# TruStick

Device independent secure
cloud storage

Thomas Kastner

# TruStick

Device independent secure
cloud storage

Master's Thesis

at

Graz University of Technology

submitted by

**Thomas Kastner**

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology
A-8010 Graz, Austria

Thu 29th Mar, 2018

Advisor: Univ.-Prof. Ph.D. Roderick Bloem

# TruStick

Unabhängiger
sicherer Cloud-Speicher

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

**Thomas Kastner**

Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie
(IAIK),
Technische Universität Graz
A-8010 Graz

Donnerstag 29 März, 2018

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter: Univ.-Prof. Ph.D. Roderick Bloem

# Abstract

Cloud services provide access to relevant data nearly everywhere at any time. In recent years the increasing usage of cloud-based applications has seriously changed our interaction with data and how they are stored. Personal data, such as photos and videos, as well as commercial data, is stored on remote servers, owned and managed by companies such as Dropbox, Amazon, Google and others. This situation makes it essential to think about, how data is stored and who can access it. Some cloud providers have implemented actions to secure client data from unauthorized access, such as server-side encryption of data. In fact, this solves the problem only partially. From a user's point of view, it is unclear who manages the cryptographic keys used for encryption and decryption. Hence, there is no way for users to observe or control who is accessing their data. Furthermore, data could be intercepted and even manipulated during transmission from and to the cloud server. In this thesis, we developed an USB-stick, called *TruStick*, encrypting data locally before transferring them to a cloud provider. We are using an embedded Linux device, acting as an USB stick, running two operating systems at once. A rich operating system, where USB emulation and data handling is done and a secure operating system monitoring the state of the platform and managing the encryption keys and cryptographic operations. With this approach we can provide data confidentiality and integrity of data during transmission over the network, as well as for data stored in the cloud. The keys used for encryption are only stored locally on the device, without a possibility to extract them from *TruStick*'s secure operating system. Thereby, *TruStick* offers a way to protect data stored on a cloud server, without transferring the responsibility of managing sensitive key material to any 3rd party, leaving the user the only person able to access and manage the stored files.

# Kurzfassung

Cloud-Dienste bieten Zugang zu persönlichen Daten von überall und zu jeder Zeit. Die stetig steigende Nutzung von Cloud-basierten Anwendungen hat unseren Umgang mit Daten und die Art wie sie gespeichert werden drastisch verändert. Private Daten, wie Fotos und Videos, sowie firmeninterne Daten werden auf Servern gespeichert, die von Cloud-Dienstleistern wie Dropbox, Amazon, Google und anderen zur Verfügung gestellt werden. Dieser Umstand macht es notwendig, sich damit zu befassen, wie solche Daten gespeichert werden und wer darauf zugreifen kann. Cloud-Anbieter haben ihrerseits bereits verschiedene Maßnahmen ergriffen, die Daten ihrer Kunden vor unbefugtem Zugriff zu schützen. Zumeist implementieren Anbieter serverseitige Verschlüsselungstechniken um die gespeicherten Daten zu sichern. Allerdings beseitigt dieser Ansatz nicht alle Probleme und Risiken, die sich aus Cloud-basierter Datenspeicherung ergeben. Aus der Sicht eines Benutzers ist es unklar, wie die kryptographischen Schlüssel für die Verschlüsselung und Entschlüsselung gespeichert und verwaltet werden. Daher gibt es keine Möglichkeit für Benutzer, zu überwachen oder zu kontrollieren, wer Zugriff auf ihre Daten hat. Darüber hinaus könnten Daten auch während der Übertragung zu einem Cloud-Server abgefangen und im schlimmsten Fall sogar manipuliert werden. Um den eben genannten Risiken sowohl bei der Datenübertragung als auch der Speicherung vorzubeugen, haben wir in dieser Arbeit einen USB-Stick entwickelt, den *TruStick*, welcher Daten noch vor dem Transfer lokal verschlüsselt und die dabei verwendeten kryptographischen Schlüssel sicher verwaltet, ohne eine Möglichkeit diese zu extrahieren. Dabei wurde ein Einplatinencomputer mit USB Gast Emulation als USB-Stick verwendet, auf dem parallel zwei Betriebssysteme arbeiten. Neben dem Hauptbetriebssystem, das die USB Emulation und die Speicherung und Verwaltung der Daten übernimmt, läuft noch ein kleineres sicheres Betriebssystem, das den Zustand des gesamten Systems überwacht und für sämtliche kryptographischen Operationen zuständig ist. Mit diesem Ansatz können wir Datenintegrität und Vertraulichkeit sowohl auf dem Cloud-Server als auch während der Übertragung sicherstellen. Die für die Verschlüsselung verwendeten Schlüssel sind dabei unter direkter Kontrolle des Benutzers, da sie den *TruStick* niemals verlassen.

## Statutory Declaration

*I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.*

| | | |
|---|---|---|
| Place | Date | Signature |

## Eidesstattliche Erklärung

*Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.*

| | | |
|---|---|---|
| Ort | Datum | Unterschrift |

# Contents

# Acronyms

| | |
|---|---|
| **AE** | Authenticated Encryption |
| **AEAD** | Authenticated Encryption with Associated Data |
| **AES** | Advanced Encryption Standard |
| **API** | Application Programming Interface |
| **DEK** | Data Encryption Key |
| **HMAC** | Keyed-Hash Message Authentication Code |
| **HSM** | Hardware Security Module |
| **IMX53 QSB** | Freescale IMX53 Quick Start Board |
| **KEK** | Key Encryption Key |
| **NW** | Normal World |
| **OCB** | Offsest Codebook Mode |
| **RSA** | Rivest-Shamir-Adleman |
| **SoC** | System-on-Chip |
| **SW** | Secure World |
| **TPM** | Trusted Platform Module |
| **TZ** | ARM TrustZone |

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Motivation

CLOUD services are an important aspect of today's life. Availability of data from everywhere, at any time, has become a common expectation of our society. Companies like Dropbox[Droa] or Amazon[Ama] offer limited cloud storage for free. Operating system distributors like Microsoft (Windows), Google (Chrome OS), or Apple (iOS) integrated cloud storage functionality into their systems (SkyDrive, Google Drive, i-Cloud). However, cloud services are not only limited to storage. Software-as-a-Service (SaaS)[TBB03] allows users to utilize software provided by cloud servers. For example, documents are created on cloud servers and files are never stored locally. At the same time servers offer the necessary programs for processing such files, e.g. *Google Docs*. Google even took one more step and integrated many services into their cloud client operating system *Chrome OS* [Goob].

Alongside with many benefits we gain from being able to access our data from anywhere, cloud services also bear risks. Data confidentiality is one of them. Files stored on public servers always face the risk of being exposed to entities or persons without being authorized by data owners.

An additional circumstance that influences the accessibility of personal data by others is the geographic location of cloud servers. Google for example distributes uploaded data to their data centers that are located in Asia, Europe and America[Gooc]. Users are left without knowledge which data center is actually used for their files. The location can play an important role since the applied law depends on the country, the server is located at. The USA PATRIOT Act[Uni] for example allows public authorities like the NSA, the CIA, and the FBI to access server data without judicial order.

Furthermore, users have no control, or information about who else might have access to the server. System administrators, as well as attackers gaining access to such a system are able to examine and even manipulate all of the data stored there. Therefore, many cloud providers do apply security mechanisms to protect stored data. Dropbox for example encrypts data stored on their servers using 256-bit Advanced Encryption Standard (AES) encryption and offers two-way authentication[Droc]. But there are still many issues left open. For example, server-side encryption raises the question, where, and how the used cryptographic keys are stored, and what happens if those keys get compromised. Furthermore, it is unclear how providers process data internally. This became clear at the last, when the NSA was accused to store the entire plaintext data streams between Google's data centers[WIR].

The fact that sensitive data, which is stored in the cloud, may be exposed to unauthorized persons in different ways makes use of client-side cryptography essential in order to achieve data confidentiality. Software tools like *Boxcryptor*[Sec] provide encryption of data before uploading it to the cloud, using state-of-the art encryption schemes with strong encryption keys. Such software-based approaches still face the problem, that the issue of protecting the key material is left to the end-user. If an attacker manages to extract the private key from a user's system, he or she will be able to decrypt and access the protected data stored in the cloud. Hence, requesting users to store their private keys without ensuring that the keys are protected against theft, constitutes a considerable attack vector for compromising private data. Thus, the major goal of a system using client-side encryption in combination with cloud storage is to provide a secure key storage. Only if the used encryption keys can be protected from unauthorized access, files in the cloud can be considered confidential.

## 1.2 Problem Statement

### 1.2.1 Key Storage

Securely storing sensitive data, such as key material implies that such files are protected from being accessed by other applications, e.g. malware. Keys that are stored directly to an operating system's filesystem are likely to be compromised by attackers. An increasing number of software applications contain software vulnerabilities that can be possibly exploited to bypass a system's built-in security mechanisms. Figure 1.1 illustrates the number of software vulnerabilities in recent years as reported by the *US National Vulnerability Database* [NIS].

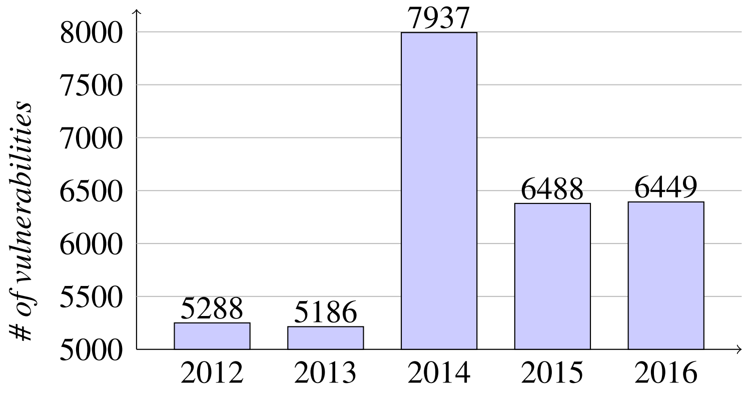


**Figure 1.1:** Number of reported software vulnerabilities per year[GFI].

If attackers are able to exploit such vulnerabilities, they can possibly access sensitive data stored on a system. In order to avoid storing plain cryptographic keys on the filesystem, different approaches are available. Security processors[KL04] or Hardware Security Modules (HSMs)[Sus11] provide ways to measure a system's state and decide, whether a system has been compromised by malware or not (see for example *Secure Boot*[Dav99] or *Integrity Measurement Architecture (IMA)*[Sai+04]). One drawback of such systems is the need for additional security relevant hardware, since costs of production are increased.

### 1.2.2 Data Confidentiality

Storing sensitive data in the cloud without applying any security mechanisms bears the risk that such data might be accessed by unauthorized persons. If attackers manage to gain access to files on a cloud server, this may have severe consequences for affected users. A recently reported incident was the theft of private files from several celebrities including private photo and video material[The]. This applies to companies storing business related data in the cloud as well. Gaining access to a company's cloud storage can be an interesting topic for industrial espionage.

Even if cloud providers do not store data in plaintext, but use cryptography to provide data confidentiality, there are many issues left to be discussed. Using server-sided encryption requires mechanisms to manage and protect used keys. A popular approach is to use key servers[BAL96] that manage and supervise access to the encryption keys. However, the weakness of such systems is the fact, that a key server forms a single point of failure. The worst case scenario involves the leakage of all keys stored on the server, exposing data of all users to attackers. Even if attackers are not able to extract the needed keys, they could delete the stored key files, making user data inaccessible, if there is no appropriate backup service. Moreover, when files are encrypted on the server, an eavesdropper could perform a Man-In-The-Middle attack(as shown in figure 1.2), intercepting file transfer to the server, if communication is not secured properly.

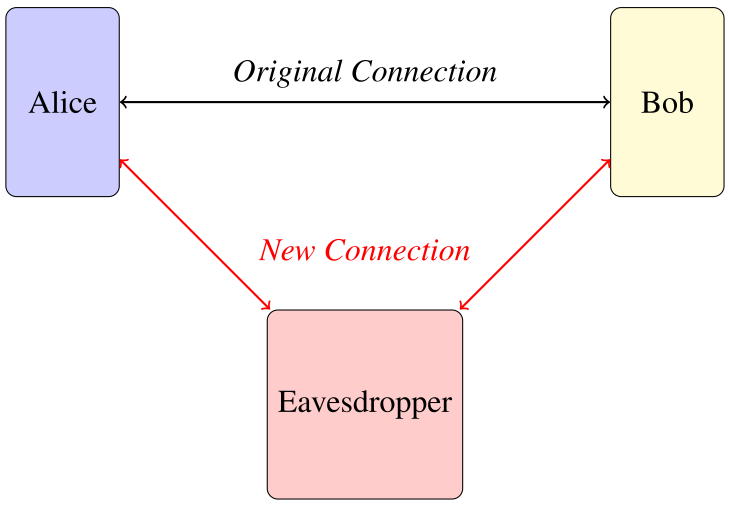


**Figure 1.2:** In a Man-In-The-Middle-Attack an attacker relays a connection between two parties who believe they are directly talking to each other.

### 1.2.3 Data Integrity

Files stored in the cloud have to be protected from being changed by accident or by malicious intention, such as Man-In-The-Middle attacks[Des11]. However, not only external attackers represent a threat to data integrity, so does the cloud itself. Cloud services as provided by Dropbox for example include file versioning, storing old file versions for thirty days before they are deleted. Hence, if we regard the cloud provider untrustworthy, a mechanism is required to ensure that the cloud provides the actual version of a file, when downloading it, as long as no other version is requested.

## 1.3 Proposed Solution

The *TruStick* presented in this thesis provides an approach of locally storing keys used for cryptographic operations. *TruStick* acts as a USB stick allowing users to encrypt files and upload them to a cloud server and vice versa. When attached to the USB port of an arbitrary device, *TruStick* identifies itself as USB mass storage device to the host. In addition to USB connectivity, the *TruStick* has a network interface used to upload and download files from a cloud storage and a Bluetooth module for receiving control commands via an Android smartphone application named *TruTalk*, which we developed for this purpose. We use Dropbox as cloud provider in this thesis. Figure 1.3 shows the block diagram of the communication between *TruStick* and connected devices.

