the case of the Cryptify Call system, which already has very effective feedback elimination algorithms in place, but this factor should be taken into consideration before deciding upon applying this solution to some other system.

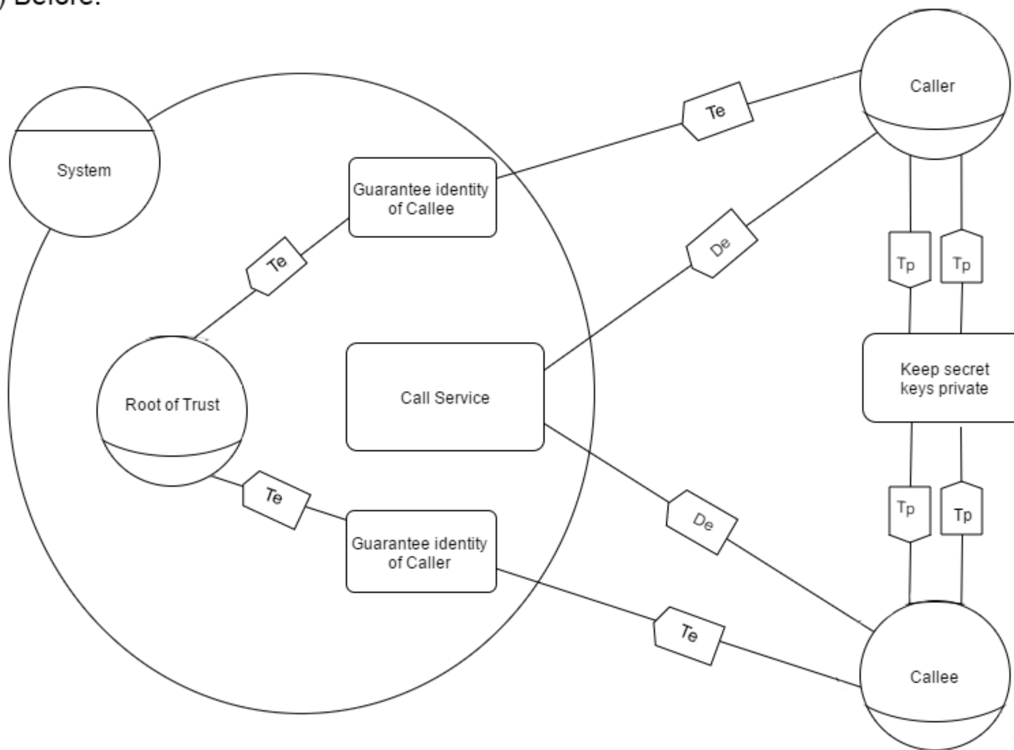
5.3 Choosing and Applying a Change Pattern

For the sake of clarity, in this section follows a step-by-step guide on how to apply either of the two change patterns identified in this chapter to some arbitrary system.

First of all, the software developer looking to use one of these change patterns needs to make sure that the trust model given in the before situation of the change scenario of both the change patterns (as the before situation is the same in both of the change patterns described in this chapter) can be mapped to the system that is to be evolved. Once this is done, the developer can choose which out of the two given change scenarios (either the change scenario shown in figure 5.1 or the one shown in figure 5.5) that is desired for the system. If neither of these change scenarios correspond to the change of trust that the system is to undergo, then neither of the two change patterns given in this chapter can be applied.

Once a change pattern has been decided upon, the architectural roles in the given architectural support template (figure 5.2 applies to both of the change patterns) need to be mapped to the corresponding components in the system's software architecture. In case that some of the roles are missing in the initial software architecture, these roles will need to be added to the architecture before proceeding.

The final step is to follow the architectural-level transformation given in the change guidance section of the chosen change patterns. This should result in an architecture that looks like the one shown in figure 5.4 or 5.6 respectively.



