University of Bremen

Department of Mathematics and Computer Science

New concept of the Android keystore service with regard to security and portability.

Master's Thesis

by

Fritjof Bornebusch

reviewed by

Dr. Karsten Sohr

Prof. Dr. Jan Peleska

supervised by

Dipl. Inf. Florian Junge

2016

Confirmation

I hereby confirm that I wrote this master thesis on my own and that I have used only the indicated references, resources, and aids.

In German:
Hiermit bestätige ich, dass ich die vorliegende Masterthesis selbstständig verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bremen, 1/13/2016 Fritjof Bornebusch

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

– Martin Fowler –

Bornebusch, Fritjof
New concept of the Android keystore service with regard to security and portability.
Master's thesis, Department 3 - Mathematics / Computer Science
University of Bremen, 2015

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0). To view a copy of this license, send an email to info@creativecommons.org, visit http://creativecommons.org/licenses/by-nc-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, California, 94042, USA.

Table of Contents

Acknowledgements **7**

List of Figures **9**

List of Listings **10**

Acronyms **13**

Glossary **15**

1 Introduction **19**

2 Background **24**

2.1 Android System Architecture 24

2.1.1 Security-Enhanced Linux 28

2.1.2 Capabilities 31

2.2 Memory Vulnerabilities 32

2.2.1 Buffer Overflow Protection 33

2.2.2 Dead Store Elimination 36

2.3 Encryption 38

2.3.1 Symmetric-Key Algorithm 38

2.3.2 Asymmetric-Key Algorithm 39

2.3.3 Mode of Operation 40

2.4 Hashing 41

2.4.1 Secure Hash Algorithm 42

2.4.2 Passphrase 42

2.5 Signature 43

3 Analysis of the Android keystore service 45
3.1 Design . . . 46
3.1.1 The Android RPC subsystem . . . 47
3.1.2 Architecture of the Android keystore service . . . 48
3.2 Application Programming Interface . . . 57
3.2.1 Java . . . 57
3.2.2 C++ . . . 58
3.3 Implementation . . . 59
3.3.1 Coding Style . . . 59
3.3.2 Library Functions . . . 60
3.3.3 SELinux . . . 65
3.3.4 Compiler Flags . . . 67
3.3.5 Logging . . . 68
3.3.6 Folder structure . . . 71
3.4 Init subsystem . . . 72
3.5 Implemented hashing algorithm . . . 74
3.5.1 Master key generation . . . 77
3.6 Implemented encryption algorithm . . . 79
3.7 Result . . . 81

4 Implementation of the new keystore service 84
4.1 Design . . . 85
4.2 Implementation . . . 85
4.2.1 Compiler Flags . . . 87
4.2.2 Folder structure . . . 90
4.2.3 Application Programming Interface . . . 91
4.3 Security . . . 91
4.3.1 Hashing . . . 92
4.3.2 Encryption . . . 93
4.4 Portability . . . 97
4.4.1 Design . . . 98
4.4.2 Privilege Separation . . . 99
4.4.3 Implementation . . . 104
4.5 Result . . . 112

5 Evaluation **114**
5.1 Android . . . 114
5.1.1 Testing . . . 115
5.2 Portable version . . . 116
5.2.1 Testing . . . 118
5.3 Result . . . 119

6 Conclusion and Future Work **120**
6.1 Conclusion . . . 120
6.2 Future Work . . . 123

Bibliography **125**

Appendices **129**

A Static analysing logs **131**

B keystore_cli(1) usage **133**

C ks(1) usage **134**

D Android keystore compiler flags **135**

E Portable keystore compiler flags **136**

F Structures for sending and receiving **138**

G Class diagram **140**

Acknowledgements

Thank you Prof. Dr. Jan Peleska, Dr. Karsten Sohr, and Florian Junge for making this thesis possible with your believing in the relevance of this topic.

This document is my own work, but it would not be in this shape without the help of friends. They spotted weak arguments, linguistic and technical errors, as well as poor sentences and made this thesis comprehensive and focused. Without their help it would not be in this shape.

Aaron Lye
: Thank you for your comments, for the discussions about the technical aspects of this document, and for checking the bibliography.

Christina Plump
: Thank your for your comments, linguistic corrections, and the discussions about weak argumentations.

Max Nitze
: Thank you for your comments, and the discussions about the technical aspects of this document.

Kai Hillmann
: Thank you for the discussions about the structure of this document.

Edina Alessa Staschewski
: Thank you for supporting me in my lazy times and pushing me back to work.

At last, a big thank you to those who are not listed by name here for supporting me when I was busy. This helped me a lot to finish my study by writing this master thesis.

List of Figures

2.1 Android system architecture 25
2.2 Android intents 27
2.3 Stack layout 34
2.4 Stack layout with canary 34
2.5 Time to brute-force 128 bit AES key 39
2.6 Electronic Codebook mode 40
2.7 Cipher Block Chaining mode 41
2.8 Keyed-Hash Message Authentication Code 43

3.1 Android keystore service design 46
3.2 The Android RPC subsystem 47
3.3 Android keystore service class diagram 49
3.4 MD5 value hashing 76
3.5 Original plain text encryption 81
3.6 *scan-build(1)* errors 82
3.7 *gcc(1)* flag errors 82

4.1 Portable keystore service design 98
4.2 Design of the portable version 99

G.1 Keystore transactions 140
G.2 Class dependency diagram 141
G.3 Class keystore 142
G.4 Classes blob and entropy 143
G.5 Class userstate 144

List of Listings

2.1 Dead store elimination example . . . 36
2.2 Dead store elimination without optimizations . . . 37
2.3 Dead store elimination with optimizations . . . 37
3.1 *BnKeystoreService*: onTransaction method implementation . . . 51
3.2 *grant_t* type . . . 52
3.3 Register keystore proxy . . . 53
3.4 *BpKeystoreService*: test method implementation in keystore service . . . 54
3.5 Accessing *BpKeystoreService* from the keystore client . . . 54
3.6 Original blob structure . . . 56
3.7 *memcpy(3)* function call . . . 60
3.8 *malloc(3)* function call . . . 61
3.9 *malloc(3)* function call in keystore service . . . 61
3.10 *atoi(3)* function call . . . 62
3.11 *atoi(3)* function call no. 1 . . . 63
3.12 *atoi(3)* function call no. 2 . . . 63
3.13 *strtol(3)* function call . . . 64
3.14 Convert string to int safely . . . 64
3.15 SELinux permissions . . . 65
3.16 SELinux users . . . 66
3.17 *has_permission* function . . . 67
3.18 Structure original implementation . . . 71
3.19 Keystore *init.rc* entry . . . 73
3.20 Original blob hashing and encryption structure . . . 75
3.21 *md5(3)* API . . . 75
3.22 Original blob hashing implementation . . . 76
3.23 Original blob compare hash implementation . . . 77
3.24 Master key generation . . . 78

3.25 Storage of the master key salt 78
3.26 Original blob encryption implementation 79
3.27 Original blob decryption implementation 80
4.1 New blob structure 86
4.2 New logging API 86
4.3 File permission check 87
4.4 Folder structure of new Android implementation 90
4.5 *EVP_DigestInit_ex(3)* function call 92
4.6 *EVP_DigestUpdate(3)* function call 93
4.7 *EVP_DigestFinal_ex(3)* function call 93
4.8 *EVP_MD_CTX_destroy(3)* function call 93
4.9 EVP_EncryptInit_ex(3) function call 94
4.10 EVP_EncryptUpdate(3) function call 95
4.11 *EVP_EncryptFinal_ex(3)* function call 95
4.12 *EVP_Cipher_CTX_cleanup(3)* function call 95
4.13 *EVP_DecryptInit_ex(3)* function call 96
4.14 *EVP_DecryptUpdate(3)* function call 96
4.15 *EVP_DecryptFinal_ex(3)* function call 96
4.16 *fork(2)* function call 101
4.17 Changing the root directory 101
4.18 Dropping privileges 102
4.19 Changing the process name 103
4.20 *send(2)* and *recv(2)* function calls 104
4.21 *socketpair(2)* function call 104
4.22 *send(2)* and *recv(2)* function calls 105
4.23 *sendmsg(2)* function call 106
4.24 *recvmsg(2)* function call 106
4.25 *signal(3)* function call 107
4.26 Structure of the portable version 108
5.1 Evaluation of the new Android version 115
5.2 Regression tests new Android version 116
5.3 Evaluation of the portable version 117
5.4 Regression tests portable version 118
A.1 *clang(1)* output 131
A.2 Keystore compiler flags output 132

B.1 *keystore_cli(1)* usage 133
C.1 *ks(1)* usage 134
D.1 Android keystore compiler flags 135
E.1 Portable keystore server makefile 136
E.2 Portable keystore client makefile 137
F.1 Structures and enumerations for sending and receiving 138

Acronyms

AES Advanced Encryption Standard 11, 28–30, 62, 65, 69–72, 84, 87, 102, 111

API Application Programming Interface 10–12, 14, 16, 18, 35–37, 43, 47–49, 53, 55–57, 59–62, 65, 66, 71, 73–76, 81–83, 87–90, 93, 99–105, 107, 111, 112

ART Android Runtime 10, 14, 15

ASLR Address Space Layout Randomization 25, 26

CBC Cipher Block Chaining 31

CLI command-line interface 45, 56, 62, 74, 89, 101, 104

DAC Discretionary Access Control 18

DEC Dead Store Elimination 27

DSA Digital Signature Algorithm 34, 45, 62, 100

ECB Electronic Codebook 30

ECDSA Elliptic Curve Digital Signature Algorithm 34

EdDSA Edwards-curve Digital Signature Algorithm 34

FLASK Flux Advanced Security Kernel 18

GUI Graphical User Interface 15, 16, 89

HMAC Keyed-Hash Message Authentication Code 33

IPC Inter-Process Communication 37, 38, 40, 43

JNI Java Native Interface 15

JVM Java Virtual Machine 11

MAC Mandatory Access Control 18

MD5 Message-Digest Algorithm 5 11, 33, 64, 70, 72, 74, 75, 82, 111

NDK Native Development Kit 10

NIST National Institute of Standards and Technology 28

NSA National Security Agency 18, 32, 34

RHEL Red Hat Enterprise Linux 19

RIPEMD RACE Integrity Primitives Evaluation Message Digest 32

RPC Remote Procedure Call 37–40, 45, 71, 111

RSA Rivest-Shamir-Adleman cryptosystem 29, 30, 45, 100

SELinux Security-Enhanced Linux 18–20, 22, 35, 53, 55, 56, 62, 75, 81, 102, 107

SHA Secure Hash Algorithm 32, 33, 64, 66, 68, 75, 82, 102, 111

SSH Secure Shell 100, 101

VM Virtual Machine 10

VPN Virtual Private Network 11, 36, 41

Wi-Fi Wireless Fidelity 11, 36, 41

Glossary

ARM Acorn RISC Machine (ARM) is a family of RISC processors developed by ARM Holdings®. ARM processors are designed for light, portable, battery-powered devices, such as smart phones, tablets, and embedded systems. 25

BSD Berkeley Software Distribution (BSD) is a Unix operation system distributed by the Computer Systems Research Group (CSRG) of the University of California, Berkeley. It is a branch of the original Unix known as Berkeley Unix. Unix was originally developed by by Ken Thompson, Dennis Ritchie, and others in the 1970s at the Bell Labs research centre. BSD is the basis for all modern BSD derivatives, like FreeBSD, OpenBSD, NetBSD, and DragonflyBSD. 11, 18, 56, 75, 90, 92, 105, 107, 108, 113, 114

C programming language C is a general purpose and imperative programming language originally developed by Dennis Ritchie at AT&T Bell Labs and first appeared in 1972. It has a low-level access to memory and uses manual memory management. For this reason it is used for implementing operating systems, e.g. BSD or GNU/Linux. 11, 12, 14, 15, 22, 23, 25, 35, 47, 49, 57, 58, 60, 61, 65, 66, 71, 72, 74, 76–78, 80, 81, 90, 94–96, 102, 111, 112

C++ programming language C++ (C plus plus) is an extension of the C programming language. It has imperative, object-oriented and generic programming features and shares the low-level memory access with C. C++ was original developed by Bjarne Stroustrup an first appeared in 1983. As it is a superset of C such combination is often written as C/C++. 11, 14, 15, 35, 45, 47–49, 58, 61, 71, 72, 74, 76–78, 80, 81, 94, 95, 102, 111, 112

DragonflyBSD DragonflyBSD is a free Unix-like operating system and an FreeBSD offspring. It is known for the 64 bit cluster file system named HAMMER. 114

Elliptic curve cryptography Elliptic curve cryptography (ECC) is a public-key cryptography approach based on elliptic curves over finite fields. This approach allows a smaller key size at the same security level compared to the discrete logarithm cryptography, e.g. used by DSA. 34

FreeBSD FreeBSD is a free Unix-like operating system with a focus on networking and storage. It is well known as a server operating systems for different purposes and implements the FreeBSD Jails to provide a system-level virtualisation. The ZFS file system is implemented by FreeBSD as well, which is designed for large storage environments. 114

GNU/Linux Linux is a Unix-like operating system that contains the Linux kernel initially conceived and created by Linus Torvalds in 1991. The Linux system contains the kernel itself as well as additional tools, e.g. for adding and manipulating files, known as the GNU tools. For this reason the name GNU/Linux is suggested by the GNU project, which originally developed such tools. 11, 18, 19, 21, 22, 37, 49, 56, 75, 90, 92, 93, 105, 107, 108, 112–114

Inode An inode is a data structure that represents a file system object in Unix-style file systems. Each inode contains the disk block location as well as various attributes. These attributes include the file owner, the file permission and meta data, like change , modify, and access time. 77

ISC license This license is a free software license written by the Internet Software Consortium (ISC). It is the preferred license of the OpenBSD project as well as related projects, such as OpenSSH or OpenSMTPD. 98

Makefile A Makefile is a text file written in a certain prescribed syntax and parsed by the make(1) utility. Such files are used to compile and link programs written in C/C++. 61, 78, 97, 98

Man page A manual page (man page) is an online software documentation on Unix or Unix-like operating systems. It is split into different sections, such as system calls (2), system administration commands and daemons (8), or general commands (1). To refer to a specific man page the following format is used: <name>(<section number>). The man(1) command shows a man page: man <section number> <name>. 78, 87, 99, 111, 112

NetBSD NetBSD is a free Unix-like operating system with a focus on portability, code clarity, and careful design across many platforms. Its motto is: "Of course it runs NetBSD". It is the most ported operating system in the world. 114

OpenBSD OpenBSD is a free Unix-like operating system with a focus on security and code quality as well as good documentation of its functionality. To achieve more security, OpenBSD includes additional security features, such as memory protection and privilege separation, and has a tradition of comprehensive code audits. Its motto is: "Secure by default" 25, 49, 74, 75, 93, 108, 110, 112, 114

OpenSMTPD OpenSMTPD is an email server developed by OpenBSD developers and is part of its base system. Its principal goals are being as secure as possible and providing an easy to understand configuration language. 75, 114

OpenSSH OpenSSH is a free implementation of the Secure Shell (SSH) protocol original developed by Tatu Yloenen. This protocol establishes an encrypted connection between two computers for remote control and is a secure replace for *telnet(1)*. Its principal goals are being as secure as possible by using state-of-the-art encryption and signature algorithms. OpenSSH is developed and maintained by OpenBSD developers. 75, 101, 103, 114

OpenSSL OpenSSL is an open-source software library. It contains implementations for SSL and TLS protocols. OpenSSL's core is written in the C programming language and implements basic cryptographic functions, e.g. for the AES encryption algorithm or the SHA hashing algorithm. 28, 67–72, 75, 82, 84, 85, 102, 111

Regression test Regression testing is a type of software testing that seeks bugs after changes have been made. Such tests execute functional or non-functional areas of an application (mostly automatically) and compare the result with the expected one defined by the programmer or software tester. If such a test fails. the developer is notified. 12, 104, 109, 114

Socket A socket connects two processes to share data between them. Each side of a socket is used for one process. A well known socket implementation are network sockets which connect a client with a server over a network using protocols like HTTP or SMTP. Another implementation are Unix domain sockets. Such sockets are used for local system communication. 41

Unix file permissions File permissions under GNU/Linux and BSD are split into 1 (execute), 2 (write) and 4 (read) for the file owner, the file owner group and others. For example, the permission 640 means the user has read-write (6), but the group has only read (4) permissions. Other users or groups are not allowed to access the file (0). Different permissions can be created by just add the basic ones. 77

User space The user space or userland is separated from the kernel space in modern operating systems. The kernel space contains the privileged operating system's kernel to manage the underlying hardware, while the user space contains the unprivileged application software. Primarily, this separation provides memory protection to protect data and functionality. In general, the user space refers to all applications or libraries that run outside the operating system's kernel. 22

Chapter 1

Introduction

Login credentials, like user name and password combinations or certificates, will become more important these days, since there are more services available which can be used with different mobile devices. Furthermore, it will be more important to use passwords of a sufficient length in order to achieve greater security as it is more difficult to guess them. One the one hand such passwords must be used. On the other hand they must be stored safely inside a system as well. If a system was compromised and all passwords were stored in plain text, a random password would not help any more. This is why encrypted or hashed storage of passwords are a crucial component of a password management system. Encrypted storage of them is necessary if the user needs the password in plain text on the client side, e.g. to login into a web application or mailbox, while hashed storage of them is needed on the server side. The server side itself does not need to know the password in plain text, as the same password leads to the same hash value, but it is not possible to process the password back from this value. For security reasons, the server side must always store passwords in their hashed form and never in plain text.

In general, the problem with passwords is to remember them. In case of a longer password it is not easy to remember it, hence, some people use the same password for different services, which can be a security vulnerability. If one service is compromised, every service that has the same password is as well. On this account offering a secure place to store such data is a crucial component of an operation system for mobile devices, like Android [1]. To allow a more comfortable access for users many services provide mobile applications (apps). The credentials of these services need to be stored somewhere on the device, so the user does not need to enter them every time an application is used.

To avoid undesirable manipulation of data, a service which manages them must implement the following mechanisms: authentication and authorisation of users as well as

verification of data. First, the user authenticates himself against a service with a secret password only these two participants know. Second, after a user has been successfully authenticated, he must be authorised to access these data as well. Different users can store different data using the same service, and these data must not be changed by a user who is not authorised to change them. The last part is the verification of data, for both writing new and reading already stored data. The user must be sure the service did not change the data and the service must be sure the data were not corrupted while writing them into the file system.

There are different approaches available to manage user credentials. We will introduce these approaches with respect to advantages and disadvantages. After working out the goals based on this analysis two research questions will be addressed in this thesis.

One approach to store the login credentials is a central Android app. These apps can be downloaded from Google Play Store and permit an easy way of installation. The preparation problem of such an application are the different layers and frameworks that will be needed. The lowermost layer is the kernel, on top of it the Dalvik Virtual Machine (VM) [2] or since Android version 5.0 the Android Runtime (ART) [3] and the framework stack including different libraries. If there is a security bug in one of those layers, every layer above is considered as compromised as well, without having an attack vector on their own, because every layer implements the Application Programming Interface (API) of its beneath layer. Another issue is the design of such an application that is by definition both a server and a client program what makes the source code less maintainable and therefore potentially less secure since both functionalities must be implemented. In general, larger source code is less maintainable than less source code as it is more complex. As this is a reduced perception, it will be discussed later in this chapter.

An alternative to a mobile application is the provision of a native Android service. Such a service runs as a system service of the operating system itself and uses only a minimum of layers to do its work. Therefore a security breach is less probable as less layers means less source code that contains issues. Additionally, there is a better separation between the server and the client what makes both more maintainable and therefore potentially more secure than the app approach. If a system service is used, the access to other parts of the operating system, like the file system, can be used in order to check whether the necessary files have the right permissions or not. In addition, there is more control about the service's design, which can increase the security of the service significantly. Google provides the Native Development Kit (NDK) [4] that allows the implementation of applications without using the third-party Dalvik process virtual machine or the ART runtime environment.

Since Android 4.0 the native keystore service [4] is available to every Android application through a public API, which allows certificates to be stored. Before that, only internal applications, like Wireless Fidelity (Wi-Fi), Virtual Private Network (VPN), or Bluetooth, were able to access this service to store their data. Third-party applications could only use the Java API [4] for encryption and decryption, and *SharedPreferences* [4] to store data. One crucial problem was the handling of the encryption keys, because they had to be stored inside the application, which can be decompiled. Decompilation is the process of transforming compiled machine or object code back into human readable source code. A Java compiler creates byte code that runs on the Java Virtual Machine (JVM) instead of creating machine code as done by C or C++. Such byte code is easier to decompile, as it has no huge abstraction from the original source code as the machine code has. Since the keystore service is public, it is possible to store login credentials more safely, without making the developer coming up with an own solution or using a third-party application. This approach is more vulnerable due to many different layers because native services use only a minimum of additional layers.

After analysing the Android keystore service [3] it was determined that it can be improved concerning different properties, for example the used hashing mechanism to make sure nothing is compromised while storing the data, the storage of user names and passwords, and the general design. Furthermore, an evaluation of Android's internal security techniques will be provided to show how attacks against the implementation can be recognized or avoided. Functions that have already been implemented, like the Advanced Encryption Standard (AES) algorithm [5], or the usage of the user's passphrase to encrypt the data will be assumed. The hashing algorithm to detect manipulations is still Message-Digest Algorithm 5 (MD5), which is no longer state-of-the-art [6]. We will see later in this thesis how this was corrected.

Storing login credentials in a secure way is also interesting for server or desktop applications, since they are stored mostly in plain text in configuration files. Therefore, one design goal is to provide a generic implementation of the Android keystore that can be ported to other operating system reliably. In order to create a portable version of the keystore, the already implemented security approaches must be evaluated. In general, there are two different ways of designing a secure application: use security mechanisms provided by the operating system itself or implement the application in a robust and secure way by default.

In this thesis the problems of Android's current keystore implementation are discussed and solutions demonstrated. The resulting design of the portable version can also be used as a reference implementation for other operating systems, such as the BSD and GNU/Linux

distributions because the indicated use cases are not limited to mobile operating systems. Furthermore, this thesis should be a help for further development in this area, through the experiences and awareness gained during this work.

The term “security” in this context does not only mean finding and fixing exploits, which would compromise the system otherwise. It includes the implementation of APIs the Android operating system provides as well, e.g. to create a service or to encrypt or hash values. Such APIs can increase the security of a service significantly. The source code structure and the used compiler flags also belong to this term. Using a standard coding style and compiler flags that support the developer to avoid adding typical issues to the code increases security and maintainability. Maintainable source code helps developers finding bugs or substantial problems what makes the code more secure in turn. For this reason, the implementation introduced in this work has a security-minded focus, instead of a performance-minded one.

The same applies to the term “portability”. It does not just mean porting a program from one platform to another by removing all warnings and errors. Rather, it means using the platform APIs the program will be ported to as well. For instance, if the second platform avoids C functions that are error-prone, such functions can be changed to use equivalent fail-safe functions and committed to the original project, which increases the security of the entire project and ports to other platforms again.

The approaches discussed above lead to the following two research questions: *Can the Android keystore service be improved in regard to security and portability?* and *Is it possible to create a portable one-to-one version based on the native Android keystore service or are source code adaptations needed?*

We start with a background chapter that conveys the basics needed to understand the further topics. The next chapter describes the result of analysing the current implementation of the Android keystore service and points out advantages and disadvantages of individual parts, as the design and the implementation. The results from this chapter will be used in the implementation chapter, which explains the new design concerning security and portability. It describes the ported version as well, in order to show that porting works as expected. The last chapter includes an evaluation of both new implementations. This chapter includes the way to port the new implementation to other platforms and which parts are needed in favour. The evaluation chapter shows how the new implementations are used and how regression tests were implemented. The conclusion chapter finalizes this thesis and gives a short overview of the work that was done and how further development for both implementations might look like.

To explain the different steps of analysing the original implementation and carve out the design for the new one, source code will be extracted and described. The entire project with its associated repository is part of the attached CD. The repository contains three different folders: *keystore*, *keystore-engine*, and *softkeymaster*. The work introduced in this thesis is focused on the *keystore* directory that contains the server as well as the client implementation. The *keystore-engine* directory contains source code that provides functions to handle different cryptosystems and the *softkeymaster* directory implements the communication with an optional crypto hardware module. The source codes located in these two directories are not part of this thesis and does not affect it.

Chapter 2

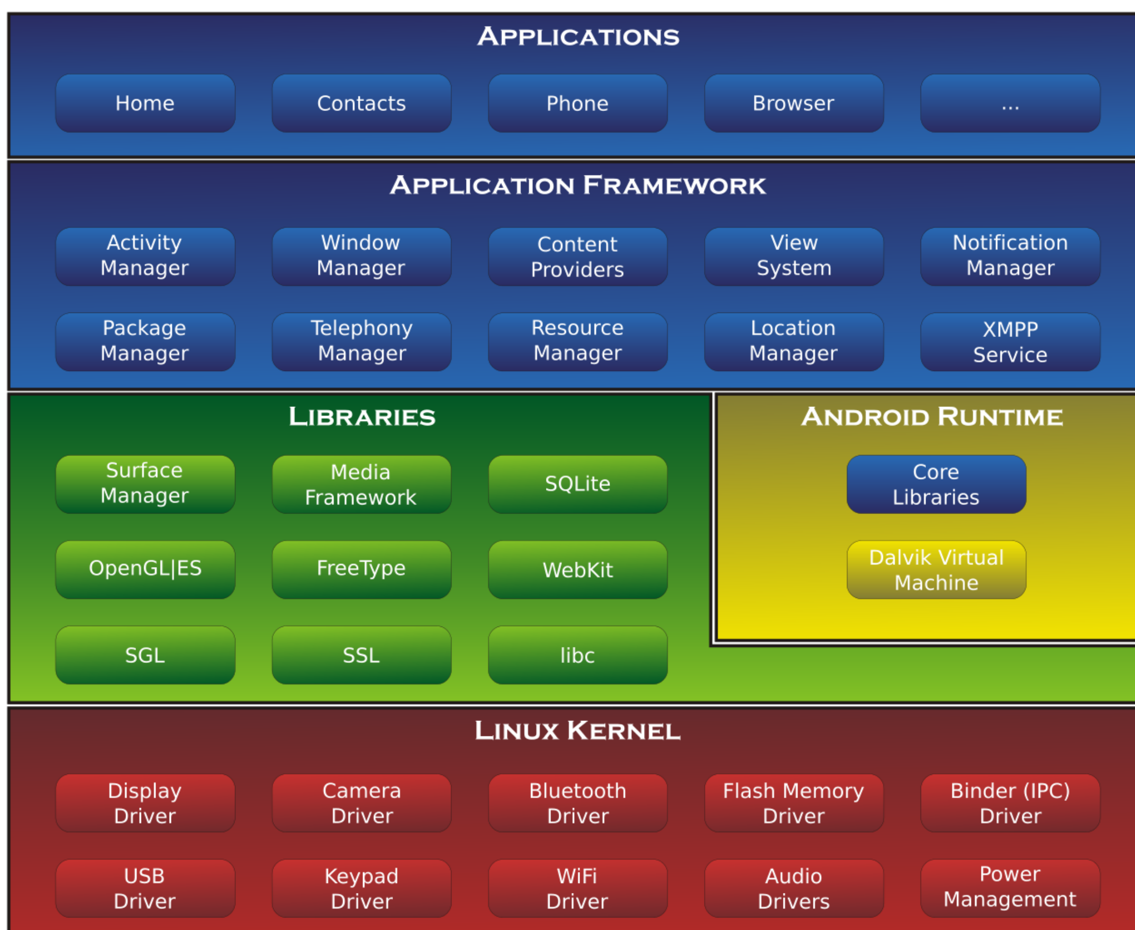
Background

This section conveys background information that is inevitable to understand this work. We will have a look at the Android system architecture, manual memory management as well as encryption and hashing algorithms. Section 2.1 explains the Android software stack, how its different layers are related to each other, and where the new conceptional design that will be introduced in this thesis takes place. Section 2.2 explains some memory problems according to source code that is written in the C programming language, as the original Android keystore service implementation is written in C/C++. Since C++ is a superset of C these issues occur in this language as well. In Section 2.3 there is an overview of different encryption algorithms. At last, Section 2.4 describes the basics of hashing, which hashing algorithms could be appropriate, their advantages and disadvantages, and how the passphrase can be handled in a secure way.

2.1 Android System Architecture

The Android system architecture is organized in a structure called stack. The stack contains different layers and each layer implements the API of its underlying layer. The lowermost is the connection to the hardware and is called the kernel. Such a structure makes it easy to add additional components into different layers without changing the entire architecture. The architecture is shown in Figure 2.1.

The lowermost layer contains all drivers and mechanisms that are necessary to communicate with the hardware or allocate resources from it. The layer above contains the libraries and as part of it the Android Runtime. Instead, the Dalvik Virtual Machine was used until Android 5.0. Later it has been replaced by ART. The library layer is divided into different layers, which contain different functionality, e.g. libc, SQLite or SSL.