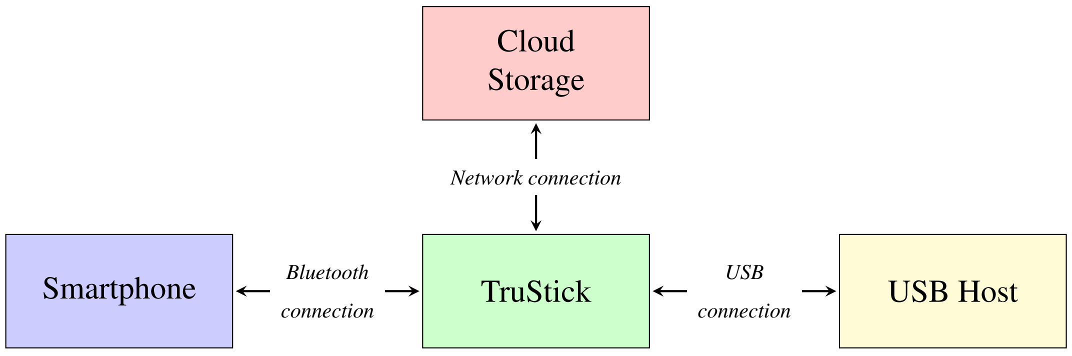


**Figure 1.3:** Trustick block diagram.

**Network connection** is used to transfer encrypted data between a cloud provider and *TruStick*.

**USB connection** provides data transfer from and to arbitrary devices identifying *TruStick* as a regular USB stick.

**Bluetooth connection** gives a user the ability to trigger operations such as encryption or decryption via smartphone.

### 1.3.1 Goals

We defined three major goals for this project to achieve a proof of concept for the *TruStick* to be applicable in practice:

**Usability** We decided to provide USB connection capabilities to *TruStick* for transferring data from any host device to *TruStick*, since USB is a wide-spread and easy-to-use interface for connecting devices and transferring data. To achieve this connection, an emulation of an USB mass storage device on our prototype device is required.

Synchronizing encrypted data with a cloud server requires a local client on *TruStick* monitoring the locally and remotely stored files. We chose Dropbox as cloud provider, since Dropbox provides a comprehensive Application Programming Interface (API), that we use to implement a small client application to synchronize encrypted user data between *TruStick* and Dropbox.

Furthermore, an easy to use command interface is required to trigger encryption and decryption operations from a remote device. We decided to implement a smartphone application therefor, because it gives an easy to understand way to control operations and manage data stored on *TruStick*. A connection between a smartphone and *TruStick* is established via Bluetooth, since most smartphones are capable of handling Bluetooth communication.

In addition to file encryption, file sharing between users is a commonly requested feature for applications dealing with cloud storage. This implies exchanging cryptographic keys between multiple *TruStick*s, in order to decrypt files received from another user.

**Security** In order to ensure data confidentiality, files have to be encrypted automatically before uploading them to a cloud server. In addition, a mechanism is required to ensure data integrity and authenticity after download from the cloud server.

Moreover, the cryptographic keys used for encryption and decryption have to be protected against software attacks.

**Scalability** Allow multiple files to be encrypted at once, combining them to a single fileset. This approach grants users the ability to manage an arbitrary number of files at once, without the need to handle each file separately.

### 1.3.2 System Architecture

To circumvent the need of additional security hardware (as discussed in section 1.2.1), such as a Trusted Platform Module (TPM)[Mor11], we decided to implement *TruStick* on an embedded system, using ARM TrustZone (TZ) technology. The basic principle of TZ is to partition a system into a Secure World (SW) and a Normal World (NW), executing separate operating systems in each world, using only one single physical processor. The SW monitors the state of the system and can overtake control anytime, if a possible software attack or similar questionable behavior is detected. This architecture is known as Trusted Execution Environment (TEE) [Gloa]. For further details about TZ see section 2.5. This environment is meant to prevent attackers from gaining access to sensitive key material, stored in the SW, by exploiting software vulnerabilities of other applications installed on the *TruStick*. We do not claim this approach to be more secure than using a Trusted Platform Module, but we want to show an alternative solution by storing the cryptographic keys and protected data in a TEE-only addressable area. In this thesis we focus on preventing remote or local software attacks. The topic of attackers having physical access to the device is not covered in this paper.

### 1.3.3 File Encryption

To ensure data confidentiality, *TruStick* encrypts files, received via USB, locally before uploading them to a cloud server. For encryption we use key encapsulation, involving the symmetric block cipher AES and the asymmetric block cipher Rivest-Shamir-Adleman (RSA). Key encapsulation significantly increases performance of encryption, since data is encrypted using fast AES encryption and only AES keys are encrypted using RSA.

The RSA keypair, used for key encryption operations, is created, when a user connects to *TruStick* for the first time via the Android application *TruTalk*. Once this initial setup is done, this key is used for any further cryptographic operations on AES keys.

Every time a file is encrypted, a new AES Data Encryption Key (DEK) is generated, and used to encrypt the file content. During the encryption process, the file's content is passed from the NW to the SW where the encryption is performed and the encrypted files are transferred back to NW.

When file encryption is completed, DEK is encrypted using the user's public RSA key. Then, the encrypted DEK is also passed on to the NW and is stored in a meta file next to the encrypted file.

To circumvent the need for an additional Keyed-Hash Message Authentication Code (HMAC) function – applied to the file after encryption to provide data integrity – we use the authenticated encryption scheme AES in Offsest Codebook Mode (OCB) mode [KR] for file encryption. Compared to similar encryption schemes providing same functionality like AES-CCM[DF03] for example, AES-OCB can be considered significantly faster[Roga].

AES-OCB generates an additional authentication tag during the encryption process, which is embedded into the ciphertext. When decrypting the ciphertext this tag is used to validate that the ciphertext has not been altered by accidental or malicious means. This generated tag is stored in the meta data file, signed with the user's RSA public key.

### 1.3.4 Filesets

In addition to encrypting single files, *TruStick* also provides a way to group multiple files to a set. Filesets can be used to combine related files that are likely to be used together. This allows users to structure files according to their personal requirements, like dividing into work-related and private files for example.

### 1.3.5 Dropbox Client

Dropbox provides a comprehensive API[Drob] to develop custom client applications. At the time of developing *TruStick* there has been no client for ARM platforms. Therefore, we have implemented a simple Dropbox client monitoring one folder where encrypted files and meta files are stored. When a new file appears, the client will automatically upload it and the corresponding meta file to the associated Dropbox account. In our design, plain files are deleted after encryption to ensure data confidentiality, even if the *TruStick* is lost or stolen. On the contrary, encrypted files remain on the *TruStick* to provide availability of data in case no network connection is possible. A drawback of this design is the fact, that the number of encrypted files is limited by *TruStick*'s storage capacity, which is likely to be smaller than the storage available in the cloud.

### 1.3.6 File Sharing

A commonly wide-spread functionality of cloud storage is the ability to share files among different users. We decided to use our smartphone app *TruTalk* for exchanging cryptographic keys, that are needed for file decryption. Thereby, users are able to exchange keys without requiring the *TruStick* to be present. To enable file sharing with encrypted files between multiple *TruStick* devices, users can export the public part of their Key Encryption Key (KEK) via Bluetooth to the *TruTalk* app and exchange them in form of a QR-code with other users. After exchanging a key, users are able to transfer such keys back to their own *TruStick* using Bluetooth again. *TruStick* imports a received key into the SW and stores it for further encryption and decryption operations.

### 1.3.7 *TruTalk*

In 2014 smartphone users totalled around 1.75 billion[eMa], making the smartphone a widely distributed portable device, that can be used to send control commands to the *TruStick* and exchange keys, without the need of carrying on the *TruStick* itself. The Android application *TruTalk* has two main purposes. It serves as command interface for uploading and downloading files and defining file sets. Uploading files can be done in two different ways: upload all files found on the *TruStick*, each encrypted as a single file, or retrieve a list of files from *TruStick* and specify a fileset with any of the listed files. For downloading, a list of uploaded files can be retrieved from the *TruStick*. Users can select entries, which will be downloaded and made available for the host computer. The second purpose is to exchange the RSA public key with other users to share encrypted files. After obtaining the public key from the *TruStick* *TruTalk* can generate a QR-code to share this keys. *TruTalk* also has a built-in QR-code reader to decode public RSA keys provided by another user's smartphone.

#### 1.3.7.1 Authentication

When launching *TruTalk* for the first time, users have to choose a username and a password, that will be stored on the smartphone. After this setup procedure any further interaction with *TruTalk* require the correct user credentials to be provided.

Furthermore, the user credentials are sent to the *TruStick*, where an RSA keypair is created in the SW for the given user. The access credentials and the keypair are stored on the filesystem in the SW and are used for verifying user authorization when issuing commands that involve the usage of cryptographic key material stored in the SW.

For this prototype we store the SHA-256[Sav] hash of the password as private preference[Gooa] of our Android application, not accessible by other applications. Considering threats that might affect a smartphone, storing the password should be done using a more advanced approach for further work.

#### 1.3.7.2 Security

Since Bluetooth is known to be prone to Man-In-The-Middle attacks, we use a shared secret key to increase security during transmission. When establishing a connection between *TruTalk* and *TruStick*, the two parties agree on a shared secret using Diffie-Hellman key exchange mechanism. After this key setup phase was successful, entire communication between the devices is encrypted using AES block cipher, with the shared secret used as AES encryption key.

#### 1.3.7.3 Graphical User Interface

Figure 1.4 shows the user interface for issuing commands to the *TruStick*. Figure 1.5 represents the interface for exchanging a public RSA key, that is encoded into the QR-code.

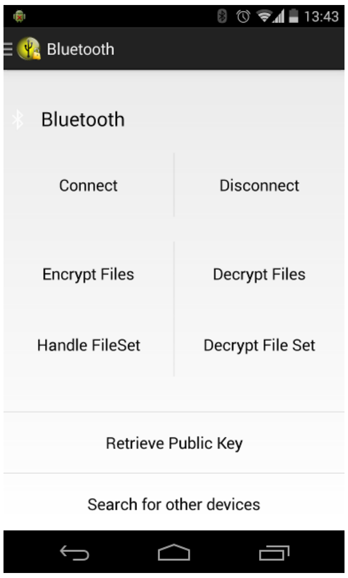


**Figure 1.4:** TruTalk Command Interface

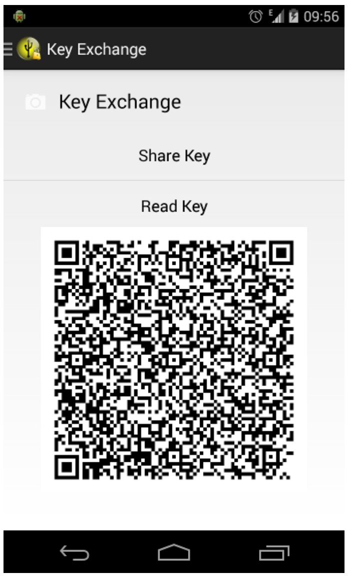


**Figure 1.5:** Key Exchange via TruTalk

### 1.3.8 Prototype

The use of ARM TrustZone (TZ) requires a System-on-Chip (SoC)[Kla04] with security extensions. To fulfill this requirement, we implemented our prototype using the Freescale IMX53 Quick Start Board (IMX53 QSB)[NXP]. The IMX53 QSB has an ARM Cortex-A8 processor, supporting TZ. We utilize ANDIX OS (see section 2.6) as SW operating system and a basic Debian Linux OS for the NW. The operating system images are loaded from a micro SD card. The iMX53 is configured to act as *USB Mass Storage Device* to an arbitrary host system. Connection between the host and the i.MX53 board is realized with the Mass Storage Gadget kernel module from the USB Gadget project[Lin]. This module simulates an USB Mass Storage Device, allowing data transfer via USB. Additionally the i.MX53 has a network connection for using cloud services and a USB Bluetooth adapter[Cona] with Bluetooth 4.0 standard for communication with our smartphone application *TruTalk*.

### 1.3.9 Outline

In chapter 2 we give an overview of topics that are relevant to understand the functionality of *TruStick*. After that, chapter 3 discusses projects dealing with similar approaches to processing sensitive data. Chapter 4 describes our implementation of *TruStick* and specifies the implementation of our Android app *TruTalk*. Chapter 5 explains our results and itemizes open issues that were not covered in this thesis. Finally chapter 6 concludes this thesis.

# 2 Preliminaries

THIS chapter gives a short introduction into the algorithms and technologies used for implementing *TruStick* including hash trees and the cryptographic algorithms Advanced Encryption Standard (AES) in Offsest Codebook Mode (OCB) and Rivest-Shamir-Adleman (RSA) as well as a brief overview about the functionality of ARM TrustZone (TZ).

## 2.1 RSA

RSA is an asymmetric block cipher named after the inventors R. L. Rivest, A. Shamir, and L. Adleman and was first proposed in 1978[RSA78]. The RSA cryptosystem can be used for encrypting or digitally signing data (see figure 2.1 and figure 2.2). Since RSA is an asymmetric cipher, a keypair, consisting of a private and a public key, is used for operation. The public key is used for encryption and verifying digital signatures and the private key for decryption and signing data. For performance reasons RSA is commonly not used for data encryption, but used in combination with a more efficient symmetric cipher, such as AES. In such a hybrid system, data is encrypted using the symmetric cipher and the symmetric key is encrypted with RSA.

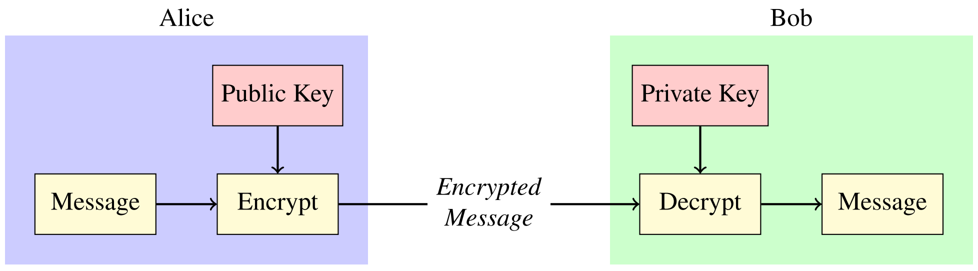


**Figure 2.1:** Message encryption with RSA.

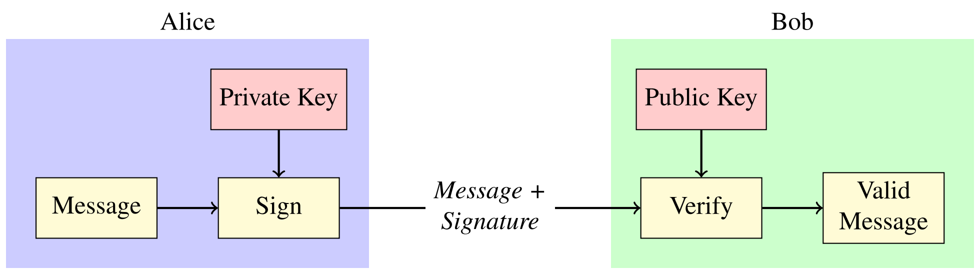


**Figure 2.2:** Digital signature with RSA.

### 2.1.1 Algorithm

This section describes the algorithms for the RSA encryption scheme, including keypair generation and encryption/decryption as discussed in [MVO96]. Listing 2.1 describes the generation of the *encryption exponent (e)*, the *decryption exponent (d)* and the *modulus (n)*. Listing 2.2 shows the usage of *(e,d,n)* for encryption and decryption of an arbitrary message *m*.