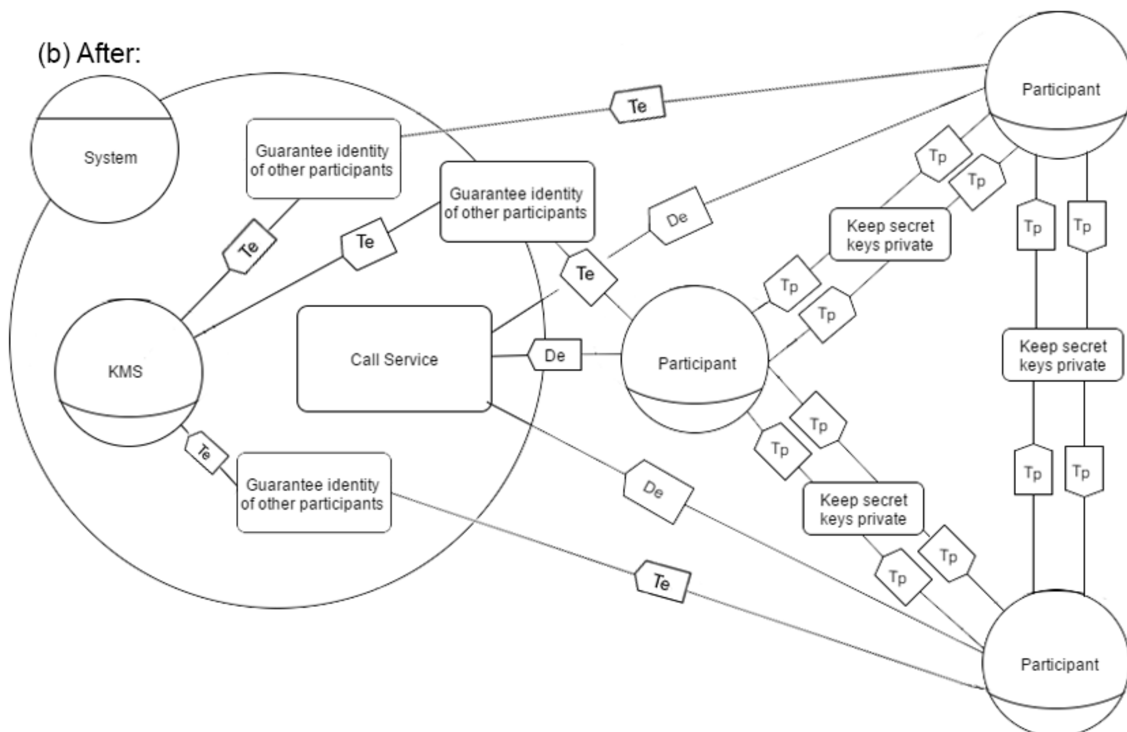


Figure 5.5: Change scenario for the alternative change pattern. In this scenario, each pair of participants in the multiparty call sets up their own separate call session, while the architecture remains otherwise unchanged.

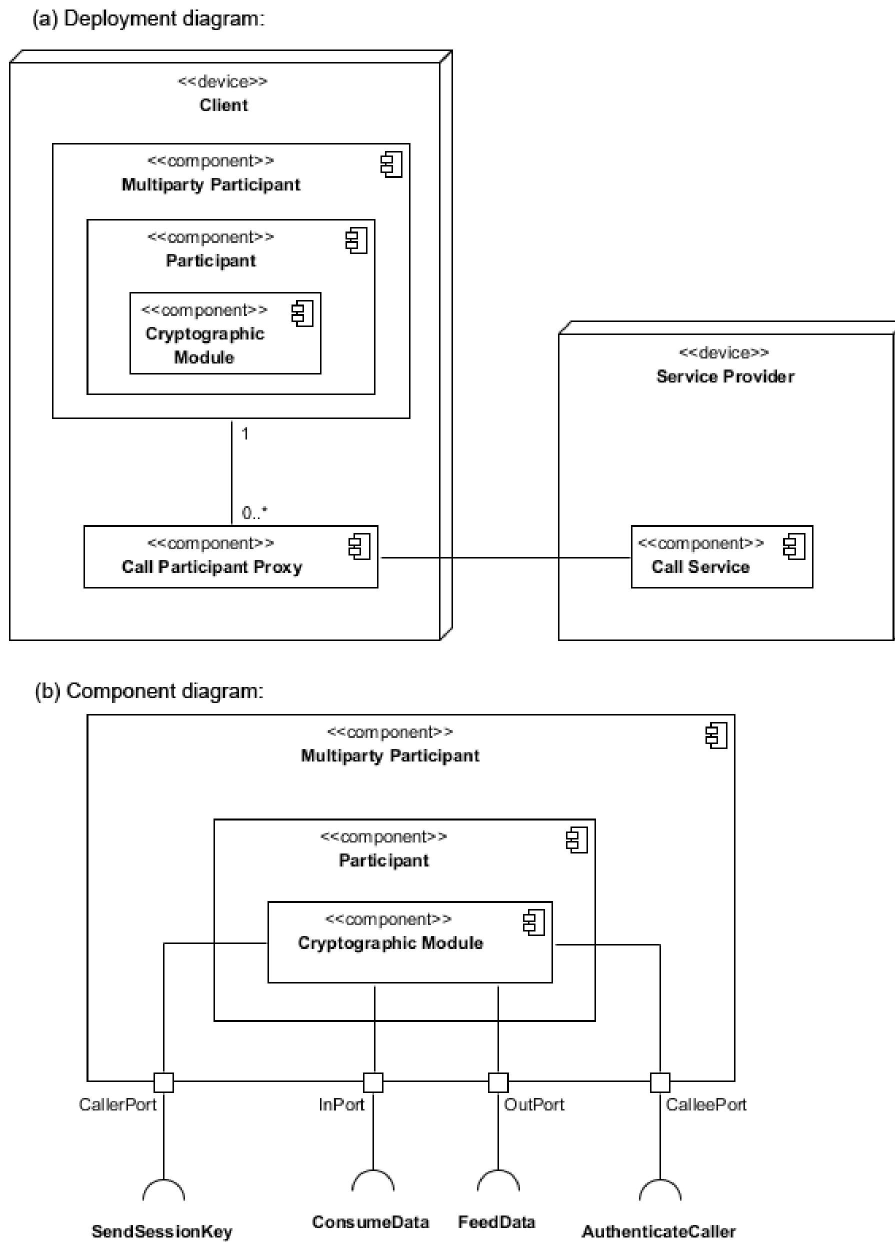


Figure 5.6: Change guidance for the alternative solution. Every Client is evolved with a Multiparty Participant component in this change pattern. The internal architecture of the Multiparty Participant component is shown in the component diagram in this figure.

6

Discussion and Conclusions

In this chapter, some aspects of the thesis will be discussed in more depth. First, a short discussion is held in section 6.1 regarding the identified change patterns and the impressions from implementing one of them. Next, a couple of other contexts where the same change patterns could be applied are mentioned in section 6.2. Thereafter, some directions which the future development of the Cryptify Call system might take are explored in section 6.3, while a few areas requiring future work from a research perspective are identified in section 6.4. Some of the challenges encountered during the thesis project are discussed in section 6.5, and finally is the thesis concluded in section 6.6.