Source: Android Developer [4]

Figure 2.1: Android system architecture

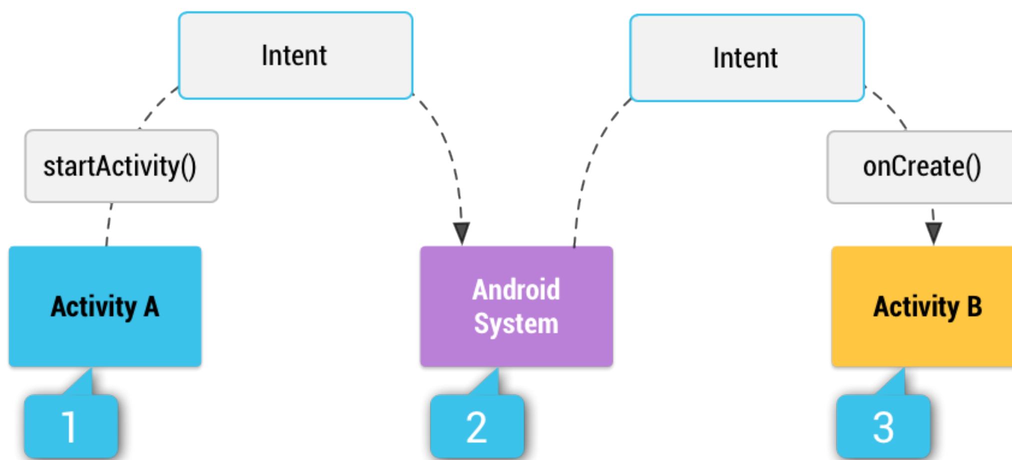
This layer shows another important part how Android apps can be developed, the *Android* Runtime. Apps using ART are the most recommended way to develop software, which can be downloaded from the Google Play Store. On the other hand it is possible to program apps directly in C/C++ without using ART, called native code. These applications are used for game engines or software that needs deep access to the file system or other operating system related resources that are not accessible using ART. It is supposed to be used in cases where performance is more important than usability. It is possible to connect this native code with Java code, called Java Native Interface (JNI) [7]. JNI allows splitting an application into two parts; the logic is more performance-related and the Graphical User Interface (GUI) to show the content. The performance part will be loaded from dynamic shared libraries into a Java-based application. Such libraries are not executable itself, but

contain pre-compiled code that can either be linked at load time or at run time. The application framework is located on top of this layer, where different parts, which contain functionality as the activity manager or the notification centre are located. The application layer is the topmost one where every implemented Android application is arranged that implements a GUI by using the Java programming language.

The approach introduced in this thesis is located between libraries layer and the applications framework, in contrast to the central app approach mentioned in Chapter 1 that uses the application layer. Obviously, the first approach needs less layers than the second one. This does not make the first approach more secure or better by definition, but there are less problematic sources that must to be considered, like different APIs to achieve functionality, and this is very important for the security of the service. Less API calls imply less code, and less code implies less errors. Besides, all you need is a secure place to store the data to avoid access for unauthorized persons. Therefore, there is no need for a GUI if there is a well-documented API, which contains function calls in order to store such data. Another benefit is the separation of functionality between the server and the client.

An Android application has the ability to contain a client of the form of a GUI and a server if it listens to intents [4]. Intents are a way to let different apps communicate with each other by sending or receiving data using a messaging object called intent. They are part of the Android application framework, but do not open a new network, or Unix socket, or do some interprocess communication, as in the traditional client-server model. Those intents are called by activities. An activity is an application component that provides a screen, to interact with the user, such as dialling a phone number or sending an email.

The activities A and B in Figure 2.2 are not necessarily individual Android apps. If such activities are two different apps the transmitted data could be a phone number that was sent to Android's phone app by tapping on a number sent by mail. For this purpose, the phone app listens to predefined intents. If both activities belong to the same app, intents can be implemented to call different activities depending on the content that should be shown. For instance, when data were received from a server over a network and such data contain contact information, a different activity is used to show such information rather than the activity to show an email. Both content is shown by tapping on the "show" button, but this button calls the different activities by calling the dedicated intent with its corresponding data.



Source: Android Developer [4]

Figure 2.2: Android intents

The *intent* follows three different steps to send data from a sender to a receiver:

1. The sender creates an *intent* with all data the receiver needs, such as the receiver itself and additional data that should be processed.
2. The *intent* is sent from the receiver to the Android System.
3. The *intent* is sent from the Android System to the receiver.

It is possible to have multiple apps that listen on one *intent*. One example is sending an email. If there are multiple email apps installed on the Android system and the user wants to send an email by tapping on it, all email apps will be shown in a list so that the user can choose one.

If the chosen software design allows a division of client and server, both parts can be implemented independently - Android applications combine both parts in one single app. This approach accomplishes more maintainable source code to make it easier to find errors, since there is less code in both implementations. Of course, there is an additional part that must be taken care of in the underlying new design, which is the way the information is sent from the client to the server and will be sent back after processing. However, there is a communication overhead because two processes are needed rather than a single one. As already mentioned this can be achieved by using sockets or inter-process communication. In a single process the information will be transferred by calling the functions that provide the functionality.

While the Android system architecture defines APIs for inter-process communication and makes sure that each part in between works as expected, the operating system itself provides further features to secure processes on a lower level. Such features are mostly implemented by the kernel and include the user management, for instance. The following section will describe GNU/Linux's Security-Enhanced Linux (SELinux) extension that extends the normal user management.

2.1.1 Security-Enhanced Linux

The classic access model on GNU/Linux- and BSD-based operating systems is Discretionary Access Control (DAC)[8]. The key concept of this model is that the owner of a file or a directory, in this case known as object, has full control of it. He can grant or revoke rights to himself, and to the associated group as well as to others. The rights are quite simple, there are only the rights to *read*, *write*, and *execute*. However, this model has weaknesses regarding a security related matter. If the user grants rights to someone else accidentally, there is no mechanism whether it is permissible or not. It is up to the user to take care of his files and directories and to grant rights only to those who require them. If he grants too many rights, such files can get compromised very easily.

SELinux [9] adds an additional layer next to the DAC layer that provides Mandatory Access Control (MAC) - *selinux(8)*. MAC refers to a type of access control which defines the ability of a subject to access an object with some sort of operations. In the case of an operating system a subject is usually a process or a thread and objects are constructs such as files or directories. SELinux was originally developed by the United States National Security Agency (NSA), which is an intelligence organization responsible for global monitoring, collection, and processing of information. It is part of the GNU/Linux kernel and implements access control security and mandatory access control using the Flux Advanced Security Kernel (FLASK) concept [10]. This concept provides flexible support for security policies by adding an additional security level to the extended attributes of each object. In this case the essential part of file systems is to define regular attributes for files and directories (objects) that store user permissions or modification times. Such attributes are essential for the object management. Furthermore, the file system provides a feature named *extended attributes* to store meta data, which are not interpreted by the file system itself, for an object. Such attributes contain the author's name, the encoding of the file, or checksums.

In the case of SELinux such meta data are security labels, stored for every object. Every process (subject) gets such a label as well. Now SELinux decides by the means of a prede-

fined policy which subject has access to which object. Every GNU/Linux distribution that enables SELinux by default, provides such a policy that contains general rules determining which process has access to which subjects. These distributions are mostly based on Red Hat Enterprise Linux (RHEL) [11] or sponsored by Red Hat, Inc.® itself. Red Hat, Inc.® is a well-known GNU/Linux distributor for enterprise companies. System users can change such policies to grant or revoke rights to implement a more restrictive or non-restrictive model. For example, there is no need for a web server to access files located under the */tmp* file system. If files are needed temporarily, like the cache for a website, the web server stores them in its own directory. So there is no policy for this scenario. Everything that is not explicitly allowed is forbidden. SELinux is separated into four different access control mechanisms that will be described below.

- **Type-Enforcement (TE)**
  TE defines a domain for every subject and a type for every object. Each type ends with *_t*, and there is no difference between the domain and type interpretation. For the web server example it means that the process has the domain *httpd_t* and the file located under */tmp* has the type *tmp_t*.

- **Role-based Access Control (RBAC)**
  This implementation defines a role for every user. For each role, rights to files can be granted or revoked. By using RBAC the powerful *root* user can be limited to only those programs that are needed for administration. All roles end with *_r*.

- **Multi-Level Security (MLS)**
  The last implementation defines different security levels and is used in security related environments, like the military, or secret services. Each object gets a secrecy level, like strictly confidential, confidential or others. Subjects get clearance for these security levels, so they are able to access objects associated with them. A typical example of this mechanism is the Bell–LaPadula model [12]. This model defines an access control by providing three security properties.

  - **Simple Security Property**
    Known as *no read-up*, which means a subject at a given security level cannot read an object at a higher security level.

  - **⋆-Property (read "star"-property)**
    Known as *no write-down*, which means a subject at a given security level cannot write an object at a lower security level.

  - **Discretionary Security Property**
    Defines an access matrix to specify the discretionary access control.

- **Multi Category System (MCS)**
  MCS uses the same code as the MLS policy and defines different categories where processes can be assigned to. Such a process must have a set of categories that is a subset of the categories a file is assigned to.

In order to handle these different implemented mechanisms and to store the necessary security context for each object and subject, SELinux uses the following format for TE and RBAC: *user:role:type:level*. For MLS/MCS a different format is used, which contains more information: *user:role:type:level:sensitivity:category*.

- **User**
  The "SELinux User" component can have multiple roles. Through these roles multiple types can be reached, as types are grouped in roles. There are three standard users available; *user_u* is the default user for a logged in system user, *system_u* is the default user for processes that start up during boot time, and *root* is the root system user. By convention users always end with *_u*.

- **Role**
  Roles are used to group security types (SELinux Type), e.g. which role can execute which types. The role of a file is always *object_r* and is just a place holder. For a process there are rules like *system_r* or *sysadm_r*. Policies can be specified to grant a role access to a set of types. By convention roles end with *_r*.

- **Type**
  The SELinux types will be added to files in order to say which role or user has access to which type. By convention types end with *_t*.

- **Level**
  This component is for the MLS / MCS access control and is divided into sensitivity and category levels. The sensitivity is the hierarchical part of MLS and contains data classification, like public, internal, confidential, and regulatory by default. The category part is for labelling resources, for example, by departments or projects. The purpose of categories is to achieve better control of security levels. Through the level field, access can be granted to specific categories at a dedicated sensitivity level. This prevents other categories from getting access to this specific file. For instance, if only

two departments should have access to confidential files, such files are associated to these departments.

2.1.2 Capabilities

The traditional GNU/Linux process model is divided into two categories of processes: privileged processes, where the effective user id is 0 (root), and unprivileged processes, where the effective user id is not 0. Each process is associated with the user who starts the process. Privileged processes bypass all permission checks of the kernel while unprivileged ones are subject to full permission checking based on the effective user id, effective group id, and supplementary group list (credentials). The user id is abbreviated as *UID* and the group id as *GID*. This model serves well if an unprivileged user starts a process because of its limited user rights. However, if the *root* user starts a process, this process has the right to read and write the entire file system and can execute any other program.

The capabilities first appeared in GNU/Linux version 2.2 and divide the privileged user into different units - *capabilities(7)*. The list of capabilities implemented by the kernel allows more control of a process, as capabilities are a per-thread attribute. This list is implemented as a kernel module and can be enabled or disabled via the *CONFIG_SECURITY_CAPABILITIES* configuration option. Each thread has three different capability sets, which can be set using the *capset(2)* function call. There are capabilities for various areas available, like using the *chroot(2)* (*CAP_SYS_CHROOT*) or the *setuid(2) (CAP_SETUID)* function call, which changes the root directory or the user id of a process.

- **Permitted**

  > *"This is a limiting superset for the effective capabilities that the thread may assume. It is also a limiting superset for the capabilities that may be added to the inheritable set by a thread that does not have the CAP_SETPCAP capability in its effective set."* capabilities(7)

- **Inheritable**

  > *"This is a set of capabilities preserved across an execve(2). It provides a mechanism for a process to assign capabilities to the permitted set of the new program during an execve(2)."* capabilities(7)

- **Effective**

  > *"This is the set of capabilities used by the kernel to perform permission checks for the thread."* capabilities(7)

Since kernel 2.6.24 a new capability set for executing files was implemented. Such capabilities are stored in the extended file attributes - already mentioned in Section 2.1.1. These capabilities are set using the *setcap(8)* command.

- **Permitted (formerly known as forced)**

  > *"These capabilities are automatically permitted to the thread, regardless of the thread's inheritable capabilities."* capabilities(7)

- **Inheritable (formerly known as allowed)**

  > *"This set is ANDed with the thread's inheritable set to determine which inheritable capabilities are enabled in the permitted set of the thread after the execve(2)."* capabilities(7)

- **Effective**

  > *"This is not a set, but rather just a single bit. If this bit is set, then during an execve(2) all of the new permitted capabilities for the thread are also raised in the effective set. If this bit is not set, then after an execve(2), none of the new permitted capabilities is in the new effective set."* capabilities(7)

It is not that easy to say, which approach is the better one; SELinux or *capabilities*. While SELinux is based on a policy, which is very powerful and allows great flexibility, the possible *capabilities* are limited and must be implemented into the process. So the developer decides which capabilities the process needs and which not. Whether SELinux or *capabilities* are used depends on the level of security that should be achieved. Besides, both are kernel modules, if they are not part of the kernel that is currently installed, they are useless.

While SELinux extends the GNU/Linux user management, each application that runs on a system can contain bugs that lead to different kinds of errors. A common programming language for user space services and commands is C. Because of its manual memory management and low-level design the developer has to take care if applications are written in this language. Some of the typical issues in C will be described in the next section.

2.2 Memory Vulnerabilities

The C programming language is widespread in native commands and services. For this reason typical problems will be introduced and solutions given in this section. Such problems

are buffer overflows and dead store elimination, which are both common security problems in C.

Because of its low-level memory access, operations, such as allocating and using memory, must be used carefully. Each buffer must be checked according to its bounds in order to avoid memory corruption and there is no automatic garbage collection that de-allocates the memory or cleans it after using. Cleaning memory and wiping its content as soon as it is no longer required is very important if code deals with cryptographic values. This makes sure that content, like e.g. keys or passphrases, is no longer accessible to any other program. If new memory is allocated for a program, the content the memory contained before allocating is still available. This section describes how to deal with those problems.

2.2.1 Buffer Overflow Protection

A buffer overflow is a very common problem that makes code vulnerable to many kinds of attacks. It allows the manipulation of code that is executed and possibly its entire behaviour as well. A normal program, like *ls(1)* that just lists files in a directory, can be manipulated to delete such files instead if such an overflow exists and the user cannot avoid it. Basically, a buffer overflow is the effect that more data will be written or read than originally expected. The code that will be written into a buffer to let it overflow can replace other code that is used by the tool itself, or another one. If more data will be read as expected, these data can contain sensitive information, like user names or passphrases, which can be an attack vector. This section describes the different kinds of buffer overflows, how they work, and how memory can be protected from such an attack.

Stack Overflow

A stack overflow is an overflow in the memory stack of a function and often manipulates its return address in order to obtain higher privileges (privilege escalation) [13] or execute code controlled by a hacker. The return address contains the address the program jumps to after it is finished. By changing this address into any other address arbitrary code can be executed.

Figure 2.3 shows a typical memory stack layout. On top of the stack are the local variables followed by the buffers the function uses, the function parameters, the saved frame pointer, and the return address. Because of the way the memory access inside a stack is laid out, it is vulnerable to buffer overflows. A buffer of 32 bytes in size allocated on the stack of a function is save to access, as long as one of the 32 bytes are used.

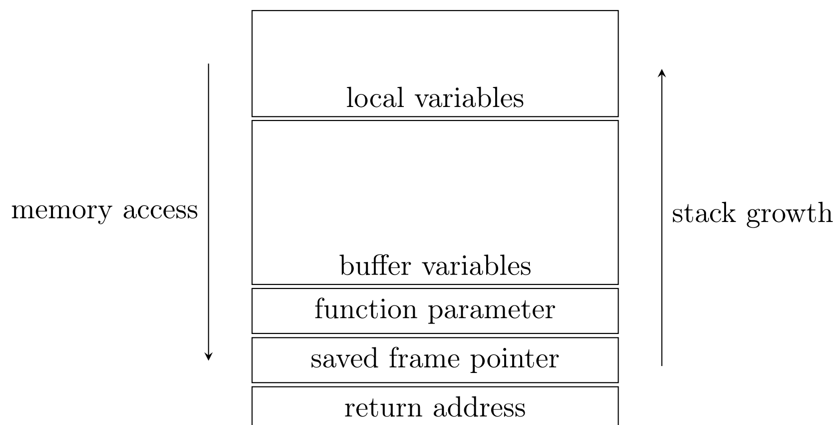


Figure 2.3: Stack layout

However, if the function tries to write the 33rd, byte the next address will be used, but this address already contains the function parameters, so they will be overridden. There is no mechanism that makes sure that the index of a buffer points to an element inside the buffer. It is up to the programmer to access the buffer properly. This can go on until the return address is overridden or even further.

One way to protect the stack from overflowing are stack canaries. A canary is a special value located on the stack right after the allocated buffers, as shown in Figure 2.4. Such a technique is provided by stack protectors like ProPolice.

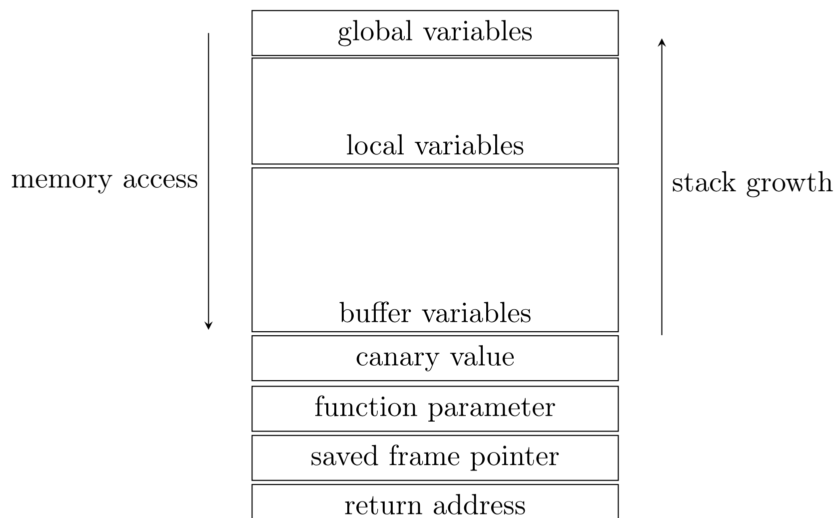


Figure 2.4: Stack layout with canary

The canary value, which is a random natural number, needs to be compared right before the function terminates. The comparable value can be a global variable that is stored above

the buffer, so an overflow cannot destroy it. If the canary value above the allocated buffers is not the same as the original one, there is a stack overflow. Such a check is going to be added by the compiler itself since not every function needs this technique. If the functions contain no buffer, there is no need for a canary. The *gcc(1)* compiler provides a stack protection that uses this technique, which is added by the *-fstack-protector* flag.

Another technique to avoid changing the return address is to use a copy of it. The return address will be copied into a global variable, like the canary value. Before a program jumps back using the return address, it will be checked against the copy whether this address is still the same or not [14].

Heap Overflow

A heap overflow is a buffer overflow caused on the heap of a program. As the stack overflow, this kind of hack can be used to execute arbitrary code. If a buffer is allocated using the *malloc(3)* function and should be 32 bytes in large, the byte at index position 32 (in C all buffers starts at index position zero) is written into the address that comes right after the buffer. If this address is executable, arbitrary code can be executed or the data stored after the buffer can be read in order to disclose information. Since the heap manages dynamic data possibly allocated by a program at runtime, it is not possible to add an overflow detection as the stack protector offers. If the size of the allocated memory is user defined, the buffer could be of any size with the result that a canary value cannot be added. The size of the stack is calculated at compile time, so the canary can be added by the compiler to detect stack overflows. To cumber or potentially prevent heap overflows, the technique to manage memory on the heap was changed in modern operating systems.

There are two common techniques implemented to reach a better heap overflow protection: The first technique is the separation of data and code to prevent payload execution by using the NX-bit. This bit stands for *No-eXecute* and is used to segregate areas of memory that store either processor instructions (code) or data. If an operating system supports such bit, areas of memory containing data can be marked as non-executable. The processor will refuse the execution of those areas in this case, so arbitrary code is not executable. OpenBSD implements a security feature named Write XOR Execute (W^X) that is based on the NX-bit. W^X makes a process' memory page either writeable or executable, but not both simultaneously unless the application requests it [15]. Android has non-executable pages by default using ARM's eXecute Never feature since Android version 2.3 [3].

The second technique to cumber heap overflows is Address Space Layout Randomization (ASLR) [16]. It randomizes the address space positions of both the stack memory and the

heap memory, which prevents an attacker from jumping to a particular exploited function in memory. If the position of such a function is not known, an attacker is not able to execute it by jumping to its location. Both methods do not really prevent heap overflows, but limit the impact (NX-bit) and cumber the jump to exploitable functions in memory (ASLR). Modern operating systems combine both methods in favour of a better overflow protection, but they are focused to prevent arbitrary code execution. It is still possible to read and write more data than expected. Hence, boundary checks of memory is still the best way to avoid heap overflows.

Even though heap overflow attacks are known for a long time, they still exist in modern software. Since the memory is allocated on the heap manually by a programmer, an overflow may occur quite easily. A well-known attack caused by a buffer overflow is the heartbleed bug [17], where the client can prevail the server sending back more data as originally requested. It is possible with this attack to steal private keys, user names and passwords from the server. This issue was solved soon after the bug has been released. Both overflow techniques reveal how important it is to make sure the buffer limit never exceeds, especially if secret data are stored. Missing checks of boundaries can cause a very huge security breach.

2.2.2 Dead Store Elimination

As already mentioned, safe clearing of buffers that contain keys or passphrases is very important in order to avoid attacks on them. The most common way is to use the *memset(3)* library function. This function is part of the standard C library and fills all addresses of a given buffer with a defined value. After this function has been finished, all previously stored values are overridden. The program given in Listing 2.1 shows how this function can be used.

```
char buf[32];
int i;

for (i = 0; i < sizeof(buf); i++)
   buf[i] = i;

memset(buf, 0, sizeof(buf));
```

Listing 2.1: Dead store elimination example

The 32 byte large buffer *buf* is filled with the value of the index variable *i*. After the loop has been finished, the entire buffer is zeroed by using *memset(3)*. This makes it impossible

to see the buffer contained all values from 0 to 31 since all values are now zero. This way passphrases and keys can be safely deleted from memory as well.

The optimizations executed by the compiler are a problem using this technique. The native code of the Android operating system is compiled using the *gcc(1)* compiler. It provides a lot of optimizations for performance, like removing unused code and for safety, as checking for uninitialised variables. However, optimizing the performance can cause a problem as shown in the following examples.

Listing 2.2 shows the resulting object code of Listing 2.1 created by the *objdump(1)* tool, and compiled without any optimizations. As can be seen, the *memset(3)* function call makes sure the buffer is cleaned.

```
4e: e8 00 00 00 00          callq  53 <main+0x53>
    4f: R_X86_64_PLT32   memset+0xfffffffffffffffc
53: b8 00 00 00 00          mov    $0x0,%eax
58: 48 8b 55 f8             mov    0xfffffffffffffff8(%rbp),%rdx
```

Listing 2.2: Dead store elimination without optimizations

Listing 2.3 shows the object code of the same example code, but with the compiler flag *-O3* of *gcc(1)*. This flag extensively optimizes for performance, and removes the *memset(3)* call, since the compiler realizes that the buffer is no longer needed after the loop. That makes this call useless from the compiler's point of view. It does not know that this is for safety reasons and there is no way to tell. This optimization is called Dead Store Elimination (DEC) and can cause a security breach for source code that uses cryptography, as passwords or cryptographic keys will not be deleted after they are no longer needed.

```
20: 75 f3                   jne    15 <main+0x15>
22: 30 c0                   xor    %al,%al
24: 48 c7 04 24 00 00 00    movq   $0x0,(%rsp)
2b: 00
2c: 48 c7 44 24 08 00 00    movq   $0x0,0x8(%rsp)
33: 00 00
35: 48 8b 54 24 28          mov    0x28(%rsp),%rdx
3a: 48 33 15 00 00 00 00    xor    0(%rip),%rdx        # 41 <main+0x41>
```

Listing 2.3: Dead store elimination with optimizations

Until now there is no way to tell the *gcc(1)* compiler to avoid optimizing for security reasons, like removing *memset(3)*, besides by using less performance-oriented flags. So there is a trade off between compiling performance-oriented or secure source code. If cryptography

is involved, it is better to use less flags that optimize for performance.

2.3 Encryption

This section explains different encryption algorithm approaches, for which purpose they are used for and which mode of operation should be used to avoid security breaches. The general goal of encryption is to send data from Alice to Bob without allowing Eve to understand or to manipulate them. If the data are read by Eve, she does not have the ability to understand their content. Only Bob can decrypt the data sent by Alice. Furthermore, Bob needs to be sure that it was really Alice who sent the data and not Eve.

There are two different kinds of encryption algorithms: symmetric and asymmetric. Symmetric algorithms use the same key for both encryption and decryption. In contrast, asymmetric encryption algorithms use two different keys, known as public and private key. Both kinds of algorithms are described in the next two paragraphs.

2.3.1 Symmetric-Key Algorithm

A symmetric key is used to encrypt and decrypt data by using the same key. A typical method is to associate the plain text with the key by using the bitwise exclusive disjunction ($\oplus$) operation in order to obtain the cipher text. For decryption, the cipher text is $\oplus$ associated with the key to get the plain text back. The encryption and decryption are very fast, but all participants need to have the same key. So it has to be distributed in a secure way to not get compromised.

A well-known symmetric-key algorithm is the Advanced Encryption Standard (AES) [18], introduced by the the United States National Institute of Standards and Technology (NIST) in 2001 as a specification for the encryption of electronic data. The underlying algorithm is the Rijndael algorithm, named after its developers Joan Daemen and Vincent Rijmen. This algorithm uses keys of 128, 192 , and 256 bit size and is available on most modern operating systems, either by an internal implementation, such as being part of the kernel, or by the public OpenSSL library [19]. Until today the AES algorithm in association with the aforementioned key sizes is considered as secure, as there are no attack vectors available that threaten the practical security of AES.

A larger key size only increases the effort for brute-force attacks. Therefore, it is only theoretically more secure as a smaller bit key size. A 128 bit key size takes $2^{128} \approx 3.402 \times 10^{38}$ steps while a 256 bit key size takes $2^{256} \approx 1.157 \times 10^{77}$ steps to find the corresponding key.

The best known attack that has practical impact reduces the steps for 128 bit down to $2^{126.1}$, for 192 bit down to $2^{189.7}$, and for 256 bit down to $2^{254.4}$ [20]. These values are the "complexity of operations" needed for an attack against the algorithm. We assume we have a machine that tries 10^{18}, keys per second, then it would take that machine $\approx 2.8912 \times 10^{12}$ years to find one 128 bit AES key:

$$\frac{2^{126.1}\ keys}{10^{18}\frac{keys}{second}} \approx 9.1176 \times 10^{19}\ seconds = 2.8912 \times 10^{12}\ years$$

Figure 2.5: Time to brute-force 128 bit AES key

Other symmetric-key algorithms are Serpent [21] and Blowfish [22]. The Serpent algorithm was a candidate for AES as well, but it is not as fast as Rijndael in both hardware and software. Blowfish is an algorithm designed by Bruce Schneier, but before the AES challenge came into existence. On mobile platforms the perfomance of such algorithms is important. As Blowfish and Serpent are both slower than AES, but more secure (Serpent) it again is a trade-off between performance and security to decide which one will be used.

2.3.2 Asymmetric-Key Algorithm

In contrary to symmetric-key algorithms an asymmetric-key algorithm consists two different keys, the private and the public key. A private key is always associated to a public key. The private key always stays private and is needed to decrypt the received data. The public key is for everyone and can be used to encrypt the data before sending them. By employing such an algorithm, it is not possible to use the keys for both encryption and decryption of the same data as long as the algorithm is seen as secure.

RSA supports the signing of data in addition to the encryption of data. Bob signs data by encrypting them using his private key. As the public key is public to the world, every user can decrypt the data using this key, but can be sure only Bob signed it. However, Alice encrypts data by using Bob's public key as only the associated private key, which Bob only knows, can decrypt them. In conclusion, encrypting data with a private key results in signing them while encrypting data with a public key results in encrypting them. In both cases only the associated key is able to decrypt the data.

A famous example is the Rivest-Shamir-Adleman cryptosystem (RSA) algorithm named after its creators (Rivest, Shamir and Adleman) [23]. Due to the fact that RSA's security relies on the complexity of the factoring problem, it uses large keys, such as 1024 bit or even larger. To be adequately secure, RSA depends on two large prime numbers. For a 1024 bit key the numbers have 309 decimals ($1024log(2) = 308.245$) and for 2048 bit 617 decimals

($2048 log(2) = 616.509$). However, a size of 1024 bit is not considered as secure any more. Today sizes of 4096 bit, which have at least 1234 ($4096 log(2) = 1233.018$) decimals, or more are recommended. Although such algorithms are not as fast as symmetric-key algorithms, they are often used for signing data or exchanging a symmetric key, since everything that is encrypted with the public key can only be decrypted with the private key. Encrypted data with the private key can be decrypted with the public key and one can be sure that only the owner of the private key can have encrypted it as long as the private key stays private.