**Listing 2.1:** RSA key generation

```
Generate two large random and distinct primes $p$ and $q$.
Compute $n = pq$ and $\phi = (p-1)(q-1)$.
Select a random integer $e$, $1 < e < \phi$, with $\gcd(e,\phi) = 1$.
Use the extended Euclidean algorithm to compute the
   unique integer $d$, $1 < d < \phi$, such that $ed \equiv 1 \mod (\phi)$.
The public key is $(n,e)$; the private key is $d$.
```

**Listing 2.2:** RSA public-key encryption

```
SUMMARY: B encrypts a message $m$ for A, which A decrypts.
B obtains A's authentic public key $(n,e)$.
Represent the message as an integer $m$ in the interval
   $[0,n-1]$.
Compute $c = m^e \mod (n)$.
Send the ciphertext $c$ to A.
To recover the plaintext $m$ A uses the private key $d$ and
   computes $m = c^d \mod (n)$.
```

## 2.2 Diffie-Hellman Key Exchange

Secure communication over an untrusted channel requires the involved parties to share a secret cryptographic key. A common way to generate such keys is the Diffie-Hellman Key Exchange Protocol[DH06]. This protocol allows two parties to generate a shared secret over an insecure channel without revealing any sensitive information to an eavesdropper. If two parties Alice and Bob want to exchange a common key, they first have to agree upon the public parameters $p$ and $g$ via the insecure channel, where $p$ is a prime number and $g$ is a primitive root modulo $p$. Thereafter, Alice and Bob both choose an integer ($a$ and $b$) that has to be kept secret and is not transmitted over the channel. Alice computes

$$A \equiv g^a \mod p \tag{2.1}$$

and sends $A$ to Bob via the insecure channel. Vice versa, Bob computes

$$B \equiv g^b \mod p \tag{2.2}$$

and sends $B$ to Alice. Now, both parties can compute the shared secret:

$$K_{AB} \equiv B^a \equiv g^{ab} \mod p \quad \textit{(Alice)} \tag{2.3}$$
$$K_{AB} \equiv A^b \equiv g^{ab} \mod p \quad \textit{(Bob)} \tag{2.4}$$

The established shared secret key $K$ can now be used to encrypt further communication over the insecure channel using a symmetric cryptographic protocol, such as AES. Figure 2.3 illustrates the required steps for a key exchange.

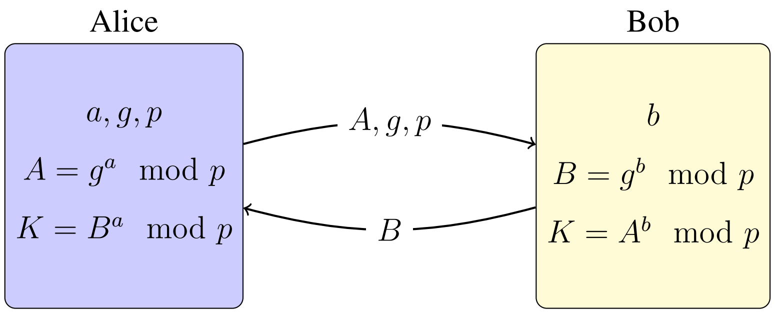


**Figure 2.3:** Basic Diffie-Hellman key exchange.

## 2.3 AES-OCB

Data encryption on *TruStick* is done using *Authenticated Encryption (AE)*, namely OCB (Offset Codebook Mode, Krovetz and Rogaway [KR]) with AES used as block cipher. The goal of AE is to provide data confidentiality, as well as authenticity. In RFC 7253 these two terms are defined as following:

> *Confidentiality is defined via "indistinguishability from random bits", meaning that an adversary is unable to distinguish OCB outputs from an equal number of random bits. Authenticity is defined via "authenticity of ciphertexts", meaning that an adversary is unable to produce any valid nonce-ciphertext pair that it has not already acquired.*

A more traditional approach of achieving confidentiality and authenticity would involve separate encryption followed by an authentication scheme, using an own key for each mechanism. AE combines these two functionality in one cryptographic operation. Since

AES-OCB is also an Authenticated Encryption with Associated Data (AEAD) scheme, it is possible to add authenticity to additional plain data, which do not need to be encrypted. This might be useful when considering a packet encryption scheme for example, where the payload has to be encrypted and authenticated and the packet header only needs authentication.

### 2.3.1 Properties

The design of OCB includes following properties [Rogb]:

- *Authenticated Encryption Scheme*: encrypted messages are both private and authenticated.
- *Indistinguishability under chosen-ciphertext attack* and *non-malleability under chosen-ciphertext attacks*.
- Fully parallelizable. Very good for encrypting messages in hardware at the highest network speeds.
- Works with any block cipher.
- OCB makes a nearly optimal number of block-cipher calls: $\lceil |M|/n \rceil + 2$.
- Only a nonce is required (but no IV).
- Only a single block-cipher key is used.
- Key setup in OCB is very cheap (typically one block-cipher call).
- OCB is very memory efficient.
- OCB generates a sequence of offsets, which it does in a very cheap way. Each offset is computed from the previous one either by xoring the previous offset by a value looked up in a small table or by doing a few shifts and xors.
- OCB can encrypt messages of arbitrary length. Messages don't have to be a multiple of the block length, and no additional padding is needed.
- Messages of all lengths are treated in a single, uniform manner.
- The length of an OCB ciphertext is the same as the length of the plaintext.
- OCB avoids 128-bit addition (which is endian-biased and can be expensive in software or dedicated hardware). It uses xors instead.
- OCB is simple to understand and implement. It uses $GF(2^{128})$ arithmetic and a Gray code, but it all comes down to some xors and shifts.
- OCB is provably secure. It provably meets its goals, as long as the underlying block cipher meets standard cryptographic assumptions.

## 2.3.2 Algorithm

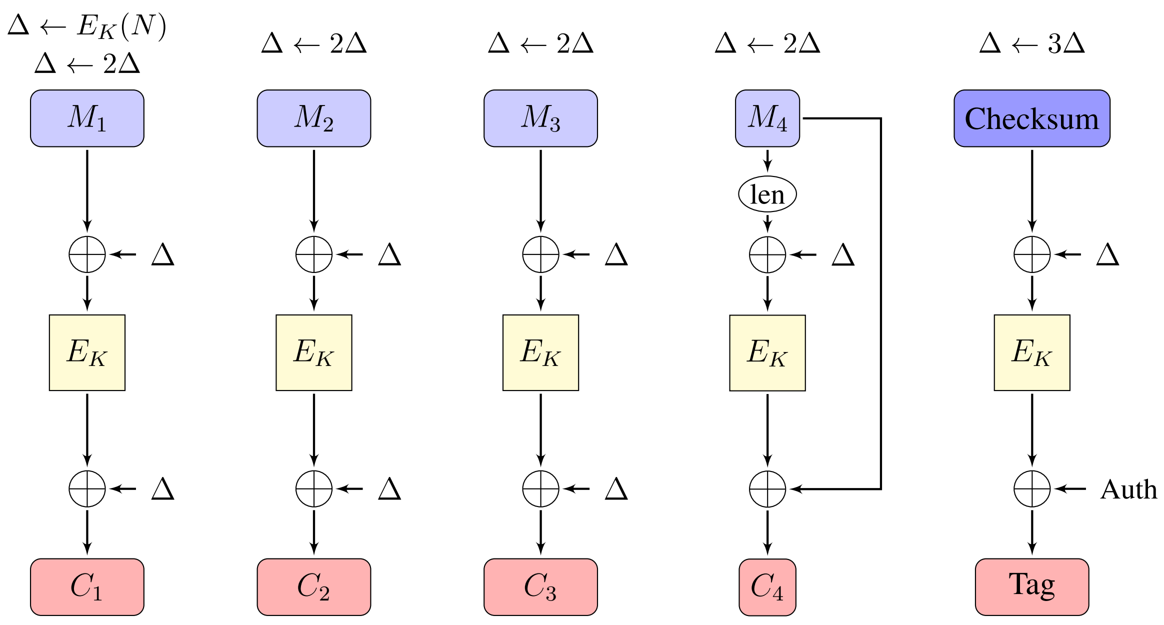
AES-OCB extends the regular encryption process by introducing an additional offset value. The initial offset is calculated by using an unique nonce of 128-bit length. This nonce is XORed with a 128-bit string that is generated from encrypting $0$ with the used AES key. The output of the XOR is again encrypted with the same AES key. This encryption finally yields the initial offset value. Data encryption is done, by XORing plain data with the offset value and encrypting the result with the given AES key. After encryption, the output is again XORed with the offset value generating the final cipher block. After processing one block, the offset value is changed by performing another XOR operation on the current offset value with a new 128-bit string.

Listing 2.3 describes an encryption process of message *M* broken into *M[1]M[2]...M[m]* blocks of size 128-bit each except the last block. Following notation is used: *K* denotes an AES encryption key with arbitrary length (128, 192, or 256 bit). *Nonce* is a 128-bit nonce and *L* represents *AES(K,0)*. Define $L(0)$ be $L$ and, for $i > 0$, let $L(i)$ be $L(i-1) << 1$ if the first bit of $L(i-1)$ is 0, and let $L(i)$ be $(L(i-1) << 1)$ **xor** 0x00000000000000000000000000000087 otherwise. Let $L(-1)$ be $L >> 1$ if the last bit of $L$ is 0, and let $L(-1)$ be $L >> 1$ **xor** 0x80000000000000000000000000000043 otherwise.

**Listing 2.3:** AES-OCB encryption algorithm

```
Offset = AES(K, Nonce xor L)
Checksum = 0
for i = 1 to m-1 do
    Offset = Offset xor L(i)
    Checksum = Checksum xor M[i]
    C[i] = Offset xor AES(K, M[i] xor Offset)
end
Offset = Offset xor L(m)
Pad = AES(K, len(M(m)] xor L(-1) xor Offset)
C[m] = M[m] xor (the first |M[m]| bits of Pad)
Checksum = Checksum xor Pad xor C[m]
FullTag = AES(K, Checksum xor Offset)
Tag = a prefix of FullTag (of the desired length)
return C[1]...C[m-1] C[m] Tag
```

Figure 2.4 shows a different representation of AES-OCB encryption process using a block diagram. The diagram should be read from left to right, whereat $\Delta$ denotes the *Offset*. For a more detailed description of OCB, see [KR].



$\Delta \leftarrow E_K(N)$
$\Delta \leftarrow 2\Delta$
$\Delta \leftarrow 2\Delta$
$\Delta \leftarrow 2\Delta$
$\Delta \leftarrow 2\Delta$
$\Delta \leftarrow 3\Delta$
$M_1$
$M_2$
$M_3$
$M_4$
Checksum
len
$\Delta$
$E_K$
Auth
$C_1$
$C_2$
$C_3$
$C_4$
Tag

**Figure 2.4:** Block diagram of OCB encryption.

## 2.4 Merkle tree

A Merkle tree or hash tree can be used to provide integrity among multiple data blocks. They provide an efficient and secure way to verify the content of large data structures. Building a Merkle tree involves applying a hash function to each data block. The yielding hash values are used as the tree's leaves. Concatenating two or more hashes creates the next level node of the tree. Most hash trees are binary taking two hash values to create a node, but they can also use more values. Repeating this procedure finally yields the *root hash* denoting the top of the tree. Figure 2.5 shows a Merkle tree built from four leaf nodes representing arbitrary data.

[Bec08] states that verifying that a leaf node is part of a given hash tree requires processing an amount of data proportional to the logarithm of the number of nodes of the tree. Compared to hash lists, where the amount of data is proportional to the number of nodes, this is a big improvement concerning efficiency. Merkle trees are used by projects such as BitTorrent [Inc], Git [Conb], and the ZFS file system.

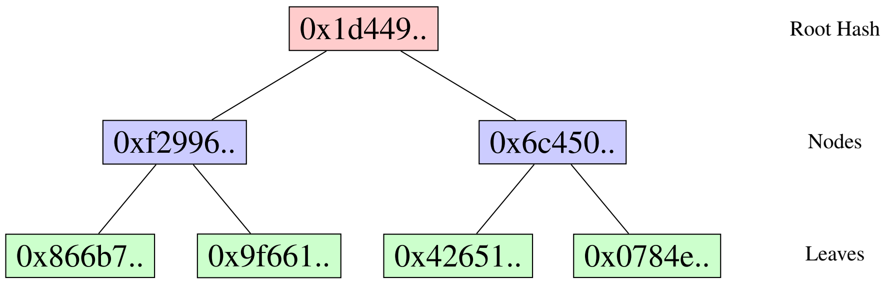


**Figure 2.5:** Example structure of a Merkle-Tree.

## 2.5 ARM TrustZone

In contrary to traditional security solutions in embedded systems where a separate security processor is required, ARM TrustZone [Lim09] makes use of security extensions, which are a feature of the processor architecture, integrated into a number of different ARM processors. These extensions introduce an additional secure mode into all relevant components of a system, such as CPU, bus interface and address space controller. The secure mode provides the possibility to split a system into a rich operating system and a much smaller, secure operating system. Since this separation is done in hardware, TZ provides a robust platform, on which the upper layers of secure software can be built. This approach solves some of the disadvantages of which external security processors suffer, such as lack of programmability or higher system costs caused by additional hardware.

The design of TZ allows a single physical processor to execute code from the secure world (SW) as well as the normal world (NW). Therefore, a controller is needed to pass control between the two worlds. To realize these world switches, a new processor mode, the secure monitor mode, has been introduced. This additional CPU mode monitors all interactions between the SW and the NW. This mode can be entered from either the SW or the NW by invoking a *Secure Monitor Call*. In case unauthorized access to security relevant memory regions or other critical resources from the NW is detected, the secure monitor mode is invoked, and the SW should react on this case. Figure 2.6 illustrates the transitions between the two worlds via the monitor mode.

The world in which the system's processor is executing is defined by a special bit in the *Secure Configuration Register (SCR)*. The value of this bit decides the world to be executed, except the case, the system is in monitor mode. In monitor mode the system is always executing in the Secure World.