6.1 The Identified Change Patterns

In chapter 5, two suitable change patterns were identified by the author in answer to the research questions specified in section 1.2. Out of the two change patterns described in chapter 5, the first solution (in which a centralized Call Coordinator is used for each multiparty call) was chosen to be implemented for the Cryptify Call system. The main reasons for this pattern to be chosen were that it only required architectural changes to a single node (the client being evolved into Call Coordinator) and that the bottleneck of the smartphone clients' performance limitations could be avoided. This solution did however require some changes to the system's trust model, as the regular participants of a multiparty conversation now have to put some additional trust in the Call Coordinator handling the call. Such tradeoffs between functionality and security are usually unavoidable though, and in this case, the benefits of using this solution vastly outweighs the downsides.

The alternative solution shown in section 5.2, in which the participants of a multiparty call each set up their own connections to each of their counterparties, could be the better choice for some systems though. This solution would in a way result in a more secure solution than the change pattern that was implemented, as no additional dependencies have to be introduced in the system's trust model. However, this solution would not be feasible to implement in the Cryptify Call system, as each additional connection would require vast amounts of performance resources

in the client devices. A typical smartphone used by the end users of Cryptify Call today would only be able to manage perhaps two or three counterparties using this solution. There are also some other downsides of this solution that would need to be tackled, most notably the risk of ending up with some of the participants of a multiparty call having conflicting views of who is participating in the call, due to a subset of the links between the participants breaking down. For a non-security critical application, this might not be a major issue, but in a security-critical system such as Cryptify Call, some sort of mechanism would need to be implemented to handle such situations.

After having evaluated the evolved version of the system, the Cryptify management was very satisfied with the implementation. This led to the decision to let the implemented code lay as the foundation for the Call Coordinator application that has now been put into production and is about to be released on the market. The most important factor in this decision was that it was possible to keep the system's security requirements basically untouched. The only changes necessary were some small modifications to the system's trust model, as the participants of a multiparty call now must put some amount of additional trust in the Call Coordinator. That it was possible to keep the necessary architectural changes rather minimal is also of great value, as the less changes that have to be made to the code, the less is the risk of introducing new bugs with the change [26]. Also from a non-functional point of view, the results of the implementation are solid. The performance evaluation conducted by the author in section 4.5 shows that it is possible to maintain a 50-party multicall without adding any additional roundtrip delay between the end users, using a MacBook Pro as Call Coordinator. Such a large number of participants is way more than what is necessary in order to support the intended use cases of the conferencing call functionality of the Cryptify Call system. However, should it at some point be necessary to further increase this limit, this can rather easily be achieved either by adding additional hardware resources to the Call Coordinator or by further improving the efficiency of its implementation.

6.2 Other Applications

While the change patterns described in chapter 5 were derived from evolving an identity-based VoIP application, the patterns themselves are not necessarily limited to only voice communication systems. The two identified change patterns can therefore be used to add multiparty support to basically any type of system where multiple parties need to exchange some sort of data in a secure way with each other. Just as long as the initial architecture is based upon a secure 1-to-1 communications paradigm, both of the patterns should be applicable in order to add support for secure multiparty communication to the architecture. Examples could range from online multiplayer games to smartphone applications of any kind that need to evolve from two-party to multiparty communication. These change patterns should also be well-suited to use on Internet of Things applications, where a multitude of devices

with embedded software need to connect to each other and exchange data. Assume for example that someone has a smart refrigerator which can communicate with its owner's smartphone. At some point, it might be necessary to evolve the refrigerator to also let it communicate with other smart devices (such as the freezer or the bathroom scales) connected to the same local area network. This could be achieved by applying either of the two change patterns presented in this thesis. As the Internet of Things is an area which is expected to grow rapidly in the near future [2], many more possible applications of these change patterns will surely present themselves in the years to come.

In order to decide whether a specific software system is suitable for being evolved according to either of the change patterns identified in chapter 5, the roles in the SI* diagram given in the architectural support template in figure 5.2 just need to be mapped to the corresponding components in the software architecture of that system. If it is possible to do so, then that architecture can be evolved according to either of the two given change patterns.

6.3 Future Work

One possible overall direction the future development of the Cryptify Call system could take is to give *all* the different client applications the capability of acting as the Call Coordinator for a single multiparty call session. This would be achieved by applying the architectural solution of the implemented change pattern, as described in section 5.1, to all client platforms, instead of just the desktop application. Doing so would mean that any of those clients would be able to set up and manage an n-party call, with n being constrained by that platform's performance restrictions. The implications of such an approach would be that a normal two-party call would follow the same control flow as a call where a third party is added. The benefits of using such a solution would mainly be to ease future maintenance, as all the clients would be able to share largely the same code and architecture (as opposed to Call Coordinators being implemented differently from the regular clients). Note that a major difference between this approach and the alternative change pattern described in section 5.2 (which was found to be unfeasible for the Cryptify Call system) is that there will still be exactly one client acting as the Call Coordinator for each individual conference call session, but that any of the participating clients can assume that role. Due to the limited performance capabilities available in most smartphones models, this solution might not be ideal however (many common smartphone models available on the market today would not be able to handle multiparty calls with more than three participants). But as long as the performance of new smartphone models continue to increase, this solution will become a more and more attractive alternative in the years to come.

The implementation of the evolved Call Coordinator could also be improved in some areas in order to further increase the overall performance of the system, if

necessary. To start with, a more advanced kind of buffer could be implemented to replace the current circular buffers queuing audio packets sent to and from the Call Coordinator. For example, a jitter buffer could be used to make sure that the incoming audio arriving to the Call Coordinator is processed at regular intervals. A jitter buffer works by dynamically adding and removing delay to the incoming audio stream in order to neutralize any deviations in the time interval between the arriving audio packets, and thus ensuring a smooth playback rate.