The ElGamal [24] and Rabin [25] cryptosystems are further known asymmetric cryptosystems. However, both algorithms could never establish what makes RSA the de facto cryptosystem of asymmetric-key algorithms.

2.3.3 Mode of Operation

The mode of operation of block-oriented cipher algorithms describes the technique how individual blocks are encrypted. Cipher algorithms, such as AES, encrypt and decrypt the content based on a fixed block size that is defined by the algorithm. AES has a fixed block size of 128 Bit. Block cipher algorithms can be used in two different main modes of operation [26].

Electronic Codebook

The Electronic Codebook (ECB) mode encrypts and decrypts every block with the key individually. The same plain text block creates the same cipher text block. Since every plain text block results in the same cipher block, the pattern of the plain text remains, e.g. every plain text A results in a cipher text 42. This is a disadvantage as the plain text structure is visible in the cipher text.

$$\forall i \in \mathbb{N}^+ : C_i = E_K(P_i)$$
$$\forall i \in \mathbb{N}^+ : P_i = D_K(C_i)$$

Figure 2.6: Electronic Codebook mode

The ECB mode encrypts the plain text block at index i (P_i) by using the encryption key E_K which results in the cipher block at index i (C_i). For decryption, each cipher block C_i will be decrypted with the decryption key D_K which produces the plain text block P_i. The key for encryption and decryption is the same. In ECB mode the same plain text block always results in the same cipher block. For this reason the plain text pattern still remains.

Cipher Block Chaining

A better approach to avoid these plain text patterns is the Cipher Block Chaining (CBC). This mode encrypts the first plain text block with the chosen key and a random initialization vector. For the second block the initialization vector is the cipher text of the previous block. This destroys the plain text pattern and identical blocks produce different cipher blocks, due to different initialization vectors are used.

$$\forall i \in \mathbb{N}^+ : C_i = E_K(P_i \oplus C_{i-1})$$
$$\forall i \in \mathbb{N}^+ : P_i = D_K(C_i) \oplus C_{i-1}$$

Figure 2.7: Cipher Block Chaining mode

The first plain text block has no previous cipher text block and this is where the random initialization vector comes in. For decryption, the cipher text block C_i will be decrypted by using the decryption key D_K, and associated with the previous cipher block C_{i-1} using the $\oplus$ operation. This gives the plain text block P_i. Each plain text block P_i is associated with the previous cipher text block C_{i-1} using the $\oplus$ operation, encrypted with the key E_K. This results in the plain text block P_i. Only if the same initialization vector is used, encryption and decryption will succeed. Therefore, the CBC mode requires the storage of this vector for the first block. If this is a random value, which is recommended, it cannot be calculated.

2.4 Hashing

Hashing is the possibility to map data of dynamic length to a value of fixed length. The context of this thesis only considers cryptographic hash functions. Such functions are one-way functions, which are easy to compute in one direction, but very hard to compute in the opposite direction. A hash value for a given value can be computed very fast, but it is not easy to compute a value back from a given hash value. Such functions can be applied to check for data corruption, since no two messages can produce the same hash value. If only one character of the message changes, the resulting hash value is completely different. It is very important to ensure that previously stored data are equivalent to those read later.

An important property that is applied to the design of hash algorithms is collision resistance. Collision resistance reveals that it is difficult to find two messages (m_1 and m_2) that lead to the same hash value: $hash(m_1) = hash(m_2)$. If such two messages can be studiously computed, a collision is found and the hash algorithm is considered as insecure.

2.4.1 Secure Hash Algorithm

The Secure Hash Algorithm (SHA) [8, 27] defines a family of cryptographic hash functions that are often used in security related projects. It is divided into three categories.

- **SHA-1**
  Defines a fixed hash size of 160 Bit, but is not considered as secure any more [28]. It was first published in 1995 and was designed by the U.S. NSA.
- **SHA-2**
  Defines a fixed hash size of 224, 256, 384, and 512 bit. This algorithm is state-of-the-art and is considered as secure today. It was first published in 2001 and was designed by the U.S. NSA.
- **SHA-3**
  Defines the same hash sizes as SHA-2, but uses a different algorithm. This algorithm was designed by Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche, but lacks implementations since it is not widely-spread.

Among these algorithms there are others available, such as Whirlpool [29] or RACE Integrity Primitives Evaluation Message Digest (RIPEMD) [30]. Whirlpool only supports a hash value length of 512 bit, while RIPEMD supports a length of 128, 160, 256, and 512 bit. The 160 bit version is the most common one, as the 256 and 512 bit version only reduce the number of hash value collisions, but do not guarantee a higher security level. The 128 bit version has a questionable security level and was replaced by the 160 bit one.

To decide which hash algorithm is the best, in terms of its security, is not that easy. It depends on the technique of creating the final hash and how many researchers have analysed the algorithm in order to find attack vectors. A hash value of 512 bit does not necessarily guarantee a higher security level than a 160 bit value does.

As already mentioned in Chapter 1, passwords should never be stored in plain text, but encrypted or hashed. The next section describes how passwords should be hashed and what one should pay attention to.

2.4.2 Passphrase

A passphrase or password is applied for authentication. Thereby the passphrase is only known by the user as his private secret, not even the service he wants to authenticate against must know this secret. The problem is: How to store this passphrase? Plain text in a database or a file system is not a good idea. If someone has access to this storage, the

passphrase is no longer a secret. However, a better approach is to hash it using cryptographic hash functions, like SHA. These hashes are vulnerable against pre-computed rainbow tables [31], where lots of common passphrase hashes will be compared with pre-computed ones. If someone has access to the hashes, it only slows the comparison of hashes down a bit, if the passphrase is weak and not "salted". A strong password contains at least twelve signs of upper- and lowercase letters, numbers and special characters. A "salted" password means a random value is attached to the password before hashing it. This increases the security level because, for example, the password *Test1234* (which is weak) results in two different hash values. As the random value is different on every system the rainbow table attack becomes less important.

A much better approach is to use a salted passphrase. This means the hash was computed from the users passphrase and a random salt that only the service knows. If the hash is stolen, it is not easy to use pre-computed hashes, since the salt is not the same for all password hashes. The Keyed-Hash Message Authentication Code (HMAC) [32] is a popular method to gain such hashes.

$$HMAC_k(M) = H((K \oplus opad)||H((K \oplus ipad)||M))$$
$$opad = \underbrace{0x5C...0x5c}_{B-times}$$
$$ipad = \underbrace{0x36...0x36}_{B-times}$$

Figure 2.8: Keyed-Hash Message Authentication Code

The final value will be computed from the hash function *H*, a secret key *K* and the message *M*. The constants *opad* and *ipad* are pre-defined in the related RFC 2104 [33], where B is the block size of the hash function. *H* is supposed to be a cryptographic hash function, the RFC standard still uses MD5, but it is no longer state of the art these days, SHA will often be used in the implementations instead. The key will be computed as follows: is the key smaller than the block size of the hash algorithm, additional zeros will be appended. If the key is larger than the block size, *K* will be replaced by *H(K)*.

2.5 Signature

Encrypting data to prevent others from manipulating it, in association with hashing algorithms to find data corruption are both useful resources if the data remain on a local system. As tools or users manage such data, these mechanisms of manipulation prevention are sufficient and the originator is known as long as the system is not compromised. If such

data, however, is sent over a network or shared using USB sticks or CDs, the originator is unknown. At least from the cryptographic point, as the file owner is not necessarily the same as the original owner or a different file system is used which does not support such file owner ids. Cryptographic signatures can be used to prevent this problem, as they ensure the sender authentication and the integrity of data. A valid signature gives the recipient a reason to believe that the data were not manipulated on its way and it was the sender who created the message.

See the following explanation of known cryptographic signature algorithms:

- **DSA**
  The Digital Signature Algorithm (DSA) is a signature algorithm proposed by the American National Institute of Standards and Technology (NIST) and developed by the NSA in 1991[34]. It is based on the discrete logarithm problem and is a variant of the ElGamal Signature Scheme.

- **ECDSA**
  The Elliptic Curve Digital Signature Algorithm (ECDSA) offers a variant of DSA based on elliptic curve cryptography [35]. The benefit of this algorithm is the smaller key size compared to DSA, due to the used elliptic curves. While DSA needs a key size of 1024 bit, 163 bit are sufficient for ECDSA. However, both share the same security level [36] what makes elliptic curve cryptography interesting for hardware with limited resources, like memory or CPU usage.

- **Ed25519**
  Ed25519 is a specific implementation of Edwards-curve Digital Signature Algorithm (EdDSA) [37]. Its design uses the Twisted Edwards curve which is an elliptic curve. This signature algorithm is designed to be fast and uses small signatures (64 byte) and small public keys (32 byte).

Chapter 3

Analysis of the Android keystore service

This chapter contains the results based on the source code analysis of the original Android keystore service. The main part is to analyse the source code with regard to security and portability. This also includes how the entire design was implemented and if it is possible to adapt it or parts of it to other platforms. The communication between the server and multiple clients is an essential part of this service and it is investigated whether the implementation is adequate to the design Android recommends for such implementations. As already mentioned in Chapter 2, C allows the insecure usage of library functions very easily. It will be evaluated how such functions can be turned into more secure ones if available. The way how key-value pairs are stored and how the service keeps track of them, e.g. by checking file permissions on every file read, is part of this part as well. Logging is a crucial part of services as the user can see if there is something wrong or not. Analysing the logging mechanism and used API finalizes the source code analysis. Particular, this is important for portability.

Another part is the analysis of a common coding style to help developers fixing bugs or developing new features. The appropriation of compiler flags can help avoid such errors, but they are introduced after analysing the source code to avoid the located errors in the future as far as possible.

In summary, this chapter describes the conceptual and implemented design of the Android keystore service, a security evaluation of the used C functions, the implemented SELinux policies, and how the service interacts with the Java-based API as well as with the command line interface client utility written in C++. Furthermore, the encryption and hashing algorithms will be evaluated regarding their security and the way they achieve it.

After each analysis, the advantages and disadvantages will be explained, in order to extract the reasons for the new design. Those reasons will be summarised at the end of this chapter to give a short overview of the new design.

3.1 Design

Figure 3.1 shows the design of Android's keystore implementation. The design itself is quite simple. There are different users on the left-hand side, who want to store their passphrases or certificates using the keystore service. Such users are system users who call either the command line client or Java/C++ based APIs see Section 3.2 for more details.

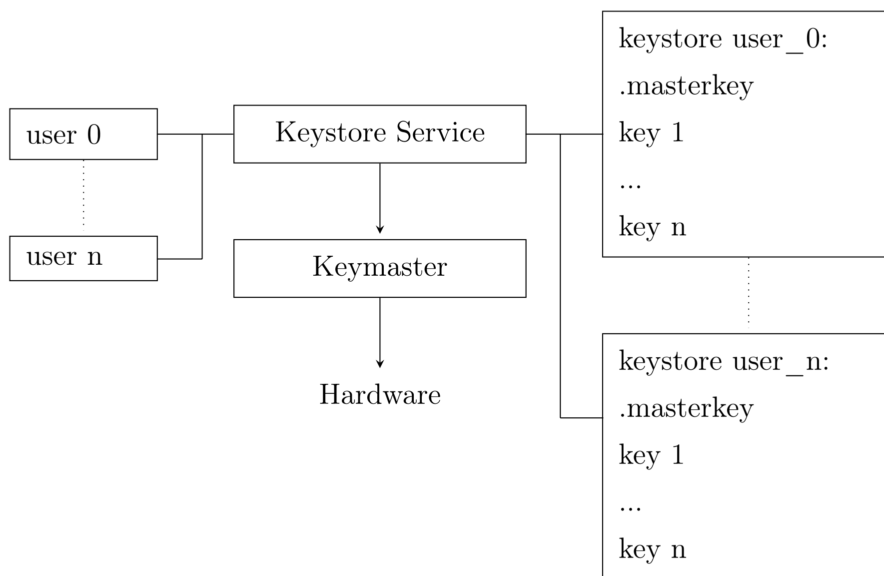


Figure 3.1: Android keystore service design

The original Android keystore service implementation stores the data in a subdirectory of emph/data/misc, which is the default directory. In this directory there are subdirectories, which contain the data of different users. The subdirectory *user_ 0* is the default for data managed by the system itself, such as VPN or Wi-Fi certificates. Each subdirectory contains the master key file and each stored key-value pair, which, in turn, contains the key of the value that is stored. The format of such files is *<userid>_ <key>*. The master key is a random 16 byte number encrypted with a key based on the smart phone owner's password. The *keymaster* provides an implementation to access a hardware module and stores the generated keys. Each key operation, like generating, signing or importing existing keys, will be executed by this implementation and delegates the operations to the crypto-enabled

hardware module [38]. In this thesis this implementation will not be touched, as only a few Android devices have such a hardware module. The main focus is to analyse the *keystore service* implementation concerning security and portability.

3.1.1 The Android RPC subsystem

The Android keystore service implements the Inter-Process Communication (IPC) subsystem of Android as its underlying design for client communication. This is the preferred way to implemented native services on Android. The RPC subsystem submits the implementation of services based on Inter-Process Communication (IPC). RPC defines a technique to call functions in other address spaces. For instance, a function is called on the client-side, but will be executed on the server side. The programmer does not need to know which underlying communication protocols, like TCP or UDP, are triggered. This makes both client and server side development easier as only the functions the services offer and the client calls must be implemented. The RPC framework itself takes care of how the data will be sent and received. Figure 3.2 shows how IPC with RPC calls interact in Android.

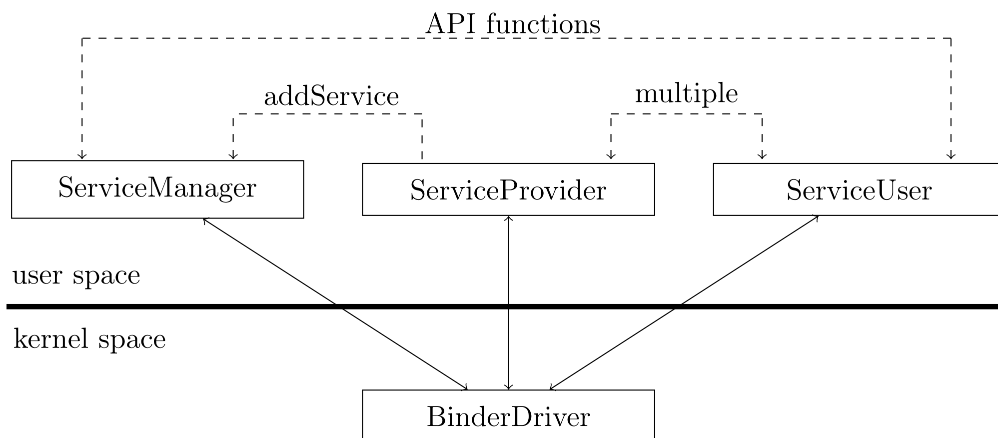


Figure 3.2: The Android RPC subsystem

The underlying implementation in Android to use RPC is the Binder framework. This framework is an Android-specific implementation to provide a interprocess-communication mechanism [39]. It was originally developed under the name OpenBinder by Be Inc.® and later by Palm Inc.® It extends the GNU/Linux IPC framework and allows processes to define interfaces, which are able to be called by other processes. The Android Binder implementation is a customized version of OpenBinder, which is not under development any more.

As seen in Figure 3.2 the *ServiceProvider*, which is the service implementation itself, implements the *Binder* interface and adds itself to the *ServiceManager* by calling its *addService* method. This method is offered by the *ServiceManager* and allows adding *Binder* objects to the system *RPC* service. It is possible to handle different clients by one singular service - *ServiceUser*. The *ServiceUser* asks the *ServiceManager* for the *Binder* object by telling it the unique name. In the case of this keystore service it is *android.security.keystore*. This name is set by the *ServiceProvider* and allows the *ServiceManager* to find it. All methods are implemented through the *Binder* interface can be called using this *Binder* object.

This approach is very powerful as it allows programming different services without concerning about IP-addresses, port numbers, or the underlying protocol. All the service needs is to implement the *Binder* interface and add itself to the *ServiceManager* under a unique name. Clients that want to communicate with this service just ask the *ServiceManager* for the service by telling it the unique name and start to communicate. Which classes need to be implemented on the server as well as on the client-side will be explained in the next section.

3.1.2 Architecture of the Android keystore service

This section describes the architecture of the Android keystore service and shows the different relationships between each individual class of the Android keystore service implementation. Since Android's IPC subsystem is used, there are additional classes needed to inherit from. As Figure 3.3 shows, the main service class is *KeystoreProxy*. This class inherits from *BnKeystoreService* and *IBinder::DeathRecipient*. The *BnKeystoreService* implements the *IKeystoreService* which again implements the *IInterface* for its part. The client is represented by the *BpKeystoreService* class. For the entire service other classes are needed, such as *Blob*, *Entropy*, *UserState*, and *KeyStore*. A more detailed view about provided class attributes and methods can be found in Appendix G.

Figure 3.3 shows how all classes are related to each other. The central server component is the *KeyStoreProxy* class in association with the *KeyStore* class (one-to-one association). The *KeyStoreProxy* implements all necessary interfaces needed to create an Android RPC-based server implementation, while the *KeyStore* class manages the different user states. Each user who uses the keystore service has its one user state. Therefore the *KeyStore* class contains an array of such objects to manage every user individually (one-to-many association). The *Entropy* object in this class is delivered to *Blob* objects and *UserState* objects used in the *KeyStore* class internally, in order to create random values (one-to-one association).

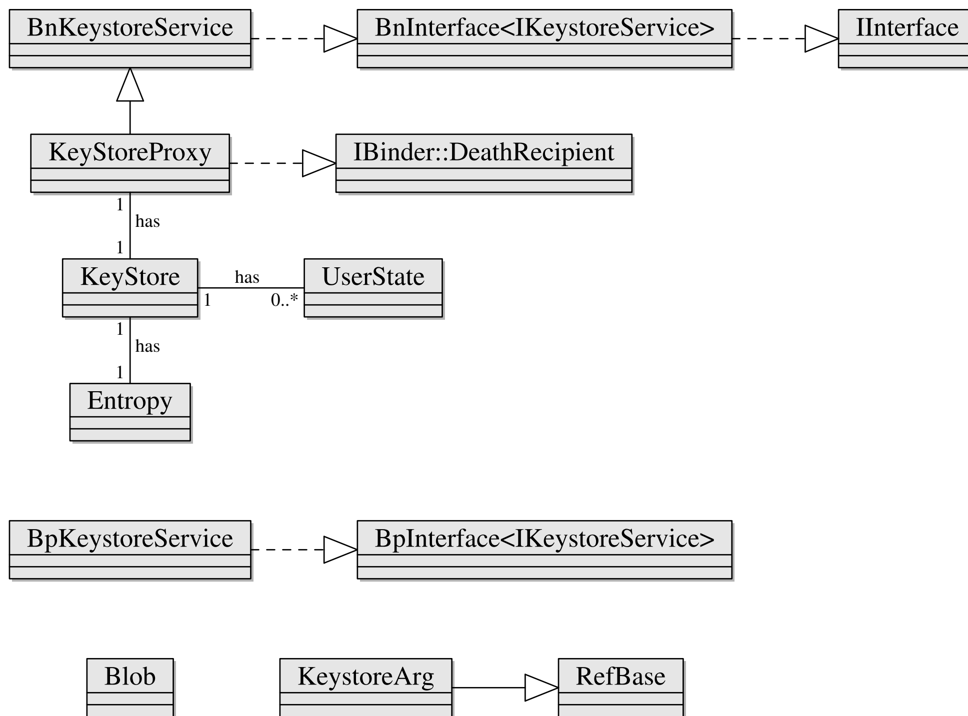


Figure 3.3: Android keystore service class diagram

The central client component is the *BpKeyStoreService* class. This class inherits the Android RPC client interface what implements the *IKeyStoreService* interface again. As the client calls the server methods, this interface needs to be implemented on both sides.

The classes *Blob* and *KeystoreArg* represents data objects that are used in different classes. Therefore, they have no dedicated association between other classes.

The purpose of all classes and interfaces as well as their advantages and disadvantages will be explained in the next sections.

IInterface

This interface is the base interface of every class that needs to communicate using the Android Binder RPC framework, which is part of the Android native framework. This framework includes classes and interfaces to write native (C++-based) applications which communicate with Java-based Android applications. The interface only provides a method named *asBinder()* that retrieves the Binder object associated with this interface.

IKeystoreService

The *IKeystoreService* is the defining interface between server and client. It covers all methods the client can call to communicate with the server. It also defines an enumeration which contains one singular enumerator that represents one transaction call for every call defined in this interface. A transaction consists of an unique identifier (*code*) to identify the transaction, which was received, the objects that contain the client's request and the server's reply (*data* and *reply*), and optional data (*flags*) as can be seen in Listing 3.1. The *Parcel* type is a container for data that are sent or received by a service, which implements Android's *Binder*-Interface [4]. The optional flags are used for normal or one-way RPC and are not considered further in this context [4]. By sending the transaction call and the associated data the server knows which internal call must be used. See Figure G.1 and Figure G.2 in Appendix G for enumerations and methods defined by this interface.

There is dedicated macro defined by the IPC binder interface which is called with the name of the keystore interface name: *DECLARE_META_INTERFACE(KeystoreService)*. This macro adds additional methods and a descriptor to the current interface description, which are called by the internal Android interface implementation in order to access it.

BnInterface<IKeystoreService>

The *BnInterface* is part of Android's framework as well and is used to implement the native end of an interface - in this case *IKeystoreService*. It contains one pre-defined method called *onTransact* that needs to be implemented as this method is called by the Android binder to send the received data to the desired service implementation. See the next section for a more detailed overview of the actual implementation of this method.

BnKeystoreService

As already mentioned, the important method that needs to be implemented is the *onTransact()* method. This method will be called as soon as the server receives a request from the client in order to process it. Listing 3.1 shows a part of implementing a service call named *test*. This call returns the current state for the user who sends this request. Those states are explained in the successive *UserState* class implementation.

As Listing 3.1 shows there is a case for every possible transaction which handles it. The *onTransact* method provides the call number (*code*), the data the client transmitted (*data*), a way to send the reply back after processing the request (*reply*), and optional data, e.g. to determine if the *blob* that should be read is encrypted (*flags*) or not.

```
status_t BnKeystoreService::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case TEST: {
            CHECK_INTERFACE(IKeystoreService, data, reply);
            int32_t ret = test();
            reply->writeNoException();
            reply->writeInt32(ret);
            return NO_ERROR;
        } break;
```

Listing 3.1: *BnKeystoreService*: onTransaction method implementation

The macro *CHECK_INTERFACE* adds an *if*-statement which checks whether the transmitted data belong to the interface or not. If not, the *PERMISSION_DENIED* flag will be returned to signal that the service is not responsible. If a new transaction call is added, only a new case must be added to deal with the data and calls the requested service call. This technique makes it very easy to add new calls or remove existing ones.

IBinder::DeathReceipient

This interface defines a callback function named *binderDied*. It notifies the process hosting the binder if an object implementing of the *IBinder* interface has gone away. This callback function can be used to close files or sockets before the hosting process will be shut down.

UserState

Every user has different states, which he resides in. These states are managed and stored by the *UserState* class. This class creates the master key and the key to encrypt and decrypt it, based on the user's password and stores the master key internally as well as encrypted in the file system for every individual user that has the permission to create a master key - see Section 3.3.3 and Section 3.5.1 for further explanations. It also reads and writes these keys and changes them if necessary. After the password has been entered and the resulting master key has been generated by using the *password* command it is possible to access data without entering the password again, until the keystore service is restarted as it is stored internally. This is not a attack vector as system services, like VPN and Wi-Fi, have their own login data to the service as we see later in this chapter. However, it needs be changed for the portable version as the use case of services and their corresponding clients on non-

mobile platforms is different. This will be discussed later in Section 4.4. An overview about all supported client commands can be found in Appendix B.

Every user is in one of the following states, none of these states will be stored permanently as they only exist while the service is running.

- **STATE_NO_ERROR**
  This is the default case, stating, that there are no errors and the user can add or manipulate every stored data.
- **STATE_LOCKED**
  If the state is locked, the master key cannot be changed. It is only possible to read but not to manipulate it. This lock can occur explicitly by calling the *lock* method or it will be locked automatically if a master key files was already generated.
- **STATE_UNINITIALIZED**
  This state indicates that there is no master key stored. If the user is in this state the master key file can be created. This state is set either by calling the *reset* method or by initializing the user state without storing a master key file previously.

KeyStore

The *KeyStore* class wraps the *UserState* class and provides additional methods, e.g. creating the file format for data that should be stored. For consistency reasons, the method identifiers provided by the *IKeystoreService* are the same as for this class. This simplifies, which method needs to be called in the server implementation. It stores vectors of *UserState* and *grant_t* objects to know each user's state and which user has access to which file. This class is the essential part of the entire implementation that reads, writes and updates existing *Blob* class objects while the *KeystoreProxy* just communicates with the client.

The *grant_t* type, as shown in Listing 3.2, contains a file with its associated user id. Thus, the server is able to find out whether a requested user id is allowed to access this file or not. This is necessary since users can grant file access to other users to share the same credentials.

```
typedef struct {
    uint32_t uid;
    const uint8_t* filename;
} grant_t;
```

Listing 3.2: *grant_t* type

KeystoreProxy

The *KeystoreProxy* class is the server-side implementation that wraps the *KeyStore* class. Even though the suffix *Proxy* suggests a client-side implementation, this class is the connection between Android's IPC framework and the actual keystore service implementation. Since it inherits the *BnKeystoreService* class, it implements all methods provided by the *IKeystoreService* in order to accept the calls from client implementations.

```
keyStore.initialize();
android::sp<android::IServiceManager> sm = android::defaultServiceManager
();
android::sp<android::KeyStoreProxy> proxy = new android::KeyStoreProxy(&
keyStore);
android::status_t ret = sm->addService(android::String16("android.
security.keystore"), proxy);
```

Listing 3.3: Register keystore proxy

Listing 3.3 shows that a *KeystoreProxy* object is added by using the key identifier *android.security.keystore* to the default service manager. The key is used for client implementations to access the *BpKeystoreService* object, which is explained later, for communication as Listing 3.5 shows. Before the proxy is added, the constructor is called and adds the address of the previously initialized keystore object.

BpInterface<IKeystoreInterface>

While the *BnInterface* defines the server-side, the *BpInterface* defines the client-side. In order to communicate with the server, the client needs to implement the same basic interface - *IKeystoreInterface*. The client will send the data using the object returned by the already implemented *remote()* method which returns an *IBinder* object - the server.

BpKeystoreService

The *BpKeystoreService* class implements the client-side implementation for the Android keystore server. Each method called from the client creates a *Parcel* object that contains the data. As already mentioned, this object is processed by the *onTransact* method.

Listing 3.4 shows the communication between the server and the client regarding the *test* API call. First of all, two *Parcel* objects are created, the one that is sent to the server (*data*) the one that contains its reply (*reply*).

```
virtual int32_t test()
{
    Parcel data, reply;
    data.writeInterfaceToken(IKeystoreService::getInterfaceDescriptor());
    status_t status = remote()->transact(BnKeystoreService::TEST, data, &
reply);
    if (status != NO_ERROR) {
        ALOGD("test() could not contact remote: %d\n", status);
        return -1;
    }
    int32_t err = reply.readExceptionCode();
    int32_t ret = reply.readInt32();
    if (err < 0) {
        ALOGD("test() caught exception %d\n", err);
        return -1;
    }
    return ret;
}
```

Listing 3.4: *BpKeystoreService*: test method implementation in keystore service

The only information the server needs in this case is the interface token. The data object will be sent afterwards by using the *transact* method that will call the *onTransact* method on the server-side internally. After evaluating the response code, the entire reply will be evaluated according to the expected values as well. In order to implement a typecast from a *Binder* object back to an interface used in the client, the macro *IMPLEMENT_META_INTERFACE(KeystoreService, "android.security.keystore")* is called after the *BpKeystoreService* is defined. This macro allocates a *BpKeystoreService* object, which is returned by the default service manager called from the client.

```
sp<IServiceManager> sm = defaultServiceManager();
sp<IBinder> binder = sm->getService(String16("android.security.keystore")
);
sp<IKeystoreService> service = interface_cast<IKeystoreService>(binder);
```

Listing 3.5: Accessing *BpKeystoreService* from the keystore client

Listing 3.5 shows how a client obtains access to the keystore implementation using the *android.security.keystore* key. The added generic client *Binder* object is returned by the default service manager and typcasted into an *IKeystoreService* object afterwards, to call the desired functions.

To communicate with an Android RPC server (*BnInterface*), a client, e.g. a CLI tool, uses the client implementation (*BpInterface*) returned by the default service manager to call the desired methods.

RefBase

This class is supposed to implement smart pointers. This kind of pointers are abstract data types that simulate a pointer while providing additional features, such as automatic memory management or bounds checking. They keep track of memory they point to and reduce bugs caused by misuse of pointers while retaining efficiency. The concept of smart pointers originates in the C++ language [40].