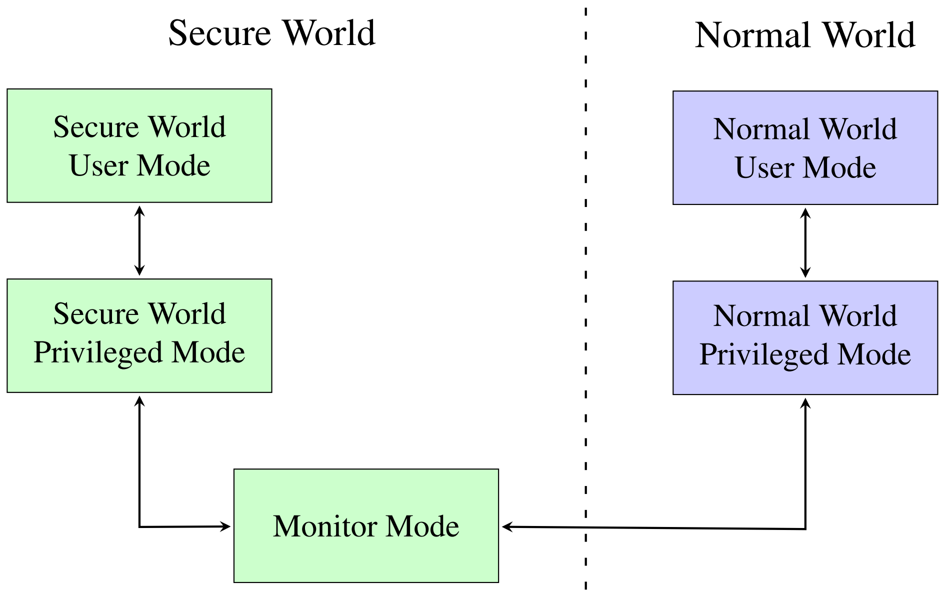


**Figure 2.6:** Separation between normal and secure world.

## 2.6 ANDIX OS

In 2014 Andreas Fitzek proposed ANDIX OS [Fiz14], an operating system designed for the SW of a system using TrustZone technology. At present, ANDIX OS can be executed on the IMX53 QSB, as well as on an adapted version of the QEMU ARM emulator with TrustZone support [Win+12]. The following section will given a short introduction into ANDIX OS. For further reading see [Fiz14].

### 2.6.1 Secure World

The kernel of the Secure World (SW) is the core component of ANDIX OS. It manages the boot process and the security functions provided by ARM TrustZone, such as protection of the SW memory. Therefore, the SW kernel handles the isolation between the NW and the SW, since the NW must not access physical memory owned by the SW.

After the boot process, the SW acts as bootloader for the NW system. This includes verifying the integrity of the NW system before boot.

At runtime, the SW monitors the NW system's state, detects possible attacks, and performs counteractions in case of an attack. When an attack is detected, the SW takes control over the system and stops execution to prevent further malicious actions, such as access to sensitive memory regions.

#### 2.6.1.1 Trusted Applications

Trusted applications are executed in the SW userspace. They are are statically linked to the SW kernel and are contained in the system image, together with the SW kernel. The SW kernel manages the initialization and separation of the Trusted Applications. NW clients can interact with Trusted Applications via *remote procedure calls (RPCs)*, allowing NW applications to exchange data and trigger operations in the SW, like for example cryptographic operations with key material stored in the SW. The interface for communication with Trusted Applications is defined by the Trusted Execution Environment Client API [Glob].

### 2.6.2 Normal World

After booting into the SW the normal world kernel is loaded and passed control by the SW. The SW stays passive and does not influence the NW process until the NW sends a request to the monitor mode, or a possible attack is detected.

### 2.6.3 World Communication

Communication between a NW application and a SW trusted application is realized by world switches. Therefore, both worlds use the monitor mode to switch modes of operation. Figure 2.7 shows the real communication path using the monitor mode and the logical communication path, provided by the TEE Client API [Glob] in the NW and the TEE Internal API [Gloc] in the SW. When an application initiates a communication channel with a trusted application, a session is opened between the NW application and the SW trusted application. This session can be used to store information, that needs to be available through the whole communication process.

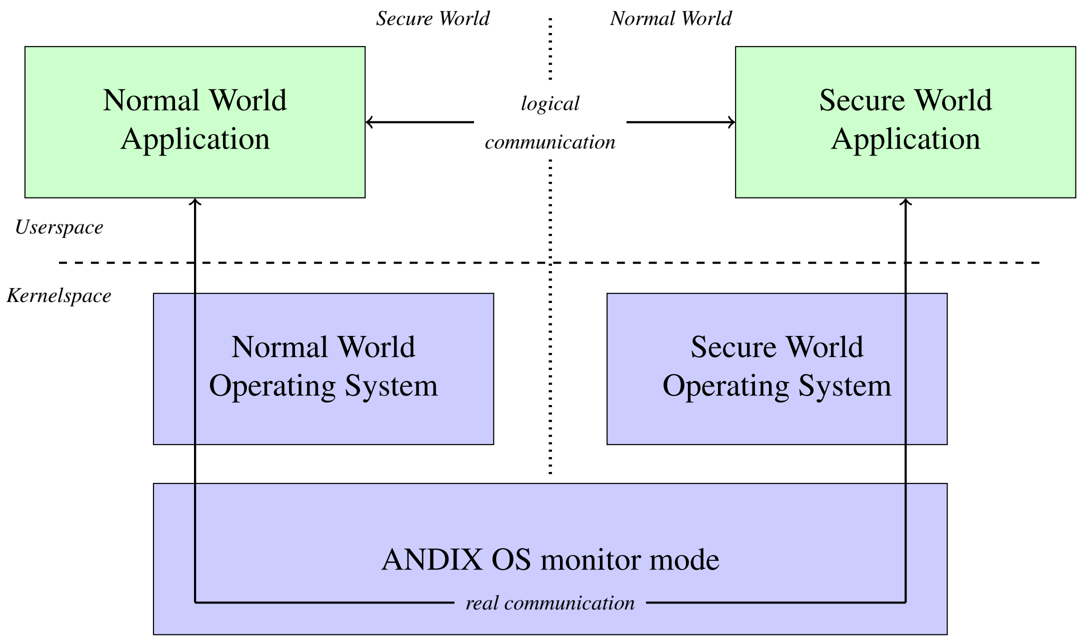


**Figure 2.7:** Communication between normal and secure applications.

# 3 Related Work

THE following chapter discusses projects related to topics covered by *TruStick*. The chapter is divided into projects dealing with cloud security (section 3.1) and projects providing local or remote attestation involving a HSM using USB technology (section 3.2).

## 3.1 Cloud based approaches

**DroidVault** Li et al. [Li+14] proposed DroidVault, a *Trusted Data Vault* for processing sensitive data on mobile devices. DroidVault makes use of TrustZone technology integrated into mobile devices executing Android operating system in Normal World. Users can retrieve sensitive encrypted data from a trusted remote server and securely decrypt it within the Secure World without exposing key material to the untrusted Android file system. The core components of DroidVault are a secure connection to a specified remote server and a I/O module to allow secure data input and display. For accessing sensitive data from a remote server, the server has to provide signed code that is loaded into the DroidVault and used for processing such data. Only if the signature can be verified by DroidVault, the code will be executed and hence the data will be available. Like *TruStick*, DroidVault uses Dropbox as remote server with the prototype, which has also been implemented using the Frescale IMX53 board. But in contrary we do not consider Dropbox to be trusted, but we limited the boundaries of trust to the *TruStick* itself. Moreover, *TruStick* allows to present decrypted data to a variety of devices, whereas DroidVault is only meant to provide data access via the smart phone.

**DFCLOUD** Shin et al. [Shi+12] introduced a project named DFCloud using an approach similar to Trustick. The DFCloud architecture involves three components: TrustZone enabled mobile client devices, a Server Manager, and backend cloud storages. Besides using TrustZone technology every mobile device has an emulated TPM located in the Secure World for key management. Crypographic file processing is done in the Secure World on the client devices. The Server Manager is in charge of distributing keys to additional devices, remote attestation of client devices, and handling third party cloud storage services. However, the DFCloud prototype was implemented on an emulated

ARM Cortex-A15 without porting the software to a hardware device. Furthermore, mobile clients are only meant to locally use downloaded files, whereas our Trustick serves as USB mass storage device, distributing files to other systems. We believe that we are able to achieve the same level of security without an additional hardware security device.

**ZTIC** Weigold et al. [Wei+08] implemented an USB stick – the *Zurich Trusted Information Channel (ZTIC)* – basically acting as Man-In-The-Middle, between online services and a client PC. Therefore, a network proxy running on the used PC passes all traffic to the USB stick. The ZTIC processes all communication from and to the client as, scanning exchanged data for sensitive operations such as bank transactions. Such transactions are intercepted and relevant information is shown on the on-board display of the stick. Only if a user presses a button on the stick, explicitly stating, that she agrees with the information shown, the communication gets passed through. This security system prevents malware and Man-In-The-Middle attacks regarding sensitive communication over the network, e.g. bank transactions.

**EMR** Akinyele et al. [Aki+11] use mobile devices to access and view self-protecting electronic medical records (EMRs) with attribute-based encryption stored on a hospital's SSL server. Therefore, a policy encryption engine parses each node of a newly submitted, XML-based, EMR record to calculate the correct access policies. Such EMRs can then be stored on external cloud storages or on mobile devices such as smartphones and can be downloaded by patients and healthcare providers with appropriate permissions.

## 3.2 USB devices

A couple of previous works use external USB devices for remote or local attestation using hardware security modules or simulating such devices. Also there are several approaches to provide a Portable Trusted Module (PTM).

**iTurtle** McCune et al. [McC+07] proposed a USB device – the *iTurtle* – aiming to verify a platform's state using remote attestation. The *iTurtle* allows a user to easily distinguish between trusted and untrusted states by providing a visual feedback, such as a LED light, stating the verification result. However, such a device has never been realized.

**MTA** Feng et al. [Fen+13b] developed a *Mobile Trusted Agent* (MTA) acting as a trusted medium between a local user and a remote verifier. A remote verifier is used to validate a platform configuration defined by measurements done by a local TPM. The MTA provides the cryptograhic functionality to establish a secure connection to this verifier and is also able to act as TPM for a host platform without a TPM. The MTA can be connected via USB using the USB Gadget library or via network. The prototype of this project was implemented on an ARM Real210 development board with a software TPM.

**PTPM** Zhang, Han, and Yan [ZHY10] introduced the *Portable TPM Based on USB Key*, which is a USB device with Java Card Runtime Environment, a little LCD display, and a button. It holds a Java Card Applet with several TPM functions implemented for locally attesting multiple devices with the same Attestation Identity Keys. Such keys are used for signing values stored in the platform configuration registers. Therefore, the assumption has to be made, that this key, as well as the Storage Root Key are migratable.

**TEEM** Feng et al. [Fen+13a] created a trusted computing module, called *Trusted Execution Environment Module (TEEM)*, that can be used by desktop computers, as well as by mobile devices. It is deployed in the Secure World on a mobile with TrustZone serving as *Mobile Trusted Module (MTM)* there. If connected to a desktop platform via USB it is configured to behave as PTM. They implemented a prototype using a ARM Real210 development board with TPM/MTM emulator, which were adapted to support more Trusted Computing Modules and crypotgraphic algorithms.

**SDM** Hein et al. [Hei+12] presented a paper using a *Secure Docking Module* (SDM) to provide decentralized method of making trust decisions. The SDM was developed for crisis management using mobile agents for information exchange in areas without reliable network infrastructure. A local decision authority is used to determine whether a mobile agent may access requested information or not. This is achieved by combining Intel's Trusted Execution Technology (TXT) with a TPM to a *Trusted Docking Station* (TDS). The TPM is used to measure the platform configuration into it's PCRs. Only if the platform running on the TDS has a configuration known to the SDM, cryptographic material needed by the platform will be released.

**SecurID** RSA distributes a device named SecurID On-Demand Authenticator, delivering a one-time password (OTP) to a user's mobile device via an SMS text message or email, turning that mobile device into a security token.

Some works related to cloud services address the topic of securing existing applications. Boxcryptor is an Open-Source application, which also automatically encrypts files like the *TruStick* before uploading them to cloud storage. Popa et al. [Pop+11] implemented a system, *CloudProof*, allowing users to detect and even proof violations of integrity to a third- party. Bessani et al. [Bes+13] introduced *DepSky*, a mechanism, that does not rely on storing data in a single cloud, but distributes encrypted data blobs to different cloud providers.

# 4 Implementation

THE implementation of *TruStick* covers multiple components that are discussed in the following chapter. Most functionality is implemented in the NW of *TruStick*, except parts using sensitive key material for cryptographic operations (see figure 4.1). Section 4.1 illustrates how USB mass storage functionality has been achieved and Section 4.2 gives an overview of file handling on *TruStick*. In section 4.3 we will discuss the functionality of our encryption application *TruCrypt* in detail. Section 4.4 describes how commands to *TruStick* are handled by our Bluetooth communication application *TruServer* and section 4.5 explains how file synchronization between *TruStick* and a cloud server is realized.

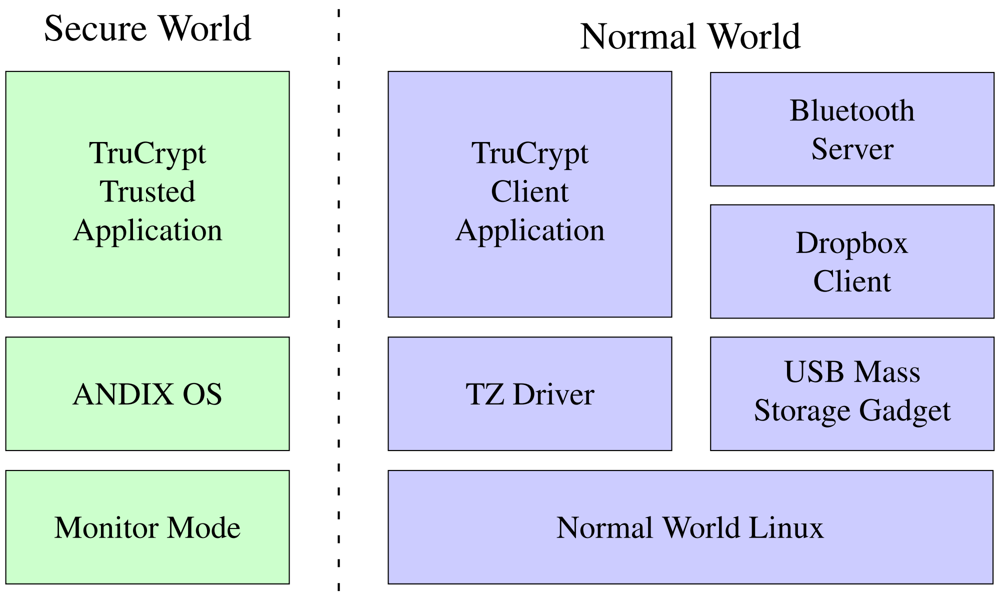


**Figure 4.1:** Separation between Normal and Secure World.

## 4.1 USB Mass Storage Gadget

*TruStick* uses the USB Mass Storage Gadget to provide USB storage functionality when attached to other devices. *TruStick* reports itself as mass storage device providing data via a special block device stored in the NW of *TruStick*. The block device only serves as gateway to transfer unencrypted data from and to *TruStick*. The following section describes in detail how this functionality is implemented.