6.4 Future Research

Hopefully the findings presented in this master's thesis can be combined with the findings of other studies in order to form a more complete picture of the whole process of software evolution. As more research is performed within this field, change patterns and other principled ways of evolution will probably become more and more important when it comes to improving and optimizing software evolution processes. However, this area still suffers from a lack of research and is as not widely embraced by the industry as it should be, where much evolution is instead done in an ad hoc way. The most obvious area where more work needs to be done when it comes to change patterns is the need to identify more patterns covering different change scenarios. In this, the two change patterns identified in this thesis could help by acting as inspiration, or possibly even as building blocks for more complex change patterns. All change patterns that are identified should ideally also be verified by applying them to real contexts in the industry. Case studies where already identified change patterns are applied to existing systems are also necessary in order to further prove the benefits of using such a principled way of evolution.

Once a sufficiently large number of change patterns covering different change scenarios have been identified, then the technique will hopefully become more widespread in its use. Ideally, all the identified patterns should be put together in some kind of database and made available online to software developers and researchers all around the world. Doing so would both promote and ease the concept of using them, as well as inspire people to contribute with their own change patterns. The pace in which new change patterns are identified could also be increased by creating some sort of framework to aid the process of coming up with patterns to solve specific change scenarios. Such a framework could be something as simple as a guidance and best-practices on how to identify suitable change patterns, or some sort of meta-template to use during the process. Perhaps it will at some point in the future also be possible to develop more complex tools that can be used in order to semi-automate the extraction of new change patterns. Change pattern research could also focus on finding ways to reuse components or whole change patterns in order to save time when identifying solutions to new change scenarios.

There are also many other aspects of security and software evolution that can be explored, such as self-adapting systems and code automatically generated from re-

quirements, two areas not touched upon during this thesis. Both these techniques, as well as others, could also be combined with the concept of change patterns. If it for example would be possible to just apply a change pattern to the software architecture of a system, and that the code would then automatically change in accordance to the new architecture, the potential cost savings on software maintenance would be huge.

6.5 Challenges

Quite a number of steps had to be performed in order to finally identify the two change patterns presented in chapter 5, and some of these steps were more challenging than others. The most time-consuming and difficult step was probably to understand and model the software architecture of the initial Cryptify Call system. The main difficulties in this originated from the fact that no complete documentation of the system architecture existed before the beginning of this project, and neither did any developer at the company have complete knowledge of the whole system. The software architecture also proved to be a quite complex affair, as the original architecture seems to have deteriorated over time in many aspects, probably due to the lack of documentation to rely on for the developers working on evolving the system.

Another challenge during the thesis was to fully grasp the MIKEY-SAKKE protocol. Such detailed knowledge was not necessary in order to just implement the evolved functionality in the system – which actually proved to be a relatively simple and straightforward process once the evolved architecture had been decided upon and modeled – but important to have in order to guarantee that no security risks had been introduced in the evolved architecture.

To translate the evolution performed on the Cryptify Call system into a context-free change pattern was also quite challenging. The main difficulty in this was to know on which level of abstraction to express the change pattern, as well as the pattern for the alternative solution which was not implemented in the Cryptify Call system. Both of the change patterns had to be abstract enough to not just apply to the Cryptify Call system, while still being relevant and useful to anyone interested in using these patterns. It took some work to get this right, but hopefully will the resulting change patterns be seen as simple enough to map to some arbitrary software system, while at the same time having detailed enough architectural-level transformations to actually help while evolving the architecture of that system.

One step that went surprisingly easy was the actual implementation of the evolved architecture in the Cryptify Call system. Even if quite some time had to be spent on the implementation part of the evolution, it was possible to carry it through without encountering any major issues. This was probably made possible through the thorough work spent on identifying different possible architectural solutions and then deciding upon one of them in collaboration with the company. However, quite

some effort and creativity was needed in order to optimize the code and thereby increase the number of participants that a single conference session can support. These optimizations were solely carried out in low-level details of the code, and did not affect the higher-level software architecture itself.

6.6 Conclusions

In order to answer the research questions specified in section 1.2 and find one or more change patterns to evolve a system from secure one-to-one into secure many-to-many communication, a suitable system had to be studied in that context. The system chosen for the task was Cryptify Call, a VoIP application used for making secure phone calls, which was evolved by adding support for secure telephone conferences. During the evolution process, two different possible architectural solutions were identified and later expressed as precise change patterns by the author of this thesis.

In order to both perform the evolution and to define the two change patterns, it was necessary to first model the software architecture of the Cryptify Call system. As no complete documentation of the system's software architecture existed at the beginning of the thesis project, the modelling had to be done mainly through reverse engineering and by interviewing senior programmers at the company. Two promising approaches – which in the end led to the two change patterns – on how to evolve the architecture were identified early on during the process. As the necessary changes to be made in both of these approaches mainly concerned the client-side of the system, most focus was spent on modelling that part of the software architecture. In addition, the system's trust model and most important security requirements – to offer authentication and confidentiality – were modeled using SI* notation. All of this modelling work was carried out by the author of the thesis alone, and then validated by experts at the company.

The change scenarios corresponding to the two change patterns were both based upon the changes made to the system's trust model and data flow. In order to define a precise solution to each of these change scenarios, the most important roles in the initial as well as the evolved software architecture for each solution had to be identified. The architectural changes in the two change patterns were then modeled by the author as architectural-level transformations, including an automatic transformation expressed in the QVT Operational language for each of the change patterns. In order to apply one of the change patterns to some other software system in the future, this process simply has to be reversed by mapping the architectural roles given by the change pattern to the corresponding components in the software architecture of the system being evolved and then follow the steps in the given transformation. A set of architectural models are given in chapter 5 in order to aid the software architect during this transformation.