KeystoreArg

The *KeystoreArg* class is used for generating new DSA and RSA keys. The keystore implementation offers the ability not just to store data, but to generate new keys as well. For the DSA key generation a vector of this class contains the parameter for the generator and the prime numbers p and q. For the RSA key generation this class stores the public exponent.

Blob

The *Blob* class covers the data stored in the file system. It provides methods to set different values, such as the version number or the type, and to write and read data. The methods for writing and reading contain the encryption, decryption and hashing part as well. For a more detailed overview of how these parts are implemented and how the structure is used to store the resulting data, see Section 3.5 and Section 3.6. The class *Blob* just manipulates a private attribute of the *struct blob* type described in Listing 3.6. This structure represents the internal composition for storing all data and will be written directly into the file system. This is possible due to the fixed size all variables and buffers of the structure have. If they were dynamic, the size of all buffers would have to be stored additionally. Therefore there is no need for an exchange format, like *JSON* or *XML*.

As can be seen in Listing 3.6, the structure has an additional attribute called *packed*. This attribute is a *gcc(1)* extension that tells it to leave out all padding between members. It makes sense to use this attribute if the structure has to match hardware. However, this attribute causes the compiler to create different code on different architectures. For instance, on the *sparc64* and *ia64* architecture the generated code of a structure that just contains an *int* variable is approximately 6 times more than on the *x86* or *x86_64* architecture [41].

```
struct __attribute__((packed)) blob {
    uint8_t version;
    uint8_t type;
    uint8_t flags;
    uint8_t info;
    uint8_t vector[AES_BLOCK_SIZE];
    uint8_t encrypted[0]; // Marks offset to encrypted data.
    uint8_t digest[MD5_DIGEST_LENGTH];
    uint8_t digested[0]; // Marks offset to digested data.
    int32_t length; // in network byte order when encrypted
    uint8_t value[VALUE_SIZE + AES_BLOCK_SIZE];
};
```

Listing 3.6: Original blob structure

Since there was a change in the way encryption and decryption was handled in the first versions of the Android keystore service, the current version needs to be stored for compatibility reasons. In the current version it is possible to decide whether a *blob* should be encrypted before storing or not in order to let apps use the keystore more productively and to delegate the encryption part down to the hardware encryption module, if the device supports it [42] - the master key will always be encrypted. That is what the *flags* attribute is for. It contains the *KEYSTORE_FLAG_NONE* or the *KEYSTORE_FLAG_ENCRYPTED* value to identify the situation. The *type* attribute will be used to store additional meta data for the blob, e.g. whether it is the master key file (*TYPE_MASTER_KEY*) or contains a key-value pair (*TYPE_KEY_PAIR*). It is possible to store an optional description, such as the salt if the *blob* is encrypted. It will be stored after the secret in plain text. The description length is stored in the *info* attribute. As already mentioned above, the attributes needed for encryption and hashing are explained in Section 3.5 and Section 3.6.

The described attributes and their functions can be simplified in the new version of the Android keystore service. For instance, the choice of whether to encrypt a *blob* or not and how does not need to be stored in an extra attribute. If the device supports a hardware crypto module, the key master implementation will recognize it and as the stored data will not leave the device, there is no situation like encrypting the data using crypto module support and decrypt it on another device that does not support such module. The version flag is only for compatibility reasons to mark the version of the Android keystore the encryption of data is done by such modules. If no crypto module is available the encryption of data is done by the service itself. It is redundant to store this functionality explicitly. At some point a new software version is used widely, so the compatibility checks make the

source code more complicated as they are no longer necessary and should be removed. As the version analysed in this master thesis is a complete fork, the point to remove such version flag for compatibility reasons has arrived. If a design decision was made that requires such a flag, it should not stay forever. If the new version is used by a sufficient number of users, such logic should be removed as its purpose is satisfied and the code is more complicated than needed.

Entropy

The *Entropy* class wraps the *open(2)* system call that calls the */dev/urandom* device to generate random numbers of a given length. This device does not block if a pre-defined entropy level falls below. Hence, it is theoretically possible that pseudo random numbers are too weak and can be pre-calculated by an attacker if there is not enough entropy available to create strong pseudo random number. The */dev/random* device blocks in this case and waits until enough entropy is available. This is clearly a consideration between performance and security. For performance reasons non-blocking function calls are better, but for security reasons it is better to generate a random number that is strong enough. A better approach generating random numbers are OpenBSD's *arc4random(3)* library functions, as they always return a random number and do not block. This, however, is only available on the 64Bit architecture [4]. Furthermore, it is questionable if there is a C++ class needed to wrap just a single C function call.

3.2 Application Programming Interface

This section explains the API provided by Android keystore service to communicate with Java- and C++-based applications and client implementations that already exist.

3.2.1 Java

The Java-based API is the mostly used one. It allows communication between an Android applications with the keystore to store certificates or passphrases. This implementation calls methods implemented by the *IKeystoreService* interface to offer all abilities the service has to Android applications [43]. There are further Java classes which use the keystore service for communication, such as KeyChain [44] or KeyStore [45].

3.2.2 C++

All API calls are defined in the *IKeystoreInterface* interface description already explained in Section 3.1.2. Hence, the client implementation with its supported functions will be described in this section.

This C++ client is implemented by the command line interface utility, named *keystore_cli*. It provides a keyword for every supported call, such as *get* to show a previously stored key or *password* to set the initial master password. Each keyword needs a pool of additional arguments like the value a key was stored in order to show it for the *get* keyword. If the keyword was used with a wrong number of arguments information how to correctly use this keyword is shown. The *$* signs means that the command will be executed by terminal emulation.

- `$ keystore_cli get foo`

  This command searches for a file that contains the key *foo* associated with the user's id and writes the found value to *stdout*, otherwise an error will be written to *stderr* that explains why the value could not be found.

- `$ keystore_cli password 53c237_9455w02d`

  This command instructs the server to create a new random master password encrypted with the password given as the 3rd parameter (53c237_9455w02d). On success the return value will be written to *stdout*. On failure the return value will be written to *stderr*.

Unfortunately, the client implementation does not implement all keywords that were supposed to be implemented. For example, it is not possible to insert a new key-value pair using the command line utility, since the implementation is marked as *TODO*. This explains that this command is not used for production, but for testing the service in order to see how it works. For a smart phone implementation design, this approach is comprehensible, as a smart phone is not designed to make use of terminal utilities. However, for a portable version the client should be revised, since it is not known on which other platforms this tool will run and the porting to these platforms should be as easy as possible. Furthermore, the API provides a large set of calls clients can execute. Beside calls to add, show or delete data there are calls to generate/import public-private keys or grant/ungrant user access to key-value pairs of other users as well. It is not quite clear why such calls need to be implemented, as there is technically no reason for such a service to generate public-private keys or to make a difference between import and insert data. It is possible to add keys

and values as key-value pairs, so why is there a dedicated function call to import data? Granting user access to key-value pairs to other users is always a subject for debate. If a password is shared between multiple services and one of them gets hacked, all others must be seen as compromised, too. For usability reasons, it is easier to remember one password instead of multiple. This is clearly a trade-off between security and usability and will be discussed further in Chapter 4.

3.3 Implementation

This section describes how the keystore service was implemented, which APIs are used, whether there is a consistent coding style, which compiler flags are used, and how possible integer overflows are treated. As already mentioned the manual memory management is a part that needs great attention, in order to avoid memory leaks or causes for program crashes. Another part is the missing overflow check in functions, provided by the standard C library (libc). This library contains different functions for memory allocation, string manipulation, or converting character values to numbers. The problem is that such functions do not always make sure that there is no overflow of integer values.

3.3.1 Coding Style

Choosing a consistent coding style is very important in software development, in particular for teams. It defines how and for what purpose macros are implemented, how the variables, objects and class names are denoted, and where the different types of brackets are set. Such a coding style makes the navigation through source code easier and helps the developer to find specific features. There is no standard coding style for C/C++ programs. While the OpenBSD project uses the kernel normal form (*style(9)*), the GNU/Linux project prefers another one described in the *Documentation/CodingStyle* kernels directory.

After analysing the Android source code it looks as it is no consistent coding style at all. The command line interface client *keyastore_ cli.cpp* shows a heavy usage of macros to handle single calls, but the server implementation does not. Such macros are preprocessor statements of *gcc(1)* that will be substituted at compile time. This complicates reading source code and prevents static analysers from finding bugs as well. Implementing the entire functionality using macros brings no benefit but speed. In an environment as the keystore, security should be in focus.

Splitting different classes or functionality into different files is less a question of coding style, but a question of clarity. There are three different files that contain the entire keystore

implementation. The *IKeyStoreService.cpp* file contains the *KeystoreArg* and *BpKeystore-Service* class as well as the *BnKeystoreService::onTranasact* method implementation. The *keystore.cpp* implements the *Entropy*, *Blob*, *UserState*, *KeyStore*, and *KeyStoreProxy* classes as well as the structures *blob* and *grant_ t* and the *main* function, which starts the service - see Section 3.3.6 for further explanations. This design gives more information to the compiler so that a better performance optimization can be obtained. However, it does not increase the clarity of code. If different classes are in the same file, the compiler can see how many of these methods are called. For example, if one method is never called, it can be optimized away. If these classes are in different files, different assemble units exist and the compiler cannot see which function is called in an external file, as it compiles every file individually. Each assemble unit will be linked together to a single binary by a linker. As already mentioned performance optimization is not as important for this kind of software as the security part is and this part starts with a clear source code layout.

3.3.2 Library Functions

This section introduces some library functions that were misused in the original keystore implementation. It explains the usage of these functions, why the usage was wrong, and how these functions can be used in a more secure way. The misuse of functions for hashing and encryption is analysed in Section 3.5 and Section 3.6.

malloc(3) function

The *malloc(3)* function allocates memory on the heap and returns the start address of it. If no memory can be allocated, the *NULL* pointer will be returned. This makes it necessary to always verify the return value of this function against *NULL*. Without such a condition check, the program can cause a crash, since there is no memory that can be accessed to store values. Listing 3.9 shows a missing return value check of *malloc(3)* that can lead to a program crash.

```
int len = 32;
char *buf = NULL;
char val[] = "Hello World";

(void)memcpy(buf, val, len); /* Segmentation fault */
```

Listing 3.7: *memcpy(3)* function call

Listing 3.7 shows what happens if the buffer, which holds the memory the value is copied into, is *NULL*. *memcpy(3)* copies *len* bytes of the *val* buffer into the *buf* buffer. More specifically, it starts reading *len* bytes from the address pointed by *val* and writes into the addresses started by the address *buf* points to. If the destination buffer contains no address, e.g. *NULL*, the source data cannot be copied as there is a valid address needed to start from. Since *memcpy(3)* is not able to handle buffers that hold no memory, a *Segmentation fault* will occur and the program crashes.

```
int len = 32;
char *buf = NULL;
char val[] = "Hello World";

if ((buf = malloc(len)) == NULL)
  return -1;

(void)memcpy(buf, val, len);

free(buf);
```

Listing 3.8: *malloc(3)* function call

If memory was previously allocated in a secure way, the data will be copied successfully without any problems as Listing 3.8 shows.

```
        *item = (uint8_t*) malloc(keyBlob.getLength());
        memcpy(*item, keyBlob.getValue(), keyBlob.getLength());
        *itemLength = keyBlob.getLength();
```

Listing 3.9: *malloc(3)* function call in keystore service

The code in Listing 3.9 was found in the original implementation of Android's keystore service. It copies the value of *keyBlob.getValue()* into the memory referred by the **item* heap object, using the standard library function *memcpy(3)*. There is no guarantee whether the **item* object has allocated memory or not what can lead to an error as Listing 3.7 shows. The examples explained above show how important it is to make sure that if memory is copied, there must be the memory large enough to store it. It is very careless to assume that there is always memory available if needed.

It should be mentioned that return values do not always have to be considered. In some cases return values could be neglected. If one buffer will be copied into another, as shown in Listing 3.8, and the second buffer is always large enough to store such data, the return value

does not have to be checked. The check of return values often leads to an error handling, either by printing the error and continue the program or by exit it. If there is a return value that indicates success or failure after calling a function, but neither the program nor the user can do anything against a failure and the program should not quit at this point such a check is unnecessary. In this case the *(void)* typecast is used to indicate such a scenario. This typecast indicates there is no return value check needed by turning the original one into *void* - this means there is no return value. *printf(3)*, for example, prints content according to the given format on *stdout* and returns the number of characters without the trailing binary zero. The return value -1 indicates a failure whether an output or encoding error occurs. Such an error should be checked before printing it and if it fails, a static text to indicate the error should be printed instead. *malloc(3)* on the other hand returns *NULL* if there is no memory available to allocate. In this case the program should quit with an understandable error message, as it would run into an error as soon as the expected memory is not accessible.

atoi(3) function

The *atoi(3)* function converts an ASCII string value to an integer value. This standard library function conforms to *ANSI C89*. In the client program of the service it converts the given user id which is a string value, as it is given to the program as a command line parameter. Since this value is controlled by the user, one always should make sure that this value is evaluated to a value the program expects. If the user enters a letter rather than a number, the program must recognize this case.

```
printf("%i\n", atoi("42"));             /* -> 42 */
printf("%i\n", atoi("test"));           /* -> 0 */
printf("%i\n", atoi("432421521451")); /* -> -1370175445 x86_64 */
printf("%i\n", atoi("432421521451")); /* -> 2147483647 armv7l */
```

Listing 3.10: *atoi(3)* function call

Listing 3.10 shows how different values will be converted by *atoi(3)*. If the value is a valid integer value, the conversion returns the correct value. If the given string cannot be converted, 0 is returned. However, the string "0" is a number and will be converted to the integer value 0. In this case, it is not determinable whether the conversation was correct or not. The last two calls show an integer overflow on an x86_64 and on an armv7l processor and both return values are completely different. It cannot be decided whether the returned value is the correct one - a number was entered - or not, at least for portable versions of

software implementing this function call.

Unfortunately, these verifications are not used in the original implementation of the keystore command line client (*keystore_ cli.cpp*), as Listing 3.11 and Listing 3.12 show.

```
        if (argc > 3) { \
            uid = atoi(argv[3]); \
            fprintf(stderr, "Running as uid %d\n", uid); \
        } \
```

Listing 3.11: *atoi(3)* function call no. 1

In this case it is assumed that the third parameter given to the client command line program can be converted without any problems. Since this value is set by the user, it must be verified that this value is an expected one.

```
    if (strcmp(argv[1], "saw") == 0) {
        return saw(service, argc < 3 ? String16("") : String16(argv[2]),
                argc < 4 ? -1 : atoi(argv[3]));
    }
```

Listing 3.12: *atoi(3)* function call no. 2

The same problem can be found in the statement shown in Listing 3.12. The fourth value will be converted without any further checks hoping the value is correct. This is not a security vulnerability, as it does not open a way to exploit neither the client nor the server. Even if the return value of *atoi(3)* is *0*, which is the *root* user id, this user is not allowed to add or manipulate data, since the SELinux implementation prohibits it - see Section 3.3.3. It can certainly lead to a side effect, as the value entered by the user is not checked at the point where it is supposed to be - before committing the data to the service. Instead, such data will be checked by the service itself. Checking user-entered parameters before calling a service's API an error message can be printed to inform the user that his input is not valid. The service does not have the link between the original input and the processed value. For this reason, user input should always be validated as soon as possible to avoid further side effects and adding unnecessary source code to the service implementation.

A way to solve this problem is to use the *strtol(3)* function, which is part of the standard C library as well. This function requires additional parameters that allow verifying whether the conversion was correct or not. Listing 3.13 illustrates how this function must be used to avoid the problems of *atoi(3)*.

```
lval = strtol(str, &ep, 10);
if (str[0] == '\0' || *ep != '\0') {
  errno = EINVAL;
  return -1;
}
if ((errno == ERANGE && (lval == LONG_MAX || lval == LONG_MIN)) ||
    (lval > INT_MAX || lval < INT_MIN)) {
  errno = ERANGE;
  return -1;
}
```

Listing 3.13: *strtol(3)* function call

The *strtol(3)* function requires three arguments: the first one is the string that should be converted, the second one is the end pointer, where the first address of the first invalid character will be stored, and the last one is the base. As we need the decimal system, this number is *10*. If the string contains characters that are not numbers, *ep* contains such characters. If there are no digits in this string, *strtol(3)* stores the original string in *ep*. In addition, if the given string is empty, nothing can be converted. If the string was converted to a number correctly, the possible return values and limits must be checked. If *strtol(3)* discovers an overflow, *LONG_MAX* is returned and *LONG_MIN* in the case of an underflow. As this function can be used as a replacement for *atoi(3)*, which returns an integer value, the return value needs to be in the interval between *INT_MIN* and *INT_MAX* additionally.

```
printf("%i\n", convert_string_to_int("42"));           /* -> 42 */
printf("%i\n", convert_string_to_int("test"));         /* -> -1 */
printf("%i\n", convert_string_to_int("432421521451")); /* -> -1 x86_64 */
printf("%i\n", convert_string_to_int("432421521451")); /* -> -1 armv7l */
```

Listing 3.14: Convert string to int safely

Listing 3.14 describes how the function introduced in Listing 3.13 is used to convert a string to an integer correctly. The last two calls show that this version is much better for a portable version as they return the same value by the time an overflow occur. If user input needs to be transformed, the *errno* variable needs to be checked as well to make sure the original value was not *-1*.

3.3.3 SELinux

Android's keystore implementation uses some custom SELinux policies to restrict different users to different access levels. As Listing 3.15 shows, each provided keystore API call, as already described in Section 3.2, has its corresponding permission. This allows the limitation of every user to a specific set of calls. C's bitwise left shift operator << makes a combination of such permissions possible, so that a user can have the right to read data (*P_ GET*) and verify them (*P_ VERIFY*) afterwards, but is not allowed to add or manipulate them.

```
typedef enum {
    P_TEST          = 1 << 0,
    P_GET           = 1 << 1,
    P_INSERT        = 1 << 2,
    P_DELETE        = 1 << 3,
    P_EXIST         = 1 << 4,
    P_SAW           = 1 << 5,
    P_RESET         = 1 << 6,
    P_PASSWORD      = 1 << 7,
    P_LOCK          = 1 << 8,
    P_UNLOCK        = 1 << 9,
    P_ZERO          = 1 << 10,
    P_SIGN          = 1 << 11,
    P_VERIFY        = 1 << 12,
    P_GRANT         = 1 << 13,
    P_DUPLICATE     = 1 << 14,
    P_CLEAR_UID     = 1 << 15,
    P_RESET_UID     = 1 << 16,
    P_SYNC_UID      = 1 << 17,
    P_PASSWORD_UID  = 1 << 18,
} perm_t;
```

Listing 3.15: SELinux permissions

Only a few system users are allowed to interact with the keystore service. Those users are *system* (AID_SYSTEM), *vpn* (AID_VPN), *wifi* (AID_WIFI), and *root* (AID_ROOT) as shown in Listing 3.16. The *system* user is the most powerful one since he is able to use every API call. The *vpn* and *wifi* user is allowed to call the *get*, *sign*, and *verify* call only. The *root* user has the lowest security level and can only call the *get* API call. Every other system user has the default permissions (*DEFAULT_ PERMS*). For instance, if a new WLAN certificate is installed, the *system* user is used by the Java-based APIs as he is the only one of the above users who can install new key-value pairs. If the certificate has

successfully been installed, the *wifi*, *vpn*, or *root* user can be used to read this certificate for further processing. Other users than these three are not allowed to access data stored by the *system* user. According to the SELinux permissions this user is the only one who can change the password for the master key. If a new key-value pair is created by a mobile application, the user id of this application is part of the key as described in Section 3.1, but all users share the same master key. Depending on the user id the service allows access or not.

```
static struct user_perm {
    uid_t uid;
    perm_t perms;
} user_perms[] = {
    {AID_SYSTEM, static_cast<perm_t>((uint32_t)(~0)) },
    {AID_VPN,    static_cast<perm_t>(P_GET | P_SIGN | P_VERIFY) },
    {AID_WIFI,   static_cast<perm_t>(P_GET | P_SIGN | P_VERIFY) },
    {AID_ROOT,   static_cast<perm_t>(P_GET) },
};

static const perm_t DEFAULT_PERMS = static_cast<perm_t>(P_TEST | P_GET |
   P_INSERT | P_DELETE | P_EXIST | P_SAW | P_SIGN
        | P_VERIFY);
```

Listing 3.16: SELinux users

As Listing 3.16 shows, the *root* user has only less API access. This is a little bit unusual because on an ordinary GNU/Linux or BSD operating system this user has full administrative access. However, the API is called from many different places, as the Java API for Android applications or the command line interface. If *root* were the only user having full access to it, every call would be called with *root* rights, which violates the least privilege principle. In this case it is reasonable to give different rights to different users, in order to limit the security issues on the system. Besides, most of the service access is done by the Java-based API, and only a few users use the CLI utility. Those who have *root* access on their devices can use the *system* user by calling the *su(1)* command. For this reason, restricting the *root* user to read stored data only, but not to manipulate them, is a reasonable way on a mobile operating system. However, for a portable version the situation is different. If there is no SELinux installed on the specific system, the permissions shown in Listing 3.16 are useless since there is nothing to execute them. So for a portable version this has to be considered and whether this approach is really helpful or not to increase security. System users on other GNU/Linux or BSD systems have already access to the

root user, but use their own unprivileged user instead.

The permissions are checked in every API method implemented by the *KeyStoreProxy* class. To check these rules, the C function *has_permission* has been implemented as seen in Listing 3.17. This function makes sure that the given user id has the permission to access a process.

```
for (size_t i = 0; i < sizeof(user_perms)/sizeof(user_perms[0]); i++) {
    struct user_perm user = user_perms[i];
    if (user.uid == uid) {
        return (user.perms & perm) &&
            keystore_selinux_check_access(uid, perm, spid);
    }
}

return (DEFAULT_PERMS & perm) &&
    keystore_selinux_check_access(uid, perm, spid);
```

Listing 3.17: *has_permission* function

As Listing 3.17 shows, the function iterates over all *user_perms* structures and checks if the current user id is the given one. If this is true, it will be checked that the current user has the permission to access this particular process. If this is false and the requested permission is part of the default permissions (*DEFAULT_PERMS*), it will be checked whether the user id has permission to access the service or not by the *keystore_selinux_check_access* function. This wraps the *selinux_check_access(3)* function provided by the *selinux(8)* framework in order to check whether the program user is allowed to execute the call or not.

3.3.4 Compiler Flags

Compiler flags can support developers in avoiding typical errors in the source code by warning about them or add additional source code to the program before compiling, e.g. warn about uninitialized variables or to enable the stack protector as described in Section 2.2.1. That is why they should always be set to reach a higher security level and to avoid further issues such flags can discover. Some of the compiler flags set in the original keystore implementation are introduced below.

As already mentioned in Chapter 2 the *gcc(1)* compiler provides different compiler flags for different purposes, like security checks or performance optimization. Since this application is critical to security, the performance-oriented options should be either omitted or if they are really necessary they should be used with great care. The compiler's documen-

tation provides a complete list of all available compiler flags [46]. The following flags are applied to Android's keystore service.

- **-Wall**
  Enables different warnings only set by one flag. It allows one to perform different checks without setting each flag individually. Some of these flags are: *Wunused-function* to find functions that are never called in the program or -*Wunused-value* to find assignments to variables that will never be read.
- **-Wextra**
  As the -*Wall* flags, it turns on different warning flags by just setting this single flag. Some of these flags are: -*Wuninitialized* to avoid accessing data of uninitialized variables or -*Wsign-compare* to recognize compare operations between signed and unsigned values. Some of these flags are equal to those set by -*Wall*, like -*Wuninitialized*, but this does not interfere the compiler.
- **-Werror**
  Turns every warning into an error. If this flag is not set, the compiler will continue compiling, but prints the warning. If this flag is set, the compiler will stop compiling as soon as an error occurs and append the specifier for the warning according to the following example: -*Werror=unused-value*. This specifier prints all unused variables found in the source code.

The applied compiler flags can increase the security and maintainability of the application. First of all, they avoid common programming issues, like uninitialized or unused variables and comparisons between signed and unsigned values. However, these flags only take care of the C code. Since the code was written in C++ mixed with C, dedicated flags for C++, such as -*Wdelete-incomplete* to prevent deleting a pointer of an incomplete type or -*Wuseless-cast* to find type casts to its own type, cannot be used. To gain more secure and maintainable code, the code needs to be split into different files that contain either C++ or C code. For these files additional flags should be added, as some flags can only handle code written in C++ or C, but not both. Further compiler flags, like -*Wstack-protector* to activate the stack protector and avoid stack overflows should be set to increase the already achieved level of security.

3.3.5 Logging

Logging is a crucial part of any server or client implementation, as it shows the user why something fails and what can be done against it, such as entering a password if it is required.

The typical case to write a log is to write the information to *standard error (stderr)*. This file descriptor is typically used to print errors or diagnostics on the text terminal. If the current application runs in the foreground, such as the *ls(1)* or *rm(1)*, this is a reasonable way to print any warnings or errors to the user. If the current application runs in the background, like a server implementation, the case is different. Since the server runs in the background, there is no dedicated *stderr* the information can be written to. Furthermore, what happens to information that is written far before the administrator logs into the server to check what the service is currently doing?

For this case, each system typically provides a logging service, such as *syslogd(8)*. This service offers an API, so other services can give any kind of information to this service which writes it into a file for further investigations. These files are normally located under */var/log* and contain information produced by any service running on the system. The information is usually stored into a file called *daemon* or if the system runs a mail server in a file named *maillog*. The service that communicates with the system's *syslogd(8)* implementation can decide for itself which file should be used to store its content. Different files to store logs without using the system's logging daemon can be configured in most applications as well.

Every information that has been written contains a priority. This priority indicates the importance of such information as defined in RFC5424 [47]. Usually, there are three kinds of priorities:

- **info / verbose**
  This priority indicates that the information just notices what the server has done at this particular moment. It can be used to show that a new client establishes a connection to the server, for example.

- **warning**
  This priority means a warning has occurred. For example, the connected client uses an unsafe hashing algorithm to hash its password in order to log in.

- **error**
  This priority is supposed to show that the server runs into an error. For example, when the client was using an incorrectly formatted email address to log in and the server does not know how to treat such an address.

There is often a fourth priority called *debug*, which is used to print debug information to *stderr*. To use this priority the server often runs in the foreground to allow the developer an easier way to fix problems located somewhere in the software. If the server runs in

the background again, all debug output is disabled, so filling the log file with unnecessary information is avoided.

On Android there is a mechanism similar to *syslogd(8)* named logcat [48]. This tool collects and views the system debug output of any service that uses Android logging API. This API provides macros to write log messages that have already described priorities. Such macros are available after including the header file *cutils/log.h*. The macros defined by the *cutils/log.h* are described in the following enumeration.

- **ALOGV**
  Writes verbose information to the Android logging system.
- **ALOGD**
  Writes debug information to the Android logging system.
- **ALOGW**
  Writes warnings to the Android logging system.
- **ALOGE**
  Writes errors to the Android logging system.

There is, however, another method available for writing logs by calling the dedicated C function *__android_log_print*. This function can be included by the *android/log.h* header file and allows a more flexible usage for writing logs. Each information can be set individually what allows a greater flexibility of creating log messages. The following enumerations describes the function parameters.

- **prio**
  This parameter contains the used priority already described above. By convention such priorities have the following format: ANDROID_LOG_<priority>, like ANDROID_LOG_DEBUG or ANDROID_LOG_INFO.
- **tag**
  This parameter identifies the process that wrote that log message. The tag allows listing all log messages of a single process and ignoring all log messages written by other processes.
- **fmt**
  The last parameter is a formatted string. This string allows logging process ids or content of variables if an error occurred internally.

It is also possible to print the tag using these macros by defining the preprocessor variable *LOG_TAG*, using the preprocessor statement *#define*. This variable must be redefined using the *#undef* statement to change it. For the developer it is more complex to find out where the *LOG_TAG* needs to be redefined as different files will be included at different places and each file can contain such definitions.

The macros above wrap C functions that allow the communication with the internal logging service. For portability reasons this can be improved, as these macros have to be redefined if other logging services are used. A better way to log information is to implement a local logging API used by the entire keystore service. This API wraps the logging API provided by the current operating system system, in this case it is the *__android_log_print* function. The source code itself needs to be changed just at this single part and not in every file that calls the macros. Besides, using C functions for logging makes the code both more maintainable and portable as macros that contain more functionality are often more complex to write.

3.3.6 Folder structure

In this section we will explain the folder structure with its implemented classes and functions. The classes themselves have already been described in Section 3.1. This should only be an overview of how the service was structured and were different logic can be found.

```
|-- Android.mk
|-- IKeystoreService.cpp
|-- defaults.h
|-- include
|   `-- keystore
|       |-- IKeystoreService.h
|       |-- keystore.h
|       `-- keystore_get.h
|-- keyblob_utils.cpp
|-- keystore.cpp
|-- keystore_cli.cpp
|-- keystore_get.cpp
`-- test-keystore
```

Listing 3.18: Structure original implementation

As can be seen in Listing 3.18 the service is separated into different C++ files (.cpp) with their corresponding header files (.h). There is also a Makefile (Android.mk) that contains data like compiler flags, names for the resulting program and files needed for the

compilation process of each one. The *test-keystore* file contains statements to test the keystore implementation using the command-line interface (CLI) client. It calls keystore API functions in various order to make sure nothing is broken, while developing. The files that implement the Android keystore service are explained in the following listing.

