



TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Konzeptionierung einer Backuplösung für mobile cryptocurrency wallets durch Verteilen des private keys auf mehrere Endgeräte

Masterarbeit

vorgelegt von: Stefan Haßferter
Studiengang: Master für Elektronische und Mechatronische Systeme
Matrikelnummer: 2588310
Erstprüfer: Prof. Dr. Oliver Hofmann
Zweitprüfer: Prof. Dr. rer. nat., Dipl.-Phys. Thomas Mahr
Betreuer: Dipl.-Inf. Steffen Blümm
Abgabedatum: 29. Januar 2019

WiSe 2018/2019

Dieses Werk einschließlich seiner Teile ist **urheberrechtlich geschützt**. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen sowie die Einspeicherung und Verarbeitung in elektronischen Systemen.

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name:

Vorname:

Matrikel-Nr.:

Fakultät:

Studiengang:

Semester:

Titel der Abschlussarbeit:

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,
 genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format auf einem Datenträger beigefügt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Ort, Datum, Unterschrift Studierende/Studierender



Inhaltszusammenfassung

Die Arbeit befasst sich mit der Konzeptionierung und Implementierung einer Backupmethode für *private keys* von Kryptowährungen. Diese werden meistens in einem Wallet gespeichert und stellen die Zugangsdaten für das auf der Blockchain gespeicherte Vermögen dar. Ein Wallet ist ein Programm, das stark einer Online-Banking-App ähnelt. Genau wie diese wird es oftmals auf dem Smartphone des Users installiert. Ist der Zugriff auf den Schlüssel durch Verlust oder Zerstörung des Geräts nicht möglich, kann gleichsam über gespeicherte Vermögen nicht mehr verfügt werden. Da die Wahrscheinlichkeit, den Schlüssel durch Erraten wiederherzustellen gegen Null geht, ist es beinahe unmöglich, den Zugang wiederzuerlangen. Aus diesem Grund sollten geeignete Maßnahmen zur Sicherung und Wiederherstellung des Schlüssels ergriffen werden. Diese sollen dem Benutzer durch die in dieser Arbeit beschriebenen Applikation sicher und komfortabel zu Verfügung gestellt werden. Der Grundgedanke besteht darin, den Schlüssel in Teile zu zerlegen und auf Geräten von Vertrauenspersonen zu speichern. Zur Wiederherstellung muss eine Untermenge dieser Teile kombiniert werden. Es werden aktuell verbreitete Verschlüsselungs- und Übertragungsmechanismen beschrieben, ebenso wie gebräuchliche Methoden, um ein Backup zu erstellen. Diese umfassen unter anderem das Rivest Shamir Adleman Kryptosystem (RSA) sowie die direkten und indirekten Kommunikationswege. Zusätzlich werden die Anforderung an ein Programm dieser Art hinsichtlich Sicherheit und Benutzerfreundlichkeit herausgearbeitet. Darauf aufbauend schließt sich die Auswahl der Komponenten und die Implementierung dieser an. Ein weiterer zentraler Punkt ist die Funktionsweise des Shamir's Secret Sharing Scheme (SSSS)-Algorithmus und die Anwendung auf die Erstellung von Sicherungskopien. Des Weiteren werden mögliche Programmerweiterungen dargelegt.



Abstract

This thesis deals with the conception and implementation of a backup method for crypto currency *private keys*. These represent the login data for the assets stored on the blockchain and are usually stored in a wallet, which is a program very similar to an online banking app. Just like this, it is often installed on the users' smartphone. If access to the key becomes unavailable due to loss or destruction of the device, stored assets can no longer be disposed of. Since the probability of recovering the key by guessing is close to zero, it is almost impossible to regain access. For this reason, appropriate measures should be taken to back up and ensure recovery of the key. It is targeted to make these securely and conveniently available to the user by the application described in this paper. The basic idea is to split the key into parts and store them on the devices of trusted third parties. For recovery a subset of these parts must be combined. It is described which methods are used to create a backup and which encryption and communication mechanisms are currently used. These include, among others, the RSA encryption method and the direct and indirect communication channels. In addition, the requirements regarding security and usability for a program of this kind are elaborated. Based on this, suitable components are selected and implemented. A further central point is the functional description of the Shamir's Secret Sharing Scheme (SSSS) algorithm and its application to the creation of backup copies. Moreover, possible program extensions are presented.



Inhaltsverzeichnis

Abkürzungsverzeichnis	III
Abbildungsverzeichnis	IV
Quellcodeverzeichnis	V
1 Einleitung	1
1.1 Ausgangssituation	1
1.2 Ziel der Arbeit	2
1.3 Veranschaulichung der zentralen Problemstellung	2
1.4 Aufbau der Arbeit	3
1.5 Projektträger	3
2 Herausforderungen und Problembetrachtung	5
2.1 Aktuelle Backupmethoden	5
2.2 Secret Sharing	9
2.3 Rivest Shamir Adleman Kryptosystem (RSA)	10
2.4 Verwendete Kommunikationsmethoden	13
3 Shamir's Secret Sharing Scheme (SSSS)	15
3.1 Grundlegende Funktionsweise	15
3.2 Sicherheitsbedenken	18
3.3 Lösung des Sicherheitsproblems	18
4 Konzeptionierung und Anforderungsanalyse	23
4.1 Programmablauf	23
4.1.1 Erstellung eines Backups	24
4.1.2 Rekonstruktion mithilfe eines Backups	25
4.2 Speicherung der Daten	26
4.2.1 Datenbankstruktur	28
4.2.2 iCloud Metadaten	29
4.3 Kommunikation zwischen den Geräten	30
4.3.1 Formen der Datenübertragung	30
4.3.2 Übertragung der Kommunikationsschlüssel	32



Inhaltsverzeichnis

4.4	Sicherheitsaspekte	34
4.5	Mögliche Funktionserweiterungen	35
5	Implementierung	37
5.1	Verwendete Komponenten	37
5.1.1	Realm Database	37
5.1.2	SSSS in Swift	40
5.1.3	MultipeerConnectivity	41
5.1.4	iCloud Drive Synchronisation	44
5.2	Verschlüsselung in Swift	47
5.3	Import und Export von Dateien	51
6	User Interface	56
6.1	Key Management	56
6.2	Vertrauenspersonen	58
6.3	Persönliche Informationen	59
7	Fazit und Ausblick	61
7.1	Fazit	61
7.2	Ausblick	62
	Literaturverzeichnis	63



Abkürzungsverzeichnis

.abf	Adorsys Backup File
AES	Advanced Encryption Standard
BIP	Bitcoin Improvement Proposals
CD	Compact Disc
DARPA	Defense Advanced Research Projects Agency
GCHQ	Government Communication Headquarters
GCM	Galois/Counter Mode
ICO	Initial Coin Offerings
ID	Identifikationsnummer
JSON	JavaScript Object Notation
OAEP	Optimal Asymmetric Encryption Padding
P2P	Peer to Peer Communication
QR-Code	Quick Response Code
RSA	Rivest Shamir Adleman Kryptosystem
SSSS	Shamir's Secret Sharing Scheme
UI	User Interface
UTI	Uniform Type Identifier



Abbildungsverzeichnis

1.1	Logo der <i>adorsys GmbH & Co. KG</i>	4
2.1	Übersicht über Backupmethoden von Kryptowährungsschlüsseln (Hardware Wallet (1), Textform(2), Text in Schlüsselcontainer(3)	7
2.2	Beispielhafte Teile eines <i>mnemonic sentence</i> in einem iOS-Wallet	8
2.3	Schaubild zur beispielhaften Übertragung einer Nachricht vom Sender zum Empfänger	11
3.1	Beispielhafte Kurve eines Polynoms in einem endlichen Körper	20
4.1	Strukturdiagramm der Erstellung eines Backups	24
4.2	Strukturdiagramm der Wiederherstellung eines Backups	26
4.3	Datenbankstruktur	28
4.4	Beispielhafter iOS Exportbildschirm einer .abf-Datei	31
4.5	Kommunikationsschlüssel als Quick Response Code (QR-Code)	32
4.6	Möglichkeit zum direkten Import einer Bilddatei in das Backupprogramm	33
4.7	Möglichkeit zum Scannen eines QR-Codes innerhalb der App mithilfe der Smartphonekamera	34
4.8	Authentifizierungsaufforderung mit <i>TouchID</i>	35
5.1	Beispiel der verschiedenen Status der <i>MultipeerConnection</i> : (a) Gerätesuchvorgang, (b) Verbindungsbestätigung und (c) <i>Ready-To-Send-State</i>	43
5.2	Beispielhafter Inhalt von iCloud Drive mit dem erstellten <i>p2pCryptoContainer</i>	45
5.3	Definition der .abf-Datei als Uniform Type Identifier (UTI) in Xcode	51
6.1	Icon für die Backupapplikation	56
6.2	Beispielhafter <i>Key Management Screen</i> mit diversen Keys	58
6.3	Beispielhafte Auflistung von Vertrauenspersonen	59
6.4	Persönliche Informationen	60



Quellcodeverzeichnis

3.1	Programmcode zum Berechnen des inversen Elements einer rationalen Zahl in einem endlichen Körper	21
5.1	Definition der <i>Buddy</i> Klasse in Realm	38
5.2	Definition der <i>KeyPart</i> Klasse in Realm	38
5.3	Definition der <i>KeyPartContainer</i> Klasse in Realm	39
5.4	Lesezugriff zum Erhalt aller <i>Buddies</i> -Instanzen	39
5.5	Verschiedene Möglichkeiten des Realm Schreibzugriffs	39
5.6	Beispiele für einen gefilterten und sortierten Datenbankzugriff	40
5.7	Erstellung der Schlüsselteile und Speicherung in der Datenbank in den vorge-sehenen Objekten	40
5.8	Rekonstruktion des Geheimnisses aus den importierten Schlüsselteilen	41
5.9	Initialisierung der benötigten Verbindungsparameter	42
5.10	Programmcode zum Initialisieren einer Session bzw. Aufrufen des Auswahl-fensters	42
5.11	Programmcode zum Senden und Empfangen von Daten	43
5.12	Beispielhafter Inhalt der <i>Buddies</i> Datei	45
5.13	Programmcode für einen Schreibzugriff auf den iCloud Ordner	46
5.14	Programmablauf zum Lesen von Daten aus einer Datei in iCloud Drive	47
5.15	Darstellung einer .abf Datei im Klartext	48
5.16	Ausschnitt einer verschlüsselten .abf Datei	49
5.17	Parametersätze zum Erstellen und Laden des Kommunikationsschlüssels	49
5.18	Erstellen und Laden des <i>private keys</i> aus der <i>KeyChain</i> und anschließendes Generieren des <i>public keys</i>	50
5.19	Ver- und Entschlüsselungsfunktion	50
5.20	Aufruf des iOS-Exportfensters	52
5.21	Programmcode zur Verarbeitung von importierten Dateien	53
5.22	Zugriff auf die Smartphonecamera zum Scannen eines QR-Codes	54
5.23	Callback Funktion zur Verarbeitung der erkannten Metadaten	54



1 Einleitung

1.1 Ausgangssituation

Kryptowährungen und allen voran deren berühmtester Vertreter, der Bitcoin, erfreuten sich im ersten Quartal 2018 eines sehr starken Zulaufs, eines regelrechten Booms. Dieser ist inzwischen zwar abgeflaut, jedoch ist die Bereitschaft, in Kryptowährungen zu investieren, ungebrochen. In der ersten Jahreshälfte 2018 wurde über Initial Coin Offerings (ICO) ein Kapital von 13,7 Milliarden Dollar generiert. [Zulauf, 2018] Der größte Unterschied zwischen den herkömmlichen zentralisierten Währungen und den dezentralisierten Kryptowährungen auf Basis der Blockchain-Technologie ist, dass der Nutzer sich ausschließlich mit seinem privaten Schlüssel authentifiziert. Dazu müssen keine personenbezogenen Daten bei einer zentralen Stelle hinterlegt werden, wie beispielsweise bei einem herkömmlichen Bankkonto. Nur über den *private key* hat der Nutzer Zugang zu seinem gespeicherten virtuellen Vermögen. Denn da aus dem *private key* der *public key* generiert wird, welcher gleichbedeutend mit der Bitcoin-Adresse ist, kann der Nutzer nachweisen, dass der auf dieser Adresse verfügbare Betrag ihm gehört. [Sixt, 2016, S. 37] Einerseits kann hierbei vollkommen anonym in Sekundenschnelle Geld überwiesen werden, andererseits birgt dies dadurch auch ein großes Risiko. Sollte der Schlüssel verloren gehen oder zerstört werden, ist kein Zugriff mehr möglich. Da ein Schlüssel aus einer zufälligen 27 bis 34 alphanumerischen Zeichenkette besteht, sind theoretisch 1.461.501.637.330.902.918.203.684.832.716.283.019.655.932.542.976 verschiedene Schlüssel möglich und die Wahrscheinlichkeit, den gesuchten Schlüssel zu erraten, ist dementsprechend klein. [Sixt, 2016, S. 12] Durch den Verlust des Schlüssels kann unter Umständen ein beträchtlicher finanzieller Schaden entstehen. Aus diesem Grund sind diverse Möglichkeiten zum Speichern von Backupkopien verbreitet, um eine Wiederherstellung des Schlüssels zu ermöglichen (vgl. Kapitel 2.1). Da das Ablegen eines solchen Dokuments an einer zentralen Stelle dem Grundsatz der Dezentralität zuwiderhandelt und stets mit Sicherheitsrisiken verbunden ist, soll die Möglichkeit einer verteilten Backuplösung untersucht werden. Im Kapitel 1.3 soll die zentrale Problemstellung anhand eines Use-Cases vorgestellt werden, um den Nutzen und die Anforderungen dieser Backupmethode zu veranschaulichen.