The solution that was chosen (by the author of the thesis in collaboration with the managers at Cryptify AB) to be implemented in the Cryptify Call system was an

architecture where one of the clients in a multiparty call acts as the coordinator for that call. The responsibilities of this Call Coordinator are to authorize all the participants of a multiparty call and to handle all exchange of data between them. The Call Coordinator therefore needs to be trusted by the regular participants in order to fulfill these obligations and not to misuse its role in any way. The only change necessary to a system's trust model in order to apply this change pattern is to add those trust relationships between the Call Coordinator and the regular participants of a multiparty call. Apart from this, none of the system's security requirements have to be modified. The main benefit with using this change pattern is that the only part of the system's software architecture that will require any changes is the client application that will be evolved into assuming the Call Coordinator role.

The alternative solution, which was not implemented, instead works by letting each participant set up a separate connection with each of its counterparties in the conversation. This alternative solution is technically more challenging but does not require any changes at all to the system's security requirements. Therefore this solution could be the better choice for some systems, but it would not be possible to implement in the Cryptify Call system due to performance limitations in the system's client applications.

Both of the identified change patterns can hopefully be applied in other contexts as well. The most obvious use would be in similar VoIP applications, but the change patterns are equally valid to any type of system built with a suitable software architecture. Hopefully will this thesis also inspire further research in the same direction, perhaps by letting the two identified change patterns be used as building blocks for more complex change patterns.

Bibliography

[1] R. Anderson, “Security engineering: A guide to building dependable distributed systems”, in, Second Edition. John Wiley & Sons, 2008, ch. 26, pp. 857–887.

[2] L. Atzori, A. Iera, and G. Morabito, “The internet of things: A survey”, *Computer Networks*, vol. 54, pp. 2787–2805, 15 2010.

[3] J. Barnes, “Software architecture evolution”, PhD thesis, Pittsburgh: Carnegie Mellon University School of Computer Science (Institute for Software Research), 2013.

[4] D. Basin, M. Clavel, J. Doser, and M. Egea, “Automated analysis of security-design models”, *Information and Software Technology*, vol. 51, pp. 815–831, 5 2009.

[5] D. Basin, J. Doser, and T. Lodderstedt, “Model driven security: From uml models to access control infrastructures”, *ACM Transactions on Software Engineering and Methodology*, vol. 15, pp. 39–91, 1 2006.

[6] R. Baskerville, “Investigating information systems with action research”, *Communication of the Association for Information Systems*, vol. 2, 1999.

[7] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice. Third Edition*, ser. SEI Series in Software Engineering. Addison-Wesley, 2013.

[8] G. Bergmann, F. Massacci, F. Paci, and T. T. Tun, “A tool for managing evolving security requirements”, in *CAiSE Forum 2011, LNBIP 107*, Springer-Verlag, 2011, pp. 110–125.

[9] H. P. Breivold, I. Crnkovic, and M. Larsson, “A systematic review of software architecture evolution research”, *Information and Software Technology*, vol. 54, pp. 16–40, 1 2012.

[10] CESG, *Cpa security characteristic: Secure voip client v2.0*, Feb. 2013. [Online]. Available: `http://www.cesg.gov.uk/publications/Documents/secure_VoIP_client_2-0.pdf`.

[11] P. Checkland and S. Holwell, “Action research: Its nature and validity”, in *Information Systems Action Research: An Applied View of Emerging Concepts and Methods*, N. Kock, Ed., New York: Springer, 2007, pp. 3–17.

[12] Cryptify, “Conference call: Pre-study on a mikey-sakke based conference call solution”, Tech. Rep., 2013.

[13] R. Davison, M. G. Martinsons, and N. Kock, “Principles of canonical action research”, *Information Systems Journal*, vol. 14, no. 1, pp. 65–86, 2004, ISSN: 1365-2575. DOI: 10.1111/j.1365-2575.2004.00162.x.

[14] A. van Deursen, E. Visser, and J. Warmer, “Model-driven software evolution: A research agenda”, Delft University of Technology, Software Engineering Research Group, Tech. Rep., 2007.

[15] B. Dick, “Postgraduate programs using action research”, *The Learning Organization*, vol. 4, pp. 159–170, 9 2002.

[16] *Doxygen*. [Online]. Available: http://www.doxygen.org/.

[17] R. Dvorak, “Model transformation with operational qvt”, in *EclipseCon 2008*, 2008.

[18] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, “Selecting empirical methods for software engineering research”, in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. Sjøberg, Eds., Springer London, 2008, pp. 285–311, ISBN: 978-1-84800-043-8. DOI: 10.1007/978-1-84800-044-5_11.

[19] S. Enderby and Z. Baracskai, “Harmonic instability of digital soft clipping algorithms”, in *Proceedings of the 15th Int. Conference on Digital Audio Effects*, 2012.

[20] N. Ernst, A. Borgida, and J. Mylopoulos, “Requirements evolution drives software evolution”, in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, 2011, pp. 16–20.

[21] N. A. Ernst, J. Mylopoulos, and Y. Wang, “Requirements evolution and what (research) to do about it”, in *Design Requirements Workshop, LNBIP 14*, Springer-Verlag, 2009, pp. 186–214.

[22] M. Felderer, “Evolution of security engineering artifacts: A state of the art survey”, *International journal of secure software engineering*, vol. 5, pp. 48–, 4 2014.

[23] D. Garlan, J. Barnes, B. Schmerl, and O. Celiku, “Evolution styles: Foundation and tool support for software architecture evolution”, in *2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on Software Architecture*, IEEE, 2009, pp. 131–140.

[24] M. Groves, “Mikey-sakke: Sakai-kasahara key encryption in multimedia internet keying (mikey)”, Tech. Rep., Feb. 2012. [Online]. Available: http://tools.ietf.org/html/rfc6509#section-2.2.