- **IKeystoreService**
  Contains implementations for the classes *BpKeystoreService* and *KeystoreArg* as well as the *BnKeystoreService::onTransact*-method implementation. All implementations were added to the Android namespace.

- **defaults**
  Contains constant declarations for the size of different keys, such as DSA or AES.

- **keyblob_utils**
  Contains functions to migrate a key from the software keymaster implementation to a hardware keymaster implementation. That implementation will be ignored in this master thesis, as hardware keymaster modules are not very common.

- **keystore**
  Contains implementations to initialize the keymaster device, the SELinux-permission checks, as well as the classes *Entropy*, *Blob* - with its corresponding structure *struct blob* -, *UserState*, *KeyStore*, *KeyStoreProxy* and the *main*-function that starts the entire service. This file contains the largest part of the service's implementation. There are different kinds of helper functions, such as *writeFully*, to write content into a file at once and its corresponding function *readFully* to read file content at once.

- **keystore_cli**
  Contains the command line interface client that implements the provided client side API calls using C-preprocessor macros.

- **keystore_get**
  Implements a function which wraps the *get* API call, to retrieve a value for a previously stored key. It is unclear why this file exists in the source tree, as its header file will never be included or used in any way in the application.

3.4 Init subsystem

The Android init subsystem is responsible for starting all system services, like keystore or network daemons, and parses a file named *init.rc*. This file contains statements written

in the Android Init Language configuration language [49] to export global variables, create symbolic links, adds content to files or creates new users for system daemons. These statements are grouped into five different categories:

- **Commands**
  Are predefined statements to create different things, like symbolic links (symlink) or to start a service (start). There are commands to mount devices into the file system (mount) or to write content into a file (write) as well.
- **Actions**
  Are named sequences of commands. Each action has a trigger it is listening on. If such a trigger is called, all action commands will be called.
- **Services**
  Are programs which will be launched by the init system and restarted if the program exits (optionally).
- **Options**
  Are modifiers for services. These modifiers avoid the start of services at start time (disabled) or change the user name for a particular service (user).
- **Imports**
  Allows the import of other *.rc* files that contain different statements. There is a general *.rc* file named *init.rc* that contains statements to start services or create users on all Android devices. There are device-dependent files as well, using the format: *init.<machine_name>.rc*. This file contains statements only called for a particular device.

The entry for the keystore service changes the user and group name and sets the default directory all data will be stored in. Furthermore, it creates a socket located under */dev/-socket/keystore* to establish a communication with the launched process - as can be seen in Listing 3.19. The file permission for this socket is 666.

```
service keystore /system/bin/keystore /data/misc/keystore
user keystore
group keystore
socket keystore stream 666
```

Listing 3.19: Keystore *init.rc* entry

The above *init.rc* entry shows that the service runs an unprivileged user named *keystore*. This prevents the system from getting compromised entirely, as an exploit only runs with the program's user rights if there is no privilege escalation. This means an invader is able to obtain privileged (root) rights by exploiting an unprivileged process.

3.5 Implemented hashing algorithm

An essential part of reading and writing data by a keystore service is to prevent corruption of content. The service must notify the user if the originally written value is not the same as the expected one. Content corruption can have many reasons, e.g. an error in the file system's implementation or the device's hard drive. Those errors must be recognized so that the user can decide what next step he wants to execute.

One way to recognize data corruption is to store another copy of it and compare the different values, while reading them. However, this leads to store data twice, which is not necessary. A far better way is to create a cryptographic hash of the content and store it instead. The benefit of this mechanism is that different content leads to different hash values and the hash values are all of a fixed size, which minimises the data that have to be stored. The original Android keystore service uses the Message-Digest Algorithm 5 (MD5) for hashing the data. This algorithm was invented by Ronald Rivest in 1992 [50] and creates a 128 bit hash value from a random message. However, the first collision has already been found in 1996 by Hans Dobbertin [51]. Although this attack did not threaten practical implementations, Hans Dobbertin recommended that MD5 should no longer be used in the future. MD5 was broken at least since 2004 [52]. The implementation of Android's keystore service started in 2012 [53], so it is unclear why they use MD5, knowing it was broken long ago. There are better hash algorithms available to ensure collision resistance, like SHA2. One could argue that the hashing value was only supposed to warn against content corruption by the file system or the hard drive itself. Then it would not need to be that cryptographically secure. However, since MD5 is no longer collision resistant, a change of the content could create the same hash, which breaks the idea of hashing content.

In Listing 3.22, the original *struct blob* implementation for hashing is shown. The important parts are the *digest* and *digested* buffer. The *digested* buffer just contains the start address of data that need to be hashed. In this case the hash contains the length of the value and the value itself. The resulting hash will be stored in the *digest* buffer.

```
    uint8_t vector[AES_BLOCK_SIZE];
    uint8_t encrypted[0]; // Marks offset to encrypted data.
    uint8_t digest[MD5_DIGEST_LENGTH];
    uint8_t digested[0]; // Marks offset to digested data.
    int32_t length; // in network byte order when encrypted
    uint8_t value[VALUE_SIZE + AES_BLOCK_SIZE];
```

Listing 3.20: Original blob hashing and encryption structure

Listing 3.20 shows how the hash is created. A look at the used *md5(3)* API shows what is necessary to create a hash from random data - see Listing 3.21. The first parameter is the buffer's start address that contains data a hash should be computed for. The second parameter describes how many bytes of the first parameter should be used. The last one contains the final hash, therefore this parameter must have space for 16 bytes (MD5_DIGEST_LENGTH) of output. The return value is a pointer to the final hash value.

```
unsigned char *MD5(const unsigned char *d, unsigned long n, unsigned char *md
    );
```

Listing 3.21: *md5(3)* API

As already mentioned, the *digested* buffer contains the start address of the input buffer and the *digest* buffer the final output. What still needs to be computed is how many data are hashed. At least the plain text value, e.g. the password, must be hashed and in this case there is space for the value type needed as well, as seen in Listing 3.22. Since encrypting the value after hashing and there is no dedicated buffer for that, the padding for AES needs to be taken care of. For this reason the length will be divided by *AES_BLOCK_SIZE* constant, which produces an integer value in this case (integer division), and multiplies it with *AES_BLOCK_SIZE* afterwards to get a multiple of this constant. As a buffer in C starts at index 0, the last index is the length of that buffer minus one: *dataLength + AES_BLOCK_SIZE - 1*.

Figure 3.4 shows the different states that are executed until the final hash is generated. The dots represent bytes that are not printable and the *0* represents the binary zero value. The *value buffer* contains the data that need to be hashed, in this case the plain text value itself. The buffer is 16 bytes in size since this is the length of the random data (the value). The *hash buffer* represents the buffer part that is used for hashing. That buffer is still the same buffer, but manipulated by pointer arithmetic shown in Listing 3.22.

```
size_t dataLength = mBlob.length + sizeof(mBlob.length);
size_t digestedLength = ((dataLength + AES_BLOCK_SIZE - 1) / AES_BLOCK_SIZE *
    AES_BLOCK_SIZE);

MD5(mBlob.digested, digestedLength, mBlob.digest);
```

Listing 3.22: Original blob hashing implementation

This buffer has the size of the computed *digestedLength*. As one can see, there are binary zeros at the start and the end. It is not that clear, why the value was moved like this and does not start at the beginning of the buffer. As every string in C ends with a binary zero, this format is a little confusing. Library functions, like *printf(3)*, only print the string until this sign occurs. If it is the first value, an iteration over the entire array is needed to show its content. The last buffer is the final hash stored in the *struct blob*'s *digest* buffer, which is *MD5_DIGEST_LENGTH* (16 bytes) in size.

value buffer:

.	.	F	.	b	b	T	.	.	.	.	.	C	.	.	u

hash buffer (*digestedLength*):

.	0	0	0	.	.	F	.	b	b	T	.	.	.	.	.
C	.	.	u	0	0	0	0	0	0	0	0	0	0	0	0

mBlob.digested:

.	.	.	.	s	=	u	u	.	'	h	.	G	.	.	.

Figure 3.4: MD5 value hashing

Listing 3.23 shows how the search for corruptions is implemented. The hash will be computed using the part of the encrypted data that represents the original value and will be compared by the *memcmp(3)* standard library function. If both hashes are the same, there is no corruption.

The used *md5(3)* API will be changed to a hashing algorithm that is state-of-the-art, as already mentioned. For the new implementation the SHA512 hashing algorithm will be implemented. It creates a 512 bit hash value and there are no attack vectors known today. The other critical part that is revealed here, is the missing return value check.

```
digestedLength = encryptedLength - MD5_DIGEST_LENGTH;
uint8_t computedDigest[MD5_DIGEST_LENGTH];
MD5(mBlob.digested, digestedLength, computedDigest);
if (memcmp(mBlob.digest, computedDigest, MD5_DIGEST_LENGTH) != 0) {
  return VALUE_CORRUPTED;
}
```

Listing 3.23: Original blob compare hash implementation

If the hash fails to be created correctly, there is no notification to the user. It is always assumed the hash will be created correctly and no error will happen. If there is no such verification, the user always assumes the hash has been computed correctly even if it was not. This should be changed as well in order to notify the user if something unexpected happens. As the above source code examples indicate, there is a lot of pointer arithmetic disposed to move the data to the appropriate locations in the *value* buffer. For security reasons, this should be avoided as far as possible because it can cause very easily memory corruption, therefore, a programmer must be very careful using this technique. Furthermore, it makes the code less maintainable as well. For an outsider it is not easy to understand what the code's actions are and this complicates the error diagnostics process. This part can also be simplified to increase the software's capability of portability and quality in general. The last part that needs to be considered is the general format of plain text that needs to be encrypted. Is it really necessary to encrypt the hash of the value? If an attacker can get access to the hash and manipulate it, the encrypted value must be changed, too. However, this can only be done if the decryption password is known, which depends on the user's password. One could argue that the hash belongs to the data and the encryption of both encapsulates data that belong together, but it does not increase the security.

3.5.1 Master key generation

As already introduced in Section 3.1, all key-value pairs are encrypted with a 16 byte random master key. The advantage of a key instead of a password is the resistance against automated brute-force attacks as a key is mostly larger and more accidental than a noticeable password. To protect the master key, the original Android implementation uses the *PKCS5_PBKDF2_HMAC(3)* [54] OpenSSL library function to create a key based on the user's password as seen in Listing 3.24. With this key the master key is encrypted to prevent unauthorized access.

```
PKCS5_PBKDF2_HMAC_SHA1(reinterpret_cast<const char*>(pw.string()), pw
    .length(), salt, saltSize, 8192, keySize, key);
```

Listing 3.24: Master key generation

Listing 3.24 creates a key based on the user's password (pw) combined with a salt value (salt) that is attached to the password inside the *PKCS5_PBKDF2_HMAC(3)* function. The salt value is a random 16 byte value that is generated every time a new master key is created. The iteration count is 8192. This count makes the derivation of keys slower the higher it is. This cumbers brute-force search attacks to find the key automated. The attacker has to use the same iteration count to start his attack. The man page recommends a value higher than 1000. The result is the derived key, which is stored in the *key* buffer.

The salt is stored at the end of the *value* buffer, which the *blob* structure contains, as can be seen in Listing 3.25. The salt is represented by the *info* pointer and its length by the *infoLength* value.

```
mBlob.info = infoLength;
memcpy(mBlob.value + valueLength, info, infoLength);
```

Listing 3.25: Storage of the master key salt

Storing the salt value is necessary to generate the same key derived from the user's password to decrypt the master key. If a key-value pair will be encrypted or decrypted, the key is derived from the user's password with the associated salt and the master key is decrypted with that particular key. After the master key has been decrypted successfully, all key-value pairs can be encrypted and decrypted. To avoid many master key decryptions, it is stored inside the *UserState* class after successful decryption as explained in Section 3.1.2.

This is a recommended way to derive keys from passwords as discussed in Section 2.4.2. Unfortunately, there are no other key derivation functions available that can be used on Android as well as on other operating systems, like *bcrypt(3)* [55] or scrypt [56], as they are not implemented in OpenSSL. *bcrypt(3)* is based on the blowfish symmetric encryption algorithm, while scrypt is based on SHA256. Both algorithms are designed to increase the costs (CPU usage, memory) per key derivation, more than PBKDF2 does. Indeed, the iterator count can be increased, but PBKDF2 has little memory and CPU usage what makes brute-force attacks using ASICs or GPUs relativly cheap. ASICs (application-specific integrated circuit) are designed for a particular use, e.g. scanning for special hash values. GPUs are designed for rapidly manipulating computer graphics and image processing. Such

circuits perform a dedicated task faster than a normal CPU does. For this reason, they can be used to implement attack vectors against encryption or hashing algorithms, which are to slow on normal CPUs.

3.6 Implemented encryption algorithm

Ensuring that no unauthorized user has access to stored passwords is a crucial part of a keystore implementation. There are different areas of protection for such data. The first one is to make sure that no other user but the keystore user has access. For this reason, all *blob*s are stored under this user id in a folder that only this user can access. The second one is that the keystore must guarantee that only the authorized user has access to them. As described in Section 3.1, the user id is part of the file name. This allows the user to verify the given user id against the stored one to permit access or not. The third one is the protection against the super user *root* and bugs in other software components of an operating system. As the super user can access every file in a file system, he can manipulate such files as well. Another problem are bugs which lead to privilege escalation, like executing code with super user rights, even if the software itself does not have such privileges. To avoid such attacks against manipulation by unauthorized users is encryption one solution. If a file is encrypted and the super user has access to it, he can still manipulate the file, but without the right password the data cannot be manipulated without recognizing it. The decryption will fail in this case, so the authorized user can be informed that there is something wrong with this file.

The keystore implementation of Android uses the AES symmetric encryption algorithm - see Section 2.3 for further explanations. The key size is set to the smallest possible value of 128 bits (*MASTER_KEY_SIZE_BITS*) by calling the *AES_set_encrypt_key* and *AES_set_decrypt_key* pre-defined OpenSSL library macros. These macros set the AES key for both encryption and decryption operation for a given key, which is the derived key based on the user's password in this case. The resulting key (*aes_key*) is used for both operations instead.

```
            uint8_t vector[AES_BLOCK_SIZE];
            memcpy(vector, mBlob.vector, AES_BLOCK_SIZE);
            AES_cbc_encrypt(mBlob.encrypted, mBlob.encrypted, encryptedLength
    , aes_key, vector, AES_ENCRYPT);
```

Listing 3.26: Original blob encryption implementation

Listing 3.26 shows how the encryption of a value is implemented. The initialization vector must be the same for encryption and decryption. Otherwise both operations are not inverse to each other. For this reason, it cannot be part of the encrypted value itself. The pre-defined OpenSSL library macro *AES_ cbc_ encrypt* expects an input buffer as the first parameter and a buffer the output is stored to as the second one. In this case both are the same, so after successful encryption, there is no way to access the plain text value. The length that should be encrypted is the length of the value as well as the length of the hash value since it is part of the encrypted value: *encryptedLength* = *digestedLength* + *MD5_ DIGEST_ LENGTH*. This does not affect the calculation of the *AES_ BLOCK_ SIZE* multiple as described in Section 3.5 since the *MD5_ DIGEST_ LENGTH* is 128 bit, too. The start address for the encryption is marked as *mBlob.encrypted* as seen in Listing 3.20. The last parameter is the mode that indicates whether this macro is used for encryption or decryption. In this case it is *AES_ ENCRYPT*.

```
            AES_cbc_encrypt(mBlob.encrypted, mBlob.encrypted, encryptedLength
    , aes_key, mBlob.vector, AES_DECRYPT);
```

Listing 3.27: Original blob decryption implementation

Listing 3.27 explains how decryption is implemented. The input and output buffer are still the same and the previously stored initialization vector is used. The length of the value that needs to be decrypted is *encryptedLength* and will be calculated by removing the part of *struct blob* that is not involved in this process from the entire file length. The last parameter is *AES_ DECRYPT* that shows the input value is now the encrypted one. After encryption, the length of the hash value (*MD5_ DIGEST_ LENGTH*) is subtracted from the length of the encrypted value, which results in the length of the original data (*digestedLength*).

Figure 3.5 shows what the buffer looks like before and after encryption. The first part is the MD5 hash value while the second part is the value itself - see Section 2.4. The structure of the plain text buffer reveals that the hash value is not followed by the data directly. This is how the encryption length is computed and that pointer arithmetic is used.

The AES encryption algorithm with a key size of 128 bit is still state-of-art, but it should be increased up to 256 bit. In a context of encryption, the stored data must be secured as far as possible because it is unknown what kinds of attack vectors come into existence in the next years. A 256 bit key size has a larger search space for keys than 128 bit. There is also a psychological aspect of changing the key size up to 256 bit. 256 bit is a higher value than 128 bit and, therefore, it suggests a higher security level.

plain text buffer:

.	.	.	.	s	=	u	u	.	’	h	.	G	.	.	.
.	0	0	0	.	.	.	F	.	b	b	T	.	.	.	.
C	.	.	u	0	0	0	0	0	0	0	0	0	0	0	0

encrypted text buffer:

.	.	.	2	.	.	.	.	.	.	.	.	C	N	.	h
x	J	.	.	4	:	f	.	.	’	.	.	.	.	5	k
U	b	w	.	.	2	.	.	.	.	.	4	v	N	l	.

Figure 3.5: Original plain text encryption

Even if there is no technical decision to change that value, it will obtain a greater acceptance of users as they think it is more secure. This will increase the acceptance of the keystore service in general.

Furthermore, the macro for encryption and decryption should be changed. As AES is a symmetric encryption algorithm, there is no need to set a key for both operations because the key is the same. The OpenSSL library provides an API called *evp_ encryptinit(3)*, which defines dedicated functions for setting the key, encrypting and decrypting the value, and saving the resulting value into the given buffer. Since these functionalities are separated, it is possible to inform the user at each point of processing if something goes wrong. In the original Android keystore service implementation neither the service nor the user will be informed if the encryption, decryption, or hashing of data fails.

3.7 Result

The result of analysing the Android keystore service arrives at the conclusion that the services implementation will be used for further development, instead of starting from scratch. The design splits different parts into different classes and uses the Android RPC service for communication. This makes further maintenance and source code reading easier, which avoids new errors and helps finding old ones. Each class wraps its own logic and is called by provided methods from other classes. However, different classes are stored in a single file and there are functions written in C inside the C++ files, which causes some problems for the compiler and the associated flags. For this reason, it is better to separate C code from *C++* code and to split different classes into different files. This increases code qual-

ity even further. For some classes, such as the *Entropy* class, it is not necessary to wrap its functionality into a *C++* class, since there is no interaction between the implemented methods that justifies the encapsulation of attributes.

Performing a static analysis using Clang's *scan-build(1)* tool, which is part of the LLVM compiler infrastructure [57], results in the Figure 3.6. This tool is used for statically analysing C/C++ source code with respect to different kinds of errors, such as uninitialized variables or values that are set, but never read. There were two different kinds of errors found by *scan-build(1)*: "unknown type name" and "expected unqualified-id". While "unknown type name" indicates a missing explicit include of a header file that defines the type name, "expected unqualified-id" often indicates a syntax error. In this case it is a subsequent fault as the correct header files are not explictly included, which defines the type name.

Adding additional *gcc(1)* compiler flags: -*Wshadow*, -*Wmissing-declarations* and -*Wold-style-cast* to analyse the source code of the original implementation results into Figure 3.7. -*Wshadow* means one variable hides another according to its visibility area. E.g. a function parameter has the same name as a local variable. The local variable hides the one given as parameter, so it is no longer accessible. -*Wmissing-declarations* means a global function is defined without a previous declaration. -*Wold-style-cast* means a C style type cast is used instead of a C++ one. In C++ different types of type casts are defined to gain better control about the resulting value - see Section 4.2.1 for further explanations. An aggregation of both logs can be found in Appendix A.

error	count
unknown type name	24
expected unqualified-id	3

Figure 3.6: *scan-build(1)* errors

gcc(1) flag	count
-Wshadow	12
-Wmissing-declarations	3
-Wold-style-cast	42

Figure 3.7: *gcc(1)* flag errors

Even if the issues in the above figures will not necessarily lead to exploitable code, it is bad coding style. Such a coding style can confuse the developer and decrease maintainability. If a compiler flag warns about an issue, it is always good to try to fix it, as such warnings have their warrant. If they cannot be fixed for some reason, make sure that they will not lead to an error.

For encryption and hashing of data, the services use an AES key length of 128 bit and the MD5 algorithm that is not considered as secure, especially in such a sensitive context. The OpenSSL library provides better hashing algorithms and greater key length to mitigate

these problems. The different return values that indicate success or failure of a function must be checked as well to inform the user if an error occurs.

Issues found while analysing the source code are solved in both versions, which are discussed in the following chapter: the new Android implementation and the portable one as far as possible. For instance, an issue located in the client-server communication cannot be solved in the portable version as there is no *IBinder* interface available. This communication has to be re-implemented completely. The main goal is to use most of the source code from the new Android implementation in the portable version. Only parts that cannot be shared between these two applications should be re-implemented. Apart from this is the API defined by the *IKeystoreService*, described in Section 3.1, providing a huge set of calls that may be necessary on Android devices, but not necessarily on other platforms. It has to be evaluated whether the set of API calls can be decreased and for which implementation.

A general problem in server development are the necessary user rights and the amount of functionality that leads to larger code size as normal utilities do. A great amount of functionality uses more code, which might lead to more errors. To compensate for this, a technique called privilege separation [13] will be implemented for this service. This technique splits the software into smaller pieces, which communicate with each other. If and how this can be implemented in both versions will be explained in the following chapter.

Chapter 4

Implementation of the new keystore service

This chapter documents the new implementation of the Android keystore service, based on the results identified in Chapter 3 as well as the portable version based on it. The first part in this chapter discusses a new service based on the results found during the analyses of the source code of the original Android keystore service. This includes the usage of a modern hash algorithm instead of MD5 and a well-documented API for encryption and decryption that allows more control about the single steps of encryption and decryption. Function calls like *malloc(3)* and *atoi(3)* calls are improved with respect to security or turned into more secure functions. Another important part in this area is the introduction of a common coding style and a more efficient usage of C and C++ source code. This includes adding additional compiler flags to support developers as well. All function calls marked as *TODO* in the CLI client are implemented to provide a fully functional client for testing the new implementation.

The second part contains the introduction of a portable version, which has been introduced in Chapter 1. All parts of the new Android keystore implementation that are portable will be shared with this version. First, this is done to avoid writing the same code twice and second, errors occurring in one version can be fixed easily in the other. Those parts that cannot be shared between both versions will be reimplemented to provide equal functionality, except for those functions that were identified as unnecessary - see Section 3.2 for more information. The design is based on privilege separation [13] to mitigate the possible attack vectors by splitting one process into several, dropping their privileges and locking those child processes down into different directories. As the portable version is based on the OpenBSD operating system, it will be described how libraries, only available on this

platform, can be used on other platforms as well, like GNU/Linux or other BSD derivatives. OpenBSD has a tradition in porting software to other platforms, such as OpenSSH or OpenSMTPD. Thus, the portable version can profit by perceptions already made in these projects.

In summary, this chapter is split into two parts: the introduction of the new Android keystore version and a portable version based on it. All differences to the original implementation will be explained and why they were chosen. Furthermore, it considers what has need to be taken care of if a portable version of an application that is limited to one singular operating system came into existence. At the end of this chapter there is a result section that gives an overview of the new and portable implementation and how they fit to the results of Chapter 3.

4.1 Design

As already mentioned in Section 3.7, the internal design only can be changed to avoid breaks of external clients that have already been implemented. This means that the API required for communication between server and client remains untouched. The SELinux implementation of roles is still the same since this is a common security feature on Android and is always enabled. For this reason, the new design will not break already implemented security features, but only extend them if necessary. For the portable version the situation is different, but this will be explained in Section 4.4.

4.2 Implementation

The new implementation of Android's keystore service is based on the results gained while analysing the original source code. For this reason, we only consider the parts that have been changed to the original implementation as all other parts are explicitly described in Chapter 3, such as the master key derivation, the *Binder* interface implementation of the keystore service, or the internal master key management by the *UserState* class.

The first part was changing the hash algorithm from MD5 to SHA512, as well as using well-documented APIs offered through the OpenSSL library. Those are low-level APIs (*EVP_ **) which allow better control of each step, from initialisation to finalisation. Such calls as well as the way they are implemented is explained in Section 4.3. To implement a more intuitive hashing and encryption functionality the *blob* structure was changed as shown in Listing 4.1.

```
    uint8_t salt[SALT_SIZE];
    uint8_t vector[AES_BLOCK_SIZE];
    uint8_t hash[SHA512_DIGEST_LENGTH];
    uint32_t length;            /* len after encryption in network byte order */
    uint8_t encrypted[VALUE_SIZE + AES_BLOCK_SIZE];
    uint8_t value[VALUE_SIZE];
```

Listing 4.1: New blob structure

Listing 4.1 shows how the new *blob* structure was implemented. The main difference to the original, which is described in Section 3.1.2, is that the buffer for the encryption text (encrypted), the plain text (value), and the salt are separated to avoid the pointer arithmetic, which is described in Section 3.5. After encrypting the plain text buffer successfully, it will be deleted by the *memcpy(3)* function. This avoids further leaking of such data. On the one hand, more space is needed in memory to hold the data, but on the other hand, the functionality to encrypt and decrypt a plain text value is more intuitive and easier to understand which increases the code quality.

The second part was splitting C source code from C++ source code, to obtain better compiler flag managing by introducing a universal coding style. An overview about the added compiler flags is described in Section 4.2.1. Furthermore, a generic logging API that wraps the Android-specific logging API functions was implemented - as shown in Listing 4.2. This is also part of the splitting process as Android-specific macros are not automatically portable to other operating systems. The underlying logging API was already introduced in Section 3.3.5.

```
void log_info(const char *msg, ...);
void log_debug(const char *msg, ...);
void log_warn(const char *msg, ...);
void log_err(const char *msg, ...);
```

Listing 4.2: New logging API

The new logging API are supposed to be used inside the entire application to create different kinds of priority messages. For portability reasons such calls can still be used, the only change is made inside such calls, which is far more maintainable than changing all calls producing logs.

At last, a function to keep track of file permissions was added. This check will be executed at every read and write access of a file and recognizes if those permissions are too powerful. Only the keystore service user should be able to write or read files that contain

either the master key or key-value pairs. If such permissions were manipulated in a way the group the keystore user belongs to or all other users are able to access them, either by accident or by an attacker, such files must be seen as compromised. Listing 4.3 shows how this check was implemented.

```
if (fstat(fd, &st) < 0)
  return SYSTEM_ERROR;

if ((st.st_uid == getuid()) && (st.st_mode & 077) != 0) {
  log_err("unprotected file: %s", filename);
  log_err("permission 0%3.3o for %s so too open", (u_int)st.st_mode & 0777,
   filename);
  return SYSTEM_ERROR;
}
```

Listing 4.3: File permission check

To check the Unix file permissions the *fstat(2)* system call is used, as it fills information, such as the user id and the inode protection mode, into the given *struct stat* structure. After calling *fstat(2)* this structure is used to retrieve the file owner's user id *st_ uid* as well as the file's inode protection mode *st_ mode*. If the file owner is the current process (keystore service), but the file permissions are too powerful - in this case it means both the group and other have the read-write-execute (RWX) mask -, the file could possibly be compromised or allows others to read them. Checking file permissions to make sure only the user has full access to the file, the inode protection mode will be combined by the bitwise AND operator (&) with the value 077. The leading 0 indicates that it is an octal numeral. If a file needs to be protected against being accessed by the user's group as well others and had a file permission mode of 640, the result would be: 0640 & 077 = 040, while a permission mode of 600 leads to: 0600 & 077 = 0. The value 0777 leads to the same value which the first operand has: 0640 & 0777 = 0640.

4.2.1 Compiler Flags

As already mentioned in Section 3.3.4, additional compiler features to gain more security and maintainability with respect to the source code can be added by compiler flags. A full list of all compiler flags used for the Android version can be found under Appendix D.

The main difference to the previous version are the separated flags for C and C++ files. As the entire program was split into different files, the compiler flags had to be separated in order to get optimal support form the *gcc(1)* compiler. These separation can be achieved by

using the *LOCAL_CFLAGS* variable for C code and the *LOCAL_CPPFLAGS* variable for C++ code located in the given Makefile. This section only describes a subset of all added flags for both languages.

- **-fstack-protector**
  This option enables the stack protector already explained in Section 2.2. The additional source code that will be added to the program by this flag forces the program to terminate and print an error to *stderr*. This error contains the stack trace of the program to help the developer to fix the bug.

- **-Wreturn-type**
  This option checks if there is a return value specified for a C function, but the code inside the function does not call the *return* statement with an object or variable of this type. This will make sure that whenever a return value is expected according to its definition the function will return such type. The only exception is the *main* function and - according to the man page - functions defined in the system headers.