1.2 Ziel der Arbeit

Die zentrale Fragestellung dieser Arbeit befasst sich damit, ob die dezentrale Speicherung eines *private keys* auf mehreren Endgeräten ohne Abstriche hinsichtlich Sicherheit, Zuverlässigkeit und Benutzerfreundlichkeit möglich ist. Hierbei soll das Shamir's Secret Sharing Scheme [Winter, 2018, S. 41] Verwendung finden und untersucht werden, ob sich dieses Verfahren für den Einsatz unter den genannten Gesichtspunkten eignet. Generell wird dabei der private Schlüssel in eine beliebige Anzahl an n Teilen (*shares*) zerlegt und bestenfalls auf genauso viele Endgeräte verteilt. Diese sollten sich dabei vorzugsweise im Besitz von Vertrauenspersonen befinden und der Zugang zu diesen Geräten sollte sichergestellt sein. Zur Wiederherstellung muss eine festgelegte Untermenge dieser Teile k verfügbar sein. Dieser Parameter ist vor der Erstellung der Schlüsselteile durch den Benutzer festzulegen und darf weder n überschreiten noch kleiner als Zwei sein. Sobald die Menge der zusammengeführten Teile x den Wert k erreicht oder überschreitet, ist eine erfolgreiche Wiederherstellung des *private keys* möglich. [Wagner, 2004, S. 2] Soll der Schlüssel beispielsweise in $n = 5$ Teile zerlegt werden und gibt der Benutzer an, dass mindestens $k = 3$ Teile vorhanden sein müssen, dann kann jede denkbar mögliche Kombination von $x \geq 3$ zusammengeführten Teilen den Schlüssel wiederherstellen.

1.3 Veranschaulichung der zentralen Problemstellung

Menschen, die viel Geld in Kryptowährungen, beispielsweise in Bitcoin, investieren, benötigen dafür ein Wallet¹. Dies wird meistens auf dem jeweiligen Smartphone installiert und alle Transaktionen werden darüber getätig. Jedoch sind sich ein Großteil der Nutzer der Gefahr nicht bewusst, dass, ein möglicher Verlust beziehungsweise die Zerstörung ihres Smartphones gleichzeitig den Verlust des Zugriffs auf das Wallet bedeutet. Dies hätte unweigerlich zur Folge, dass das darin gespeicherte Vermögen unwiederbringlich verloren wäre, da es durch den im Wallet gespeicherten *private key* signiert wurde. Somit kann auf den Bestand nur zugegriffen werden, solange der Schlüssel verfügbar ist. Aus diesem Grund sollte stets ein Backup angelegt werden, um den Schlüssel notfalls wiederherstellen zu können. Einen zusätzlichen Schutz bietet es, den Schlüssel nicht zentral an einem Ort aufzubewahren, da hierbei die Gefahr auf unbefugten Zugriff geringer ist. Dieser Prozess wird mittels der in dieser Arbeit beschriebenen iOS-Applikation ermöglicht. Ein weiter Vorteil dieser App ist, dass der jeweilige User frei entscheiden kann, welchen seiner Mitmenschen er in dem Maße vertraut, dass er Teile seines Geheimnisses mit diesen teilen will.

¹ein Programm, das wie ein digitaler Geldbeutel fungiert und die Transaktionen durchführt bzw. speichert



1.4 Aufbau der Arbeit

Generell soll sich die vorliegende Masterarbeit in die drei Teile Aufzeigen der technischen Herausforderungen und Problemstellungen, Anforderungsanalyse für eine Backupapplikation und Implementierung eines Prototyps gliedern. Im ersten Teil sollen die Herausforderungen und Problemstellungen, welche ein Programm zum Erstellen von Sicherheitskopien allgemein aufwerfen, erläutert werden. Daraufhin soll die Anwendung des SSSS auf den gespeicherten privaten Schlüssel eines *cryptowallets* evaluiert und die Anforderungen an eine Applikation dieser Art herausgearbeitet werden. Hierzu zählen unter anderem die sichere Aufbewahrung der Schlüsselteile und der verschlüsselte Austausch dieser Teile zwischen den Endgeräten. Anschließend werden die Anforderungen in einem Prototyp auf Basis einer iOS-Applikation programmtechnisch umgesetzt. Hierbei werden zwar viele iOS-Boardmittel verwendet, jedoch ist das Konzept als solches theoretisch auch auf andere Betriebssysteme übertragbar. Sämtliche Ergebnisse werden in der Arbeit zusammengefasst. Es ist nicht Gegenstand der Arbeit, die verwendeten Protokolle und Mechanismen hinsichtlich ihrer Sicherheit zu überprüfen. Es wird von ausreichender Sicherheit und Zuverlässigkeit dieser ausgegangen.

1.5 Projektträger

Diese Masterarbeit wurde in Kooperation mit der Firma *adorsys GmbH & Co. KG* durchgeführt. Um die Hintergründe und das fachliche Interesse, welches dazu geführt hat, diese Arbeit zu unterstützen, einordnen zu können, soll das Unternehmen kurz vorgestellt werden. Die *adorsys GmbH & Co. KG* ist ein im Jahre 2006 in Nürnberg gegründetes mittelständisches IT-Unternehmen. Es wird aktuell von dem Geschäftsgründer Francis Pouatcha Nouy-euwe und seinem Stellvertreter Stefan Hamm geleitet und beschäftigt derzeit 110 Mitarbeiter an den Standorten Nürnberg und Frankfurt am Main.² Der Firmenschwerpunkt liegt auf individuellen Softwarelösungen für Kunden aus der Finanz- und Versicherungsbranche. Mit dem Motto, „Wir entwickeln Software für eine digitale Zukunft“, begleiten die eingesetzten Projektteams die Kunden von der ersten Idee bis hin zum fertigen Produkt. Dabei ist es unabhängig davon, ob ein bestehende Software modernisiert werden soll oder die Lösung zu einem neuen Geschäftsmodell fehlt. Auf ihrer Internetseite beschreibt sich das Unternehmen selbst wie folgt:

„Die adorsys ist ein seit 2006 bestehendes innovatives IT-Unternehmen für zielgenaue, individuelle und exklusive IT-Lösungen. Wir decken eine Vielzahl fachlicher und technologischer Themen ab und bieten die komplette Projektrealisierung aus einer Hand.“

²Stand November 2018



1 Einleitung

Von Projektmanagement, Businessanalyse und Anforderungsentwicklung, Softwarearchitektur und -entwicklung über Development Services bis zur Betriebsvorbereitung.“ [adorsys, 2017]

Da das Unternehmen sich intensiv mit den neuen Trends und Technologien im Bereich Finanzen beschäftigt, ist das Thema rund um Blockchain und Kryptowährungen ein Bereich, der immer wieder in den Fokus rückt. Im Rahmen vieler Projekte und wissenschaftlicher Arbeiten wird hier Know-How zusammengetragen und Kompetenz aufgebaut. Da das sichere Backup und Wiederherstellen des *private keys* eine zentrale Problemstellung der Kryptowährungen darstellt, soll durch diese Arbeit ein Ansatz zum Lösen dieses Umstandes geschaffen werden. In Abbildung 1.1 ist das aktuelle Firmenlogo der *adorsys GmbH & Co. KG* dargestellt.



Abbildung 1.1: Logo der *adorsys GmbH & Co. KG*

Quelle: entnommen aus [adorsys, 2019]



2 Herausforderungen und Problembetrachtung

In diesem Kapitel soll ein Einblick gegeben werden, welche Schwierigkeiten ein Programm zur Sicherung und Wiederherstellung von Schlüsseln für Kryptowährungen aufwirft.

2.1 Aktuelle Backupmethoden

Wie bereits in Kapitel 1.1 erwähnt, geht die Wahrscheinlichkeit zur Wiederherstellung eines verlorenen Schlüssels für einen Kryptowährungsaccount durch Erraten gegen Null. Zusätzlich sind digitale Speichermedien anfällig für Störungen und Beschädigungen. Aus diesem Grund haben sich bei *cryptocurrency wallets* gewisse Vorgehensweisen zum Erstellen einer Sicherungskopie etabliert. Generell gibt es zwei Arten von Sicherheit. Zum einen die Zugriffssicherheit (engl. *security*), welche sicherstellen soll, dass nur autorisierte Personen Lese- und/oder Schreibrechte auf bestimmte Datensätze haben. Dies wird meistens durch Zugriffsbeschränkungen wie Benutzerlogins oder Verschlüsselung der Daten erreicht. Der zweite Aspekt ist die lokale Sicherheit auf Vorhandensein (engl. *safety*), sprich die Gewährleistung, dass immer mindestens auf eine Kopie desselben Datensatzes zugegriffen werden kann. Dies wird dadurch erreicht, dass die Datei auf mehreren, separaten Datenträgern gespeichert wird. Somit ist mit hoher Wahrscheinlichkeit immer mindestens eine Kopie intakt. Ein Backup für ein *cryptocurrency wallet* sollte unbedingt immer unter Berücksichtigung beider Gesichtspunkte erstellt werden. Das bedeutet, ein Backup sollte unter allen Umständen in verschlüsselter Form gespeichert werden. Ein unverschlüsseltes Backup ist beispielsweise vergleichbar mit einer Kreditkarte, auf welcher der zugehörige Pin notiert ist. Jeder, der in irgendeiner Weise Zugriff auf das Backup erhält, kann sich des darin gespeicherten Vermögen bedienen. Zusätzlich sollte es, falls möglich, immer eine Vielzahl an Backups an verschiedenen Stellen und in unterschiedlicher Art und Weise geben. Hier sind der Phantasie des Nutzers keine Grenze gesetzt. Im Folgenden sollen die populärsten Backupvarianten kurz genannt werden. Anschließend wird jede Art genauer betrachtet.

- Ein **Online Wallet** ist zugänglich über einen Webbrowser. Somit ist der *private key* auf dem Server des Anbieter gesichert. Dies stellt eine komfortable Möglichkeit, jedoch ist viel Vertrauen gegenüber dem Betreiber notwendig, da der *private key* für den User nicht direkt zugänglich ist.



- Ein Backup als **chiffrierter Text** ist bei einer Vielzahl von Wallets möglich, dabei wird der *private key* als Text oder Satz (*mnemonic sentence*) ausgegeben und muss vom User selbst gespeichert werden. Dies bietet ein breites Spektrum an Möglichkeiten. Der User ist allerdings selbst in der Pflicht für Sicherheit zu sorgen.
- Ein **Hardware Wallet** ist ein externes Speichermedium, auf dem der *private key* gespeichert wird. Er ist sicher, da alle Vorgänge auf dem Gerät ausgeführt werden, jedoch teuer und es wird immer ein zusätzliches Gerät benötigt.
- Eine **Wallet.dat Datei** ist eine Datei, die alle Transaktionen und Schlüssel des Wallet enthält. Sie wird auf einen zusätzlichen Datenträger gespeichert und muss regelmäßig erneuert werden um aktuell zu bleiben. Der User muss selbst für die Sicherheit dieser sorgen.