[25] C. B. Haley, R. Laney, J. D. Moffett, and B. Nuseibeh, “Security requirements engineering: A framework for representation and analysis”, *IEEE Transaction of Software Engineering*, vol. 34, pp. 133–153, 1 2008.

[26] C. Jones, *Software Quality - Analysis and Guidelines for Success*. Boston, Massachusetts: International Thomson Computer Press, 1997.

[27] J. Jürjens, *Secure Systems Development with UML*. Springer, 2005.

[28] ——, “Automated security hardening for evolving uml models”, in *ICSE’11*, ACM, 2011, pp. 986–988.

[29] N. Kock, “The three threats of action research: A discussion of methodological antidotes in the context of an information systems study”, *Decision Support Systems*, vol. 37, no. 2, pp. 265–286, 2004.

[30] D. LeBlanc and M. Howard, “Writing secure code”, in, Second Edition. Microsoft Press, 2002, ch. 4.

[31] M. Lehman, “On understanding laws, evolution, and conservation in the large-program life cycle”, *Journal of Systems and Software 1*, pp. 213–221, 1980.

[32] F. Massacci, J. Mylopoulos, and N. Zannone, “Security requirements engineering: The si* modeling language and the secure tropos methodology”, in *Advances in Intelligent Information Systems*, ser. Studies in Computational Intelligence, Z. Ras and L.-S. Tsay, Eds., vol. 265, Springer Berlin Heidelberg, 2010, pp. 147–174, ISBN: 978-3-642-05182-1. DOI: 10.1007/978-3-642-05183-8_6. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-05183-8_6.

[33] T. Mens, S. Demeyer, M. Wermelinger, R. Hirschfeld, S. Ducasse, and M. Jazayeri, “Challenges in software evolution”, in *IEEE Eighth International Workshop on Principles of Software Evolution*, 2005, pp. 13–22.

[34] T. Mens and S. Demeyer, *Software Evolution*. Springer-Verlag, 2008.

[35] *Meta object facility (mof) 2.0 query/view/transformation specification*, Version 1.2, Object Management Group (OMG), Feb. 2015.

[36] D. Mitropolous, G. Gousios, and D. Spinellis, “Measuring the occurrence of security-related bugs through software evolution”, in *16th Panhellenic Conference on Informatics*, 2012, pp. 117–122.

[37] L. Montrieux, M. Wermelinger, and Y. Yu, “Challenges in model-based evolution and merging of access control policies”, in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, ACM, 2011, pp. 116–120.

[38] A. Nhlabatsi, B. Nuseibeh, and Y. Yu, “Security requirements engineering for evolving software systems: A survey”, *International journal of secure software engineering*, vol. 1, pp. 51–, 1 2010.

[39] Q. Ni, E. Bertino, J. Lobo, C. Brodie, C. Karat, J. Karat, and A. Trombeta, “Privacy-aware role-based access control”, *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, pp. 1–31, 3 2010.

[40] R. Olimid, “About the key escrow properties of identity based encryption schemes”, *Journal of mobile, embedded and distributed systems*, 2012.

[41] S. S. Al-Riyami and K. G. Paterson, “Certificateless public key cryptography”, 2003.

[42] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering", *Empirical Software Engineering*, vol. 14, pp. 131–164, 2 2009.

[43] A. Shamir, "Identity-based cryptosystems and signature schemes", in *Advances in Cryptology*, ser. Lecture Notes in Computer Science, G. Blakley and D. Chaum, Eds., vol. 196, Springer Berlin Heidelberg, 1985, pp. 47–53, ISBN: 978-3-540-15658-1. DOI: 10.1007/3-540-39568-7_5. [Online]. Available: http://dx.doi.org/10.1007/3-540-39568-7_5.

[44] V. T. Toth, *Mixing digital audio*, Aug. 2000. [Online]. Available: http://www.vttoth.com/CMS/technical-notes/68-mixing-digital-audio.

[45] T. T. Tun, Y. Yu, C. Haley, and B. Nusibeh, "Model-based argument analysis for evolving security requirements", in *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, IEEE, 2010, pp. 88–97.

[46] S. Wenzel, D. Poggenpohl, J. Jürjens, and M. Ochoa, "Specifying model changes with umlchange to support security verification of potential evolution", *Computer Standards and Interfaces*, vol. 36, pp. 776–791, 2014.

[47] T. Wood-Harper, "Research methods in information systems: Using action research", in *Research Methods in Information Systems*, E. Mumford, R. Hirschheim, G. Fitzgerald, and T. Wood-Harper, Eds., Amsterdam: North-Holland, 1985, pp. 169–191.

[48] D. T. Yeh, "Digital implementation of musical distortion circuits by analysis and simulation", PhD thesis, Stanford University (Department of Electrical Engineering), 2009.

[49] C. Youngblood, "An introduction to identity-based cryptography", 2005.

[50] K. Yskout, R. Scandariato, and W. Joosen, *Change patterns catalog*, 2011. [Online]. Available: https://distrinet.cs.kuleuven.be/software/changepatterns/additional/catalog.pdf.

[51] ——, "Change patterns: Co-evolving requirements and architecture", *Software and Systems Modeling*, vol. 13, no. 1, Feb. 2014.

[52] E. S.-K. Yu, "Modelling strategic relationships for process reengineering", PhD thesis, University of Toronto (Department of Computer Science), 1995.

[53] Y. Yu, T. T. Tun, A. Tedeschi, V. N. L. Franqueira, and B. Nuseibeh, "Openargue: Supporting argumentation to evolve secure software systems", in *2011 19th IEEE International Requirements Engineering Conference*, IEEE, 2011, pp. 351–352.