We use the Linux USB Gadget Framework API to provide USB mass storage functionality with the IMX53 QSB. This framework allows systems to act in USB *device* (slave) role. The necessary hardware requirement for this feature is a USB controller that supports acting as such a slave device, which is provided by the IMX53 QSB. The Gadget Framework API is split into several layers (see figure 4.2):

- **Peripheral Controller Drivers** are the only layers directly communicating with the hardware. They implement the Gadget Drivers and support an arbitrary number of such drivers, but only one can be used at a time.
- **Gadget Drivers** use calls to the controller drivers and implement hardware-neutral USB functions.
- **Upper Layers** build the upper boundaries for Gadget drivers connecting to further drivers or frameworks in the operating system.

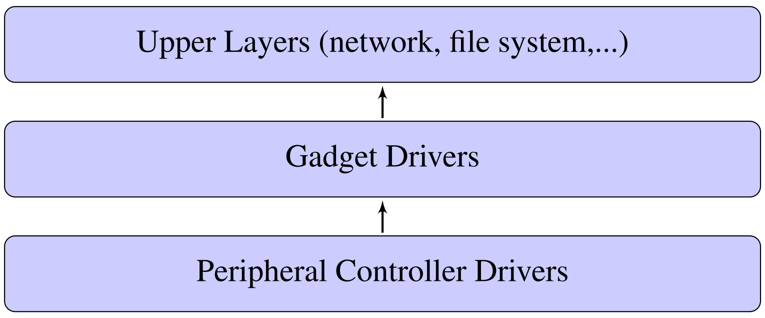

Upper Layers (network, file system,...)
Gadget Drivers
Peripheral Controller Drivers

**Figure 4.2:** Linux-USB Gadget Framework API layers.

The Gadget Framework already comes with several public available gadget drivers, providing common USB functionality. One of the provided drivers is the **Mass Storage Gadget**, which we utilize to make the IMX53 QSB act as USB storage device to arbitrary USB hosts. The gadget takes an arbitrary file or block device as backing store, which has to be specified when the appropriate kernel module is loaded, and presents it to a host as SCSI disk drive. On *TruStick*, the backing store is a separate partition on the SD card. For interoperability reasons the partition was formatted as FAT32 partition, thus file sizes are limited to 4 GB.

Integrating the *Mass Storage Gadget* into *TruStick* only requires USB support to be configured into the NW Linux kernel. USB support is contained in Linux kernel 2.2.7 and later. With an appropriately configured kernel, loading the USB kernel module works flawlessly on the IMX53 QSB. One consideration that has to be made though, is that manipulating data in the backing store, while the mass storage device is connected to a USB host, might cause unwanted behavior or even destroy data on the device, since USB hosts do not expect data on the device to be changed, when it is connected.

## 4.2 File Handing

We have implemented three applications, interacting with each other, to achieve the goal of securely managing files and storing them on a cloud server. Figure 4.3 shows the basic interaction between these applications within *TruStick*.

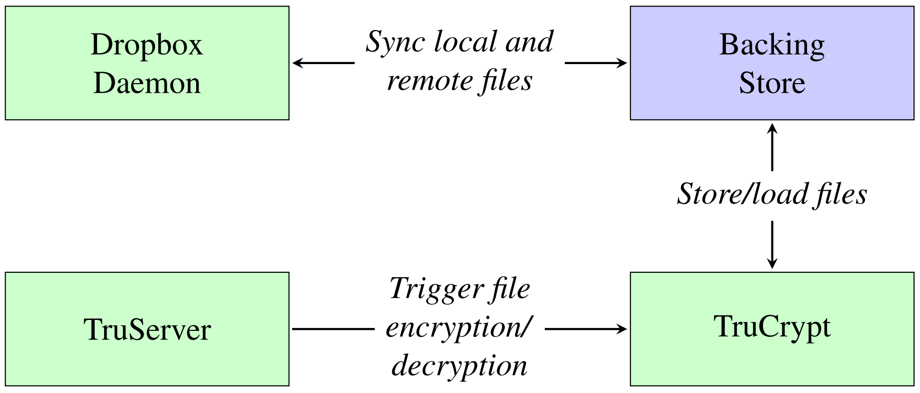

Dropbox Daemon
Sync local and remote files
Backing Store
Store/load files
TruServer
Trigger file encryption/ decryption
TruCrypt

**Figure 4.3:** Interaction between applications and backing store.

- *TruCrypt* (section 4.3) is responsible for file encryption and decryption. This also includes combining files into filesets and verifying data integrity after decryption. *TruCrypt* is built of the client application in the NW and the trusted application in the SW. Commands to *TruCrypt* are issued by *TruServer*.
- *TruServer* (section 4.4) provides the Bluetooth communication interface required by our *TruTalk* app(see section 4.6) and triggers the encryption/decryption process performed by the *TruCrypt* application. Furthermore, *TruServer* reports the currently available files to *TruTalk*.
- *Dropbox Daemon* (section 4.5) monitors the local file status and synchronizes files with Dropbox.

## 4.3 TruCrypt

*TruCrypt* is the client application for interaction with the SW trusted application.

When performing an encryption operation, *TruCrypt* reads the unencrypted input file block-wise and transfers the it to the SW. After encryption the encrypted file blocks are transferred back to the NW and *TruCrypt* stores them in a file in the backing store, monitored by the *Dropbox Daemon*. Vice versa, file decryption works in the same way. The *TruCrypt* client transfers an encrypted file to the SW and stores the decrypted file afterwards. In addition, *TruCrypt* also processes meta files and is responsible for user management.

The *TruCrypt* trusted application is executed in the SW and performs encryption and decryption of file blocks. In addition, the trusted application also encrypts the AES keys used for file encryption, since these keys have to be stored in meta files in the NW. Furthermore, the *TruCrypt* trusted application creates the Merkle-Tree structure, when dealing with filesets.

Table 4.1 lists all available commands supported by *TruCrypt*. The commands are invoked by a user via the Android application *TruTalk* (see section 4.6) and are transmitted via Bluetooth to the *TruServer* (see section 4.4). The *TruServer* application then invokes the *TruCrypt* application with the appropriate arguments.

**Table 4.1:** Available commands for TruCrypt.

| *Command* | *Description* |
|---|---|
| `CREATE_USER` | Create a new user |
| `GET_PUB_KEY` | Export the user's public RSA key |
| `SET_PUB_KEY` | Import a public RSA key |
| `ENCRYPT_FILE` | Encrypt single file |
| `ENCRYPT_SET` | Encrypt set of files |
| `DECRYPT_FILE` | Decrypt single file |
| `DECRYPT_SET` | Decrypt fileset |
| `SHARE_FILE` | Share file/set with another user |

### 4.3.1 User Management

Before the *TruCrypt* application can be used for file encryption, a user has to be created by invoking TryCrypt with the `CREATE_USER` command. *TruStick* does not accept any other commands before a user has been set up. As argument a username and a password have to be passed when invoking the command. The *TruCrypt* application will then establish a connection to the SW trusted application and relay the command to the SW. There, the user credentials will be stored on the SW filesystem, alongside with a newly randomly generated RSA keypair bound to this specific user. The transmitted user credentials have to be provided along with every command issued to *TruStick* verify the user's identity.

At the moment *TruStick* does not support multiple users. If the *CREATE_USER* command is issued a second time, the credentials and RSA keypair of the first user will be deleted. Since the RSA private key is not stored elsewhere, this could result in loss of data, in case the key has already been used for encrypting files.

### 4.3.2 Public Key Transfer

When a user has been created, *TruCrypt* can export the user's RSA public key from the SW to the NW. Since the file encryption keys are encrypted with this RSA key, it is necessary to be able to export the local public key and import additional public keys to provide file sharing among multiple *TruStick* devices (see section 4.3.7).

When invoking the `GET_PUB_KEY` command, the trusted application reads the locally stored public RSA key and passes it to the NW. On the other side, the `SET_PUB_KEY` command is invoked with a public key and a username from another *TruStick* device. These two arguments are passed to the SW and are stored on the SW's filesystem.

### 4.3.3 Metafiles

The Data Encryption Key (DEK) and authentication tag belonging to a certain file are stored in a separate meta file. With this approach we can increase performance of modifying encrypted files, since operations like adding or releasing a key from a file can be performed without modifying the encrypted file itself. Basically, a meta file for a single encrypted file contains the encrypted DEK, including the signed authentication tag. If a file is shared with another user (see section 1.3.6) the new encrypted DEK is appended to the meta file. In this way, an arbitrary number of additional users can be added with minimal computational cost.

Since the DEK is only stored in the meta data file, if this file gets lost or corrupted, there is no way to restore the original file content.

Meta files are used for storing information related to the encrypted files and contain the encrypted data encryption key (DEK). Figure 4.4 and figure 4.5 show the fields contained in a meta file for single encrypted files and filesets. In case of a single encrypted file, the filename of the metafile is equal to the corresponding file with the ending *.mf* appended. For filesets, a user has to provide a name for the set on her smartphone when issuing the encryption command. This name is then used to denote the metafile, again with the appended *.mf* suffix.

| |
|---|
| File header |
| Username |
| Encrypted AES Data Encryption Key |
| Signed Authentication Tag |

**Figure 4.4:** Meta file content for a single file.

| |
|---|
| File header |
| Username |
| Encrypted AES Key Encryption Key |
| Filename A |
| Encrypted AES Data Encryption Key A |
| Filename B |
| Encrypted AES Data Encryption Key B |
| ... |
| Signed Merkle-Tree Root Hash |

**Figure 4.5:** Meta file content for a fileset.

- **File Header** Byte sequence to identify the file as metafile by *TruCrypt*. Also used for distinguishing between metafiles for single files or filesets.
- **Username** The username as provided to *TruStick* during authentication.
- **Encrypted AES Data Encryption Key** The AES key used for encrypting the file
- **Encrypted AES Key Encryption Key** The AES key used for encrypting data encryption keys in case of a fileset.
- **Signed Authentication Tag/Merkle-Tree Root Hash** The tag or root hash created after encryption used for verifying file integrity.

### 4.3.4 Filesets

Basically, the encryption process does not change, when dealing with a set (see figure 4.10). Each file is encrypted separately with it's own DEK. But instead of encrypting the DEKs with the KEK directly, a new AES KEK is generated and used for encrypting the DEKs. This newly created AES key is then encrypted using the KEK and stored in the meta file along with all other DEKs. Figure 4.6 illustrates this key hierarchy.

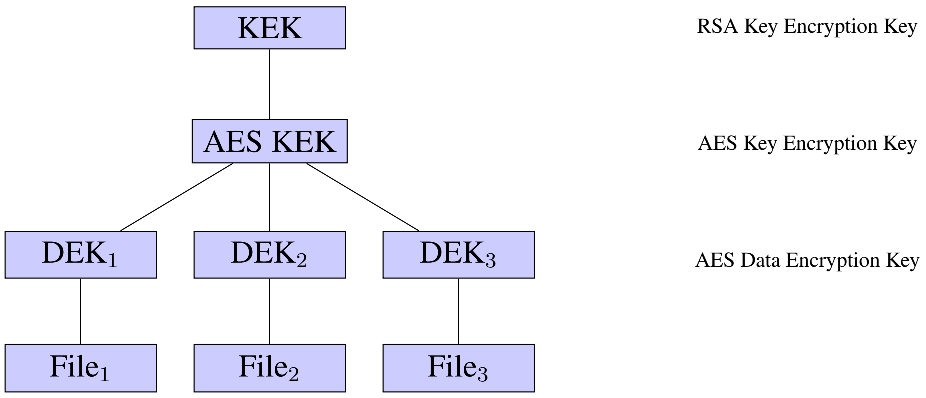


**Figure 4.6:** Key hierarchy of a fileset.

To verify the integrity of every file in a set a basic approach could be to apply a hash function to each file and store these hashes for example. Since this has to be done for each file every time a fileset is decrypted, such a solution would consume a lot of computation time for large files.

Since we use AES-OCB for file encryption, every encrypted file yields an own authentication tag after encryption. We decided to use a more efficient method and make use of the generated authentication tags by storing those tags in a dedicated data structure, known as Merkle-Tree. In our implementation, a Merkle-Tree builds a binary tree from the tags, yielding in one single value at the root node of the binary tree. The authentication tags are used as the tree's leaves. A node in the next tree level is created by applying a pre-image resistant SHA-256 hash function to two adjacent tags that are concatenated. For the next level, again the hashes of two nodes are concatenated and hashed. This procedure is repeated until the root of the tree is reached. The root hash is signed and stored in the meta file of a fileset. This allows an application to verify the root hash during decryption without the need for additional hash operations on the files themselves. Figure 4.7 shows a Merkle-Tree structure built of four authentications tags generated by encrypting four different files, resulting in two intermediate hashes from which the root hash is calculated.

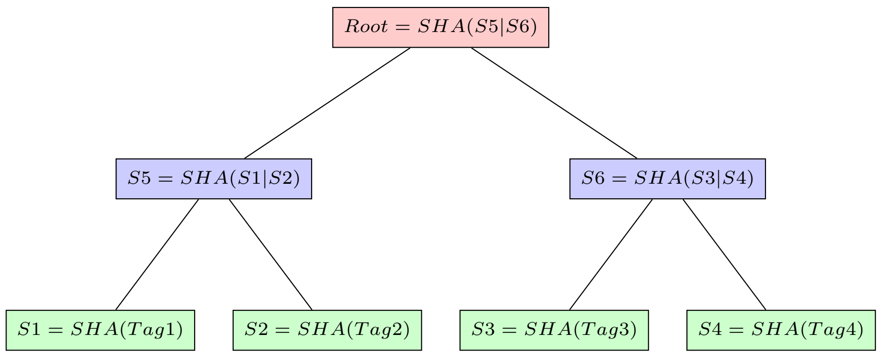


**Figure 4.7:** Merkle-Tree built from authentication tags.

### 4.3.5 Encryption

The encryption process of *TruStick* provides two different modes of operation. `ENCRYPT_FILE` triggers the encryption of one single files, each with a corresponding metafile (see section 4.3.3), containing the encryption key and other additional information used for decryption. Whereas, `ENCRYPT_SET` invokes the encryption of multiple files, with one metafile for all related files. Basically the encryption process is the same for single files and filesets. Therefore, we will discuss single file encryption first and, based on this discussion, the additional steps for fileset encryption afterwards. Figure 4.8 shows a sequence of operation diagram for the encryption process.

#### Single file encryption

`ENCRYPT_FILE` encrypts one or multiple files and creates an own metafile for each encrypted file. The filenames are passed as arguments along with the `ENCRYPT_FILE` command. The encryption process includes several steps to be performed:

1. The *TruCrypt* client application opens a session to the trusted application. During session creation the user's RSA keypair is loaded from the SW filesystem and stored in the session context.