Ein **Online Wallet**, wie beispielsweise *MyWallet* von *blockchain.info*, ist im eigentlichen Sinn kein Backup. Allerdings ist es eine Möglichkeit, seinen *privat key* zu speichern und vor Zerstörung zu schützen. Hierbei ist der Wallet nicht auf dem Endgerät gespeichert, sondern auf den Servern des Betreibers. Zugriffen wird darauf über einen Browser und eine Internetseite. Dieses Verfahren ähnelt sehr einem herkömmlichen Onlinebankingportal. Der User authentifiziert sich mit einer Kombination aus WalletID und Passwort, anstatt mit seinem *private key*. Sollte ein Bestandteil verloren gehen, kann es erneuert werden. Der Schlüssel ist davon nicht betroffen. Da ein Online Wallet komplett ohne zusätzliche Software auf dem Endgerät benutzt werden kann, sind sie der bequemste Weg, um mit Kryptowährungen zu handeln. Zusätzlich bieten sie meistens einen Exchangeservice an, das heißt, sie tauschen reale Währungen in Kryptowährungen und zurück. Da der User jedoch keine Zugriff auf den Schlüssel hat, mit welchem die eigentlichen Transaktionen durchgeführt werden, besteht hier ein hohes Sicherheitsrisiko, da der Benutzer auf die Vertrauenswürdigkeit des Anbieters angewiesen ist und sich darauf verlassen muss, dass dieser keinen Missbrauch der gespeicherten Daten betreibt. Um dieses Problem zu umgehen, gibt es Varianten der Online Wallets, welche dem User nach Erstellung des Kontos eine Datei zum Download anbieten, beispielsweise <https://www.myetherwallet.com>. Dieses enthält zwar den *private key*, jedoch muss sich hier selbstständig um ein Backup bemüht werden. [brokervergleich, 2019]

Eine weit verbreitete Möglichkeit ist es, den Schlüssel als **chiffrierten Text** zu speichern, auszudrucken oder in anderer Art und Weise zu konservieren. Es gibt zwei gängige Methoden, um den Text zu speichern. Bei Verwendung eines Bitcoin Improvement Proposals (BIP) 32/44 konformen Wallets kann der *private key* als Zeichenkette ausgegeben werden.¹ Diese kann dann nach Benutzerwunsch gespeichert werden. Einige Beispiele sind:

- Ausdrucken und sichere Aufbewahrung

¹ein BIP ist ein kurzes Dokument mit Vorschlägen zur Verbesserung der Bitcoin Technologie

- Erstellen eines QR-Codes und sichere Verwahrung
- Speichern in einer verschlüsselten Datei
- Ablegen in einem Schlüsselcontainer²

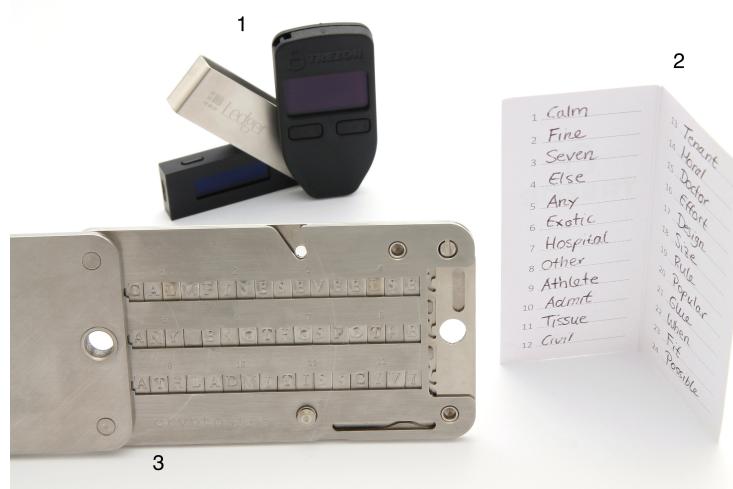


Abbildung 2.1: Übersicht über Backupmethoden von Kryptowährungsschlüsseln (Hardware Wallet (1), Textform(2), Text in Schlüsselcontainer(3))

Quelle: entnommen aus [hardware wallets.net, 2017]

Im BIP 39 wird eine andere Möglichkeit zur Generierung einer Klartextrepräsentation beschrieben. Hierbei wird aus dem Schlüssel ein Satz gebildet, ähnlich einer Eselsbrücke, ein so genannter *mnemonic sentence* (vgl. Abbildung 2.2). Dieser Umstand ermöglicht es, dass der *private key* sich vom Nutzer wesentlich einfacher gemerkt und gegebenenfalls rekonstruiert werden kann als eine 28-stellige Kombination aus Hexadezimalwerten. [bitcoin community, 2019] Dies wird unterstützt durch die Tatsache, dass ähnlich klingende Wörter nicht zulässig sind. Das Verfahren bietet zusätzlich den Vorteil, dass der *key* auf einzelne und unabhängige Teile aufgeteilt wird. Diese Anteile können, auf verschiedene Medien verteilt, gespeichert werden. Somit kann ein Dritter, der in Besitz eines Teiles kommt, den Schlüssel nicht rekonstruieren. Im Gegensatz zum SSSS-Algorithmus werden hier alle Teile für eine erfolgreiche Wiederherstellung benötigt.

Die wahrscheinlich kostspieligste Backupvariante ist der Einsatz eines **Hardware Wallets**. Dies ist ein externes Speichermedium, das den *private key* enthält. Zum Tätigen von Transaktionen muss das Hardware Wallet mit einer kompatiblen Wallet Applikation verbunden werden. Dabei beauftragt die Applikation die Überweisung, durchgeführt wird sie allerdings auf dem Hardware Wallet selbst. Aus diesem Grund muss das Hardware Wallet stets mit dem Gerät, das

²Ein Schlüsselcontainer ist eine kleine Box aus Metall, in die der Schlüssel eingestanzt oder als Buchstabenplättchen eingelegt worden ist



2 Herausforderungen und Problembetrachtung

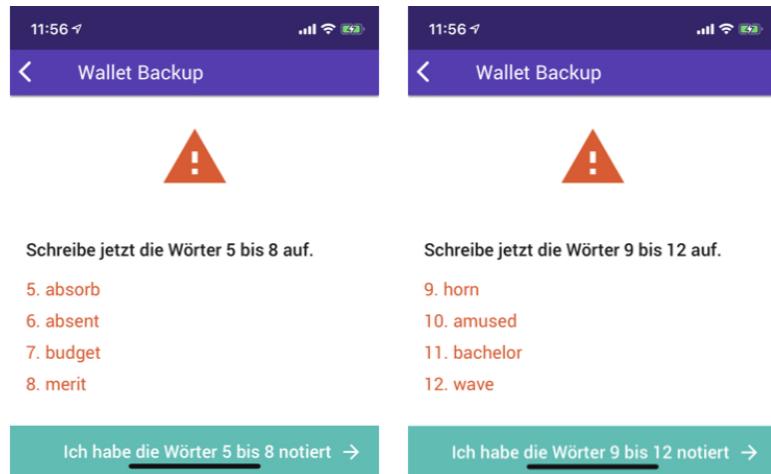


Abbildung 2.2: Beispielhafte Teile eines *mnemonic sentence* in einem iOS-Wallet

Quelle: eigene Darstellung

die Wallet Applikation besitzt, verbunden sein. Der Zwang des physische Vorhandenseins eines zusätzlichen Geräts, stellt ein enormes Maß an Schutz gegen Missbrauch und Diebstahl dar, da der *key* selbst das Wallet nie verlässt. Trotz des Umstands, dass ein zusätzliches Gerät benötigt wird, ist der Komfort eines Hardware Wallet inzwischen ähnlich dem eines Online-Wallets, allerdings mit einem wesentlich höheren Sicherheitsstandard. [hardware wallets.net, 2019] Ein optimales Szenario wäre die Verwendung von zwei Hardware Wallets mit demselben Schlüssel, eines zum Durchführen von Transaktionen und eines als Backup an einem sicheren Ort.

Eine weitere Möglichkeit ist das Kopieren der **Wallet.dat Datei**. Diese ist eine Art Datenbankdatei, die von vielen Wallets angelegt wird und in der alle Schlüssel und Transaktionen gespeichert werden. Das Backup besteht darin, dass diese Datei kopiert und sicher abgelegt wird. Gute Orte sind beispielsweise ein USB-Stick oder eine CD, um offline zu bleiben, oder aber Cloudservices wie Dropbox und iCloud. Sollte das Gerät, auf dem sich das Wallet befindet, beschädigt werden, kann der Ist-Zustand des letzten Backups durch Einspielen der .dat-Datei in einem neuen Wallet wiederhergestellt werden. Kritisch ist bei dieser Möglichkeit wiederum erneut die Sicherheit des Backups. Wie anfänglich dargelegt, sollte mit Dateien dieser Art sehr verantwortungsbewusst umgegangen werden und die Datei stets verschlüsselt gesichert werden. [O'Sullivan, 2018]

Abschließend sei gesagt, dass ein gutes Backup immer eine Kombination aus allen verschiedenen Varianten ist, da eine allein keine vollständige Sicherheit bietet. Zusätzlich sollte ein Backup immer verschlüsselt sein, denn man kann nie sicher sein, wer Zugriff auf das Backup erlangt. Dies gilt vor allem an Speicherorten, über die man selbst keine Kontrolle hat, wie zum Beispiel Online-Plattformen.



2.2 Secret Sharing

Der Secret Sharing Prozess ist ein Vorgang, bei dem ein Geheimnis, das *secret*, auf eine Gruppe von Personen aufgeteilt wird. Niemand kennt das komplette Geheimnis, sondern stets nur einen Teil davon. Gemeinsam kann allerdings aus den einzelnen Teilen, den *shares*, das Geheimnis rekonstruiert werden. Diese Idee des verteilten „Speichern“ eines Geheimnisses ist nicht neu. Das gängigste Beispiel ist ein Piratenkapitän, der seine Schatzkarte zerrißt und unter seiner Besatzung aufteilt, damit sie nur gemeinsam den Schatz bergen können. Heutzutage findet der Vorgang des *secret sharings* vor allem in der Geschäftswelt und im Militär Verwendung. Es handelt sich hierbei fast immer um den Zugang zu Geheiminformationen, der aus Sicherheitsgründen nur gemeinsam von einer Gruppe Personen erfolgen darf. Beispiele dafür sind der Zugriff auf bestimmte Bankkonten oder der Zugang zu hochgeheimen militärischen Daten, wie die Zertifikate beziehungsweise Schlüssel zum Verschlüsseln von Top-Secret-Informationen oder sogar den Abschusscodes von Raketen. [Wätjen, 2018, S. 295] Hierbei ist es aber immer möglich, dass einer aus dem Kreis der Geheimnisträger nicht anwesend sein kann, sei es durch Krankheit, Urlaub oder andere Verpflichtungen. Hier kommen moderne Schwellwertverfahren (*threshold secret sharing*) zum Einsatz. Ein Vertreter dieser Art ist das in Kapitel 3 vorgestellte Shamir's Secret Sharing Scheme. Hierbei wird das Geheimnis S auf eine Gruppe von n Personen aufgeteilt. Zusätzlich wird ein Schwellenwert für die Rekonstruktion t (*threshold*) festgelegt. Dies ist die Anzahl der Personen, die mindestens benötigt wird, um das Geheimnis wiederherzustellen. Somit kann jede beliebige Gruppe an Vertrauenspersonen, deren Anzahl x größer als t ist, das Geheimnis wiederherstellen. Bei einem *sicheren* Schwellwertverfahren verfügen die Beteiligten nach dem Kombinieren von $x < t$ Teilen nicht über mehr Informationen als ein Außenstehender, welcher keine Schlüsselteile besitzt.



2.3 Rivest Shamir Adleman Kryptosystem (RSA)

Das Rivest Shamir Adleman Kryptosystem (RSA) wurde von Ron Rivest, Adi Shamir und Len Adleman 1977 erfunden. Es war das erste öffentlich bekannte Public-Key Verschlüsselungsverfahren. Gleichzeitig ist es noch heute das wichtigste und am meisten verwendete Verfahren für sichere Kommunikation. [Buchmann, 2016, S. 168] Ergänzend sei erwähnt, dass Clifford Cocks von den Government Communication Headquarters (GCHQ) bereits 1975 eine ähnliche Idee hatte, die allerdings strengster Geheimhaltung unterlag und erst 1997 veröffentlicht wurde. [Wätjen, 2018, S. 77]

Ein Public-Key-Kryptosystem unterscheidet sich von einem damals üblichen Verschlüsselungsverfahren in einem grundlegenden Punkt. Bei der herkömmlichen Methode wird derselbe Schlüssel beim Ver- und Entschlüsselungsvorgang benutzt. Diese Vorgehensweise reicht von einfachen Beispielen von Substitutionschiffren³ und Transpositionsschiffren⁴ über die Enigma der Nationalsozialisten bis zu moderner Advanced Encryption Standard (AES)-Verschlüsselung. Bei all diesen Methoden kann der Empfänger die Nachricht nur entschlüsseln, wenn er den Algorithmus oder den Schlüssel kennt, welcher zum Verschlüsseln benutzt wurde. Aus diesem Grund muss der Schlüssel immer auf einem sicheren Weg zwischen den beteiligten Parteien ausgetauscht werden, wie beispielsweise durch einen Boten oder einen anderen Kommunikationskanal, der nicht mit der eigentlichen Nachricht in Verbindung gebracht werden kann. Im Gegensatz dazu benutzt ein Public-Key-Kryptosystem ein Schlüsselpaar. Es besteht aus einem öffentlichen Schlüssel (*public key*), welcher versendet oder in einem Verzeichnis gespeichert werden darf, und aus einem privaten (*private key*), der unter allen Umständen geheim gehalten werden sollte. Der öffentliche Schlüssel wird zum Verschlüsseln von Nachrichten verwendet, der private zum Entschlüsseln. Sollte Person A mit Person B kommunizieren wollen, erhält A den öffentlichen Schlüssel von B und verschlüsselt die Nachricht damit. Dann sendet er seine verschlüsselte Nachricht an B. Diese benutzt ihren privaten Schlüssel und entschlüsselt die Nachricht. Da der Schlüssel nur Person B bekannt sein sollte, kann ausschließlich sie die Nachricht lesen. Alle Stationen dazwischen, wie beispielsweise der Server des Internetproviders, können die Nachricht nicht lesen. Aus diesem Grund spricht man hierbei auch von *End-to-End-Encryption*. Abbildung 2.3 zeigt ein Schaubild für die Übertragung einer Nachricht zwischen Person A und B.

Name	öffentlicher Schlüssel
Person A	ZTkJkUDhVguHxwj19jVTO9WxlUiBgXKL
Person B	IONcDcXBYQBfBfY6t5iqm5sbrpFflCap
Person C	tlanX81d61AqrPmlfBv2yUcP4dryWSF0
Person D	OPPilk6H7uR8kkbMNDyJ38Q4dUnz1c1U
Person E	To1mlkRVNY4pNLn6hBnmAVHP124aln4i
...	...