A

Different Merging Algorithms

There are plenty of algorithms that can be used to merge multiple audio signals together, each one with its own benefits and drawbacks [19]. A couple of different such algorithms were considered to be used in the evolved Cryptify Call system instead of the simple clipping algorithm described in section 4.4.1, but in the end it was decided that the clipping algorithm worked sufficiently good to use even in conversations with a large number of participants. The reason for this is that the PCM audio format used in the Cryptify Call system includes a large enough amplitude headroom in the audio packets to ensure that clipping very rarely has to occur. During testing, clipping was found to occur only when a large number of speakers were speaking simultaneously. In the case of a telephone conference, it will in any case be impossible to discern a single voice if too many people are talking at the same time, regardless of what merging algorithm is being used, and therefore the clipping algorithm was decided to be sufficient for the Cryptify Call system for the time being. Some of the other algorithms that were considered during the evolution process are discussed below however, as these algorithms might be of interest to use in a system where clipping would occur more frequently when using the clipping algorithm, or perhaps for merging some other kind of data altogether.

The implemented clipping algorithm is expressed as a function of time in equation A.1, for comparison. Z_{min} and Z_{max} in the equation represent the smallest and largest PCM values allowed, $A_1 \dots A_n$ represent the individual audio streams sent by the participants and Z represents the final merged signal. The graph in figure A.1 shows the clipping function being used to merge two sound waves (the dashed lines in the graph) with different frequencies and amplitudes. The combined sound wave is shown by the thicker line in the graph. Clipping occurs at several locations in the merged sound wave, but in a real conversation this will rarely be the case due to the large amplitude overhead available in the PCM specification used by the Cryptify Call system.

$$Z(t) = min(Z_{max}, max(Z_{min}, A_1(t) + A_2(t) + \cdots + A_n(t))) \tag{A.1}$$

One of the alternatives that was tried out instead of using the clipping algorithm, was to use linear attenuation. Linear attenuation simply works by adding together all the separate signals and then dividing the result by the number of signals. This

algorithm will always yield a result within the allowed range, but the amplitude of the merged sound wave resulting from this algorithm is directly correlated to the number of participants, regardless of how many of them that are currently speaking, and thus the resulting audio would become more and more silent as more participants connect to the call. Therefore this algorithm would not work well in Cryptify Call or some other VoIP system, as the number of simultaneous speakers is usually limited to just one or two, regardless of the total number of participants in the call. The linear attenuation algorithm is expressed mathematically in equation A.2 and an example of its use is shown in the graph in figure A.2. Note that the resulting sound wave is free from distortion or clipping, but that it does not utilize the full amplitude range available, resulting in a loss of volume.

One way of making the linear attenuation algorithm more useful for a VoIP application is to first identify and discard any silent sound packets, before running the algorithm only on the remaining packets. This way, the loss of volume would normally not be as discernable as when dividing by the total number of participants. Linear attenuation could also be of interest to use in applications where it is some other kind of data than audio that needs to be merged.

$$Z(t) = \frac{A_1(t) + A_2(t) + \cdots + A_n(t)}{n} \tag{A.2}$$

Yet another alternative to the clipping algorithm is to use a method proposed by Viktor T. Toth [44]. In this method, each PCM value in the merged signal is calculated by taking the union of the values of the corresponding PCM samples in the signals to be merged. In order for this algorithm to work though, each of the samples first have to be normalized to a value ranging between 0 and 1. This is done by dividing all the values with the largest allowed PCM value (Z_{max}). After the union of all the signals thereafter has been calculated, getting the final result is achieved by multiplying back Z_{max}. This algorithm is expressed mathematically in equation A.3 and an example is shown in the graph in figure A.3. The merged audio wave clearly utilizes the full amplitude range, but the sound wave looks to be distorted at some points. This distortion means that the algorithm is not very suitable for merging voice audio data, but could possibly work better for merging other types of data.

$$Z(t) = Z_{max} \cdot |\bigcup_{i=1}^{n} \frac{A_i(t)}{Z_{max}}| \tag{A.3}$$

A better solution could possibly be to instead calculate the hyperbolic tangent of the combined sound waves, using the tanh function [48]. Just as when using Viktor Toth's method, the values need to be normalized, but the result is guaranteed to be distortion-free while at the same time utilizing most of the available amplitude range. This function would therefore be the best alternative for a VoIP application where the clipping algorithm would clip the merged audio stream too frequently.

The algorithm is expressed mathematically in equation A.4, and an example of its use is visualized in the graph in figure A.4.

$$Z(t) = Z_{max} \cdot \tanh(\frac{A_1(t) + A_2(t) + \cdots + A_n(t)}{Z_{max}}) \tag{A.4}$$

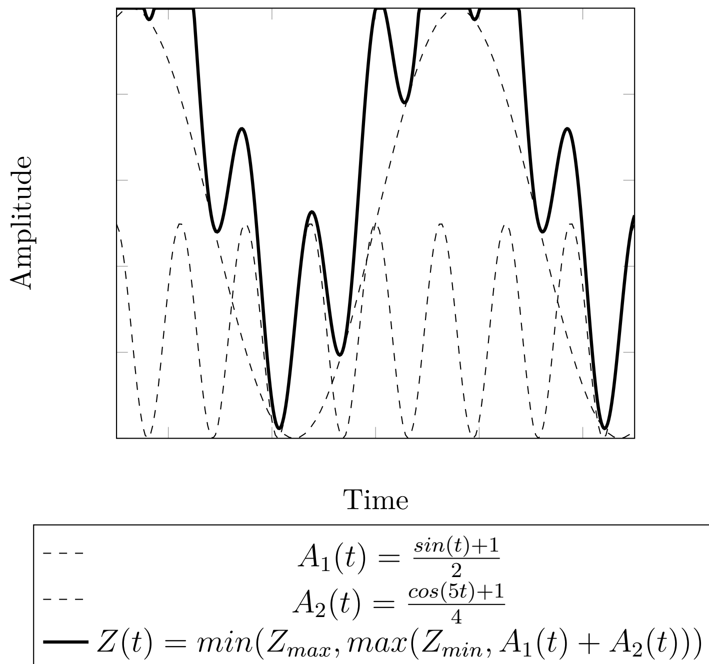


Figure A.1: The clipping function. Every time the merged audio signal ($Z(t)$) would go outside of the allowed PCM values, the signal is simply clipped to the minimum or maximum value allowed by the PCM specification respectively.