- **-Wshadow**
  This option warns if a local variable shadows another local variable, parameter or global variable. Shadowing means it has the same name as the variable or object it shadows.

- **-Wuninitialized**
  This option warns if a variable or object is read, but has not been written before. In C all variables are uninitialized except if they are static. Not initializing these variables can lead to a security breach or program crash, as they contain the same data previously written into this memory area. This often causes indeterministic behaviour of the program.

- **-Wold-style-cast (C++ only)**
  This option warns if there is an old-style cast to a non-void type. C++ defines new-style casts that are less vulnerable to unintended effects. Such casts include *dynamic_cast*, *static_cast*, *reinterpret_cast*, and *const_cast*.

  - **static_cast**
    This option performs a safe and relatively portable cast of one type to another. This cast option is often used to explicitly document a cast, the compiler would perform automatically, e.g. *bool* to *int*. While this shows the programmer's

intention, a static cast is necessary to convert a *void* pointer into a different type. This cast makes an implicit cast explicit, which supports the developer while reading the source code.

- **dynamic_cast**
  This option uses runtime information to perform a conversion, in contrast to a static cast. This operator is used to perform a *downcast*, e.g. cast a base class pointer into a derived class pointer. This can be dangerous if a variable is accessed that is not part of the base class.

- **reinterpret_cast**
  Such a cast should be used with great care as it converts data into stream bytes and a byte array must always be large enough to store the data, like converting an address of an *int* value into an *unsigned char* pointer. It is often used in low-level applications. It reminds the reader that an unsafe conversion takes place and that the programmer is responsible for the result.

- **const_cast**
  This option adds or removes the *const* and *volatile* qualifications. This avoids calling a non-*const* function on a *const* object. If the design requires such a declaration, this cast operator is used to remove the objects's *const* qualifier to call its method.

- **-Wreorder (C++ only)**
  This option warns if the order of the given class attribute initializers does not match the order in which they must be executed. If a class defines two attributes named i and j, but the class attribute initializer specified by the constructor initializes j first and i afterwards, this warning will be triggered.

- **-Woverloaded-virtual (C++ only)**
  This option warns if a method declaration hides a virtual function of its base class. E.g. the base class contains a method declaration for *void f()* and the derived class contains a method named *void f(int)*. The first method is hidden by the second one, so calling the first one will lead to a compile failure.

All warnings are turned into errors using the -*Werror* flag, as the compile process stops at the point a warning occurs. This makes sure that no warning is overlooked and could lead to a failure at runtime.

4.2.2 Folder structure

The folder structure displayed in Listing 4.4 represents the new Android keystore service implementable. Compared to the original one described in Section 3.3.6 all classes were split into different files and source code written in C was separated from that written in C++. Furthermore, if no object orientation was needed, it was removed and rewritten in C as in *blob.c*, *entropy.c*, *log.c* and *permission.c*.

```
|-- Android.mk
|-- IKeystoreService.cpp
|-- blob.c
|-- entropy.c
|-- include
|   `-- keystore
|       |-- IKeystoreService.h
|       |-- blob.h
|       |-- defines.h
|       |-- entropy.h
|       |-- keyblob_utils.h
|       |-- keystore.h
|       |-- keystore_get.h
|       |-- keystoreproxy.h
|       |-- log.h
|       |-- permission.h
|       `-- userstate.h
|-- keyblob_utils.cpp
|-- keystore.cpp
|-- keystore_cli.cpp
|-- keystore_get.cpp
|-- keystored.cpp
|-- keystoreproxy.cpp
|-- log.c
|-- permission.c
|-- test-keystore
`-- userstate.cpp
```

Listing 4.4: Folder structure of new Android implementation

The file name mirrors the class names that have already been described in Section 3.1. Therefore, in this section only the additional file names will be explained regarding its content:

- **blob**
  Contains the rewritten structure for storing data as well as its functions in C. Most of the code was reused, since the C++-methods were changed into C-functions and all C++ type casts had to be removed.

- **entropy**
  Contains a C-function to retrieve the entropy by using the */dev/urandom* device. As there is no need for object orientation here, it was removed. Furthermore, the device is closed after receiving the random values instead of closing it in the destructor as implemented before. This prevents the device from being opened as long as the *Entropy* object exists, which can avoid a regular close if the application crashes during this time.

- **log**
  Contains the new logging API that wraps the operating system specific ones. These files (.c and .h) were completely reimplemented.

- **permission**
  Contains all functions necessary to perform SELinux-permission checks. As in the *blob* file, all old casts had to be removed as well as C++ objects, like *String8*. Such objects were changed to *const char*-pointers. The strings are already internally represented as const char pointers.

4.2.3 Application Programming Interface

The Application Programming Interface (API) remains unchanged for two reasons. First, the Android application stack already implemented it and this stack had to be changed as well if some functions the keystore interface offers were deleted. However, this is not part of this thesis. Furthermore, Android applications that have already implemented the API would be constrained in their execution. Second, the command-line-interface client does still support all calls the keystore service API offers. This increases maintenance and allows better testing by the developer, as he does not need to implement an Android application that calls every Android API that uses the keystore implementation.

4.3 Security

In this section, we will introduce the hashing and crypto APIs for the new design as already mentioned in Section 3.7. It shows how they were implemented and which different steps

should be considered. Furthermore, the subsections describe the benefits regarding security and maintainability of these APIs to give a short overview of how they work and why they should be used.

4.3.1 Hashing

As already mentioned in Section 3.5, the MD5 hashing algorithm cannot be considered secure any more. The first part regarding the new implementation is to use a state-of-the-art algorithm, so an adequate level of security is gained, for the hashing of data. In the second part the implemented API will be changed in order to be more maintainable and to allow more control about the different steps of hashing. This new API is *EVP_DigestInit(3)* and is part of the OpenSSL library. These functions are more generic and allow the change of hashing algorithms in an easy way without changing the entire function. Since there is no dedicated function to create an SHA hash as there is for MD5, the usage of a new algorithm will cause a change of the entire function call. This can be avoided by using more generic functions as explained in the following sections.

EVP_DigestInit_ex(3) function call

This call initializes the hash context regarding the chosen hash algorithm. In this case, SHA with a 512 bit hash value will be used. The last parameter of this function is the *ENGINE* that allows the usage of another implementation of the chosen hash algorithm. The default value for this case is *NULL*, as can be seen in Listing 4.5. Before the context can be initialized, there is a context needed. The *EVP_MD_CTX_create(3)* API call provides this functionality.

```
ctx = EVP_MD_CTX_create();
if (!EVP_DigestInit_ex(ctx, EVP_sha512(), NULL)) {
  log_err("EVP_DigestInit_ex failed");
  goto fail;
}
```

Listing 4.5: *EVP_DigestInit_ex(3)* function call

EVP_DigestUpdate(3) function call

The *EVP_DigestUpdate(3)* function stores the computed hash value in the given context. In this case the hash value of the encrypted value is computed to make sure the encrypted value was not changed during storage.

```
if (!EVP_DigestUpdate(ctx, blb->encrypted, sizeof(blb->encrypted))) {
  log_err("EVP_DigestUpdate failed");
  goto fail;
}
```

Listing 4.6: *EVP_DigestUpdate(3)* function call

EVP_DigestFinal_ex(3) function call

The *EVP_DigestFinal_ex(3)* function finalizes the hashing process and it writes the hash value into the given *hash* value. The *len* value contains the length of the given buffer. As this function is generic with respect to different hash algorithms, it is not certain each buffer has the same size.

```
if (!EVP_DigestFinal_ex(ctx, blb->hash, &len)) {
  log_err("EVP_DigestFinal_ex failed");
  goto fail;
}
```

Listing 4.7: *EVP_DigestFinal_ex(3)* function call

EVP_MD_CTX_destroy(3) function call

The last function that needs to be called is the *EVP_MD_CTX_destroy(3)*. This function cleans up the context and frees memory that was allocated internally. It is very important not to forget this function. Otherwise a memory leak occurs.

```
EVP_MD_CTX_destroy(ctx);
```

Listing 4.8: *EVP_MD_CTX_destroy(3)* function call

The implementation of this API reveals that it is quite generic and hash algorithms can be changed easily. This increases the maintenance of the source code for further development.

4.3.2 Encryption

This chapter introduces the encryption API implemented in the new version of Android's keystore service and explains reasons why it is a better approach to use it. Each used call will be described according to its parameters and return values in order to provide

additional information to the user why both encryption or decryption failed. For example, decryption fails if a wrong password is used or if the initialisation vector is not the same than for encryption. In fact, the initialisation vector is stored in the file system, but it can still be corrupted because of file system or hard drive errors. As the original encryption algorithm is AES, the new design of the Android keystore uses the same algorithm as well. This algorithm is state-of-the-art today and there is no reason to change it.

EVP_EncryptInit_ex(3)

This function call is used to initialize all necessary data involved in encrypting a buffer. Such data are the cipher context, the type, the encryption engine, if necessary, and the key with its initialization vector. The cipher context stores all information needed to encrypt or decrypt a buffer, such as the key and the chosen encryption algorithm. The encryption engine is optional, as it allows the usage of alternative implementations of the specified algorithm. As Listing 4.9 shows the new encryption algorithm is still AES, but with a key size of 256 bit. This key size is the largest being supported for AES and guarantees a maximum of security at the encryption level. The *EVP_CIPHER_CTX_init(3)* function that is part of the OpenSSL library as well, must be called additionally to initialize the cipher context without any arguments.

```
EVP_CIPHER_CTX_init(&ctx);
if (!EVP_EncryptInit_ex(&ctx, EVP_aes_256_cbc(), NULL, key, blb->vector)) {
  log_err("EVP_EncryptInit_ex failed");
  goto fail;
}
```

Listing 4.9: EVP_EncryptInit_ex(3) function call

EVP_EncryptUpdate(3)

The *EVP_EncryptUpdate(3)* functions writes the cipher text according to the chosen encryption algorithm and plain text into the buffer named as *encrypted*, as Listing 4.10 shows. This buffer has to be of the same size as the *value* buffer is combined with the size of 128 bits defined by the *AES_BLOCK_SIZE* constant in order to have a buffer large enough to store each encrypted block.

```
if (!EVP_EncryptUpdate(&ctx, blb->encrypted, &len, blb->value, sizeof(blb->
  value))) {
  log_err("EVP_EncryptUpdate failed");
  goto fail;
}
```

Listing 4.10: EVP_EncryptUpdate(3) function call

EVP_EncryptFinal_ex(3)

Padding is enabled by default and this is done by the last step. It encrypts the final data, that is any data remain in the partial block, as can be seen in Listing 4.11. For this case the *PKCS* padding is used. The sum of the *len* and the *tmplen* value results in the length of data that were encrypted. This value must be remembered as it is necessary to decrypt the value correctly.

```
if (!EVP_EncryptFinal_ex(&ctx, blb->encrypted + len, &tmplen)) {
  log_err("EVP_EncryptFinal_ex failed");
  goto fail;
}
```

Listing 4.11: *EVP_EncryptFinal_ex(3)* function call

EVP_CIPHER_CTX_cleanup(3)

The final part after encrypting a buffer is to clean up all information needed for encryption and free all internal allocated buffers. That is what the *EVP_CIPHER_CTX_cleanup(3)* function does, as can be seen in Listing 4.12. It is important to call this function in order to avoid memory leaks.

```
EVP_CIPHER_CTX_cleanup(&ctx);
```

Listing 4.12: *EVP_Cipher_CTX_cleanup(3)* function call

EVP_DecryptInit_ex(3)

The decryption works the same way as the encryption as it is the inverse operation. The OpenSSL library provides functions for this purpose as well. First, the cipher context has to be initialized. There is no dedicated function to initialize the context itself for encryption

and decryption. That is why the *EVP_CIPHER_CTX_init(3)* must be called for both, as Listing 4.13 shows. Only the function to initialize the cipher context with the encryption algorithm, the key, and the initialisation vector is different from the one used for encryption.

```
EVP_CIPHER_CTX_init(&ctx);
if (!EVP_DecryptInit_ex(&ctx, EVP_aes_256_cbc(), NULL, key, blb->vector)) {
  log_err("EVP_DecryptInit_ex failed");
  goto fail;
}
```

Listing 4.13: *EVP_DecryptInit_ex(3)* function call

EVP_DecryptUpdate(3)

As described in Section 4.3.2, the *update*-function encrypts plain text into the cipher text. By inverting this process the *EVP_DecryptUpdate(3)* function decrypts the cipher text and stores the resulting plain text into the *value* buffer, as can be seen in Listing 4.14.

```
if (!EVP_DecryptUpdate(&ctx, buf, &len, blb->encrypted, blb->length)) {
  log_err("EVP_DecryptUpdate failed");
  goto fail;
}
```

Listing 4.14: *EVP_DecryptUpdate(3)* function call

EVP_DecryptFinal_ex(3)

This function decrypts the padding previously encrypted, as can be seen in Section 4.3.2. The sum of the *len* and *tmplen* values is the length of plain text data.

```
if (!EVP_DecryptFinal_ex(&ctx, buf + len, &tmplen)) {
  log_err("EVP_DecryptFinal_ex failed");
  goto fail;
}
```

Listing 4.15: *EVP_DecryptFinal_ex(3)* function call

The final step is to call the *EVP_CIPHER_CTX_cleanup(3)* as already introduced in Section 4.3.2 to make sure every allocated memory will be freed. These functions submit a better source code structure than the original encryption implementation as their purpose

is more obvious because of their names. The purpose of a function should always be clear by its name and not by its given parameters. This simplifies the source code and increases its quality significantly.

The benefit of implementing these functions instead of using the *AES_ cbc_ encrypt* API introduced in Section 3.6 is that in each step the return value can be checked to provide more detailed information to the user. Furthermore, only one key is needed and there is no need to call additional functions to set an encryption as well as a decryption key. This can be confusing, as AES uses the same key for both operations, but these functions suggest that two different keys are needed. There is a flag needed to tell the function if the data are supposed to be encrypted or decrypted as well. Due to the fact that all functions for encryption starts with the prefix *EVP_ Encrypt* and all functions for decryption starts with the prefix *EVP_ Decrypt*, each function's purpose is obvious. There is no dedicated parameter (flag) needed to change the behaviour of a function as in the original keystore encryption implementation.

The last important point are the man pages available for all *EVP_ ** library functions. These man pages help the developer to use such functions, which avoids the occurrence of errors since they explain the function's behaviour as well as the expected return values. Specifically, in an environment as the keystore service, the safe storage of user credentials is a crucial component.

4.4 Portability

This section describes the new design and its corresponding implementation for the portable version of the Android keystore service. It will explain why the design is slightly different in some cases and which parts remain the same for both implementations. Furthermore, it will give an overview of how services can be developed in general to gain more robustness against external attacks and how OpenBSD-only functions can still be used even in portable versions of the software. Some of these functions behave in a more secure way and increase the security of system significantly.

Figure 4.1 illustrates the design of the portable version. The entire service was split into two single processes *Blob* and *Service*. The *Blob* service manages the read and write of data and the *Service* process communicates with clients. It sends the data of the clients to the *Blob* process which reports back whether this was successful or not. It also requests the already stored data from the *Blob* process to send it back to the requested client.

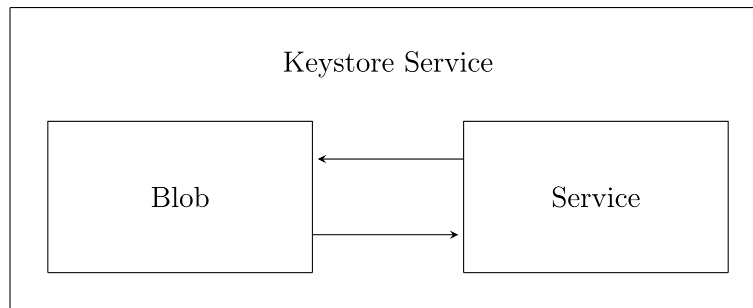


Figure 4.1: Portable keystore service design

This is the principal part of privilege separation, splitting one single process into multiple where each process is responsible for a certain task. The challenge is to find out what the certain tasks are and how many processes are needed to encapsulate them. Separating a single process into too many processes makes the code that handles the communication more complex. However, separating it into too less processes can be a security issue. A good trade-off is to look at the functionality and choose the number of processes after that. The keystore implementation talks to different clients by using a defined API, manages different user states and stores data into the file system. The different states depend on the master key file and are returned to the client. As this functionality is located in the middle of the client communication and storage of data, but is more related to the communication side, the new design splits the single process into the two already described above.

How the process separation was implemented, which library calls are necessary, and which aspects need to be considered if such a design was chosen will be explained in the following section.

4.4.1 Design

The new design implements the already introduced privilege separation technique and reuses the storage capability of the original implementation. Furthermore, the internal behaviour was simplified to decrease the lines of source code, which reduces the potential number of errors. The new design includes a client API for communication with the service that is able to interact with other programming languages, such as *Python* or *Lua*. This makes the design more powerful since it can be used with many different applications.

Figure 4.2 shows the new design as well as how the communication between the server and the client was implemented.

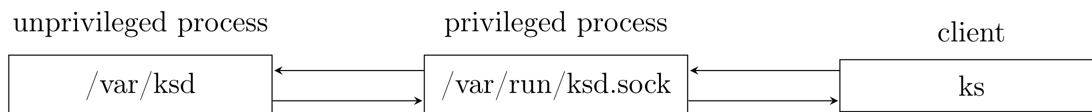


Figure 4.2: Design of the portable version

The entire implementation is split into a server named keystore daemon (ksd) and a client named keystore client (ks). As the daemon uses privilege separation, it is split into two processes internally. The following enumeration explains the three different parts, what purpose they have and why they are necessary:

- **Privileged process**
  This process is the main part of the entire service and forwards the requests sent by the client to the unprivileged process as well as its responses back to the client. Basically, this process is just an intermediate between the client and the unprivileged process at runtime. To communicate with the client, a local *Unix* TCP socket is used, located under */var/run/ksd.sock*. At start time, this process forks itself and the child becomes the unprivileged process. The man page can be found under *ksd(8)*.

- **Unprivileged process**
  This process is isolated under */var/ksd*. So it is not possible to access files or directories located above this directory. It runs under a non-root user called *_ksd* as well. The underscore as prefix to the user name is a convention to separate normal users from more privileged users used by system daemons or commands. This user has to be added to the system before using the keystore daemon.

- **Client**
  The client implements the API provided by the server and is called *ks*. It reads the command line parameters necessary for each server call, sends these data and processes the received data in return. The utility is CLI-oriented as this avoids the dependency to a GUI. The man page can be found under *ks(1)*.

4.4.2 Privilege Separation

Privilege separation is a technique to limit the possible security breaches of services and utilities. It splits one parent process into multiple child processes, runs every child process under an unprivileged user and locks this child process to a dedicated directory. These child processes cannot leave their dedicated directory and can only communicate with other processes using messages instead of function calls.

Imagine there is a mail server installed on a system. This mail server has different tasks, like reading its own configuration and the user database, and writing log files and emails. If these tasks are done by one single process, it needs to run under a user that has the privileges to perform all these tasks. Furthermore, it must navigate through the entire file system, as the user database is located under */etc/passwd* and the log files under */var/log* on a GNU/Linux or BSD system. This makes the single service extremely powerful what is not necessary. A more security-related approach is to split a single process into multiple, lock each down into its dedicated directory and change the user to an unprivileged one. Now, the processes only execute a subtask and do not need a powerful user id which makes them more manageable and secure.

This approach needs another kind of communication between the parent process and its children and even between the children among themselves. For this reason, an additional implementation is needed to manage such communication. Sockets will be used in this case as they implement an easy way of communication between individual processes. If there is one single process and each subtask is implemented in a different file, only their provided APIs must be called to communicate between them. As privilege separation consists of multiple isolated processes the communication cannot be implemented by calling APIs built up of C functions. This fact makes such an implementation a bit more complex, but has a high advantage on the security level as a security breach in one process cannot influence others.

It should be mentioned that separation of processes is not always necessary. If the program does not need to access the file system or network sockets, there is no benefit in splitting a process into multiple children. For instance, a calculator that supports the first rules of arithmetic and just prints the result to *stdout* does not need to be separated, as there are no dedicated subtasks.

The following sections describe what calls are necessary to implement a privilege separated process.

Splitting Process

To split a single process into a parent and a child process, the *fork(2)* function call is used. After calling this function it returns the new child process id to the parent and *0* to the child process. These return values make it possible to run different code on both the parent and the child side in order to start the main loop or open files.

```
switch ((child = fork())) {
  case -1:
    /* error handling */
  case 0:
    /* child */
  default:
    /* parent */
}
```

Listing 4.16: *fork(2)* function call

Listing 4.16 shows a *fork(2)* function call. The different *case* labels allow the handling of each individual process including the error. After the *switch*-section, the code for both processes is the same. For this reason the statements executed by the parent process, after the *default*-label, often just return the child process id to store it for further communication.

Changing the root directory

The *chroot(2)* function call changes the root directory. The default root directory is / for all processes. If this directory is changed, e.g. to */var/log*, it is not possible for the process to navigate to a higher directory, e.g. */var*, as it is locked down to the new root directory.

```
if (chroot("/var/log") == -1) {
  /* error handling */
}
if (chdir("/") == -1) {
  /* error handling */
}
```

Listing 4.17: Changing the root directory

As Listing 4.17 illustrates, the process has a new root directory called */var/log*. This function returns *0* on success and *-1* on failure. After the process has been locked down into the new root directory, the *chdir(2)* function is called as described in Listing 4.17. This function sets the current directory of a process, in this case the new root directory. On success it returns *0* and *-1* on failure.

Dropping privileges

After a new root directory has been chosen, it is necessary to drop the privileges from the *root* user down to an unprivileged user. It is not possible to do this first, as the *chroot(2)*

call needs such privileges. If anyone could call this function, it would be possible to break out of the *chroot* environment. For this reason, we always change the root directory before dropping privileges.

```
if ((pw = getpwnam(UNPRIV_USER)) == NULL) {
  /* error handling */
}
if (setgroups(1, &pw->pw_gid) ||
    setresgid(pw->pw_gid, pw->pw_gid, pw->pw_gid) ||
    setresuid(pw->pw_uid, pw->pw_uid, pw->pw_uid)) {
  /* error handling */
}
```

Listing 4.18: Dropping privileges

Listing 4.18 shows how privileges can be dropped by using an unprivileged user. The *getpwnam* function call returns a pointer of the *struct passwd* type which contains all information stored in the password database files - *passwd(5)*. If the pointer is *NULL*, the given user was not added previously. *setgroups(2)* sets the group access list (supplementary group) of the current process to the new group id. Each user on GNU/Linux or BSD systems is at least a member of one group named *primary group*. It is also possible to add a user to other groups named *supplementary group*. This call resets all *supplementary group* ids to the *primary group* id. The *setresgid(2)* call sets the real, effective and saved group ids of the current process and *setresuid(2)* does the same but, for the user ids. In both cases it is the unprivileged user that owns this process now. Typically, all three groups and user ids (real, effective, and saved) are the same. Those ids will be used in different cases, such as file creation, doing some unprivileged work as a privileged user, or affecting sending signals to processes. However, they are more often used in commands, such as *su(1)*, or environments that will both not affect the keystore implementation.

After these calls have been finished, the current process drops its privileges from the *root* user and runs under the unprivileged *UNPRIV_ USER* user.

Changing the child process name

If a child is created, it inherits the name of its parent as well. This can be complicated when one tries to find out the client's purpose according to the name. If both processes have the same name, e.g. *keystore*, which one is the service for client communication and which one takes care of storing data? For this reason, it is necessary to change the child's name to show its purpose. This can either be done by using the *prctl(2)* library function

since GNU/Linux 2.6.9 or by using OpenBSDs *setproctitle(3)* as seen in Listing 4.19. While *setproctitle(3)* always succeeds *prctl(2)* returns a *0* on success and *-1* on failure.

```
/* GNU/Linux since 2.6.9 */
if (prctl(PR_SET_NAME, "[keystore] storage") == -1) {
  /* error handling */
}
/* OpenBSD */
setproctitle("[keystore] storage");
```

Listing 4.19: Changing the process name

As OpenBSD has a tradition in developing services and utilities that use privilege separation, it provides all necessary functions, even for portable versions. The mail server contains a directory named *openbsd-compat* that contains functions only implemented in OpenBSD. This makes it possible to use the same code on different platforms, but only include additional header files. Unfortunately, this only works if the underlying functions or process structures support it. On the Android platform the process name cannot be changed by *setproctitle(3)* as its way of splitting a name is not supported.

Communication between parent and child

The communication between the parent and its child or children is very important, as there is no possibility to transform data from one process to another by calling functions. Communication between multiple processes can be established in many ways.

The easiest way is to use files and a defined format for communication. The data will be converted into this format, e.g. CSV, XML, or JSON, before writing it and will be read and converted back by another process. This method has a few advantages, one being that additional exchange formats must be supported by each process and the data must be converted into the format and back for further processing. A better approach is to use native data structures offered by the programming language, like *struct*s. APIs to transmit these data are the *send(2)* and *recv(2)* system call. These APIs provide functions for sending and receiving data and are part of the systems socket library.

Listing 4.20 shows how the functions to communicate with processes are used. The first parameter is the file descriptor - *fd*. Each process has its own descriptor it listens on for incoming data. The *msg* structure as defined in *recv(2)* holds the data for sending and receiving. It contains an array of data with its length. The data can be of any type from numbers or strings to structs.

```
if (sendmsg(fd, &msg, flags) == -1) {
  /* error handling */
}
if (recvmsg(fd, &msg, flags) == -1) {
  /* error handling */
}
```

Listing 4.20: *send(2)* and *recv(2)* function calls

The receiver just needs to know how many data are sent in order to allocate enough memory. The man pages describe possible flags as well, e.g. in order to avoid blocking while communicating.

The creation and managing of file descriptors for such a communication is a bit complicated. This must be done before *fork(2)* is executed, as later the child is forked from the parent and it is not possible to link them together again. The last parameter *sp* is an array of two integer values that contains the descriptors for sending and receiving - as can be seen in Listing 4.21. These descriptors represent the start and the end of a single pipe.

```
if (socketpair(AF_UNIX, SOCK_STREAM, PF_UNSPEC, sp) == -1) {
  /* error handling */
}
```

Listing 4.21: *socketpair(2)* function call

Listing 4.16 shows how the parent and child parts are split and can be managed individually. After *fork(2)* has been called, both processes have the same descriptors. Since the processes communicate using a pipe, only two ends of the pipe must remain. For this reason, the redundant descriptors must be closed in each process. The child closes the descriptor located at index *0* using the *close(2)* function and the parent the one at index *1*. After calling this function only the single ends of the pipe remain.

4.4.3 Implementation

This section describes the portable version of Android's keystore at the implementation level. This includes both the way the client communicates with the server as well as the internal server communication and the differences to the original implementation and why these changes have been made. As C++ is only used to communicate with the Android *Binder*-Interface, all files containing C++ were removed and rewritten in C as discussed in the following sections. This includes even service related classes, such as *KeystoreProxy* and

KeyStore. The *UserState*-class was also removed entirely as it only stores the master key as well as different states of each user for a better key management. However, this approach complicates the service implementation and was changed into an implementation that reads the master key on every request the client sends to the server. The *UserState* stores the master key internally to allow the user accessing data without entering a password. The problem with it is that it does not prevent unauthorized users from reading values of keys, for example. If two different users have access to the Android system user *system*, key-value pairs leak from on user to another. This was changed by storing data per user who can login into the system as we can assume that the user protects his login data against unauthorized users.

In conclusion, the entire portable version is written in the C programming language. This avoids issues with different kinds of compiler flags as well and keeps the source code more uniform as there is no mixture of functionality between C and C++.

The following sections describe the client-server communication, the used signal handler to stop the server reasonably, and the main differences between the Android and the portable version of the keystore service.

Communication between processes

As already mentioned, the client communicates with the server using a local Unix TCP socket. For such a communication the *send(2)* and *recv(2)* system calls were used as these calls need a connected state. The TCP protocol establishes such a connection. These functions need a socket descriptor to send the data to or receive them from, the data itself, the length of such data and additional flags if needed as Listing 4.22 shows.

```
if (send(fd, req, sizeof(*req), MSG_NOSIGNAL) < 0) {
  fprintf(stderr, "could not send data");
  goto close;
}
if (recv(fd, res, sizeof(*res), 0) < 0) {
  fprintf(stderr, "could not receive data");
  goto close;
}
```

Listing 4.22: *send(2)* and *recv(2)* function calls

The data that encapsulate those for both sending and for receiving are two structures: *ks_req* (req) and *ks_res* (res). The first structure represents the client request and contains the user id the client runs under, the master password and the key-value pair if needed.

The second one represents the request and contains the error value, so the client can handle the response accordingly and the value stored previously if requested. The *fd* value is the socket a connection was made to. Both structures as well as the used enumeration to indicate the type of request or response (init, insert etc.) can be found under Appendix F. The flag *MSG_NOSIGNAL* that is added to *send(2)* avoids sending the *SIGPIPE* signal. The sending of such a signal requires a signal handler to handle it. Instead, the return value is used to indicate an error. The *recv(2)* function call requires no extra flag.