³Chiffriermethode, bei welcher jeder Buchstabe des Klartextes durch ein anderes Zeichen oder einen anderen Buchstaben ersetzt wird

⁴Verschlüsselungsmethode, bei der die Reihenfolge der Buchstaben vertauscht wird

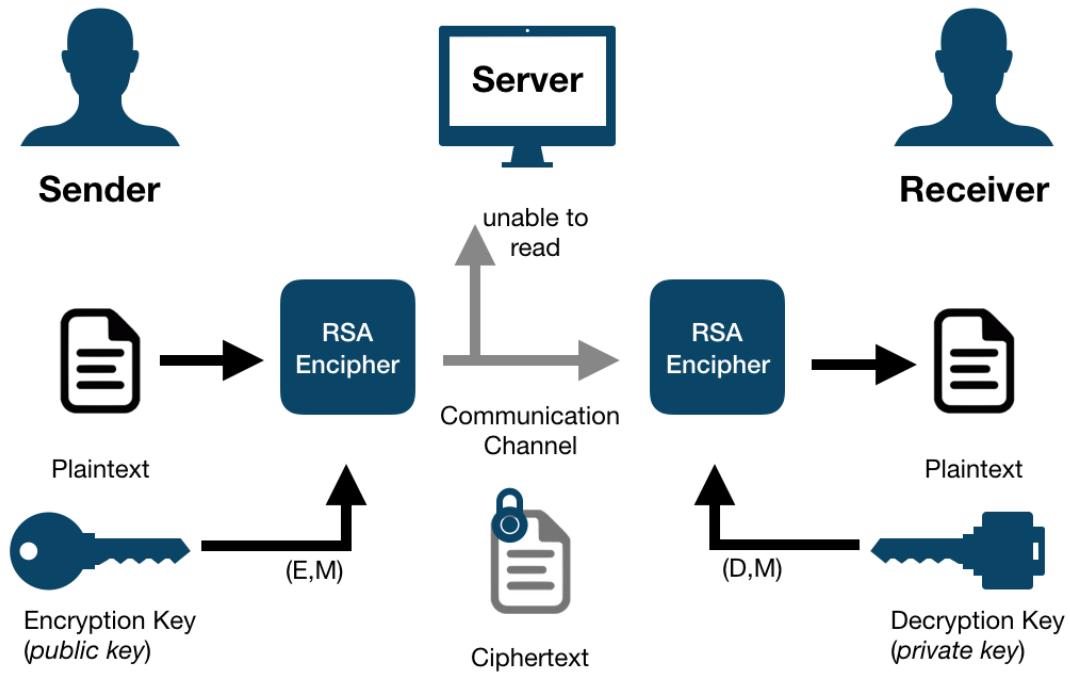


Abbildung 2.3: Schaubild zur beispielhaften Übertragung einer Nachricht vom Sender zum Empfänger

Quelle: eigene Darstellung

Zur Generierung des Schlüsselpaars werden erst zwei zufällige Primzahlen p und q gewählt, die sich in der Länge um wenige Stellen unterscheiden. Das Produkt aus diesen Zahlen definiert den *RSA-Modul* n und bildet die Basis für die Generierung des Schlüsselpaars. [Wätjen, 2018, S. 77]

$$n = pq \quad (2.1)$$

Die Bitlänge von n ist die Länge des Schlüssels und direkt proportional zu der Sicherheit des Schlüssels. In dem in dieser Arbeit vorgestellten Programm beträgt die Schlüssellänge 4096 Bit bzw. 512 Byte. Dies gilt zu diesem Zeitpunkt als sehr sicher.

Der Algorithmus wählt eine natürliche Zahl e (*encrypt*) aus, für die gilt: [Buchmann, 2016, S. 169]

$$1 < e < \varphi(n) = (p-1)(q-1), \gcd^5(e, (p-1)(q-1)) = 1 \quad (2.2)$$

Diese bildet in Verbindung mit dem *RSA-Modul* den öffentlichen Schlüssel. Durch den erweiterten euklidischen Algorithmus kann wiederum d (*decrypt*), also der private Schlüssel, berechnet werden, wofür gilt: [Buchmann, 2016, S. 169]

$$1 < d < (p-1)(q-1), de \equiv 1 \pmod{(p-1)(q-1)} \quad (2.3)$$

⁵Größter gemeinsamer Teiler



2 Herausforderungen und Problembetrachtung

Wie hieraus erkennbar ist, hängt die Sicherheit der Schlüssel stark von den gewählten Primzahlen ab, beziehungsweise davon, wie schwer der Modul faktorisierbar, also in seine Primfaktoren zerlegbar ist. Deshalb kann niemand beweisen, dass RSA wirklich sicher ist, es gilt nur als schwierige, sehr langwierige und rechenintensive Operation, eine derart große Zahl in ihre Primfaktoren zu zerlegen, vor allem dann, wenn die einzigen Faktoren zwei sehr große Primzahlen sind. [Buchmann, 2016, S. 172] Aus diesem Grund ist es beinahe unmöglich, den privaten Schlüssel aus dem öffentlichen Schlüssel zu berechnen. Sollte aber eine neue, schnellere Faktorisierungsmethode entwickelt werden oder sollten jemals Quantencomputer einsatzbereit sein, wird RSA als Verschlüsselungsmethode unbrauchbar werden. Ein weiterer Ansatz ist, viele Prozessoren parallel zu betreiben und so die Dauer der Faktorisierung zu verringern. Allerdings steigt der Energieverbrauch dabei enorm an. Laut einem Vortrag von Daniel Bernstein während eines Workshops der Defense Advanced Research Projects Agency (DARPA), würde ein Computer, welcher einen 2048 Bit RSA-Schlüssel in einem Jahr knackt, die gleiche Energie benötigen, die die Sonne innerhalb dieses Jahres auf die Erde schickt. [steemit, 2017] Im Vergleich dazu würde ein Standard Desktop Computer (2.2 GHz AMD Opteron mit 2 GB RAM) für diesen Schlüssel etwas mehr als 6.4 Quadrillionen Jahre benötigen. [digicert, 2019] Basierend auf diesen Tatsachen kann momentan davon ausgegangen werden, das RSA mit ausreichend großen Primzahlen eine sichere Kommunikation ermöglicht. Zusätzlich sind in den verwendeten RSA-Algorithmen weitere Mechanismen eingebaut, um die Sicherheit zu erhöhen. Auf diese wird an dieser Stelle nicht näher eingegangen, da es den Rahmen dieser Arbeit sprengen würde. Eine genauere Betrachtung des RSA-Kryptosystems kann den Literaturquellen [Wätjen, 2018] und [Buchmann, 2016] entnommen werden.



2.4 Verwendete Kommunikationsmethoden

Ein zentraler Punkt eines Backups ist die Übertragung der zu speichernden Daten von Punkt A zu Punkt B. Hierbei ist es sehr wichtig, dass die Übertragung vertraulich ist und möglichst schnell vorstatten geht. Auch eine Rückmeldung über eine erhaltene Nachricht ist wünschenswert, jedoch nicht immer mit einfachen Mitteln realisierbar. Für die Vertraulichkeit und Sicherheit sorgt bei der in dieser Arbeit entwickelten Anwendung das in Kapitel 2.3 beschriebene RSA-Verfahren. In diesem Kapitel wird zwischen zwei Kommunikationswegen unterscheiden:

- Die **direkte Kommunikation**, bei der direkt über die Schnittstellen des Endgeräts untereinander kommuniziert werden kann. (Bluetooth, Airdrop, etc.)
- Die **indirekte Kommunikation** wobei über eine dritte Instanz, meist einen Server, kommuniziert wird. (E-Mail, Whatsapp, etc.)

Bei der **direkten Kommunikation**, auch *Peer to Peer Communication (P2P)* genannt, werden die Daten direkt zwischen zwei Endgeräten ausgetauscht. Ein gutes Beispiel ist der Austausch von Bildern und Musik über Bluetooth oder Infrarot. Hier werden die Daten direkt und ohne einen Server oder ähnliche Netzwerkinfrastrukturen übertragen. Heute wird diese Art der direkten Kommunikation oftmals für drahtlose Verbindung von Peripheriegeräten verwendet. Einige Beispiele sind kabellose Eingabegeräte wie Controller für Spielkonsolen, Bluetoothkopfhörer oder Funkfernbedienungen. Hier sind die zu übertragenden Datenmengen und die Entfernung gering. Im Gegensatz dazu ist die Menge an Daten, welche zwischen Smartphones ausgetauscht werden sollen, stark angewachsen. Aus diesem Grund haben sich hier andere Übertragungsprotokolle etabliert. Die Geräte des Apple Konzerns setzen auf das hauseigene Protokoll *AirDrop*, wohingegen Android überwiegend mit einem Protokoll namens *Wifi-Direct* arbeitet. Eine direkte Kommunikation, komplett ohne Netzwerkschnittstellen, stellt das Lesen eines QR-Codes da. Hier ist das Übertragungsmedium das Licht, das auf den Kamerachip fällt. Das Betriebssystem decodiert die Nachricht und öffnet beispielsweise eine Internetseite oder verbindet sich mit einem drahtlosen Netzwerk.

Vorteil des direkten Kommunikationsweges ist die Übertragung quasi ohne Verzögerung, da die einzige Limitierung die Übertragungsgeschwindigkeit des jeweiligen Netzwerkprotokolls beziehungsweise der Netzwerkschnittstelle darstellt. Gleichzeitig kann man von seinem Kommunikationspartner stets eine Rückmeldung über den momentanen Zustand der Übertragung und nach Beendigung eine Bestätigung für eine erfolgreiche Übermittlung erhalten. Sollte die Übertragung fehlerhaft sein, kann sie sofort erneut angestoßen werden. Der Nachteil dieser Art der Kommunikation besteht darin, dass die Reichweite stark begrenzt ist und sich somit beide Parteien in unmittelbarer räumlicher Nähe befinden müssen.



2 Herausforderungen und Problembetrachtung

Der direkten Datenübertragung steht die **indirekte Kommunikation** gegenüber. Hierbei werden die Daten vom Sender zu einer speziellen zentralen Stelle geschickt, um von dort an den Empfänger weitergeleitet zu werden. Im Heimnetz kann die zentrale Stelle der Router sein, bei der Internetkommunikation der Server eines Providers. Die wohl inzwischen bekanntesten Vertreter der letzteren Art sind die E-Mail und das Messagingprogramm *Whatsapp*. Über diesen Weg lassen sich Textnachrichten und vielerlei Dateitypen sehr einfach von einem Ende der Welt an das andere verschicken. Daher ist es nicht verwunderlich, dass die Nutzerzahlen von *Whatsapp* inzwischen bei circa 1,5 Milliarden liegen und täglich rund 65 Milliarden Nachrichten versendet werden. [statista, 2018]

Bei der indirekten Kommunikation werden die zu versendeten Daten verpackt, mit einer Adresse versehen und über das Internet an den zentralen Server gesendet. Dieser leitet die Nachricht an den Empfänger weiter, sobald dieser für den Server erreichbar ist. Aus diesem Grund ist keine räumliche Nähe zwischen Sender und Empfänger nötig. Allerdings muss durch wesentlich aufwendigere Sicherheitsmechanismen die Authentizität des Senders und die Unverfälschtheit der Nachricht gewährleistet werden. Zusätzlich hat die Asynchronität dieser Übertragungsweise Vor- und Nachteile. Einerseits kann die Nachricht zeitversetzt zugestellt werden, wenn der Empfänger vorübergehend keine Netzwerkverbindung hat. Andererseits muss die Rückmeldung des Empfangs bei der Zieladresse meistens manuell durchgeführt werden. Eine solche Kommunikation kann durchaus einige Zeit in Anspruch nehmen, je nach Datenmenge und Auslastung der Server oder Netzwerkkommunikationskanäle. Zusätzlich ist es möglich, dass Nachrichten *verloren* gehen oder das die Zustellung erheblich länger dauert als nötig.

Somit muss bei der Konzeptionierung einer Backup-Applikation Folgendes analysiert werden:

- Welche Datenmengen und -typen sollen übertragen werden?
- Welche Arten von Kommunikation sollen dem Nutzer zur Verfügung stehen?
- Auf welchen Wegen können die benötigten Schlüssel für eine sichere Kommunikation zwischen Sender und Empfänger ausgetauscht werden?

Im Anschluss müssen Konzepte erarbeitet werden, um diese Anforderungen umzusetzen. Dies ist Gegenstand des Kapitels 4.



3 Shamir's Secret Sharing Scheme (SSSS)

Das *Shamir's Secret Sharing Scheme* ist ein Vertreter des in Kapitel 2.2 vorgestellten Prozesses des *Secret Sharings*. Es wurde 1979 von *Adi Shamir* in seinem Dokument *How to Share a Secret* [Shamir, 1979] publiziert und ist ein Schwellwertverfahren (*Threshold Secret Sharing*). Hierbei wird das Geheimnis S auf eine bestimmte Anzahl von n Personen aufgeteilt, wobei jede ein Teil des Geheimnisses (*share*) erhält. [Wätjen, 2018, S. 296] Zusätzlich wird der Schwellenwert t gewählt. Dieser Wert gibt die Anzahl der Personen an, die mindestens ihre Schlüssel kombinieren müssen, um S zu reproduzieren. Da bei SSSS alle *shares* gleichwertig sind, kann zum Wiederherstellen von S jede beliebige Untermenge k von n ihre Schlüsselteile vereinen, solange gilt $k \geq t$. Eine Gruppe von $k \leq (t - 1)$ ist allerdings nicht in der Lage, nähere Informationen über das Geheimnis zu erfahren. Ihr Wissen ist vergleichbar mit einer außenstehenden Person, welche keine Geheimnisteile besitzt. [Shamir, 1979] Dieses Verfahren ist dahingehend sinnvoll, dass sich die Geheimsträger gegenseitig kontrollieren können und sich somit die Wahrscheinlichkeit des Missbrauchs verringert. Da nicht alle Personen beziehungsweise deren Schlüsselteile an der Rekonstruktion teilnehmen müssen, ist es sehr flexibel einsetzbar.