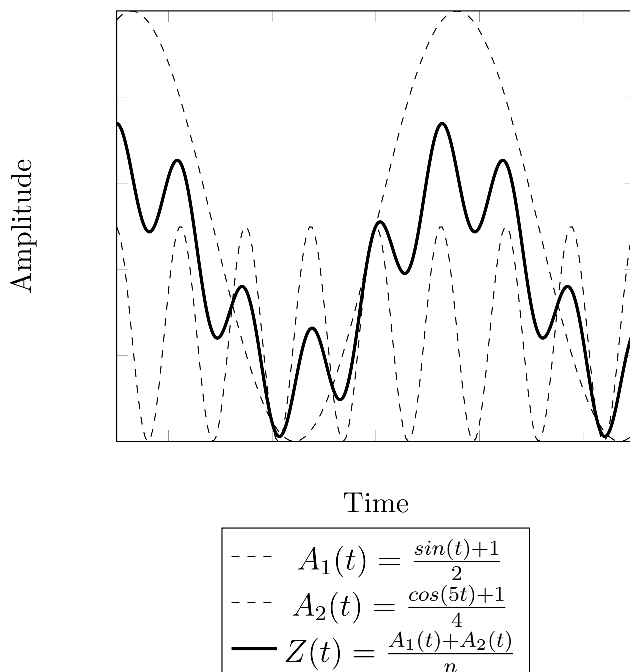


Figure A.2: Linear attenuation. The merged audio signal is guaranteed to be distortion free, but as the amplitude of the signal corresponds negatively to the number of speakers in the conversation, the volume of the resulting signal would become very low in a conversation with many participants.

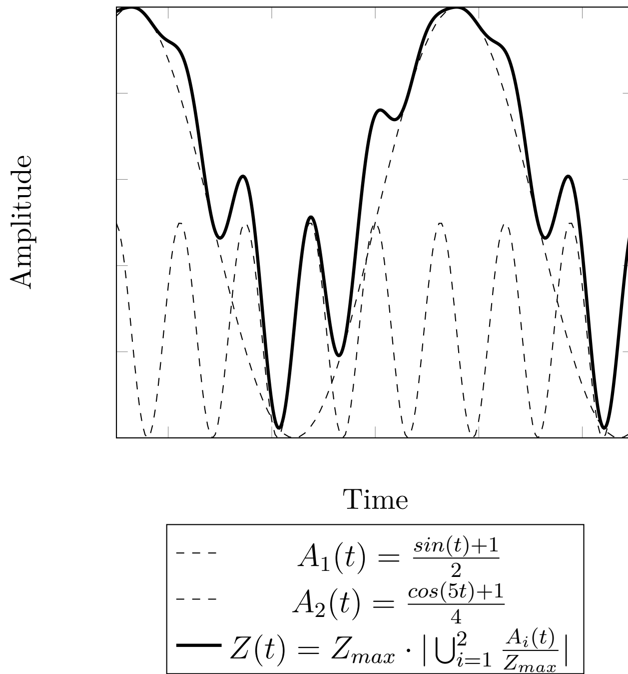


Figure A.3: Viktor Toth's method. This more complex algorithm utilizes the full amplitude range, but the merged sound wave will be distorted at some points.

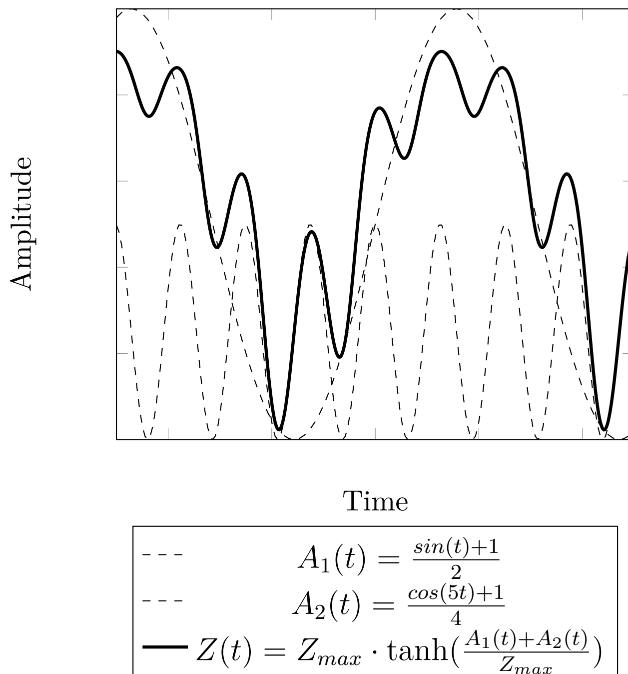


Figure A.4: The hyperbolic tangent (tanh) function. This algorithm does not utilize the full amplitude range, but it is guaranteed to be distortion free, and should therefore be a favorable alternative for applications where the simple clipping function does not work very well.