For internal communication between the father and the child process, the already introduced *socketpair* system call is used. This means that there is no connected state and different calls are needed. Such calls are *sendmsg(2)* and *recvmsg(2)*, as can be seen in Listing 4.23 and Listing 4.24. These calls require a special structure named *msghdr* for communication that is part of the systems socket library, too. This structure contains the data that are sent or received.

```
if (sendmsg(fd, &msg, 0) == -1) {
  log_err("sendmsg failed: %s", strerror(errno));
  return -1;
}
```

Listing 4.23: *sendmsg(2)* function call

Listing 4.23 shows how the *sendmsg(2)* call is used. The first argument represents the file descriptor the *msg* will be sent to or received from and the last argument contains additional flags that are not needed in this case.

```
if (recvmsg(fd, &msg, 0) == -1 ) {
  log_err("recvmsg failed: %s", strerror(errno));
  return -1;
}
```

Listing 4.24: *recvmsg(2)* function call

Similar to Listing 4.23, Listing 4.24 shows how a *msg* sent to a file descriptor will be received. There are no additional flags needed as well.

Signal handler

Daemons in C must implement a signal handler, which is called if one of the signals the daemon is listening for is received. A signal handler can be registered using the *signal(3)*

library function as shown in Listing 4.25. Without registering such signals, allocated resources like file descriptors or opened sockets could not be closed, as the program would terminate instantly. Another reason why signal handlers are important is that each parent should take care of its children. If the parent terminates, each child should terminate as well. A child process that has no parent process any more is called orphan and should be avoided. It is bad style as such processes need to be terminated manually afterwards and could block resources necessary for the next daemon start.

```
signal(SIGTERM, parent_sig_handler);
signal(SIGINT, parent_sig_handler);
signal(SIGCHLD, parent_sig_handler);
```

Listing 4.25: *signal(3)* function call

The first argument is the signal that needs to be caught and the second one is the function that is called as soon as the signal was received. The signals used in the above signal handler calls will be described in the following enumeration to explain why they are necessary.

- **SIGTERM**
  This signal will be sent to a process to request its termination. Catching this signal allows the process to release allocated resources or terminate its children.
- **SIGINT**
  This signal will be sent to the process by using the Ctrl-C shortcut. The purpose of this signal is similar to *SIGTERM*.
- **SIGCHLD**
  This signal will be sent to the process to notify it that one of its children was terminated. This signal is important for privilege separation, as the children are a significant part of the entire service, and it fails if one child is not available.

Compiler flags

OpenBSD provides templates for compiling and linking C files to a single program. For userland tools the template is called *bsd.prog.mk* and has to be included at the end of the Makefile - *.include <bsd.prog.mk*. This includes some default flags, such as *02* for optimization or *pipe* to use pipes rather than temporary files for communication between the different stages of the compilation process. Some additional flags are used, like *-Wuninitialized* or

-*Wshadow*, which are already explained in Section 4.2.1. The Makefiles for both the client and the server implementation are located in Appendix E.

Folder structure

Listing 4.26 shows the different folders with their contained files created by *tree(1)*. The server (*ksd*) and client (*ks*) only have separated files to build each program (Makefile). All files needed by both programs are in the same directory on top.

```
|-- LICENSE
|-- Makefile
|-- README.md
|-- blob.c
|-- blob.h
|-- defines.h
|-- entropy.c
|-- entropy.h
|-- ks
|   `-- Makefile
|-- ks.1
|-- ks.c
|-- ksd
|   `-- Makefile
|-- ksd-api.c
|-- ksd-api.h
|-- ksd.8
|-- ksd.c
|-- ksd.h
|-- log.c
|-- log.h
|-- msg.c
|-- msg.h
|-- proc.c
|-- proc.h
|-- socket.c
`-- socket.h
```

Listing 4.26: Structure of the portable version

There is a file containing the ISC license and a short description (README.md) about the project as well, which will not further be discussed. The following listing will give an overview of all significantly changed files or created to port the original Android keystore

service to OpenBSD.

- **blob**
  For privilege separation new functions were added to fork the child process and manage the requests as well as send the responses back to the client. The child will be locked down to the */var/ksd* directory and drop its privileges after forking.

- **entropy**
  Wraps the OpenBSD extension *arc4random(3)* instead of using */dev/urandom*. This function will always return a random number and there are still portable versions in OpenSSH and OpenSMTPD.

- **ks.1**
  Contains the man page for the client. *1* means *General Commands*, which this command is.

- **ks**
  Contains the client and implements all calls necessary for communicating with the server. Those commands will be explained in a subsequent section.

- **ksd-api**
  Contains the API implemented by clients. Such functions are named after the call they encapsulate: *api_reset* to reset the keystore or *api_insert* to insert a new key-value pair. Each function needs a callback function implemented in the client, which will be called after sending data and receiving data from the server. Functions to send and receive will be called internally to avoid problems in client implementations.

- **ksd.8**
  Contains the man page for the server. *8* means *System Manager's Manual*, which this server is.

- **ksd**
  Contains the server implementation. The already introduced signal handler will be registered as well as forking the client process and starting the main loop to receive client requests.

- **msg**
  Contains the implementation for internal communication between parent and child process.

- **proc**
  Contains functions to change the name of processes.

- **socket**
  Implements the client-server communication by creating a Unix socket located under */var/run/ksd.sock* to receive client requests and send responses back.

Differences to the new Android keystore service implementation

The API offered by the server was simplified and focused on calls covering the necessary use cases. Each call is secured by a passphrase request called in the API to avoid client implementations without such a request. All supported flags are listed in Appendix C. The following enumeration explains the calls implemented by the command-line-interface client of the portable version.

- **add (A)**
  This call adds a new key-value pair for a given key.

- **delete (D)**
  This call deletes a key-value pair for a given key. This is the inverse function to *add*.

- **get (G)**
  This call requests the stored value for a given key.

- **init (I)**
  This call initializes the keystore of the user who calls the client (*ks(1)*) with the entered master password. The master password must be entered twice for confirmation.

- **reset (R)**
  This call deletes the entire keystore of the user who calls the client (*ks(1)*). This is the inverse function to *init*.

Using these calls every additional key or keystore manipulation operation can be implemented. The update of an existing value only comprises the calling of the already implemented *delete* and *add* operations. Changing the master password just needs to remember all stored key-value pairs, reset the keystore and initialize it again before adding back all remembered key-value pairs. As the client uses the command line interface, a shell script can implement such functionality by combining the base operations. There is no need for a dedicated *update* or *change master password* operation. For the same reason, calls to

generate keys for encryption or signing, such as RSA or DSA, are not supported. There are commands already available, like *ssh-keygen(1)* or *openssl(1)* to create SSH keys or certificates for web servers. Hence, there is no need to implement such functionality again. On Android the situation is different as APIs already implement the keystore service and rely on it. Besides, on a mobile operating system CLI commands are not often used, so adding key generation to the service directly is reasonable in this case.

There is another difference regarding the length of data that can be stored. While the Android implementation is able to store 32KB, the portable version can only store 2KB. This has two reasons. The first reason is the avoidance of storing certificates and private keys and the focus on storing user names and passwords. Why is that so?

The first reason is that handling such data differs between desktop or server operating systems and mobile operating systems. If we have a look at OpenSSH, which is a widely used implementation the of Secure Shell (SSH) protocol, all used certificates and private keys will be protected by the implementation itself. If the private key has too powerful rights or does not belong to the user it used to be, this implementation will stop establishing a connection to the server and print a message to *stderr*. The only user who is able to see the private key is the owner as well as the *root* user. If one of these users is compromised, the system needs to be reinstalled or better changed entirely. The keystore does not protect the user's credentials in that case, therefore, the attacker could install a key logger that sends him all of the user's keyboard input. It would indeed protect such data if the computer was stolen or confiscated by the authority and the file system was not encrypted. However, this should always be the case.

The second reason is how those functions for sending and receiving messages work internally. Each socket has a maximum of bytes for sending and receiving. On OpenBSD 5.8 its 4KB, but can be changed by using the *setsockopt(2)* system call with the *SO_SNDBUF* flag, for the sending buffer, and the *SO_RCVBUF* flag, for the receiving buffer. Moreover, those functions block while calling them. Here, there are two problems to deal with. The first is the size limit. This could, of course be increased up to 32KB, but what if a private key or certificate is more than 32KB in size? In this case it would be better to avoid a static buffer and use a dynamic buffer instead to be more flexible. A socket handling of dynamic data, however, cannot be changed by *setsockopt(2)*, as such values should always be static to avoid errors - e.g. negative numbers or numbers too big to set, to implement the storage of data that have dynamic size. The calls for transmitting such data must be non-blocking. This can be achieved by setting additional flags to the system calls itself, such as *MSG_DONTWAIT* and implement the *poll(2)* system call to monitor the file descriptor

to be notified whether there is data to read or write. As this master thesis is only a proof-of-concept to check whether it is possible to make an Android native service or parts of it portable, this feature is not implemented and will be discussed later in the future work section.

4.5 Result

This chapter contains the discussion of the new implementation of Android's keystore service as well as the portable version. The result of this implementation will be summarized in this section.

The Android version is implemented based on the results of analysing the original source code as already described in Chapter 3. The source code was split into different files to separate the logic written in C from that written in C++. This allows better handling of compiler flags and supports avoiding source code issues, such as bugs and coding style problems. Furthermore, the logging mechanism was encapsulated to provide a generic API that wraps the provided *syslog* API. The generic API is used for the entire application, so a portable version just needs to change the wrapped calls and there is no change of logging calls inside the application necessary. Library calls, such as *malloc(3)* or *atoi(3)*, check the return value to make sure that the call was successful or was changed into a function that prevents integer overflows. At last, the implemented hash function (MD5) was changed to SHA512, which is state-of-the-art, as there is no vulnerability known in practice. The API for encryption (AES) was simplified to use one that is described in a man page and gives an overview of its behaviour as well as the expected return values. The hash algorithm uses this API as well as OpenSSL implements generic functions for encryption and hash algorithms. Chapter 1 describes the consideration to additionally sign the data.

Let us have a short look at what signing of data will ensure. Data should be signed to make sure the receiver knows the data were sent by the original sender and not manipulated by someone else instead. In case of the keystore service the sender is the file system and the receiver the service itself. The files, however, are written into a directory only accessible to the keystore user, except *root*. As this user is the most powerful system user (administrator), it is not reasonable to make a service "*root*-save". The user expects that *root* has access to every tool or file. First, it will confuse the user and second, it is more important to avoid unprivileged system users to get access to those files, as most system services use these. This increases the potential to compromise them as well as their corresponding unprivileged user. The Android version protects the manipulation of data by *root* by using the *keystore_ cli(1)*

command through SELinux policies, but it is always possible to manipulate data by using the normal userland tool-chain, like *rm(1)*. So there is no one else that can change the data. This makes the signing of data useless, except that they will be sent over an insecure medium, like the Internet or a memory card, which is not the case.

As already described in Chapter 3, the internal design as well as the provided client API remain the same to be backward compatible with existing implementations. This allows changing of the original implementation to the new one without any problems. Only previously stored data should be added again as this part has changed entirely.

The portable version implements privilege separation that locks the unprivileged process down into a dedicated directory to prevent exploits from compromising the entire system. The way the data will be stored remains the same as in the Android version. Only the communication between internal processes and clients was reimplemented. Furthermore, the count of functions the client is able to call was reduced to basic functionality, to keep the system as small as possible. Additional functionality can be implemented by other clients or by combining existing ones using *Shell* scripts. At last, the size of data that can be stored was changed to 2 KByte instead of 32 KByte because the large value only makes sense if certificates or keys will be stored, but services like OpenSSH do already care about such data. Consequently, the use case is slightly different regarding mobile operating systems compared to desktop or server operating systems.

Chapter 5

Evaluation

This chapter contains the evaluation of the work introduced and implemented during this thesis. It will discuss the advantages and disadvantages of both the new Android keystore service and its portable implementation. Furthermore, it describes how both implementations can be used and how the Android version interacts with applications, which have already implemented its API. Implemented regression tests are also presented, which cover different cases to make sure that the service as well as the client work as expected. Finally, the results of both evaluations are summarized to show what they reveal and what both implementations can achieve.

5.1 Android

As already described, the provided API calls to communicate with the CLI client as well as the Java API compared to the original implementation remain the same for compatibility reasons. In this section the new version of Android's keystore service will be evaluated regarding its ability to write and read data using the CLI client, as seen in Listing 5.1. This evaluation does not show all calls the client is able to carry out, as they work analogously to the rest. It is only meant to demonstrate that the new implementation of Android's keystore service does what it is supposed to do.

As Listing 5.1 shows, the client executes a subset of the implemented keystore API calls with their needed parameters. This subset was chosen to show the main functionality of the service. The *reset* call resets the entire user keystore, while *password* creates a new master passphrase if none exists. The *insert* call inserts a new key-value pair into the user's keystore and *get* prints the value previously stored for a given key. This command does not need a password, as the master key is stored for every user in the service internally.

```
$ stop keystore && start keystore # restart keystore service
$ keystore_cli reset
No error
$ keystore_cli password foo
No error
$ keystore_cli insert key value
No error
$ keystore_cli get key
value$
```

Listing 5.1: Evaluation of the new Android version

Once the service has been restarted, the master key must be reloaded. The *password* command will deal with it, as it creates a new master key if none has been created before and reloads it if it already exists. All implemented calls can be found in Appendix B.

Due to the fact that terminal applications on Android are not used as often as on server or desktop operating systems, additional output will be displayed directly - *No error*. A common behaviour style for command-line tools on BSD or GNU/Linux is the following: they only print an error, but if everything went successful nothing is printed, for instance, the *rm(1)* command. If a file or directory could not be deleted, an error is printed. There is no reason to print a text that says "file or directory deleted successfully", as the user assumes the command will run successfully with the chosen parameters anyway. This helps other programs cascaded by operators, such as the pipe (|), which uses the output (stdout) of one program as input (stdin) of the next program. If the *password* call was successful, inserting a new key-value pair can be started. To show whether a program runs successfully or not, only a return value is needed to reveal its state.

As already explained, only the internal functionality was changed to be more secure and maintainable. This includes the source code as well as the data that will be stored. The defined API was not changed in order to avoid issues with implementations that already use the service, such as different Java classes delivered with the Android software stack.

5.1.1 Testing

This section describes the *test-keystore* script that implements statements to test the Android keystore service using the Android debugger bridge (*adb(1)*) command and is based on the original test script, but simplified. The *adb(1)* command allows communication between the Android device and the system it is connected to, e.g. to execute commands on the device itself. The output generated on the device is displayed on the system. This

script compares two different files: *test-keystore.baseline* and *test-keystore.log*, as Listing 5.2 shows. The -> sign shows the test that was executed. The *Shell* control operator *&&* (double ampersand) will be interpreted as logical *AND*. When this operator is used, the second command will only be executed if the first one succeeds - returns *0*. *PASSED* is displayed only if all tests were successful, otherwise *FAILED* will be displayed and the *diff(1)* command shows which output was printed instead of the expected one.

```
$ ./test-keystore
START
-> reset keystore and reinit as system user
-> root does not have permission to run test
-> test add, get and delete key-value pair
-> lock the store as system
comparing ./test-keystore.baseline and ./test-keystore.log
PASSED
```

Listing 5.2: Regression tests new Android version

As can be seen in the above output, four different cases are tested. These cases are described by comments following after the -> sign. The reason why not every call will be tested is that the main focus re-implementing this service was the internal design and the storage logic, which is covered by the provided tests. Of course, these regression tests can be improved, but this is future work. The *test-keystore.baseline* file contains the expected output set by the test-script and the *test-keystore.log* contains the output produced by the *adb(1)* command that calls the *keystore_ cli* command. For comparison the *diff(1)* command is used to show the differences between two files. If both files share the same output, the *keystore_ cli* command works as expected.

5.2 Portable version

In this section the portable version will be evaluated regarding its usage and possibilities to implement further client software.

As already mentioned in Chapter 4, this version only implements a basic feature set that allows it to be combined to a more complex feature, such as updating an already added key-value pair or updating the master password. To achieve this, a combination of different return values is needed for *Shell* scripts to decide whether this call was successful or not. In general, the *0* value means success and a value *!= 0* means failure. The *Shell* control operator *;* (Semicolon) executes all commands separated by this sign - no matter what

the return value of the former command was. Clients written in other languages, such as *Python*[1] or *Lua*[2], can be implemented as well, using the provided API. They have libraries to call native C functions and process the result in the concrete programming language. However, this will not be discussed in this thesis.

Listing 5.3 shows how the keystore server can be started and how the client interacts with its API to call the appropriate functions. The *&* (single ampersand) is a control operator that runs the given command in the background and the # (hashtag) initiates a comment.

```
$ sudo /usr/local/sbin/ksd &  # start server
$ ps -aux | grep ksd          # show ksd related proceses
root      21628  ...  0:00.03 ksd: [priv] (ksd)
_ksd       3044  ...  0:00.00 ksd: blob (ksd)

$ ks -R                       # reset a keystore
passphrase:
$ ks -I                       # init a keystore
passphrase:
confirm passphrase:
$ ks -A -k key -v value       # add new key-value pair
passphrase:
$ ks -G -k key                # get value of key
passphrase:
value$
$ ks -D -k key                # delete key-value pair of key
passphrase:
```

Listing 5.3: Evaluation of the portable version

As can be seen in the above listing the client needs a minimum of arguments to call each API function. The call for a passphrase is added into every call instead of adding the passphrase as an additional argument that can be seen by an attacker. It will never be shown and can be piped in or written to *stdin* by redirecting *stdout* of another program, either by using the pipe (|) or the stream redirection symbol >.

Since this version is portable to other BSD or GNU/Linux operating systems, SELinux was removed as it is not portable to other operating systems than GNU/Linux. Furthermore, it is not enabled by default on all GNU/Linux systems. On the one hand, this could cause attack vectors as every user is able to use the client completely, so bugs will cause

[1] https://docs.python.org/3/c-api/
[2] http://www.lua.org/pil/24.html

greater damage on the system. On the other hand, additional source code is not needed to check if GNU/Linux is either implemented or supported.

OpenBSD does support a system call named *pledge(2)* and appeared in OpenBSD 5.8. It specifies a group of system call categories, such as *malloc* to allow calling the *malloc(3)* family of functions or *rw* to call most types of *I/O* operations. If a system call is called that is not part of the specified categories, the kernel will not call it and the program terminates. As *pledge(2)* forces an application into a restricted-service operating mode it is always possible to decrease the permissions, but not to increase them.

Features implemented by a kernel that are used in applications are always difficult to port if different operating systems should be supported. While compiling the application, a verification whether a feature is implemented and enabled or not has to be executed, to mask or unmask the logic that calls this feature. This can be implemented by C's pre-processor directive *#define* to make sure a desired feature is implemented. As not all BSD versions support *selinux(8)* or *pledge(2)*, such information can be given to the pre-processor in order to mask the source code responsible for calling it. However, this makes the code more complicated. In general, the program should not just rely on kernel security features. Design techniques, like privilege separation, should be considered as they significantly increase the security of a program and library functions should always be checked for expected return values. Such security features should only be used in addition, as the program itself has to be programmed in a way that tries to avoid attacks against it.

5.2.1 Testing

This section describes the regression tests implemented for the portable version of Android's keystore service. Those tests call every command implemented by *ks(1)* and will write the output to *stdout*, as seen in Listing 5.4.

```
$ regress/ks.sh
could not connect to server
ks: unknown error occured

$ regress/ks.sh
test1234$
```

Listing 5.4: Regression tests portable version

All tests are executed by calling the script *regress/ks.sh*. The first example in the above listing shows that the keystore service (*ksd(8)*) must be started in order to execute all

regression tests. The second call reveals what output the script produces to demonstrate all tests ran successfully. The only output is the chosen value *test1234* printed by *ks -G -k <key>*. The entire test script is part of the *portable-keystore* repository that contains the portable version of the new Android keystore service implementation.

It is not necessary to provide any further information. If an error occurs during the execution of *ks(1)*, it will be written to *stderr*, as implemented by the program itself. Even in an automated testing environment this produces enough output to help finding errors. The return value of the script will be *0* if everything runs successfully or *1* if something goes wrong, since those are the values returned by the *ks(1)* program. The system that calls the script must only automatically check its return value and whether it is *1* and send a notification to the developer afterwards. Furthermore, implementing tests this way makes it easy to reproduce them for a more detailed error diagnosis. The developers can call the tests themselves and do not depend on a complex test setting.

5.3 Result

This chapter shows how the new Android keystore implementation as well as the portable version are tested. The regression tests are based on Shell-scripts as shown in Section 5.1 and Section 5.2. These tests can easily be extended in order to include additional test scenarios in the future since only the basic functionality (init, reset, add, get, delete) has been covered by now. The tests show that both implementations perform their tasks and work as expected. Furthermore, they confirm that no bugs regarding the basic functionality of both services and clients were introduced while implementing the new design. As the tests, which are based on the client, show that the communication between server and client works, there are no additional tests needed in this area, e.g. a test if the server process has been started successfully before executing further tests. If the services do not start the clients will terminate with an error.

There is one area in this context, test cases based on the client do not cover the functionality adequately and this is the encrypted storage of data. If there is an error in this case, the client could still show previously stored data as it cannot validate whether the data is stored encrypted or not. In this particular part the regression test must look into the file system to verify that an encrypted file was added with the specified parameters, like the user id and password, but this is future work.

In conclusion, the implementations introduced in this thesis are a solid basis for further development of both keystore services.

Chapter 6

Conclusion and Future Work

This chapter summarizes the thesis and gives an overview of the work that was done and what the benefits according to the original Android keystore service implementation are. Furthermore, there is an outlook about further development of both versions, which contains ideas and improvements not touched during this work.

6.1 Conclusion

This master thesis analyses the Android keystore service with regard to security and portability and offers an approach to increase both, according to the first research question: *Can the Android keystore service be improved in regard to security and portability?* To address the portability aspect, a portable version for OpenBSD was implemented based on the new Android implementation introduced in this work, which is the result of analysing the original one.

With respect to security, the first goal was to analyse the used encryption and hash algorithms according to their state-of-the-art approach. This includes the analyses of the authentication, authorization, and verification mechanisms of data in the original implementation as well. If highly sensitive data, such as passwords, have to be stored, it must be guaranteed that no trespasser has access to them. Data corruption is another important part in this area, as the user needs to be informed if the data he received from the service were not the same as those he stored originally. Both areas, the secure storage of data and the recognition of data corruption, are crucial parts of this service. As the user trusts the application to manage the data fairly, but if the service does not, it cannot be trusted any more, what makes it useless. For this reason the already implemented hashing and encryption algorithms were evaluated with respect to being state-of-the-art today.

Unfortunately, the hash algorithm implemented in the original keystore version is MD5, which is considered as compromised for over a decade, but is still in use. The problem with MD5 is that it is no longer collision resistant, which downgrades the recognition of manipulations. For this reason, MD5 was replaced by the SHA512 algorithm in the new implementation. This algorithm is more secure as there are no attack vectors known that lead to a collision attack. The encryption algorithm is still AES as it is state-of-the-art today, only its key size was changed from 128 bit up to 256 bit, as it is the highest possible value and increases the security significantly. A larger encryption key increases the security level as it increases the number of possible keys - see Section 3.6 for further explanations. Furthermore, the AES algorithm is fast regarding encryption and decryption and this is important on mobile platforms that are limited with respect to memory and CPU usage.

The second goal was to analyse the source code as well as the entire design of the original keystore. This includes the implementation of Android-specific APIs for services, the usage of APIs provided by C/C++ libraries, like the standard C library or OpenSSL, and the usage of *gcc(1)*'s compiler flags. Android provides an API for implementing services and their corresponding clients named the *Binder* interface, which is based on RPC. This interface defines C++ interfaces that need to be implemented for both the server and the client to establish a RPC-based communication between both sides. The original keystore implementation implements such interfaces, so there was no further source code change at this point as this is the recommended way to implement services.

The original implementation used an undocumented macro named *AES_ cbc_ encrypt*, for both encryption and decryption. This was changed to the *EVP_ ** functions provided by the OpenSSL library. Such functions give more control of each single step while encrypting plain text or decrypting cipher text. Their behaviour, such as return values or global error values that will be set is a failure occurs, are described in man pages, which supports the development significantly. The same API was implemented for hashing as it behaves the same way. Furthermore, the insecure function call *atoi(3)* was changed to *strtol(3)*, as *atoi(3)* does not take care of integer overflows and has a different behaviour regarding return values on several platforms. For a portable version, this had to be unified as the different behaviour could lead to additional errors, which are necessary to avoid.

As the *gcc(1)* compiler supports different flags to support the developer avoiding programming errors, it was evaluated which additional flags can be used to achieve more robust source code. The original implementation is written in C/C++ and the logic of both languages was mixed together into single files what decreases the maintainability. Such logic was separated for two different reasons. First, for maintainability reasons, as C with its

procedural paradigm is not suitable with C++, which is an object-oriented one. Second, to obtain more support by the compiler flags *gcc(1)* offers since they can work better if different flags will be used per language and not per file. Some flags are C- or C++-only and cannot be used for a heterogeneous file.

With respect to portability, the first goal was to evaluate, which parts of the new keystore service are possible to port to another platform and which are not. Porting the entire service is too difficult as Android-specific libraries as the *Binder* interface or *selinux(8)* policies need to be ported as well. In the first step, the service was analysed to find out, which parts are considered for a portable version. In the second step, these parts needed a further investigation related to their implemented APIs, e.g. the logging API. Logging is essential for services as it reveals problems and issues to the user. The entire services used the Android-specific logging API which had to be unified by providing a generic logging interface that wraps the platform specific one. New implemented APIs as described above were selected to the effect that they are available on other platforms. At last, only the source code that contains the functionality for key-value pair storing was ported as the rest was too Android- or GNU/Linux-specific, such as *selinux(8)*. The approach introduced in this thesis is the providing of a portable version of the new Android keystore service that is as platform-independent as possible.

Due to the fact that the *Binder* interface could not be ported, the service calls for client communication had to be implemented again and were simplified at once in the portable version, as some calls are redundant. The new Android version does not change such calls as it would provoke a break with APIs and applications already implementing and using this service. Such simplification was possible since neither an application nor a library relies on the portable version as it is an implementation from scratch. To document the server as well as the client of the portable version, man pages were written: *ksd(8)* (server) and *ks(1)* (client). For instance, these man pages document the server's API calls implemented by the client for communication.

The client was implemented by using the server's client API located under *ksd-api.h*. This API defines all functions to communicate with the server and as it was written in C, it is possible to connect it with other programming languages that support an interface for calling C functions. Thus, the service can be used in other projects by just implementing its API and the entire service does not need be changed.

As already mentioned the portable version is based on OpenBSD and shows what needs to be considered while porting an Android application to other platforms in general. The second research question was: *Is it possible to create a portable one-to-one version based*

on the native Android keystore service or are source code adaptations needed? As the functionality that contains the key-value pair storage is the only part that can be ported, the rest had to be implemented again. The main difference between these two versions is the privilege separation implemented in the portable version. This technique decreases the potential of compromising an entire system, as each process is locked down to a directory below the root directory by using the *chroot(2)* library function and runs under its own unprivileged user id. This means it is not possible for a process to get access to files or directories above the root directory set for the process. If the process gets compromised, it is only possible to read or write files the process has access to and which are below the chrooted directory. Files, such as */etc/passwd*, are protected from getting compromised. That makes many exploits less probable. Unfortunately, this technique cannot be adapted to Android as the underlying init system configuration of the keystore service as well as the *Binder* interface do not support it.

In summary, this thesis has investigated the original Android keystore service implementation, discussed it and offer an approach that is more secure and more portable as the original one. Even if there are parts of the service that cannot be ported to other platforms, the essential part was and resulted successfully into a portable version. Such a version can be the origin for ports to other BSD derivatives or GNU/Linux distributions. Furthermore, this thesis reveals what must be considered if an Android application should be ported to one of these platforms so that other developers can profit from the knowledge obtained in this work.

6.2 Future Work

This section gives an overview of the further development for both keystore versions, based on the development described in Chapter 4.

The advantages of the new Android implementation can be easily integrated into the original implementation as it does not change its behaviour or introduces additional functionality. Whether these changes will be integrated some day depend on the original projects developers and can not be decided at that time. However, all changes are available by a public source code repository[1], which supports the integration and further development.

The portable version can be secured by implementing multiple children for a better split of each task. The parent process still runs with *root* privileges and does only have to start and stop all of its children. Based on this, there is at least one child needed to communicate

[1]https://github.com/alokat/platform_system_security

with each client application and chrooted down into */var/run*. Another child can be used to store all master keys independently from all key-value pairs. One child manages the key-value pairs of all users and a second one - chrooted in another directory - manages all master keys. If someone breaks into the process, managing all key-value pairs it is not possible to access the master keys. Furthermore, support for different encryption algorithms, such as Blowfish or Twofish [58], can be implemented to let the user choose between multiple ones and not rely on a single one.