3.1 Grundlegende Funktionsweise

Das SSSS basiert auf dem Prinzip der Polynominterpolation. Im zweidimensionalen Raum definieren zwei Punkte, $P_1(x_1, y_1)$ und $P_2(x_2, y_2)$, exakt eine Gerade. Mit drei Punkten, $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ und $P_3(x_3, y_3)$, kann eine Parabel bestimmt werden, mit vier Punkten eine kubische Funktion, usw. Im Allgemeinen lässt sich ein Polynom vom Grad t durch mindestens $(t + 1)$ Punkte festlegen. [Shamir, 1979] Aus einem Polynom lassen sich aber unendlich viele Punkte berechnen, somit kann die benötigte Anzahl an *shares* recht einfach generiert werden.

Das folgende Beispiel soll die grundlegende Funktionsweise verdeutlichen. Es verwendet allerdings statt der Arithmetik der endlichen Körper die Arithmetik der ganzen Zahlen und ist somit, genau genommen, kein eigentliches Beispiel des SSSS. Die grundlegenden mathematischen Konzepte sind aber identisch.



3 Shamir's Secret Sharing Scheme (SSSS)

Will man allgemein ein Geheimnis S mit einem (t, n) -Schwellwertverfahren¹ aufteilen, müssen zuerst t und n festgelegt werden, dass gilt:

$$1 < t \leq n \quad (3.1)$$

Anschließend wird ein allgemeines Polynom vom Grad $t - 1$ aufgestellt:

$$f(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \dots + a_{k-1} \cdot x^{k-1} \quad (3.2)$$

Die Parameter a_i werden zufällig gewählt, wobei $a_0 = S$ gesetzt wird. Anschließend wird für jede Person ein Teil berechnet mit:

$$f(0 < i < n) = s_i \quad (3.3)$$

Es erhält jede Person einen Punkt $P_i (i, s_i)$. Das Backup ist jetzt komplett.

Zur Wiederherstellung kombiniert man mindestens t Punkte. So kann mithilfe der Lagrange-Methode (Gleichung 3.4) [Buchmann, 2016, S. 294] das Ausgangspolynom bestimmt und $a_0 = S$ abgelesen werden.

$$L(x) = \sum_{i=1}^t y_j \prod_{j=1, j \neq i}^t \frac{x_j - x}{x_j - x_i} \quad (3.4)$$

Da a_0 gleichbedeutend mit S ist, wird nur dieser Koeffizient benötigt. Aus diesem Grund wird die optimierte Lagrange-Methode (Gleichung 3.5) verwendet, da das Berechnen aller Vorfaktoren ineffizient und unnötig ist.

$$L(0) = \sum_{i=1}^t y_j \prod_{j=1, j \neq i}^t \frac{x_j}{x_j - x_i} \quad (3.5)$$

Zahlenbeispiel

Erzeugung der Schlüsselteile

Es sei gegeben:

- $S = 5486$
- $t = 3$
- $n = 6$

¹sprich: t aus n Schwellwertverfahren



Aus $t = 3$ und $a_0 = S$ ergibt sich das allgemeine Polynom zu:

$$f(x) = S + a_1 \cdot x + a_2 \cdot x^2 \quad (3.6)$$

Die übrigen Konstanten werden zufällig gesetzt auf:

$$a_1 = 3653$$

$$a_2 = 4865$$

Damit ergibt sich das Polynom:

$$f(x) = 5486 + 3653 \cdot x_1 + 4865 \cdot x^2 \quad (3.7)$$

Es berechnen sich die einzelnen Werte mit:

$$f(1) = 5486 + 3653 \cdot 1 + 4865 \cdot 1^2 = 14004$$

$$f(2) = 5486 + 3653 \cdot 2 + 4865 \cdot 2^2 = 32252$$

$$f(3) = 60230$$

$$f(4) = 97938$$

$$f(5) = 145376$$

$$f(6) = 202544$$

Somit ergeben sich die *Shares* zu

$$s_1 = (1, 14004), s_2 = (2, 32252), s_3 = (3, 60230),$$

$$s_4 = (4, 97938), s_5 = (5, 145376), s_6 = (6, 202544)$$

Wiederherstellung des Geheimnisses

Es sollen die Schlüsselteile s_4 , s_5 und s_2 und die optimierte Lagrange-Methode (Gleichung 3.5) benutzt werden.

$$s = a_0 = \sum_{i=1}^t y_j \prod_{j=1, j \neq i}^t \frac{x_j}{x_j - x_i} \quad (3.8)$$

Damit Gleichung 3.8 gilt, müssen die Indizes der verwendeten s_i geändert werden.

$$P_1(x_1, y_1) = (4, 97938), P_2(x_2, y_2) = (5, 145376), P_3(x_3, y_3) = (2, 32252)$$



Aus Gründen der Übersichtlichkeit werden die Summanden m_i einzeln berechnet.

$$m_1 = \frac{x_2}{x_2 - x_1} \cdot \frac{x_3}{x_3 - x_1} = \frac{5}{5-4} \cdot \frac{2}{2-4} = -5$$

$$m_2 = \frac{x_3}{x_3 - x_2} \cdot \frac{x_1}{x_1 - x_2} = \frac{2}{2-5} \cdot \frac{4}{4-5} = \frac{8}{3}$$

$$m_3 = \frac{x_1}{x_1 - x_3} \cdot \frac{x_2}{x_2 - x_3} = \frac{4}{4-2} \cdot \frac{5}{5-2} = \frac{10}{3}$$

$$S = \sum_{i=1}^t y_i \cdot m_i$$

$$S = -5 \cdot 97938 + \frac{8}{3} \cdot 145376 + \frac{10}{3} \cdot 32252 = 5486$$

3.2 Sicherheitsbedenken

Nach informationstechnologischem Standard gilt ein *Secret Sharing System* als sicher, wenn eine Person mit weniger als t *shares* keinerlei Informationen über das Geheimnis erlangen kann. Jedoch hat das oben beschriebene Verfahren unter diesem Gesichtspunkt einen sicherheitstechnischen Schwachpunkt. Denn da die Information t und somit das allgemeine Polynom sowie die Information $a_0 = S$ öffentlich bekannt sind, kann mit zunehmender Anzahl von gesammelten *shares* das *secret* immer enger eingegrenzt werden. Der Angreifer macht sich dabei den Umstand zunutze, dass die Reihenfolge der Zahlen durch die Menge der natürlichen Zahlen festgelegt ist, nämlich $0, 1, 2, 3, 4, \dots$. Gleichzeitig kann ein Graph einer Funktion nur in einer beschränkten Art und Weise zufällig verlaufen und in Verbindung mit einem Polynom aus Gleichung 3.2 keine Sprünge oder ähnliches enthalten. Dadurch wird die Anzahl von unendlich möglichen Geheimnissen drastisch reduziert.

3.3 Lösung des Sicherheitsproblems

Um den in Kapitel 3.2 dargestellten Umstand zu umgehen, wird sich der Eigenschaften der endlichen Körper in der Mathematik bedient. Ein Endlicher Körper ist ein Konstrukt, das eine endliche Menge an Zahlen besitzt. Realisiert wird ein solches Gebilde durch den *Modulo Operator* `mod`, der auch oft mit dem % Zeichen dargestellt wird. Mathematisch ergibt er den *Rest* einer Division zweier ganzer Zahlen. Somit ergibt beispielsweise $19 \bmod 5 = 4$.

Anwendung im SSSS-Algorithmus



Zusätzlich zu den bereits bekannten Parametern S, s_i, t und n , wird eine zusätzliche Konstante eingeführt. Es wird eine Primzahl p gewählt, dass gilt $p > S, p > n$
somit ergibt sich: [Wätjen, 2018, S. 298]

$$f(x) = S + \sum_{i=1}^{t-1} a_i x^i \in \mathbb{Z}_p[x] \quad (3.9)$$

Anschließend werden die *shares* auf eine ähnliche Weise berechnet, wie im ersten Beispiel. Allerdings muss hierbei der *Modulo* Operator mod berücksichtigt werden, sodass berechnet wird:

$$f(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \dots + a_{k-1} \cdot x^{k-1} \mod p \quad (3.10)$$

Das Geheimnis kann über eine ähnlich angepasste Lagrange-Methode wiederhergestellt: [Wätjen, 2018, S. 300]

$$S = L(0) = \sum_{i=1}^t y_j \prod_{j=1, j \neq i}^t x_j (x_j - x_i)^{-1} \mod p \quad (3.11)$$

Zahlenbeispiel

Um den Vergleich zu vereinfachen, wird das Beispiel mit denselben Parametern wie jenes aus Kapitel 3.1 dargestellt.

gegeben:

- $S = 5486$
- $t = 3$
- $n = 6$
- $p = 11047$

Analog zu Kapitel 3.1 ergibt sich das Polynom zu

$$f(x) = S + a_1 \cdot x + a_2 \cdot x^2 \mod p \quad (3.12)$$

Die Konstanten sind:

$$a_1 = 3653$$

$$a_2 = 4865$$

Damit ergibt sich:

$$f(x) = 5486 + 3653 \cdot x_1 + 4865 \cdot x^2 \pmod{11047} \quad (3.13)$$

$$f(1) = 5486 + 3653 \cdot 1 + 4865 \cdot 1^2 \pmod{11047} = 2957$$

$$f(2) = 5486 + 3653 \cdot 2 + 4865 \cdot 2^2 \pmod{11047} = 10158$$

Analog dazu:

$$f(3) = 4995$$

$$f(4) = 9562$$

$$f(5) = 1765$$

$$f(6) = 3698$$

Spätestens hier macht sich der Unterschied der beiden Verfahren bemerkbar, da keine Aussage darüber getroffen werden kann, wie der Graph verläuft (vgl. Abbildung 3.1). Denn anders als bei einer Parabel üblich steigt der Wert nicht kontinuierlich. Durch den *Modulo* Operator wird jede Zahl oberhalb von 11046 *umgebrochen*. Der inverse Vorgang ist nicht möglich, da der Wert der Division nicht gespeichert wird. Denn $22095 \% 11047 = 11048 \% 11047 = 1$ Aus diesem Grund ist es nicht möglich, das Geheimnis ohne ausreichend *shares* einzugrenzen.

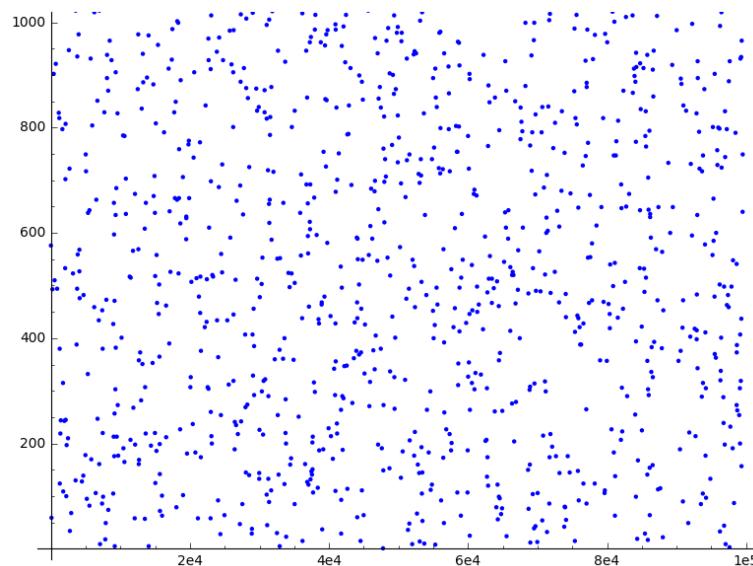


Abbildung 3.1: Beispielhafte Kurve eines Polynoms in einem endlichen Körper

Quelle: entnommen aus [Wolfmankurd, 2017]



Die *shares* sind somit:

$$s_1 = (2957), s_2 = (2, 10158), s_3 = (3, 4995),$$

$$s_4 = (4, 9562), s_5 = (5, 1765), s_6 = (6, 3698)$$

Wiederherstellung des Geheimnisses

Bei der Wiederherstellung der Koeffizienten muss eine Besonderheit der Endlichen Körper berücksichtigt werden, denn es existieren darin nur ganze Zahlen. Dabei stellt

$$m_1 = x_2 \cdot (x_2 - x_1)^{-1} \cdot x_3 \cdot (x_3 - x_1)^{-1} = 5 \cdot (5 - 4)^{-1} \cdot 2 \cdot (2 - 4)^{-1} = -5$$

kein Problem dar, doch im Folgenden muss dieser Umstand berücksichtigt werden:

$$m_2 = x_3 \cdot (x_3 - x_2)^{-1} \cdot x_1 \cdot (x_1 - x_2)^{-1} = 2 \cdot (2 - 5)^{-1} \cdot 4 \cdot (4 - 5)^{-1} = 8 \cdot 3^{-1}$$