2. After creating the session the client application invokes the generation of a new random 128-bit AES key and a 128-bit nonce in the SW. The key and the nonce are used for file encryption and are temporarily stored as a session parameters in the current session between the *TruCrypt* client application and the trusted application.

3. On successful key creation, the file to be encrypted is loaded from the NW filesystem in blocks of 1MB. These blocks are sequentially transferred to the SW where the trusted application encrypts them, using the previously generated AES key. When a file block has been encrypted, the encrypted block is transferred back to the NW and stored in the resulting output file.

4. After encryption of the last file block, AES-OCB yields the authentication tag used for verifying file integrity during decryption. This tag is stored temporarily within the *TruCrypt* session context.

5. In the next step, the trusted application encrypts the AES encryption key using the user's public RSA key and passes the encrypted key to the NW. When receiving the encrypted key, the client application creates the metafile for the encrypted file and stores the encrypted key in the metafile.
6. Finally, the trusted application signs the authentication tag with the user's RSA private key and sends it to the NW, where it is stored in the metafile as well.

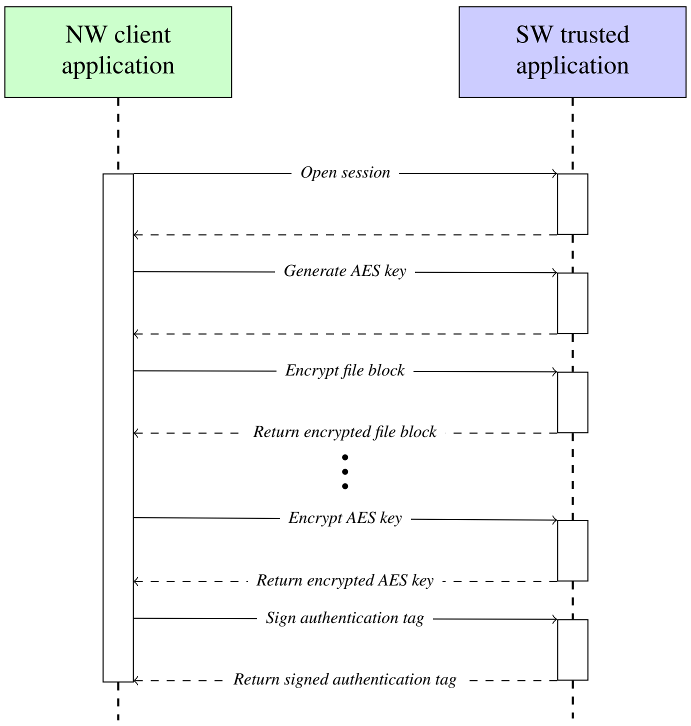
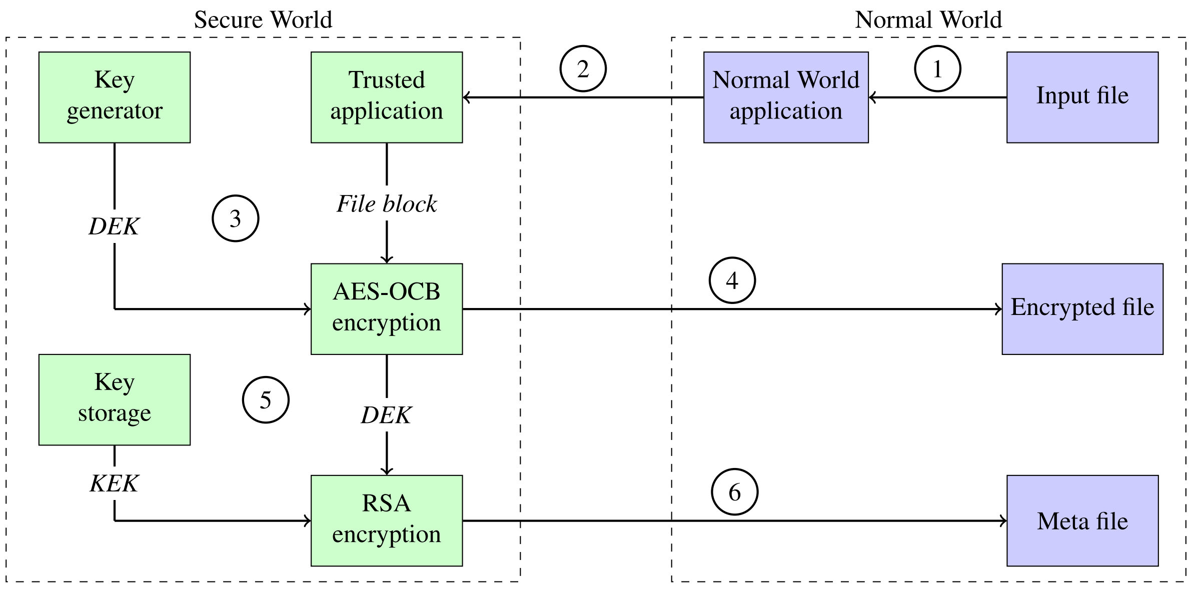


**Figure 4.8:** File encryption sequence of operation diagram

Figure 4.9 illustrates the encryption process as a block diagram, including the file transfer from NW to SW, the AES encryption using a newly generated DEK and the encryption of the DEK using the locally stored KEK. After successful encryption, the encrypted DEK is transferred back into the NW and stored in the associated meta file. Operations taking place in the SW of *TruStick* are denoted by green colored blocks. Normal world operations in contrary are represented by blue colored blocks.



(1) Sequentially read a file in blocks of 1MB

(2) Transfer the read block to SW

(3) Generate a random *DEK* and encrypt the file block using AES-OCB encryption

(4) Send the encrypted file block back to NW and store it in a file

(5) Load the *KEK* and encrypt the *DEK* using RSA encryption

(6) Store the encrypted *DEK* and the signed authentication tag in a meta file in the NW

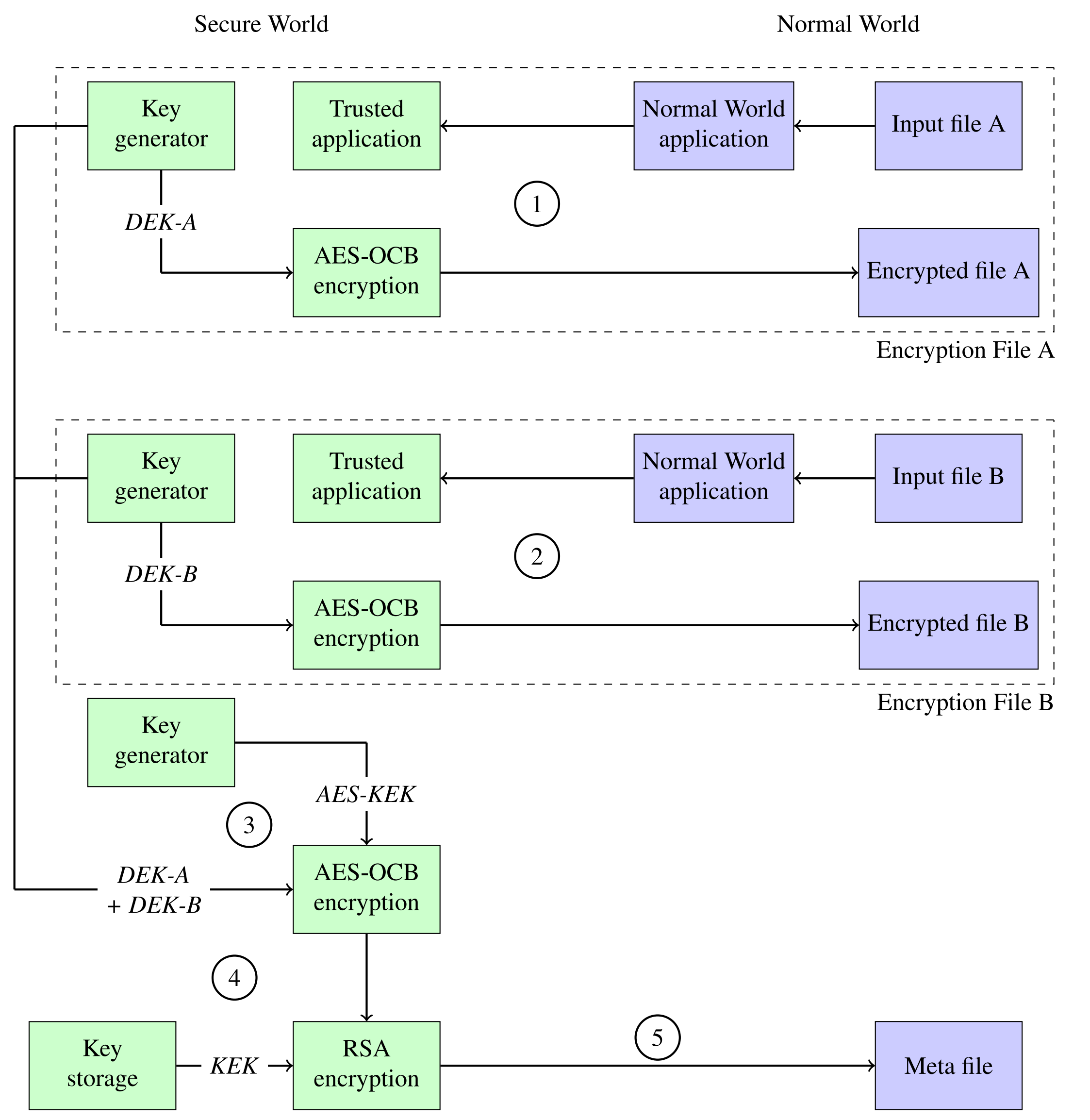
**Figure 4.9:** Single file encryption.

### Fileset encryption

In contrast to single file encryption, there are several differences when combining multiple files into a fileset:

1. For each file contained in the fileset, an own AES data encryption key is generated.
2. The AES data encryption keys are not directly encrypted with the user's RSA public key, but a new AES key encryption key is generated and used for encrypting the AES data encryption keys. This key encryption key is then encrypted with the RSA public key and stored in the metafile along with all encrypted file encryption keys.
3. The AES-OCB authentication tags are used for building a Merkle-Tree structure. This is done by respectively concatenating two authentication tags and applying a SHA-256 hash functions on them. Thus, the tags depict the leaves of a binary tree, and the resulting hashes the first level nodes. These nodes are now again concatenated, two at a time, and again hashed. The resulting hash values represent the second level of the tree. This procedure is repeated until there is only one single hash value left. This last hash represents the root of the tree. The root hash is signed by the trusted application, using the user's RSA private key, and is then transferred back to the NW, where it is stored in the metafile.

Figure 4.10 describes the encryption of a fileset containing two files as a block diagram. As a first step the two files are loaded into the SW and encrypted as already discussed in the first part of this section. Additionally, the two data encryption keys generated for each file are encrypted with a third generated AES key encryption key. This *AES-KEK* is then encrypted using the locally stored RSA key encryption key. The fileset's root hash and the encrypted *AES-KEK* are finally transferred to the NW and stored in the meta file belonging to this fileset. Again, green blocks denote SW operations and NW operations are colored blue.



① Single file encryption process for file A

② Single file encryption process for file B

③ Generate a new AES key and encrypt the DEKs for file A and B

④ Load the KEK and encrypt the AES-KEK using RSA encryption

⑤ Store the encrypted AES-KEK in the meta file alongside with the signed root hash of the Merkle-Tree, created from the authentication tags of the AES-OCB file encryption

**Figure 4.10:** Fileset encryption.

### 4.3.6 Decryption

Similar to file encryption there are two modes of operation for file decryption. `DECRYPT_FILE` is used for decrypting single encrypted files and `DECRYPT_SET` invokes decryption of a fileset. Both modes of decryption require the encrypted files and the corresponding metafiles as arguments. Again we will discuss single file decryption first and then discuss the differences when dealing with filesets. Figure 4.11 shows a sequence of operation diagram for the file decryption process.

#### Single file decryption

1. The *TruCrypt* client application starts the decryption process by reading the encrypted AES key from the metafile. After opening a session with the trusted application, the encrypted AES key is sent to the SW. The trusted application decrypts this key with the user's private RSA key, which has been loaded into the session context during session creation.

2. If the AES key has been successfully decrypted, the client application reads the encrypted file in blocks of 1MB. The read blocks are sequentially sent to the SW and decrypted by the trusted application. Afterwards, the decrypted blocks are transferred back to the NW and stored in the result file.

3. After decrypting the last block, the client application reads the signed authentication tag from the metafile and transfers it to the SW. The trusted application verifies the signature with the user's public RSA key and compares the received tag with the tag yielded from file decryption. If these two tags are equal, the integrity of the decrypted file has been verified.

#### Fileset decryption

Basically the decryption process for filesets is the same as for single files, besides the fact that there is only on single metafile for all files contained in the set and that the meta file contains an extra key encryption key used for decrypting the data encryption keys:

1. In the first step, the encrypted AES key encryption key is loaded from the meta file and decrypted in the SW and stored within the session context.

2. Afterwards,the encrypted data encryption keys are loaded and decrypted in the SW using the previously loaded key encryption key.

3. These keys are then used for decrypting the appropriate file.

4. After every encryption of a file, AES-OCB yields an authentication tag that is temporarily stored in the *TruCrypt*'s session context.

5. After all files have been decrypted the corresponding Merkle-Tree structure is created from all tags.

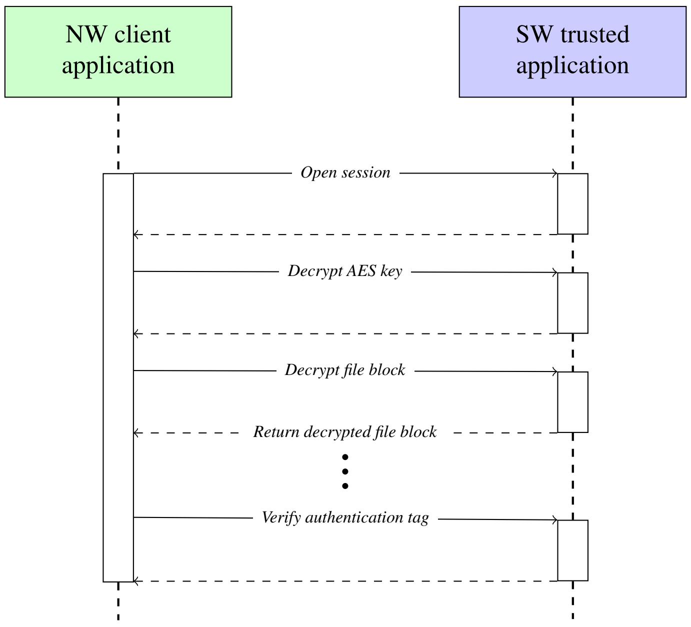


**Figure 4.11:** File decryption sequence of operation diagram

6. Afterwards, *TruCrpyt* loads the signed root hash from the metafile and verifies it in the SW using the user's public RSA key. If the root hash of the created tree is the same as the one loaded from the metafile, the integrity of the fileset has been verified.