As the portable version of the Android keystore implementation is based on OpenBSD, it is not usable on other platforms automatically, such as GNU/Linux distributions or other BSD derivatives like FreeBSD, NetBSD, or DragonflyBSD since it uses OpenBSD-only library functions in some cases. For such a version, projects like OpenSSH or OpenSMTPD can be considered because they have portable versions already. Making an application as portable as possible creates more complex source code as there are many operating system dependent cases to look at. The portable version of Android's keystore service discussed in this thesis follows the OpenBSD approach: creating two versions of a software, one for the actual operating system and one portable version that includes the operating system dependent functions. This keeps both versions as small as possible since the portable version just includes the operating system dependent functions, but the actual application source code does not need to be changed. The portable implementation is available by a public repository[2] as well, which facilitates the further development.

Even if there are regression tests already implemented for both versions, further testing is needed to make sure no sensitive information leak to unauthorized persons and that the implementations are as robust as possible. The same applies to the clients as their usage is very important for the acceptance of the entire projects.

[2]https://github.com/alokat/keystore-portable

Bibliography

[1] Jeremy Andrus and Jason Nieh. Teaching operating systems using Android. In *Proceedings of the 43rd ACM technical symposium on Computer science*, pages 613–618, 2012.

[2] Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi, and Soo-Mook Moon. Evaluation of Android Dalvik virtual machine. In *The 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 115–124, 2012.

[3] The Android Source Code. http://source.android.com/source/index.html. accessed: 03/25/15.

[4] Android Developers. https://developer.android.com/index.html. accessed: 03/25/15.

[5] Henk C. A. van Tilborg, editor. *Encyclopedia of Cryptography and Security*. Springer, 2005.

[6] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *Advances in Cryptology - 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 19–35, 2005.

[7] Marian Bubak, Dawid Kurzyniec, Piotr Luszczek, and Vaidy S. Sunderam. Creating Java to Native Code Interfaces with Janet. *Scientific Programming*, 9(1):39–50, 2001.

[8] Henk C. A. van Tilborg and Sushil Jajodia. *Encyclopedia of Cryptography and Security, 2nd Ed*. Springer, 2011.

[9] Bill MacCarty. *SELinux - NSA's open source security enhanced linux: beating the o-day vulnerability threat*. O'Reilly, 2005.

[10] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, Dave Andersen, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security policies. In *Proceedings of the 8th USENIX Security Symposium*, 1999.

[11] Faye Coker. Taking advantage of SELinux in Red Hat Enterprise Linux. http://www.redhat.com/magazine/006apr05/features/selinux/, 2015. accessed: 06/19/15.

[12] Ebru Celikel Cankaya. Bell-lapadula confidentiality model. In *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 71–74. 2011.

[13] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[14] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 409–417, 2001.

[15] Theo de Raadt. Exploit mitigation techniques. http://www.openbsd.org/papers/ven05-deraadt/index.html. accessed: 10/25/2015.

[16] Hector Marco-Gisbert and Ismael Ripoll. On the Effectiveness of NX, SSP, RenewSSP, and ASLR against Stack Buffer Overflows. In *2014 IEEE 13th International Symposium on Network Computing and Applications, NCA*, pages 145–152, 2014.

[17] Carey Williamson, Aditya Akella, and Nina Taft. Proceedings of the 2014 Internet Measurement Conference. ACM, 2014.

[18] Joan Daemen and Vincent Rijmen. Rijndael/aes. In *Encyclopedia of Cryptography and Security*. 2005.

[19] John Viega, Matt Messier, and Pravir Chandra. *Network security using OpenSSL - cryptography for secure communications*. O'Reilly, 2002.

[20] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full AES. *IACR Cryptology ePrint Archive*, 2011:449, 2011.

[21] Eli Biham, Ross J. Anderson, and Lars R. Knudsen. Serpent: A New Block Cipher Proposal. In *Fast Software Encryption, 5th International Workshop*, pages 222–238, 1998.

[22] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption, Cambridge Security Workshop*, pages 191–204, 1993.

[23] Zuhua Shao and Yipeng Gao. Certificate-based verifiably encrypted RSA signatures. *Trans. Emerging Telecommunications Technologies*, 26(2):276–289, 2012.

[24] Yvo Desmedt. ElGamal Public Key Encryption. In *Encyclopedia of Cryptography and Security*. 2005.

[25] David Pointcheval. Rabin Cryptosystem. In *Encyclopedia of Cryptography and Security*. 2005.

[26] Bart Preneel. Modes of Operation of a Block Cipher. In *Encyclopedia of Cryptography and Security*. 2005.

[27] Dirk Fox. Secure Hash Algorithm SHA-3. *Datenschutz und Datensicherheit*, 37(2):104, 2013.

[28] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference*, pages 17–36, 2005.

[29] Paulo S. L. M. Barreto and Vincent Rijmen. Whirlpool. In *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 1384–1385. 2011.

[30] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In *Fast Software Encryption, Third International Workshop*, pages 71–82, 1996.

[31] Oguzhan Kara and Adem Atalay. Preimages of hash functions through rainbow tables. In *The 24th International Symposium on Computer and Information Sciences*, pages 304–309, 2009.

[32] Harris E. Michail, Athanasios P. Kakarountas, Athanasios Milidonis, and Costas E. Goutis. Efficient implementation of the keyed-hash message authentication code (HMAC) using the SHA-1 hash function. In *Proceedings of the 2004 11th IEEE International Conference on Electronics*, pages 567–570, 2004.

[33] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. http://tools.ietf.org/html/rfc2104, 1997. accessed: 05/19/15.

[34] Sung-Ming Yen and Chi-Sung Laih. Improved Digital Signature Algorithm. *IEEE Trans. Computers*, 44(5):729–730, 1995.

[35] Don Johnson, Alfred Menezes, and Scott A. Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). *Int. J. Inf. Sec.*, 1(1):36–63, 2001.

[36] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). http://tools.ietf.org/html/rfc4492, 2006. accessed: 10/23/15.

[37] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *IACR Cryptology ePrint Archive*, 2011:368, 2011.

[38] Nikolay Elenkov. Android security internals. page 100, 2015.

[39] Thorsten Schreiber. Android Binder - Android Interprocess Communication. In *Seminarthesis - Network and Data Security*. Ruhr-Universität Bochum, 2011.

[40] Daniel R. Edelson. Smart Pointers: They're Smart, But They're Not Pointers. In *Proceedings of the C++ Conference*, pages 1–20, 1992.

[41] Roland Dreier. Why you shouldn't use __attribute__((packed)). http://digitalvampire.org/blog/index.php/2006/07/31/why-you-shouldnt-use-__attribute__packed/. accessed: 07/10/15.

[42] Kenny Root. Keystore: Add flag for blobs to be unencrypted. https://github.com/alokat/platform_system_security/commit/f9119d6414f43ef669d64e9e53feb043eda49cf3, 2013. accessed: 03/25/15.

[43] Java-based implementation of Android keystore interface. https://android.googlesource.com/platform/frameworks/base/+/master/keystore/java/android/security/KeyStore.java. accessed: 07/21/15.

[44] Android KeyChain class. http://developer.android.com/reference/android/security/KeyChain.html. accessed: 07/21/15.

[45] Android KeyStore class. http://developer.android.com/reference/java/security/KeyStore.html. accessed: 07/21/15.

[46] Warning Options - Using the GNU Compiler Collection (GCC). http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html. accessed: 07/02/15.

[47] R. Gerhards. The Syslog Protocol. https://tools.ietf.org/html/rfc5424, 2009. accessed: 10/05/2015.

[48] Logcat. https://developer.android.com/tools/help/logcat.html. accessed: 10/05/2015.

[49] Android Init Language. https://android.googlesource.com/platform/system/core/+/master/init/readme.txt. accessed: 10/05/2015.

[50] R. Rivest. The MD5 Message-Digest Algorithm. https://tools.ietf.org/html/rfc1321, 1992. accessed: 06/30/15.

[51] Hans Dobbertin. The Status of MD5 After a Recent Attack. *CryptoBytes - The technical newsletter of RSA Laboratories*, 2(2), 1996.

[52] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. *IACR Cryptology ePrint Archive*, 2004:199, 2004.

[53] Kenny Root. Move keystore from frameworks/base. https://github.com/alokat/platform_system_security/commit/a91203b08350b2fc7efda5b1eab39e7541476b3a, 2012. accessed: 03/25/15.

[54] B. Kaliski. PKCS no. 5: Password-Based Cryptography Specification Version 2.0. https://tools.ietf.org/html/rfc2898, 2000. accessed: 10/27/15.

[55] Friedrich Wiemer and Ralf Zimmermann. High-speed implementation of bcrypt password search using special-purpose hardware. In *International Conference on ReConFigurable Computing and FPGAs*, pages 1–6, 2014.

[56] C. Percival and S. Josefsson. The scrypt Password-based key Derivation Function - draft-josefsson-scrypt-kdf-03. https://tools.ietf.org/html/draft-josefsson-scrypt-kdf-03, 2015. accessed: 10/27/2015.

[57] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization*, pages 75–88, 2004.

[58] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Chris Hall. On the Twofish Key Schedule. In *Selected Areas in Cryptography*, pages 27–42, 1998.

Appendices

Appendix A

Static analysing logs

```
$ clang++ -analyze keyblob_utils.cpp
clang: warning: argument unused during compilation: '-analyze'
keyblob_utils.cpp:39:14: error: unknown type name 'uint8_t'
static const uint8_t SOFT_KEY_MAGIC[] = { 'P', 'K', '#', '8' };
             ^
clang: warning: treating 'c-header' input as 'c++-header' when in C++ mode,
    this behavior is deprecated
clang: warning: argument unused during compilation: '-analyze'
keystore.h:59:1: error: unknown type name 'size_t'
size_t get_softkey_header_size();
^


$ clang++ -analyze methods.h
clang: warning: treating 'c-header' input as 'c++-header' when in C++ mode,
    this behavior is deprecated
clang: warning: argument unused during compilation: '-analyze'
methods.h:33:21: error: unknown type name 'DSA'
    void operator()(DSA* p) const {
                    ^


methods.h:51:18: error: expected unqualified-id
typedef UniquePtr<RSA, struct RSA_Delete> Unique_RSA;
                 ^
```

Listing A.1: *clang(1)* output

```
system/security/keystore/keyblob_utils.cpp: In function 'uint8_t*
    add_softkey_header(uint8_t*, size_t)':
system/security/keystore/keyblob_utils.cpp:45:10: error: no previous
    declaration for 'uint8_t* add_softkey_header(uint8_t*, size_t)' [-Werror=
    missing-declarations]
 uint8_t* add_softkey_header(uint8_t* key_blob, size_t key_blob_length) {
          ^
system/security/keystore/IKeystoreService.cpp: In constructor 'android::
    KeystoreArg::KeystoreArg(const void*, size_t)':
system/security/keystore/IKeystoreService.cpp:32:54: error: declaration of '
    data' shadows a member of 'this' [-Werror=shadow]
 KeystoreArg::KeystoreArg(const void* data, size_t len)
                                                      ^
system/security/keystore/IKeystoreService.cpp:87:36: error: use of old-style
    cast [-Werror=old-style-cast]
            size_t ulen = (size_t) len;
                                    ^
system/security/keystore/IKeystoreService.cpp:89:43: error: use of old-style
    cast [-Werror=old-style-cast]
            *item = (uint8_t*) malloc(ulen);
                                           ^
system/security/keystore/IKeystoreService.cpp: In member function 'virtual
    int32_t android::BpKeystoreService::password(const android::String16&)':
system/security/keystore/IKeystoreService.cpp:215:5: error: declaration of '
    password' shadows a member of 'this' [-Werror=shadow]
     {
     ^
system/security/keystore/IKeystoreService.cpp:648:64: error: shadowed
    declaration is here [-Werror=shadow]
     uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
                                                                ^
system/security/keystore/IKeystoreService.cpp:769:21: error: declaration of '
    int32_t flags' shadows a parameter [-Werror=shadow]
            int32_t flags = data.readInt32();
                     ^
```

Listing A.2: Keystore compiler flags output

Appendix B

keystore_cli(1) usage

```
$ keystore_cli
no action given
usage:  keystore_cli test
        keystore_cli get        <key>
        keystore_cli get_pubkey <key>
        keystore_cli insert     <key> <value>
        keystore_cli del        <key> [uid]
        keystore_cli del_pubkey <key> [uid]
        keystore_cli exist      <key> [uid]
        keystore_cli saw        <key> [uid]
        keystore_cli reset
        keystore_cli password   <value>
        keystore_cli lock
        keystore_cli unlock     <key>
        keystore_cli zero
```

Listing B.1: *keystore_cli(1)* usage

Appendix C

ks(1) usage

```
usage:   ks -A -k key -v value
         ks -D -k key
         ks -G -k key
         ks -I
         ks -R
```

Listing C.1: *ks(1)* usage

Appendix D

Android keystore compiler flags

```
LOCAL_CFLAGS := -fstack-protector
LOCAL_CFLAGS += -Wall
LOCAL_CFLAGS += -Wcast-qual
LOCAL_CFLAGS += -Wchar-subscripts
LOCAL_CFLAGS += -Wcomment
LOCAL_CFLAGS += -Werror
LOCAL_CFLAGS += -Wextra
LOCAL_CFLAGS += -Wformat
LOCAL_CFLAGS += -Wformat-security
LOCAL_CFLAGS += -Wmissing-declarations
LOCAL_CFLAGS += -Wparentheses
LOCAL_CFLAGS += -Wreturn-type
LOCAL_CFLAGS += -Wshadow
LOCAL_CFLAGS += -Wsign-compare
LOCAL_CFLAGS += -Wstrict-aliasing
LOCAL_CFLAGS += -Wswitch
LOCAL_CFLAGS += -Wtrigraphs
LOCAL_CFLAGS += -Wuninitialized
LOCAL_CFLAGS += -Wunused
LOCAL_CFLAGS += -Wno-unused-parameter

LOCAL_CPPFLAGS := -Wctor-dtor-privacy
LOCAL_CPPFLAGS += -Wnon-virtual-dtor
LOCAL_CPPFLAGS += -Wreorder
LOCAL_CPPFLAGS += -Wold-style-cast
LOCAL_CPPFLAGS += -Woverloaded-virtual
LOCAL_CPPFLAGS += -Wsign-promo
```

Listing D.1: Android keystore compiler flags

Appendix E

Portable keystore compiler flags

```
.PATH:      ${.CURDIR}/..

PROG=   ksd

SRCS=   socket.c proc.c blob.c msg.c entropy.c log.c ksd.c

MAN=    ksd.8

BINDIR=     /usr/local/sbin

LDADD=     -lcrypto

CFLAGS+=        -Wall -Wstrict-prototypes -Wmissing-prototypes
CFLAGS+=        -Wmissing-declarations
CFLAGS+=        -Wshadow -Wpointer-arith -Wcast-qual
CFLAGS+=        -Wsign-compare

.include <bsd.prog.mk>
```

Listing E.1: Portable keystore server makefile

```
.PATH:      ${.CURDIR}/..

PROG=     ks

SRCS=     ksd-api.c ks.c

MAN=      ks.1

BINDIR=     /usr/local/bin

CFLAGS+=            -Wall -Wstrict-prototypes -Wmissing-prototypes
CFLAGS+=            -Wmissing-declarations
CFLAGS+=            -Wshadow -Wpointer-arith -Wcast-qual
CFLAGS+=            -Wsign-compare

.include <bsd.prog.mk>
```

Listing E.2: Portable keystore client makefile

Appendix F

Structures for sending and receiving

```
enum req_type {
  NONE,
  RESET,
  INIT,
  INSERT,
  DELETE,
  GET
};

enum res_type {
  NO_ERROR,
  NO_MASTER_KEY,
  NO_KEY,
  KEYSTORE_ALREADY_RESETED,
  KEYSTORE_ALREADY_EXISTS,
  MASTER_KEY_ALREADY_EXISTS,
  KEY_ALREADY_ADDED,
  VALUE_CORRUPT,
  WRONG_PASSWORD,
  SYSTEM_ERROR
};

struct ks_msg {
  char passwd[PASSWORD_MAX];
  char key[KEY_MAX];
  char value[VALUE_MAX];
};

struct ks_res {
  enum res_type res;
```

```
  struct ks_msg msg;
};

struct ks_req {
  int req;
  uid_t uid;
  struct ks_msg msg;
};label
```

Listing F.1: Structures and enumerations for sending and receiving

Appendix G

Class diagram

Figure G.1: Keystore transactions

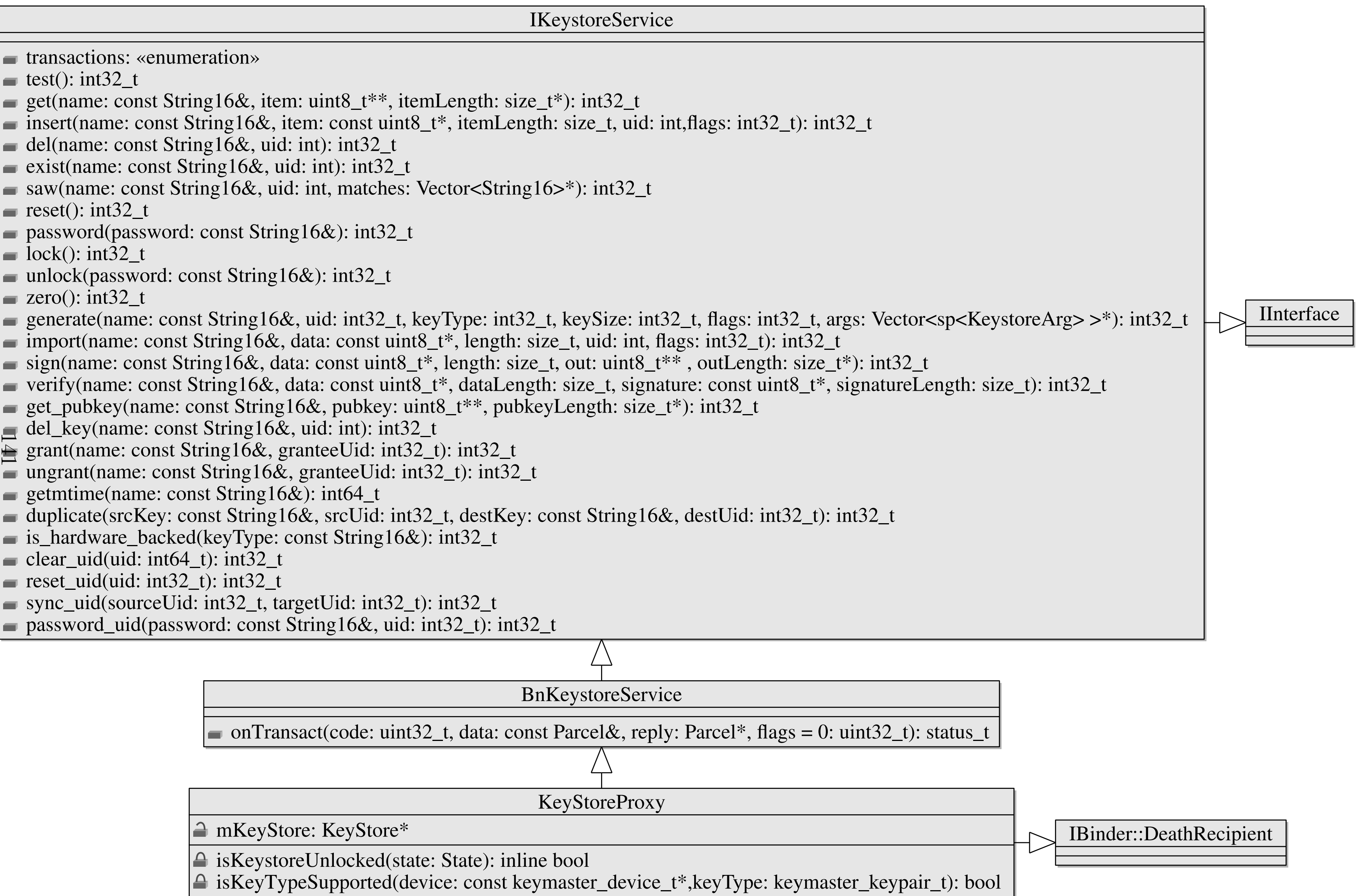


Figure G.2: Class dependency diagram

KeyStore

sOldMasterKey: static const char*
sMetaDataFile: static const char*
sRSAKeyType: static const android::String16
mEntropy: Entropy*
mDevice: keymaster_device_t*
mMasterKeys: android::Vector<UserState*>
mGrants: android::Vector<grant_t*>
mMetaData: keystore_metadata_t

KeyStore(entropy: Entropy*, device: keymaster_device_t*)
~KeyStore()
getDevice() const: keymaster_device_t*
initialize(): ResponseCode
getState(uid_t uid): State
initializeUser(pw: const android::String8&, uid: uid_t): ResponseCode
copyMasterKey(src: uid_t, uid: uid_): ResponseCode
writeMasterKey(pw: const android::String8&, uid: uid_t): ResponseCode
readMasterKey(pw: const android::String8&, uid: uid_t): ResponseCode
getKeyName(keyName: const android::String8&): android::String8
getKeyNameForUid(keyName: const android::String8&, uid: uid_t): android::String8
getKeyNameForUidWithDir(keyName: const android::String8&, uid: uid_t): android::String8
reset(uid: uid_t): bool
isEmpty(uid: uid_t) const: bool
lock(uid: uid_t): void
get(filename: const char*, keyBlob: Blob*, type: const BlobType, uid: uid_t): ResponseCode
put(filename: const char*, keyBlob: Blob*, uid: uid_t): ResponseCode
del(filename: const char *, type: const BlobType, uid: uid_t): ResponseCode
saw(prefix: const android::String8&, matches: android::Vector<android::String16> *, uid: uid_t): ResponseCode
addGrant(filename: const char*, granteeUid: uid_t): void
removeGrant(filename: const char*, granteeUid: uid_t): bool
hasGrant(filename: const char*, uid: const uid_t) const: bool
importKey(key: const uint8_t*, keyLen: size_t, filename: const char*, uid: uid_t, flags: int32_t): ResponseCode
isHardwareBacked(keyType: const android::String16&) const: bool
getKeyForName(keyBlob: Blob*, keyName: const android::String8&, uid: const uid_t, type: const BlobType): ResponseCode
getUserState(uid: uid_t): UserState*
getUserState(uid: uid_t) const: const UserState*
getGrant(filename: const char*, uid: uid_t) const: const grant_t*
upgradeBlob(filename: const char*, blob: Blob*, oldVersion: const uint8_t, type: const BlobType, uid: uid_t): bool
importBlobAsKey(blob: Blob*, filename: const char*, uid: uid_t): ResponseCode
readMetaData(): void
writeMetaData(): void
upgradeKeystore(): bool

Figure G.3: Class keystore

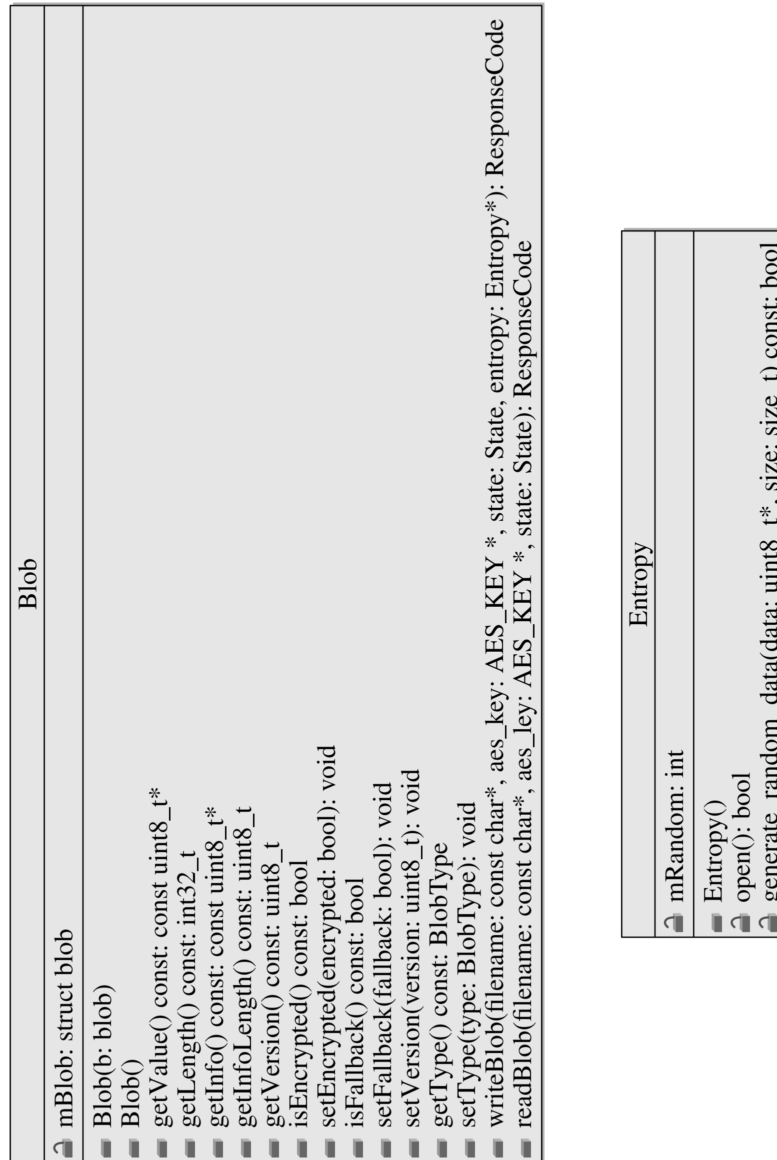


Figure G.4: Classes blob and entropy

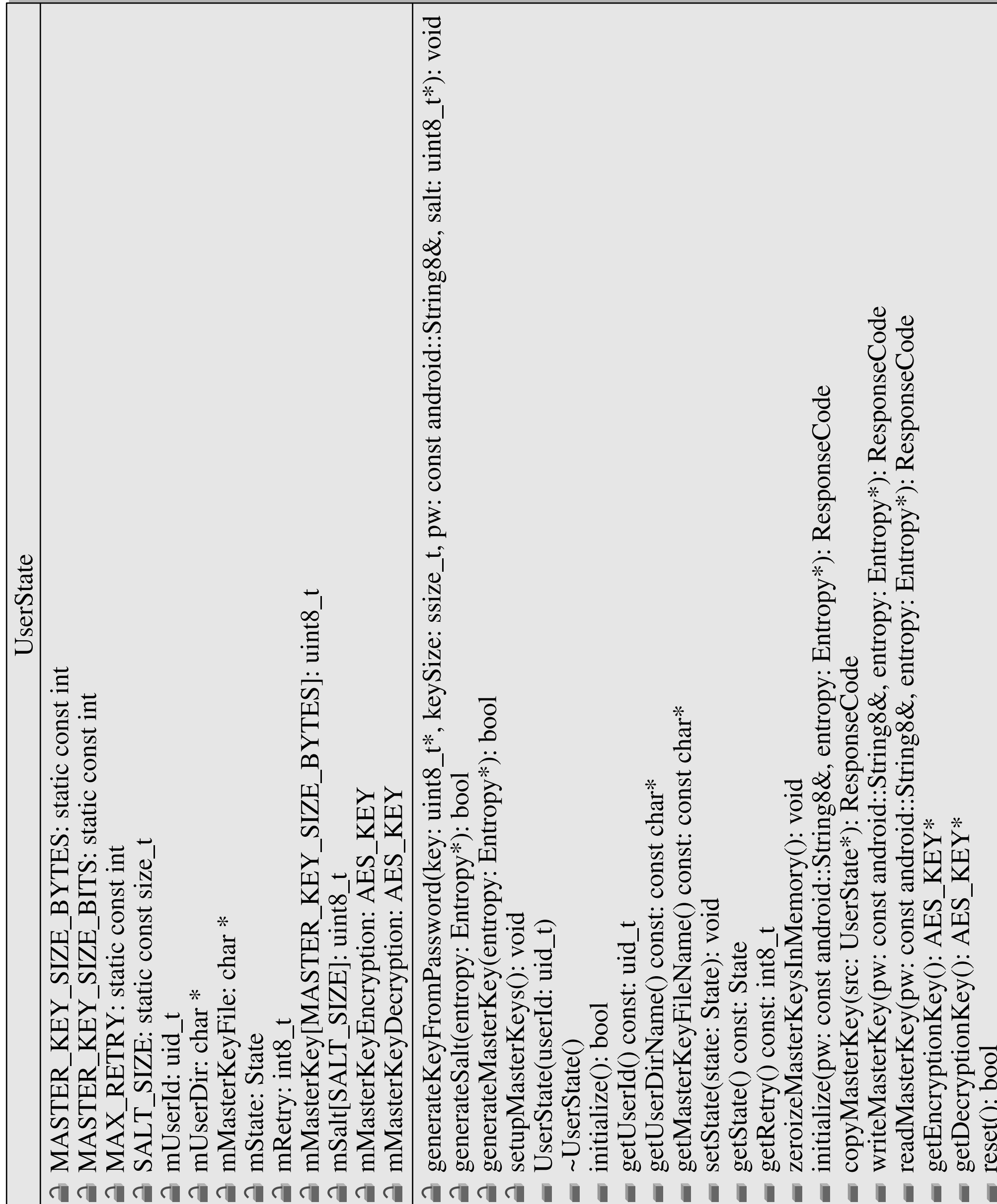


Figure G.5: Class userstate