Da in einem endlicher Körper alle Grundrechenoperationen möglich sein müssen, muss es im Körper \mathbb{Z}_{11407} eine Zahl geben, für die gilt:

$$3 \cdot x = 1$$

Diese kann entweder aus einer Multiplikationstabelle abgelesen oder durch Ausprobieren ermittelt werden. In der hier vorliegenden Größe ist die Möglichkeit des Ausprobieren meistens effizienter, als eine Tabelle anzulegen. Hierfür kann beispielsweise der folgende Swift-Programmcode verwendet werden.

```
1 let field = 11047;
2 let numerator = 10;
3 let denominator = 3;
4 for i in 0 ..< field {
5     if (denominator * i) % field == numerator {
6         print("corresponding Integer found: \(i)")
7         break
8     }
9 }
```

Quellcodebeispiel 3.1: Programmcode zum Berechnen des inversen Elements einer rationalen Zahl in einem endlichen Körper

Im Grunde probiert das Programm alle Zahlen aus, bis es diejenige findet, welche die Bedingung $3 \cdot x = 1$ erfüllt. In diesem Beispiel ist das die Zahl 7365, denn es gilt $3 \cdot 7365 \bmod$



$$11047 = 1.$$

Somit ergibt sich:

$$m_1 = 8 \cdot 3^{-1} \in \mathbb{Z}_{11047} = 8 \cdot 7365$$

Analog dazu gilt:

$$m_3 = x_1 \cdot (x_1 - x_3)^{-1} \cdot x_2 \cdot (x_2 - x_3)^{-1} = 4 \cdot (4 - 2)^{-1} \cdot 5 \cdot (5 - 2)^{-1} = 10 \cdot 3^{-1} \in \mathbb{Z}_{11047} = 10 \cdot 7365$$

Nach Gleichung 3.11:

$$S = \sum_{i=1}^t y_i \cdot m(i) \mod p$$

$$S = m(1) \cdot y_1 + m(2) \cdot y_2 + m(3) + y_3 \mod 11047$$

$$S = -5 \cdot 9562 + 8 \cdot 7365 \cdot 1765 + 10 \cdot 7365 \cdot 10158 \mod 11047$$

$$S = 852082690 \mod 11047$$

$$S = 5486$$



4 Konzeptionierung und Anforderungsanalyse

Für die Generierung der Schlüsselteile selbst wird der SSSS-Algorithmus verwendet. Eine weitere Hauptaufgabe dieser Applikation ist die Verwaltung der Schlüsselfragmente und die Zuordnung zu den Profilen der Vertrauenspersonen. Dies ist nötig, da sichergestellt werden muss, dass jede Person einen anderen Teil bekommt und, dass im Falle einer wiederholten Übertragung des Schlüsselteils stets der identische Teil übertragen wird. Im folgenden Kapitel soll auf die Anforderungen an eine Applikation für sichere Backups eingegangen und generelle Konzepte zur Erfüllung eben jener Anforderungen vorgestellt werden.

In Kapitel 1.3 wurde ein kurzer Überblick über die Anforderungen und die Zielsetzungen an ein Sicherungsprogramm gegeben. Bei genauerer Betrachtung der Thematik entsteht eine Vielzahl von Fragestellungen, die sich zentral um zwei Themengebiete drehen. Der erste Themenbereich ist die Speicherung der Daten, der die Fragestellungen *Was soll gespeichert werden?*, *Wie soll gespeichert werden?* und vor allem *Warum soll gespeichert werden?* beinhaltet. Die zweite Problemstellung ist die Übertragung von Daten, wobei sich folgenden Fragestellungen ergeben: *Was soll übertragen werden?* und *Wie soll übertragen werden?*

4.1 Programmablauf

Zur besseren Übersicht der verwendeten Mechanismen und Prozesse, die in den nachfolgenden Kapiteln vorgestellt werden, soll zuerst eine Übersicht über den Programmablauf und die Funktionsweise der Applikation gegeben werden. Allgemein gliedert sich das Programm in zwei verschiedene Abläufe:

- Erstellung eines Backups
- Wiederherstellung eines Schlüssels

Beide Workflows werden jeweils in einem Schaubild graphisch dargestellt.

4.1.1 Erstellung eines Backups

Die Erstellung eines Backups läuft in sechs unterschiedlichen Schritten ab.

1. Abrufen des zu sichernden Geheimnisses

Abrufen des Geheimnisses oder Eingabe durch den Nutzer, Erstellung der Schlüsselteile, Anlegen der Datenbank für Metadaten

2. Erstellung eines Kommunikationsschlüsselpaars

Erstellung der Kommunikationsschlüssel, falls noch nicht vorhanden

3. Einbinden von Vertrauenspersonen

Importieren von PublicKeys der Vertrauenspersonen, Vervollständigung der *Buddy*-Profile durch Metadaten wie Kontaktfoto, Nickname und zusätzliche Informationen

4. Versenden eines Schlüsselteils

Aufbau eines verschlüsselten Kommunikationskanals, Versenden eines Schlüsselteils

5. Import des Schlüsselteils

Import des Schlüsselteils auf dem Gerät der Vertrauensperson

6. Wiederholung von Schritt drei bis fünf

Wiederholung Schritts drei bis fünf, bis Anzahl der *Buddies* ausreichend ist

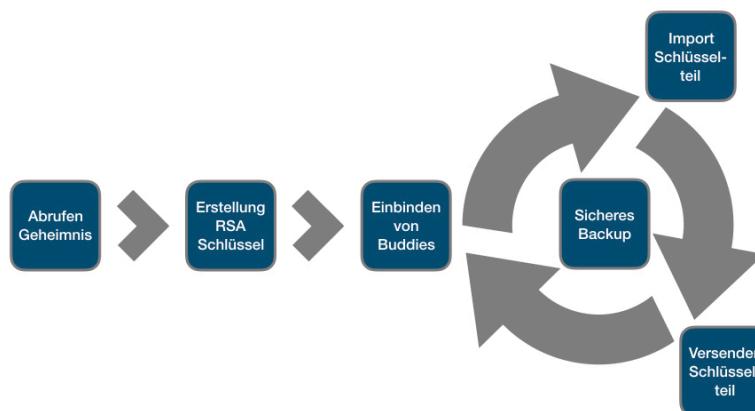


Abbildung 4.1: Strukturdiagramm der Erstellung eines Backups

Quelle: eigene Darstellung

Zu Beginn muss das Geheimnis in das Programm importiert werden. Dies kann entweder über Texteingabe geschehen oder indem das Programm im Wallet gespeicherten *private key* ausliest. Die zweite Möglichkeit bietet hierbei größere Sicherheit, weil der Schlüssel nie in Klartext vorliegen muss. Da diese Arbeit nur eine Machbarkeitsstudie darstellt und ohne die übergeordnete Walletapplikation auskommt, ist nur die erste Version möglich. Nach dem Import ist es notwendig, die öffentlichen Kommunikationsschlüssel der *Buddies* zu importieren. Hierbei wird



gleichzeitig ein Profil angelegt, das unter anderem diesen Schlüssel der zugehörigen Person zuordnet und abspeichert. Nach erfolgreichem Import kann ein Schlüsselteil zufällig ausgewählt und durch den jeweiligen Kommunikationsschlüssel verschlüsselt versendet werden. Durch die Verschlüsselung kann ausschließlich der gewünschte Empfänger das Schlüsselfragment entpacken und weiterverarbeiten. Er speichert daraufhin die empfangenen Informationen in seiner Datenbank ab. Die Schritte zwei, drei und vier werden mit unterschiedlichen Personen wiederholt, bis die Anzahl der verteilten *shares* für ein sicheres Backup ausreichend ist. Diesen Parameter kann der Geheimnisträger selbstständig wählen.

4.1.2 Rekonstruktion mithilfe eines Backups

Die Wiederherstellung eines Schlüssel aus einem Backup besteht aus fünf Schritten.

1. **Erstellung eines Kommunikationsschlüsselpaars**

Erstellung der Kommunikationsschlüssel, falls noch nicht vorhanden

2. **Austausch der Kommunikationsschlüssel**

Austausch der Kommunikationsschlüssel zwischen dem Geheimnisträger und dem Besitzer eines Schlüsselteils

3. **Übertragung der Schlüsselteile**

Übertragung aller gespeicherten Schlüsselteile zurück an den Geheimnisträger

4. **Wiederholung von Schritt zwei und drei**

Wiederholung von Schritt zwei und drei, bis ausreichend Teile gesammelt wurden

5. **Wiederherstellung des Geheimnisses**

Wiederherstellung des Geheimnisses nach Überschreitung des Grenzwertes

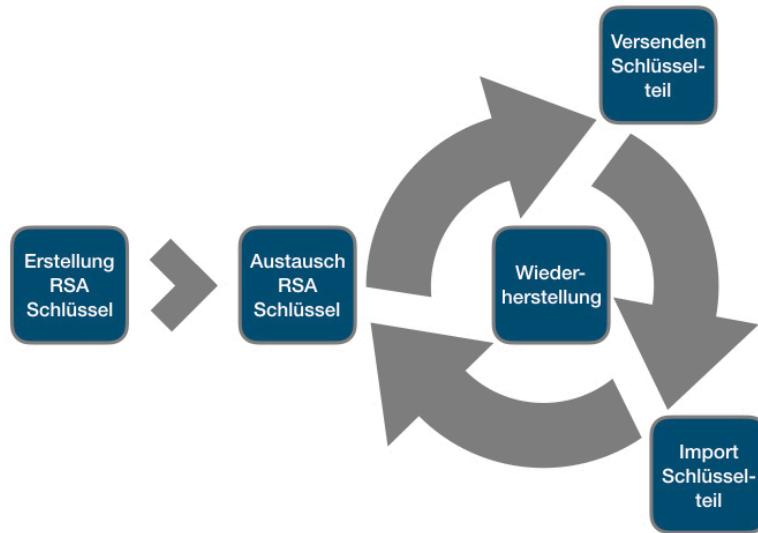


Abbildung 4.2: Strukturiertes Diagramm der Wiederherstellung eines Backups

Quelle: eigene Darstellung

Aus einem Backup muss der *private key* nur dann wiederhergestellt werden, wenn das Gerät zerstört oder verloren gegangen ist. Da der Kommunikationsschlüssel gleichfalls verloren ist, muss dieser zuerst neu generiert werden. Anschließend muss der Geheimnisträger die *public keys* der beteiligten Personen importieren. Gleichzeitig muss auf Seiten der Vertrauenspersonen der Schlüssel, der mit dem Profil des Geheimnisbesitzers verknüpft ist, aktualisiert werden. Erst dann ist eine erfolgreiche Kommunikation zwischen deren Endgeräten möglich. Sind die Schlüssel ausgetauscht, sollen die *shares* an den Besitzer übertragen werden. Auf dessen Gerät wird der Schlüsselteil gespeichert. Diese Prozedur wird entsprechend oft wiederholt, bis eine ausreichende Anzahl an Teilen vorhanden ist und der verlorene *private key* wiederhergestellt werden kann.

4.2 Speicherung der Daten

Wie bereits erwähnt, lauten die zentralen Fragestellungen der Speicherung von Daten: *Was soll gespeichert werden? Wie soll gespeichert werden? Warum soll gespeichert werden?* In diesem Fall genauer: *In welcher Form sollen die relevanten Informationen gespeichert werden? Welche Daten müssen überhaupt persistent gehalten werden? Aus welchem Grund ist es notwendig diese Daten auf dem Gerät abzulegen?* Essentielle Bestanteile der Daten sind die Schlüsselteile selbst und die *Profile* der Vertrauenspersonen. Jedes dieser Geheimnisbruchstücke muss alle für die Wiederherstellung relevanten Daten, wie die Nummer des Schlüsselteils und den festgesetzten Schwellenwert, enthalten. Um die redundante Speicherung dieser Daten zu vermeiden, sollen zusammengehörige Schlüsselteile in einer Art Container gespeichert werden.