### 4.3.7 File sharing

To share a file or a fileset the DEK located in the meta file is decrypted and re-encrypted with the new imported public key. This new encrypted key is finally stored in the meta file in addition to the file owner's key. After the meta file has been updated by the Dropbox client, a user owning the appropriate private part of a KEK can decrypt the shared files or sets using his or her *TruStick*. In this thesis we did not implement any mechanisms to automatically copy files between Dropbox accounts. This has to be done manually by the users.

If multiple users using *TruStick* want to share a file or fileset, *TruStick* is capable of exchanging these files without decrypting them before. Therefore, users have to exchange their public RSA keys via *TruTalk* (see section 4.6). Once a user has imported a public RSA key of another user, she can issue the `SHARE_FILE` command to *TruStick*. When receiving the `SHARE_FILE` command *TruCrypt* will load the meta file of the file or set to be shared, and read the AES data encryption key, when dealing with a single file or the AES key encryption key in case of a fileset. This key will then be encrypted with the previously imported public RSA key of the second user. This newly encrypted key is then added to the meta file and uploaded to the cloud server. Since the data files themselves are not modified in this process the signed authentication tags remain valid and the user receiving the file can verify file integrity after decryption.

#### Cancel file share

The approach of using meta files to store additional decryption keys also makes it easy to cancel a file share if it is no longer desired. Therefore, the additionally added decryption key has to be removed from the meta file. Since the decrypted data encryption keys are not stored anywhere, but in the meta file of a file or fileset, users will not be able to recover the DEK.

## 4.4 TruServer

*TruServer* provides the communication interface between the smartphone app *TruTalk* and *TruStick*. This includes receiving commands from *TruTalk* and relaying them forward to *TruCrypt*. Furthermore, *TruServer* reports available files for encryption and decryption to the user.

### 4.4.1 Communication

*TruServer* is started during the NW boot process. It launches a Bluetooth connection thread awaiting a Bluetooth connection request. To distinguish between a *TruTalk* connection request and a connection request sent by any other device, *TruServer* expects a unique id to be sent along with the request. If this unique id does not match, *TruServer* rejects the request. If a connection request is issued by another device via the *TruTalk* application, a shared secret - an AES encryption key - is established using the Diffie-Hellman key exchange protocol. If the shared secret has successfully been exchanged, *TruTalk* sends the user's username and password encrypted with the shared secret to *TruStick*. On *TruStick* the user credentials are forwarded to the SW via *TruCrypt*. If the user credentials could be verified, *TruServer* is ready to receive further commands from *TruTalk*.

### 4.4.2 File Monitoring

When files are transferred to the *TruStick* via USB while attached to a host device, *TruServer* monitors the dedicated partition used as USB backing storage. When files are added or removed from this storage, changes are reported to the *TruTalk* app via Bluetooth.

After successfully encrypting a file or fileset, the original unencrypted files are deleted, to prevent unauthorized access, in case an attacker manages to gain access to the device.

## 4.5 Dropbox Daemon

We have developed a simple Dropbox client using the Dropbox Developer API[Drob] to monitor and synchronize the USB backing store of *TruStick*. The *Dropbox Daemon* is started as background daemon on NW system start. Given that a network connection is available, the daemon periodically checks, if the local stored encrypted files are up to date with the files on the server and updates them in case any changes to the files are reported.

To connect the *Dropbox Daemon* to an existing Dropbox account, the daemon has to provide an app key, that has to be created via the Dropbox web interface, alongside with the user's access credentials. In our thesis, we do not support connection to multiple accounts, since the app key is stored within the source code of the daemon application.

## 4.6 TruTalk

We developed an Android application in order to control cryptographic operations and monitor file status on the *TruStick* via *TruServer*. Therefore, we implemented a thread providing Bluetooth communication capabilities and a QR- code reader used for exchanging public RSA keys between multiple *TruStick* devices. Section section 4.6.1 describes how communication between *TruTalk* and *TruServer* is established and how authentication is achieved. Section 4.6.2 gives an overview of issuing commands to *TruStick*. Section 4.6.3 explains how public RSA keys can be exchanged with other devices using QR-codes.

### 4.6.1 Communication

#### Initial setup

When *TruTalk* is launched for the first time, users are asked to choose a username and a password, that are stored locally on the device and have to be provided each time *TruTalk* is launched. After these access credentials have been set up, *TruTalk* starts to search for *TruStick* devices via Bluetooth. If a device is discovered, a Diffie-Hellman key exchange is initiated, yielding a common AES encryption key, that is known by *TruTalk*, as well as by *TruStick*. If the key exchange has been successful, *TruTalk* encrypts the local user credentials with the AES key and sends them to *TruStick* to register the user (see section 4.3.1).

This procedure is the most critical operation in the setup phase, since the user credentials are used to authenticate *TruTalk* to *TruStick* and are used for authorizing encryption and decryption operations. Hence, it is advisable to ensure, that no other Bluetooth devices are in range when the initial setup is done.

### 4.6.2 Commands

Figure 4.12 illustrates sequence of operations for issuing a command from *TruTalk* and execution on *TruStick*. After a Bluetooth connection has been established, a shared AES key is created using the Diffe-Hellman key exchange protocol. This secret key is used to encrypt any communication between *TruTalk* and *TruStick* and is kept until the Bluetooth connection is closed. The next time a connection is established, a new key has to be generated. Following commands are available on *TruTalk*:

- `CONNECT`: Establishes a Bluetooth connection to a *TruStick* device, including the Diffie-Hellman key exchange protocol for creating a shared secret used to encrypt any subsequent communication and transmitting the user credentials for authenticating to *TruStick*. On success, *TruStick* accepts file operation commands.
- `DISCONNECT`: Closes the Bluetooth connection and discards the shared secret. For further communication, the `CONNECT` command has to be resent.
- `ENCRYPT FILES`: Shows a dialog listing all unencrypted files located in the backing store of *TruStick*. Users can select one or multiple files, that will be encrypted and uploaded to the cloud. If multiple files are selected, each will be shown as single file in the decryption dialog.
- `DECRYPT FILES`: Shows a dialog listing all single encrypted files that have been downloaded by the *Dropbox Daemon*. Users can select one or multiple files, that will be decrypted and available to a USB host via the USB mass storage gadget.
- `ENCRYPT FILESET`: Encrypts multiple files that have been chosen from the provided list of available files, combining them to one set, that is represented to the user in the decryption dialog. When this command is selected, the user is asked to enter a name for the fileset.
- `DECRYPT FILESET`: Decrypts a fileset chosen from the file dialog. All files contained in the set will be available via the USB mass storage gadget.
- `GET PUBLIC KEY`: Requests the public RSA key from *TruStick* and stores it within the application context. After this has been done, the `SHARE KEY` command is available. Once the public key has been retrieved, it is permanently stored on the smartphone and can be shared at any time, without requiring the *TruStick* to be present.
- `SHARE KEY`: Converts the previously requested public RSA key into a QR-code that can be read by other *TruTalk* devices.

- `READ KEY`: Utilize the smartphone's camera to read a public RSA key encoded as QR-code and import it to the app storage. If the smartphone does not have a camera, this command is not available. If a connection to a *TruStick* device is present, when the key is decoded, the key is immediately sent to the *TruStick*. Else, the key is stored and transmitted as soon as a Bluetooth connection is available.

The corresponding commands triggered on *TruStick* are listed in table 4.1.

In addition to the commands listed in table 4.1 that are directly relayed to *TruCrypt*, *TruTalk* can implicitly send a `LIST_FILES` command when invoking any encryption or decryption command. This request will return two file lists. A list of all encrypted files, currently stored on *TruStick* available for decryption, and a list of all files located in the USB backing store, that can be encrypted.

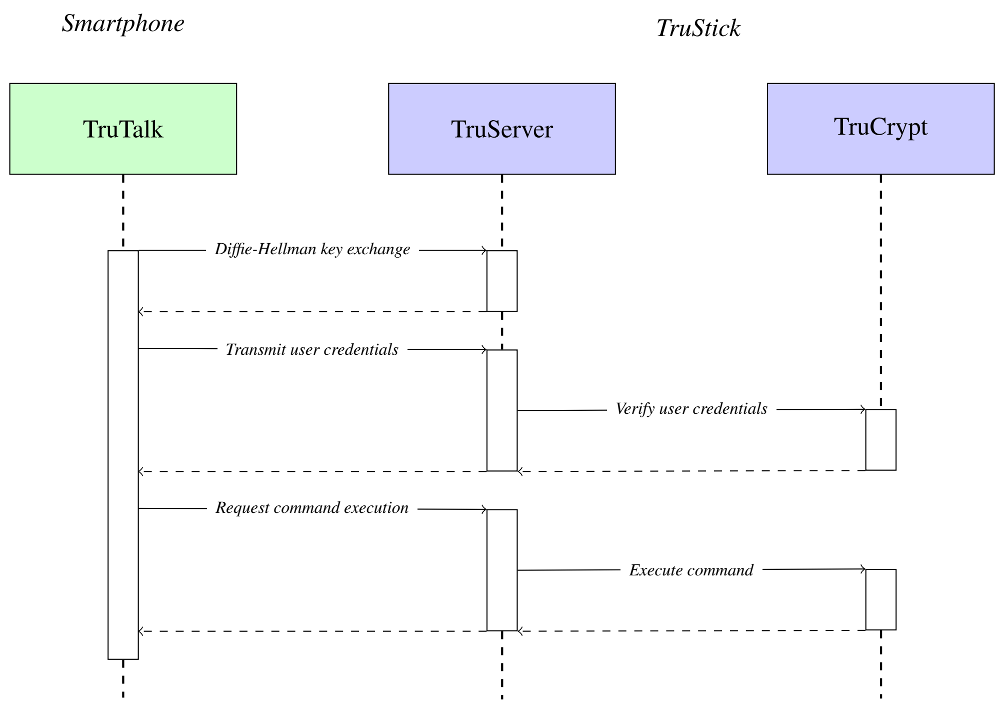


**Figure 4.12:** Authentication + command execution

### 4.6.3 Key Exchange

*TruStick* offers functionality to share encrypted files with other users. Therefore, users have to exchange their public RSA keys to be able to decrypt shared files. For this reason, we have implemented a QR-code generator within *TruTalk*, that converts a public RSA key byte array into a QR-code and displays it on the smartphone. Furthermore we also have implemented a QR-code reader using the smartphone's built-in camera, that is able to read RSA keys encoded as QR-code. To achieve QR-code functionality, we use the open source *ZXing (Zebra Crossing)*[ZXi] image processing library, that offers QR-code reading and creating capabilities.

# 5 Evaluation

IN this chapter we summarize the outcome of this thesis. Section 5.1 discusses the results and compares them to the initially stated goals. Section 5.2 states to which extent goals were not achieved and describes further problems that are still left open for future work.

## 5.1 Results

One primary goal of this thesis was to provide a device, capable of dealing with sensitive data stored in an untrusted environment, such as cloud storage. Moreover, we defined a major goal to be usability, since this is a major concern when dealing with security aspects. Security enhancing devices or software often face the problem, that they require established technical knowledge for correct handling. We reduced the required skills to the operation of a simple smartphone application. Using a smartphone application for controlling data handling on *TruStick* provides a widely accepted and well-understood manner of controlling a device.

Furthermore, we defined the goal of providing data integrity and confidentiality to be a fundamental part of *TruStick*. Therefore, we chose the symmetric cipher AES in OCB mode for data encryption and strong asymmetric RSA cipher for encrypting the AES keys. The combination of those two ciphers yields an efficient way of dealing with data of arbitrary size. Moreover, AES-OCB provides data confidentiality and integrity at the same time by creating an additional tag for each encryption process that is used to verify data integrity during decryption. Since we implemented *TruStick* on an embedded device, performance of cryptographic operations also played an important role in the decision which encryption scheme to use. Table 5.1 shows the results of performing encryption and decryption operations using AES in OCB mode on *TruStick*. We also implemented encryption and decryption using AES-CCM in order to compare the performance of AES-OCB. The evaluation of both encryption schemes shows that AES-OCB can be considered approximately twice as fast as AES-CCM.

**Table 5.1:** Encryption and decryption results with AES-OCB.

| *Bytes* | *Encryption[ms]* | *Decryption[ms]* |
|---|---|---|
| `4 KB` | 19 | 10 |
| `8 KB` | 29 | 11 |
| `1 MB` | 1584 | 853 |
| `2 MB` | 3618 | 1832 |
| `10 MB` | 18813 | 9526 |

**Table 5.2:** Encryption and decryption results with AES-CCM.

| *Bytes* | *Encryption[ms]* | *Decryption[ms]* |
|---|---|---|
| `4 KB` | 25 | 29 |
| `8 KB` | 38 | 37 |
| `1 MB` | 3598 | 3115 |
| `2 MB` | 7180 | 6089 |
| `10 MB` | 35928 | 30445 |

Our third major goal targets scalability. By introducing filesets that can hold an arbitrary number of files without the need of dealing with each file separately during encryption or decryption, our approach is also applicable to vast projects containing many files without loosing the comfort of issuing a single command to process all files or a subset of them at once.

## 5.2 Open Problems

Even though we were able to fulfill our major goals, there are still several issues left uncovered in this thesis.

Control commands issued to *TruStick* by *TruTalk* are secured using AES encryption with a newly generated key exchanged via Diffie-Helman key exchange. Nevertheless, an attacker could intercept encrypted messages and resend them, performing a replay attack. An applicable countermeasure to such attacks could include tagging each message with a session id and a unique number, invalidating a message after being transmitted once.

Another issue concerns our implementation of a simple client, used for synchronizing data stored on *TruStick* with Dropbox. Our client does not include a mechanism of sharing files with other users and managing permissions each user has. Currently this has to be done manually by a user via the web interface of Dropbox. If a user is granted write access to files shared by another user, he or she could manipulate the meta file, deleting the file owner's data encryption key, preventing the owner from being able to decrypt his own file.

Finally, currently there is no mechanism to notify users sharing content, that there have been changes on the server. Users have to download and decrypt files manually to determine whether the content has changed. A system, such as distributed hash tables, as used by BitTorrent [Inc] could be used to circumvent this process by informing a user via push notifications on the smartphone every time something has changed.

# 6 Conclusions

THE goal of this thesis was to implement a device providing a secure and easy-to-use way to guarantee data integrity when dealing with data stored on a cloud server. We managed, to create a device emulating USB mass storage for transferring data between *TruStick* and arbitrary USB host systems using the Freescale IMX53 Quick Start Board. The *TruStick* utilizes ARM TrustZone technology, executing two operating systems on a single physical core. We use the secure world operating system Andix OS for securely storing cryptographic key material and, hence, for executing cryptographic operations on data transferred from the normal world. Thereby, the cryptographic keys are stored safely in the secure world, which also monitors the system's state and takes countermeasures, if possible malicious actions are detected.