Im Falle der *Buddies* ist es entscheidend, die Zuordnung von *communication public keys* und den *Nicknames*, also den vom User festsetzbaren Spitznamen zur Identifizierung der Person, festzuhalten um die Kommunikation zu ermöglichen. Auch muss ein Speicherbereich für die von diesem *Buddy* empfangenen Schlüsselteile geschaffen werden, um ihm bei Bedarf seine Schlüsselteile zurück liefern zu können. Da zwischen diesen zwei Grundobjekttypen vielerlei Beziehungen bestehen, soll zur Speicherung eine relationale Datenbank verwendet werden. Der große Vorteil eines Datenbanksystems ist die Möglichkeit des gezielten Zugriffs auf bestimmte Objekte, beziehungsweise die einfache Filterung der Datenmenge auf die gewünschte Menge. Da eine Vielzahl an Situationen im Lebenszyklus einer solchen Backupapplikation möglich ist, erleichtert diese Art der Speicherung die Verwaltung der Daten ungemein. Es muss sicher gestellt sein, dass jeder *Buddy* ein unterschiedliches Schlüsselteil bekommt beziehungsweise dass jedes Schlüsselteil erst einmal versendet worden ist, bevor eins mehrfach verteilt wird. Außerdem kann eine Vertrauensperson ihr Gerät verlieren. Damit das Backup wieder vollständig wird, muss es möglich sein, das *richtige* Teil erneut zu übertragen. Auch um dem User die Möglichkeit zu bieten, einzusehen, wie viele Teile er von einem Schlüssel schon verteilt hat beziehungsweiseen welche Teile eine bestimmte Person hat, müssen diese Metadaten gespeichert werden. Zusätzlich muss zur Wiederherstellung der Schlüssel eine Möglichkeit existieren, die Schlüsselteile asynchron zu sammeln. Das bedeutet, dass sich nicht alle Vertrauenspersonen zur selben Zeit am selben Ort aufhalten müssen, sondern, dass der Geheimnisträger die Personen nacheinander aufsucht. Damit eine Wiederherstellung möglich ist, müssen die einzelnen Schlüsselteile wiederum nach Verwendungszweck sortiert gespeichert werden. Denn erst, wenn der Schwellenwert überschritten wird, kann der Schlüssel wiederhergestellt werden. Eine große Erleichterung zur Wiederherstellung der Schlüssel bietet die Methode, einen gewissen Satz an Metadaten auf einer Onlineplattform, in diesem Fall auf iCloud, zu speichern. Hierzu gehören vor allem, die *Identifikationsnummern (IDs)* der *Key-Container*, die jeweiligen beteiligten *Nicknames* und der jeweilige Schwellenwert. Mit diesen Informationen, lässt sich eine Wiederherstellung, signifikant erleichtern, da der User die Informationen besitzt, welche Personen von welchem Schlüssel Teile gespeichert haben und somit eine Art geführte Wiederherstellung möglich ist.

4.2.1 Datenbankstruktur

Wie in Kapitel 4.2 erwähnt, sollen die Daten in einer *relationalen Datenbank* gespeichert werden. Eine Datenbank dieser Art ist in der Lage, Beziehungen zwischen Datenbankobjekten aufzubauen und zu verarbeiten. Das bedeutet, dass wenn ein Datenbankeintrag ausgelesen wird, der in Relation zu einem anderen steht, im gleichen Zug auf die Inhalte des zweiten Eintrags zugegriffen werden kann, ohne einen weiteren expliziten Datenbankaufruf durchzuführen. Dies ermöglicht es, eine Art Datenbaum anzulegen, ohne die IDs der entsprechenden Objektinstanzen separat speichern zu müssen. In Abbildung 4.3 ist der Aufbau dieser Datenbank graphisch dargestellt.

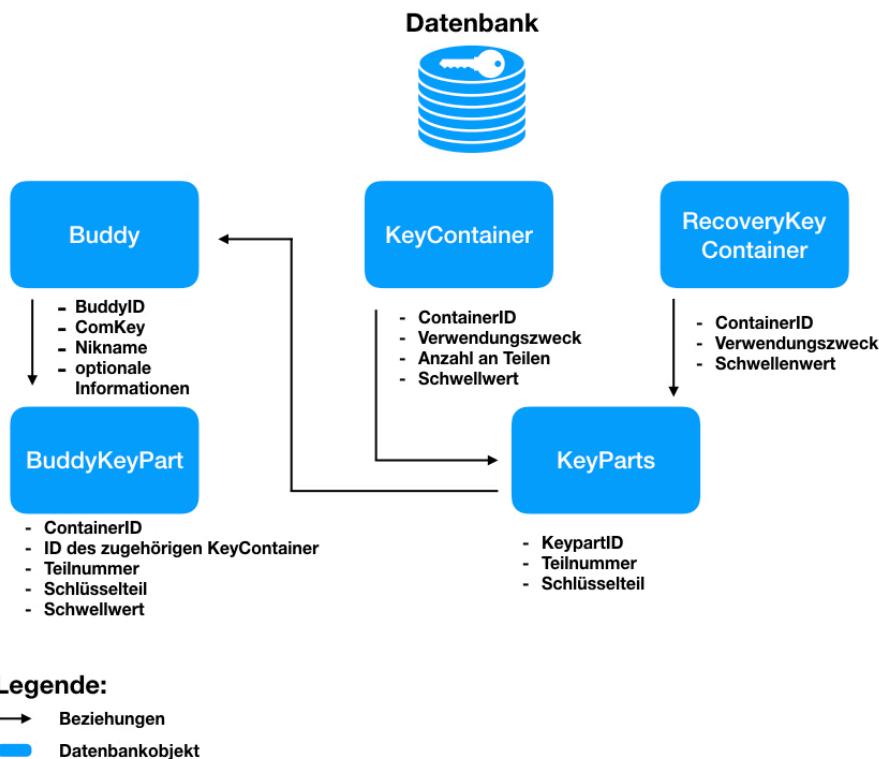


Abbildung 4.3: Datenbankstruktur

Quelle: eigene Darstellung

Es gibt fünf unterschiedliche Objekttypen, wobei jeder spezialisierte Datensätze beinhaltet. Diese sind in Abbildung 4.3 unter dem jeweiligen Datenbankobjekt aufgelistet. Gleichzeitig sind die Beziehungen zwischen den Objekten durch Pfeile dargestellt. Generell beinhaltet jedes Objekt eine ID, um sicherzustellen, dass eine Instanz eines Objekts nur einmal angelegt wird und anschließend aktualisiert werden kann. Die beiden Containerobjekte *KeyContainer* und *RecoveryKeyContainer* enthalten neben den *KeyParts* alle benötigten Metadaten, die für



die Verteilung beziehungsweise Wiederherstellung der Kryptowährungsschlüssel oder allgemein der Geheimnisse benötigt werden. Der *KeyContainer* wird bei der Erstellung des Backups verwendet und beinhaltet, neben allen beteiligten Schlüsselteilen, den Verwendungszweck des Schlüssels, sowie die Gesamtanzahl der Teile und den Schwellenwert. Wie der Name suggeriert, kommt der *RecoveryKeyContainer* bei der Wiederherstellung des Geheimnisses zum Einsatz. Hierbei ist nur der Schwellenwert interessant. Zusätzlich werden hier diejenigen *KeyParts* gespeichert, welche von einer Vertrauensperson auf das Gerät des Schlüsselbesitzers zurückübertragen wurden. Im Objekttyp *KeyParts* wird der eigentliche Schlüsselteil, die Teilnummer und die Personen, an die der Schlüsselteil verteilt wurde, gespeichert. Diese Personen bekleiden je eine Instanz der *Buddy*-Klasse. Neben dem zwingend erforderlichen Kommunikationsschlüssel und einem *Nickname* können zusätzliche Informationen wie ein Kontaktfoto und beispielsweise eine E-Mail-Adresse oder Telefonnummer gespeichert werden, um die Zuordnung zu erleichtern. Das Konstrukt der *Buddies* besitzt zusätzlich einen Speicherbereich, in dem diejenigen Schlüsselteile als *Buddy-KeyParts* abgelegt werden, die von dieser Person zur Speicherung empfangen wurden. Hierbei sind neben dem eigentlichen Schlüsselteil vor allem die Informationen, zu welchem Schlüssel das Teil gehört, die *ID des zugehörigen Containers*, die Teilnummer und der Schwellenwert interessant. Da diese Datenbank über eine Vielzahl an vertraulichen Informationen verfügt, muss sie in jedem Fall verschlüsselt und somit vor unbefugtem Zugriff geschützt werden.

4.2.2 iCloud Metadaten

Der Hintergrund, Metadaten in iCloud zu speichern, besteht darin, den Prozess der Wiederherstellung zu vereinfachen. Hierbei werden ausschließlich die IDs der *KeyContainer* in Verbindung mit den beteiligten *Buddies* gespeichert. Und hier wiederum nur der *Nickname* des *Buddies* und die Teilnummer des betreffenden *KeyParts*. Sollte das Smartphone verloren gehen, kann diese Informationen auf das neue Gerät heruntergeladen werden. Somit muss sich der Nutzer nicht daran erinnern, wem er welchen Teil welches Schlüssels gegeben hat, sondern kann gezielt die jeweiligen Personen kontaktieren. Die sensiblen Daten werden nicht auf iCloud hochgeladen, da diese Datei nicht dort verschlüsselt abgelegt werden kann. Der Grund dafür liegt darin, dass, bei Verlust des Smartphones, der Schlüssel für die Datei gleichsam verloren geht und somit der Zugriff auf diese nicht möglich wäre. Der iCloud-Zugriff bietet dem Nutzer, falls er die Applikation auf mehreren iOS-Geräten installiert hat, die Möglichkeit abzurufen, welche Vertrauenspersonen welche Teile von welchem Schlüssel innehalt, auch wenn das Gerät selbst nicht über die Schlüsselteile verfügt.



4.3 Kommunikation zwischen den Geräten

In dieser Applikation sollen die beiden in Kapitel 2.4 beschriebenen Kommunikationsmethoden enthalten sein. Hier kommen die schon bereits erwähnten öffentlichen Kommunikationsschlüssel zum Einsatz. Denn da diese Schlüssel vom Gerät abhängig sind, kann nur das Bestimmungsgerät die Informationen lesen.

4.3.1 Formen der Datenübertragung

Generell können Daten in mehreren Formaten übertragen werden. In dieser Applikation kommen davon zwei zum Einsatz:

- **Datenstrom**

Übertragung der Daten ohne Zwischenspeicherung oder Umwege durch Übertragungswege dritter Seite

- **Dokument**

Schreiben der Daten in eine Datei/Dokument und Übertragung als Ganzes

Bei der Datenübertragung als Datenstrom werden die Daten nicht erst gespeichert, sondern direkt und kontinuierlich versendet. Beispiele sind hierfür Internettelefonie oder auch der Messagingdienst Whatsapp. Als Nutzer bemerkt man den Austausch an Daten nicht und muss beispielsweise nicht jede neue Nachricht explizit in das Programm laden. Im Gegensatz dazu steht die Übertragung als Dokument/Datei. Hierbei werden alle zu übertragenden Informationen in eine Datei geschrieben. Diese Datei kann auf beliebigem Wege übertragen. Sie kann vom User aktiv in jedem Programm geöffnet werden, das den vorliegenden Dateityp unterstützt. In dieser Applikation wird dafür das eigens angelegte Dateiformat ..abf verwendet. Übertragen werden die Informationen in beiden Fällen im JavaScript Object Notation (JSON)-Format. Dies ist ein standardisiertes Format, das für Übertragungen sehr häufig verwendet wird. Es besteht aus Text in der *Key-Value-Syntax*, was bedeutet, dass jeder Wert einen *Namen* besitzt. Aus diesem Grund eignen sich JSON-Files sehr zur Verarbeitung in einer objektorientierten Programmiersprache. Um die Daten vertraulich zu halten, wird der Text des JSON-Files vor dem direkten Versenden oder Abspeichern in einer Datei mit dem *public key* des Empfängers unleserlich gemacht. Für die Übertragung von Dateien stellt iOS eine Vielzahl an Möglichkeiten bereit. Diese reichen vom Versenden über *AirDrop*, *Nachrichten* und *E-Mail* über Speichern auf Onlineplattformen wie *iCloud* bis hin zu Methoden von Drittanbietern. Hierzu gehört beispielsweise das Übertragen durch *Threema* und *Whatsapp* oder der Upload auf *Dropbox*. In Abbildung 4.4 wird das Auswahlfenster zum Teilen einer .abf-Datei gezeigt. Hierbei ist das iOS-Betriebssystem oder das Drittanbieterprogramm für die Kommunikation verantwortlich. Somit findet sie komplett außerhalb des Backupprogramms statt. Für eine direkte und programminterne Kommunikation

stellt Apple ein Framework bereit. Dies wird in Kapitel 5.1.3 näher beschrieben. Es basiert komplett auf der Kommunikation durch Datenströme.

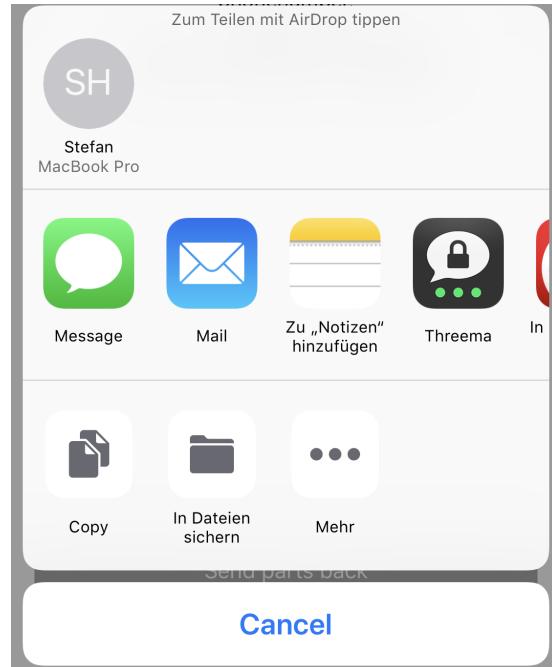


Abbildung 4.4: Beispielhafter iOS Exportbildschirm einer .abf-Datei

Quelle: eigene Darstellung

4.3.2 Übertragung der Kommunikationsschlüssel

Um die zu versendenden Information mit dem *public key* des Empfängers verschlüsseln zu können, muss dem Gerät der jeweilige Schlüssel zur Verfügung stehen. Auch hierbei sollen die beiden in Kapitel 4.3.1 genannten Kommunikationskanäle zur Verfügung stehen. Dabei wird der *public key* immer als QR-Code dargestellt. Dies bietet die Möglichkeit, sowohl den Code graphisch anzuzeigen als auch als Bilddatei abzuspeichern. Die Abfolge der schwarzen und weißen Pixel stellt hierbei eine codierte Information dar. Das Zielgerät kann diese decodieren und weiterverarbeiten. Diese Art der Datenübertragung eignet sich sehr gut für die Übertragung von kleinen Datenmengen, wie beispielsweise Links zu Internetseiten oder Netzwerkschlüsseln. Da ein *public key* für die Veröffentlichung bestimmt ist und somit keine sicherheitskritischen Daten enthält, müssen diese Daten nicht verschlüsselt werden.