We designed a file structure including a meta file for storing encrypted data and all relevant information required to decrypt and also share files without the need to explicitly decrypt the files before. The use of meta files for storing additional encrypted keys limits the operations to be executed in order to share files among users to decrypting and re-encrypting the used encryption key, without touching the data file itself.

In addition, *TruStick* offers the ability to group commonly used files together to filesets. With this approach users can easily manage a large number of files without the need to process each file on its own. Since an extra key encryption key is used for encrypting the data encryption key, it is sufficient to re-encrypt this one key, if a fileset is to be shared. Thus, we have implemented an efficient way of sharing encrypted files, also requiring only one decryption and one encryption operation.

Furthermore, we have implemented two communication channels for *TruStick*. Firstly, a Dropbox client application responsible for synchronizing locally encrypted files with the appropriate cloud storage. Secondly an Android application used for issuing file operation commands to *TruStick* and report the current file status to a user.

Though, the goal to implement a notification mechanism, notifying users about file changes in the cloud, using distributed hash tables, has been skipped.

Nevertheless, this thesis shows the successful proof of concept implementation of a device, using client-side encryption to provide device independent cloud storage.

# Bibliography

[Aki+11] Joseph A. Akinyele, Matthew W. Pagano, Matthew D. Green, Christoph U. Lehmann, Zachary N.J. Peterson, and Aviel D. Rubin. "Securing Electronic Medical Records Using Attribute-based Encryption on Mobile Devices". In: *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '11. Chicago, Illinois, USA: ACM, 2011, pp. 75–86. ISBN 978-1-4503-1000-0. doi:10.1145/2046614.2046628. `http://doi.acm.org/10.1145/2046614.2046628` (cit. on p. 22).

[Ama] Amazon.com, Inc. *Amazon Cloud Drive*. `http://www.amazon.de/gp/feature.html?ie=UTF8&docId=1000655923` (Retrieved 04/05/2017) (cit. on p. 1).

[BAL96] PGP BAL's. "Public Key Server". In: *The Computer Law Resource* (1996), pp. 1–2 (cit. on p. 3).

[Bec08] Georg Becker. "Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis". In: (2008) (cit. on p. 16).

[Bes+13] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds". In: *Trans. Storage* 9.4 (Nov. 2013), 12:1–12:33. ISSN 1553-3077. doi:10.1145/2535929. `http://doi.acm.org/10.1145/2535929` (cit. on p. 23).

[Cona] Conrad Electronic GmbH & Co KG. *Bluetooth®-Stick 4.0 Conrad (Raspberry Pi® kompatibel)*. `http://www.conrad.at/ce/de/product/448876/Bluetooth-Stick-40-Conrad-Raspberry-Pi-kompatibel` (Retrieved 04/05/2017) (cit. on p. 9).

[Conb] Software Freedom Conservancy. *git –fast-version-control*. `https://git-scm.com` (Retrieved 03/05/2017) (cit. on p. 16).

[Dav99] Derek L Davis. *Secure boot*. US Patent 5,937,063. Aug. 1999 (cit. on p. 3).

[Des11] Yvo Desmedt. “Man-in-the-middle attack”. In: *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 759–759 (cit. on p. 4).

[DF03] R. Housley D. Whiting and N. Ferguson. “Counter with CBC-MAC (CCM)”. In: (Sept. 2003). `http://www.rfc-editor.org/info/rfc3610` (cit. on p. 6).

[DH06] W. Diffie and M. Hellman. “New Directions in Cryptography”. In: *IEEE Trans. Inf. Theor.* 22.6 (Sept. 2006), pp. 644–654. ISSN 0018-9448. doi:10.1109/TIT.1976.1055638. `http://dx.doi.org/10.1109/TIT.1976.1055638` (cit. on p. 12).

[Droa] Dropbox, Inc. *Dropbox*. `https://www.dropbox.com` (Retrieved 04/05/2017) (cit. on p. 1).

[Drob] Dropbox, Inc. *Dropbox Developer API*. `https://www.dropbox.com/developers` (Retrieved 04/05/2017) (cit. on pp. 7, 40).

[Droc] Dropbox, Inc. *How secure is Dropbox?* `https://www.dropbox.com/help/27` (Retrieved 04/05/2017) (cit. on p. 1).

[eMa] eMarketer. *Smartphone Users Worldwide Will Total 1.75 Billion in 2014*. `http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536` (Retrieved 04/05/2017) (cit. on p. 7).

[Fen+13a] Wei Feng, Dengguo Feng, Ge Wei, Yu Qin, Qianying Zhang, and Dexian Chang. “TEEM: A User-Oriented Trusted Mobile Device for Multi-platform Security Applications”. In: *Trust and Trustworthy Computing*. Ed. by Michael Huth, N. Asokan, Srdjan Čapkun, Ivan Flechais, and Lizzie Coles-Kemp. Vol. 7904. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 133–141. ISBN 978-3-642-38907-8. doi:10.1007/978-3-642-38908-5_10. `http://dx.doi.org/10.1007/978-3-642-38908-5_10` (cit. on p. 23).

[Fen+13b] Wei Feng, Yu Qin, Dengguo Feng, Ge Wei, Lihui Xue, and Dexian Chang. “Mobile Trusted Agent (MTA): Build User-Based Trust for General-Purpose Computer Platform”. In: *Network and System Security*. Ed. by Javier Lopez, Xinyi Huang, and Ravi Sandhu. Vol. 7873. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 307–320. ISBN 978-3-642-38630-5. doi:10.1007/978-3-642-38631-2_23. `http://dx.doi.org/10.1007/978-3-642-38631-2_23` (cit. on p. 22).

[Fiz14] Andreas Fizek. “Development of an ARM TrustZone aware operating system ANDIX OS”. MA thesis. Graz University of Technology, 2014 (cit. on p. 18).

[GFI] GFI Blog. *Report: Most vulnerable operating systems and applications in 2013*. `http://www.gfi.com/blog/report-most-vulnerable-operating-systems-and-applications-in-2013` (Retrieved 04/05/2017) (cit. on p. 2).

[Gloa] GlobalPlatform. *GlobalPlatform made simple guide: Trusted Execution Environment (TEE) Guide*. `http://www.globalplatform.org/mediaguidetee.asp` (Retrieved 04/05/2017) (cit. on p. 6).

[Glob] GlobalPlatform. *TEE Client API Specification*. `http://www.globalplatform.org/specificationsdevice.asp` (Retrieved 04/05/2017) (cit. on pp. 18, 19).

[Gloc] GlobalPlatform. *TEE Internal API Specification*. `http://www.globalplatform.org/specificationsdevice.asp` (Retrieved 04/05/2017) (cit. on p. 19).

[Gooa] Google, Inc. *Android Developers - Storage Options*. `http://developer.android.com/guide/topics/data/data-storage.html` (Retrieved 04/05/2017) (cit. on p. 8).

[Goob] Google, Inc. *Chromium OS - The Chromium Projects*. `http://www.chromium.org/chromium-os` (Retrieved 04/05/2017) (cit. on p. 1).

[Gooc] Google, Inc. *Google Data Centers*. `http://www.google.com/about/datacenters/inside/locations` (Retrieved 04/05/2017) (cit. on p. 1).

[Hei+12] Daniel M. Hein, Ronald Toegl, Martin Pirker, Emil Gatial, Zoltán Balogh, Hans Brandl, and Ladislav Hluchý. "Securing Mobile Agents for Crisis Management Support". In: *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*. STC '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 85–90. ISBN 978-1-4503-1662-0. doi:10.1145/2382536.2382550. `http://doi.acm.org/10.1145/2382536.2382550` (cit. on p. 23).

[Inc] BitTorrent Inc. *The Original BitTorrent Client*. `http://www.bittorrent.com` (Retrieved 03/05/2017) (cit. on pp. 16, 47).

[KL04] Ho Won Kim and Sunggu Lee. "Design and implementation of a private and public key crypto processor and its application to a security system". In: *Consumer Electronics, IEEE Transactions on* 50.1 (Feb. 2004), pp. 214–224. ISSN 0098-3063. doi:10.1109/TCE.2004.1277865 (cit. on p. 3).

[Kla04] Jeff Klaas. *System-on-a-chip*. US Patent 6,816,750. Nov. 2004 (cit. on p. 9).

[KR] Ted Krovetz and Phillip Rogaway. *The ocb authenticated-encryption algorithm*. `http://tools.ietf.org/html/rfc7253` (Retrieved 04/05/2017) (cit. on pp. 6, 13, 15).

[Li+14] Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Prateek Saxena. "DroidVault: A Trusted Data Vault for Android Devices". In: (2014) (cit. on p. 21).

[Lim09] ARM Limited. "ARM Security Technology - Building a Secure System using TrustZone Technology". In: (2009) (cit. on p. 17).

[Lin] Linux USB Project. *Linux-USB Gadget API Framework*. http://www.linux-usb.org/gadget (Retrieved 04/05/2017) (cit. on p. 9).

[McC+07] Jonathan M. McCune, Adrian Perrig, Arvind Seshadri, and Leendert van Doorn. "*Turtles All The Way Down: Research Challenges in User-Based Attestation*". In: *Workshop on Hot Topics in Security (HotSec)*. http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA520948. 2007 (cit. on p. 22).

[Mor11] Thomas Morris. "Trusted Platform Module". In: *Encyclopedia of Cryptography and Security* (2011), pp. 1332–1335 (cit. on p. 5).

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN 0849385237 (cit. on p. 12).

[NIS] NIST - National Institue of Standards and Technology. *National Vulnerability Database*. https://nvd.nist.gov (Retrieved 04/05/2017) (cit. on p. 2).

[NXP] NXP Semiconductors. *i.MX53 Applications Processors: Advanced Multimedia, ARM® Cortex®-A8 Core*. http://www.nxp.com/products/automotive-products/microcontrollers-and-processors/arm-mcus-and-mpus/i.mx-application-processors/i.mx53-applications-processors-advanced-multimedia-arm-cortex-a8-core:IMX53_FAMILY (Retrieved 04/05/2017) (cit. on p. 9).

[Pop+11] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. "Enabling security in cloud storage SLAs with CloudProof". In: *USENIX Annual Technical Conference*. 2011 (cit. on p. 23).

[Roga] Phillip Rogaway. *Authenticated-Encryption Software Performance: Comparison of CCM, GCM, and OCB*. http://web.cs.ucdavis.edu/~rogaway/ocb/performance (Retrieved 04/05/2017) (cit. on p. 6).

[Rogb] Phillip Rogaway. *OCB: Background*. http://web.cs.ucdavis.edu/~rogaway/ocb/ocb-back.htm (Retrieved 04/05/2017) (cit. on p. 14).

[RSA] RSA. *SecurID – On-Demand Authenticator*. http://www.emc.com/security/rsa-securid.htm (cit. on p. 23).

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems". In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN 0001-0782. doi:10.1145/359340.359342. http://doi.acm.org/10.1145/359340.359342 (cit. on p. 11).

[Sai+04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. "Design and Implementation of a TCG-based Integrity Measurement Architecture." In: *USENIX Security Symposium*. Vol. 13. 2004, pp. 223–238 (cit. on p. 3).

[Sav] John J. G. Savard. *A Cryptographic Compendium - Description of SHA-1 and SHA-256*. `http://www.quadibloc.com/crypto/mi060501.htm` (Retrieved 04/05/2017) (cit. on p. 8).

[Sec] Secomba GmbH. *Boxcryptor - Encryption for cloud storage*. `https://www.boxcryptor.com` (Retrieved 04/05/2017) (cit. on p. 2).

[Shi+12] Jaebok Shin, Yungu Kim, Wooram Park, and Chanik Park. "DFCloud: A TPM-based secure data access control method of cloud storage in mobile devices". In: *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. Dec. 2012, pp. 551–556. doi:10.1109/CloudCom.2012.6427606 (cit. on p. 21).

[Sus11] Laurent Sustek. "Hardware Security Module". In: *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 535–538 (cit. on p. 3).

[TBB03] Mark Turner, David Budgen, and Pearl Brereton. "Turning software into a service." In: *Computer.* 36.10 (2003), pp. 38–44 (cit. on p. 1).

[The] The Washington Post. *Leaks of nude celebrity photos raise concerns about security of the cloud*. `http://www.washingtonpost.com/politics/leaks-of-nude-celebrity-photos-raise-concerns-about-security-of-the-cloud/2014/09/01/59dcd37e-3219-11e4-8f02-03c644b2d7d0_story.html` (Retrieved 04/05/2017) (cit. on p. 3).

[Uni] United States, Department of Justice. *USA Patriot Act*. `http://www.justice.gov/archive/ll/highlights.htm` (Retrieved 04/05/2017) (cit. on p. 1).

[Wei+08] Thomas Weigold, Thorsten Kramp, Reto Hermann, Frank Höring, Peter Buhler, and Michael Baentsch. "The Zurich Trusted Information Channel – An Efficient Defence Against Man-in-the-Middle and Malicious Software Attacks". In: *Trusted Computing - Challenges and Applications*. Ed. by Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch. Vol. 4968. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 75–91. ISBN 978-3-540-68978-2. doi:10.1007/978-3-540-68979-9_6. `http://dx.doi.org/10.1007/978-3-540-68979-9_6` (cit. on p. 22).

[Win+12] Johannes Winter, Paul Wiegele, Martin Pirker, and Ronald Tögl. "A Flexible Software Development and Emulation Framework for ARM TrustZone". English. In: *Trusted Systems*. Ed. by Liqun Chen, Moti Yung, and Liehuang Zhu. Vol. 7222. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 1–15. ISBN 978-3-642-32297-6. doi:10.1007/978-3-642-32298-3_1. `http://dx.doi.org/10.1007/978-3-642-32298-3_1` (cit. on p. 18).

[WIR] WIRED.com. *Report: NSA Is Intercepting Traffic From Yahoo, Google Data Centers*. `http://www.wired.com/2013/10/nsa-hacked-yahoo-google-cables` (Retrieved 04/05/2017) (cit. on p. 2).

[ZHY10] Dawei Zhang, Zhen Han, and Guangwen Yan. “A Portable TPM Based on USB Key”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS ’10. Chicago, Illinois, USA: ACM, 2010, pp. 750–752. ISBN 978-1-4503-0245-6. doi:10.1145/1866307.1866419. http://doi.acm.org/10.1145/1866307.1866419 (cit. on p. 23).

[ZXi] ZXing Project. *ZXing, open-source image processing library*. https://github.com/zxing (Retrieved 04/05/2017) (cit. on p. 43).