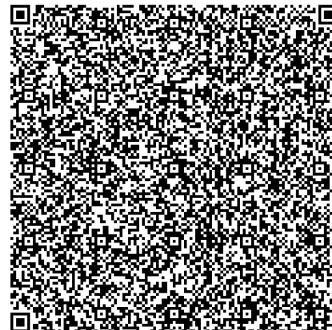


Abbildung 4.5: Kommunikationsschlüssel als QR-Code

Quelle: eigene Darstellung

In Abbildung 4.5 ist ein beispielhafter Schlüssel abgebildet. Bilddateien können, ähnlich wie in Kapitel 4.3.1, versendet werden. Um den Schlüssel in das Backupprogramm zu importieren, kann die Datei entweder direkt mit diesem Programm geöffnet werden, vgl. Abb 4.6, oder aber über eine spezielle Oberfläche innerhalb der App mit Hilfe der Smartphonekamera gescannt werden, vgl. Abb 4.7.

Die Möglichkeit des Scannens des QR-Codes ist sehr benutzerfreundlich und fast unmöglich zu missbrauchen, da sich hierbei beide Geräte in unmittelbarer Nähe befinden müssen. Somit kann der Geheimnisträger seinen Gegenüber kontrollieren.



Abbildung 4.6: Möglichkeit zum direkten Import einer Bilddatei in das Backupprogramm

Quelle: eigene Darstellung

Generell lässt sich sagen, dass die Methode zum Übertragen von Daten als Datei in iOS einfach und komfortabel ist und durch die in Kapitel 2.3 vorgestellte Methode der Verschlüsselung für die meisten Anwendungszwecke in puncto Sicherheit und Integrität ausreichend ist. Möchte der Nutzer jedoch ein höheres Niveau dieser beiden Aspekte, ist es möglich, Daten ohne das Verwenden von programmexternen Methoden und Netzwerkstrukturen auszutauschen. Es muss dabei nicht auf den Komfort der Einfachheit verzichtet werden.

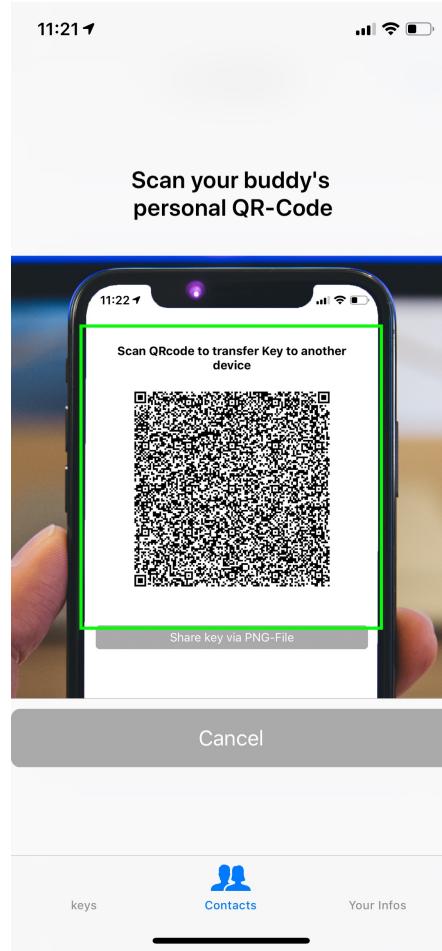
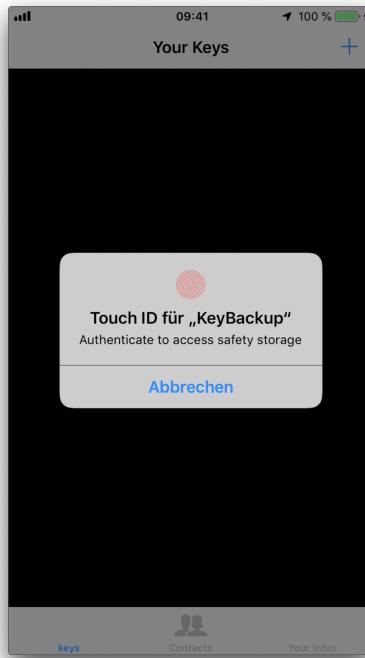


Abbildung 4.7: Möglichkeit zum Scannen eines QR-Codes innerhalb der App mithilfe der Smartphonekamera

Quelle: eigene Darstellung

4.4 Sicherheitsaspekte

Wie bereits erwähnt, ist die sichere Speicherung aller Daten von höchster Wichtigkeit. Das Verschlüsseln von Nachrichten wurde in den vorherigen Kapiteln ausgiebig behandelt. Die Datenbank wird symmetrisch verschlüsselt, somit darf der Schlüssel nicht öffentlich bekannt sein und muss sicher verwahrt werden. Dazu wird er in der iOS eingebetteten *Keychain* abgespeichert. Dies ist ein hochsicherer Bereich des iOS-Betriebssystems, welcher für die Speicherung sensibler Daten konzipiert worden ist und auf den nur durch expliziten Wunsch des Users zugegriffen werden kann. Hier muss sich der Nutzer authentifizieren. Dies geschieht bei neueren iOS-Geräten fast immer über den Fingerabdruck *TouchID* oder über einen Scan der Gesichtsmerkmale *FaceID*. Dadurch ist gewährleistet, dass nur der Besitzer selbst auf den in der *KeyChain* gespeicherten Schlüssel und somit auf die Datenbank der Applikation zugreifen kann. Abbildung 4.8 zeigt eine beispielhafte Authentifizierungsaufforderung für *TouchID*.

Abbildung 4.8: Authenifizierungsauflöderung mit *TouchID*

Quelle: eigene Darstellung

4.5 Mögliche Funktionserweiterungen

Zusätzlich zum eigentlichen Programmablauf sind einige weitere Funktionen denkbar, die aber in dieser Arbeit nicht implementiert werden, da sie deren Rahmen übersteigen würden. Ein Aspekt ist die Sicherstellung der Verfügbarkeit der Schlüsselteile. Denn sollte einer der Vertrauenspersonen sein Gerät zerstören oder verlieren, ist dieses Fragment des Sicherungskopie zugleich unerreichbar. Meldet sich diese Person bei allen Geheimnisträgern die Schlüsselteile bei ihr gespeichert haben, können sie erneut übertragen werden. Geschieht dies nicht und weitere Vertrauensträger verlieren ihrer Teile ebenfalls, kann das Backup unbrauchbar werden, da die erforderliche Anzahl an Schlüsselfragmenten nicht mehr verfügbar ist. Eine Lösung wäre, dass sich die App, die das Backup erstellt hat, bei den Geräten, die ein Teil besitzen, „erkundigt“, ob das Teil noch verfügbar ist und gegebenenfalls den Geheimnisträger auf den Verlust aufmerksam macht. Somit kann er das Teil erneut übertragen. Dieser Prozess des Überprüfens kann zum Beispiel über ein Push-Nachricht realisiert werden. Jedoch ist der Aufwand zur Umsetzung dieser Funktionalität nicht unerheblich. Somit ist sie erst für zukünftige Programmversionen vorgesehen. Einen Ausweg bietet das manuelle *Deaktivieren eines Buddies*. Somit besteht immerhin die Möglichkeit den Status des Backups zu aktualisieren. Eine weitere denkbare und überaus nützliche Funktion ist das entfernte Löschen von Schlüsselteilen auf Geräten von *Buddies*, falls sich an der Beziehung zwischen dieser Person und dem Besitzer des



Geheimnis etwas ändert und man dieser Person das Teil entziehen möchte. Hierbei könnten erneut Push-Nachrichten zum Einsatz kommen. Denkbar ist auch die Möglichkeit, sich nicht von den Geräten von Personen abhängig zu machen, sondern die Schlüsselbruchstücke auf ebensoviele verschiedene Onlineplattformen hochzuladen oder auf diverse E-Mail-Accounts zu verteilen. Hierbei muss die Frage der Sicherheit geklärt werden, wobei ein Schlüsselteil allein eine relativ unkritische Information darstellt. Der Aufwand für einen Angreifer, in mehrere gesicherte Onlineplattformen einzudringen, um der Teile habhaft zu werden, steigt enorm mit der Anzahl der Teile beziehungsweise der Speicherorte. Im Gegenzug dazu sind diese Teile für den Besitzer stets zugänglich und auch vor versehentlichem Verlust geschützt.



5 Implementierung

In diesem Kapitel soll eine Übersicht über die Frameworks und iOS spezifischen Mechanismen gegeben werden, die in dieser Arbeit verwendet werden.

5.1 Verwendete Komponenten

Bei der Entwicklung von Software werden oftmals fertige Komponenten von Drittanbietern eingebunden. Sie sind ähnlich einem Baustein und bieten eine bestimmte Funktionalität, wie das Erstellen einer Datenbank oder die direkte Kommunikation zwischen mehreren Geräten. Diese Bausteine nennt man *libraries*, zu deutsch Bibliotheken. In diesem Kapitel sollen die verwendeten Bibliotheken und deren Implementierung kurz vorgestellt werden.

5.1.1 Realm Database

Wie in Kapitel 4.2 erwähnt, sollen die Daten in einer relationalen Datenbank gespeichert werden. Diese Aufgabe übernimmt die Realm Datenbank. Sie ist ein Datenbank-Framework der Firma Realm und stellt in der Open-Source-Variante alle grundlegenden Datenbankfeatures zur Verfügung. In der kostenpflichtigen Version *Realm Platform* kann die Datenbank um Features wie serverbasierte Synchronisation und *Realtime Collaboration* erweitert werden. [realm, 2019b] Da die Datenbank in der in dieser Arbeit vorgestellten Applikation nur auf dem einen Gerät verfügbar sein soll, ist die kostenlose Variante ausreichend. Die *Realm Database* unterstützt die Verschlüsselung der gesamten Datenbank mit AES-256+SHA2 mit einem Schlüssel der Länge 64 Byte. [realm, 2019a] Da AES ein symmetrisches Verschlüsselungsverfahren ist, muss zum Öffnen stets derselbe Schlüssel verwendet werden. Daher wird er nach erstmaligem Anlegen in der in Kapitel 4.4 erwähnten *iOS-Keychain* gespeichert. Bei Programmstart muss der User sich über *TouchID* oder *FaceID* authentifizieren. Anschließend kann der Schlüssel geladen und somit die Datenbank geöffnet werden. Laut dem Hersteller betragen die Leistungseinbußen durch die Ver- und Entschlüsselung weniger als zehn Prozent. [realm, 2019a] Die abzulegenden Datenbankobjekte müssen nach einem gewissen Schema aufgebaut sein. Dies soll mit Hilfe dreier aus Kapitel 4.2.1 bekannter Objekten veranschaulicht werden: *Buddy*, *KeyPart* und *KeyPartContainer*.



5 Implementierung

```
1 class Buddy: Object {
2
3     override static func primaryKey() -> String? {
4         return "BuddyID"
5     }
6
7     @objc dynamic var BuddyID:String!
8     @objc dynamic var commPublicKey:Data!
9     @objc dynamic var nickname:String
10    @objc dynamic var contactPicture:String?
11    @objc dynamic var additionalInformation:String?
12
13    let keyOwner = LinkingObjects(fromType: KeyPartSSSS.self,
14                                   property: "buddies")
15 }
```

Quellcodebeispiel 5.1: Definition der *Buddy* Klasse in Realm

Das Codebeispiel 5.1 zeigt einen Ausschnitt der Klasse *Buddy* in der Art und Weise, wie sie das Realmframework vorschreibt. Jede Klasse muss von der Elternklasse *Object* abgeleitet sein. Zusätzlich zum Datentyp muss eine Variable das Präfix `@objc dynamic` tragen. Dies sind Relikte aus der Zeit, als iOS noch in Objective-C und nicht in Swift programmiert wurde. Denn der Kern der Datenbank wurde noch nicht nach Swift portiert. Eine der Besonderheiten der Datenbankstruktur sind die *LinkingObjects*. Sie stellen die Beziehung zu einem zugehörigen Objekt vom Typ *KeyPartSSSS* her, genauer gesagt sind sie jeweils ein Verweis auf exakt das Objekt, welches diesen *Buddy* in dessen Variable `"buddies"` enthält. Somit kann stets abgerufen werden, welcher KeyPart mit welchem *Buddy* geteilt worden ist, ohne die Beziehung manuell in einem extra Verzeichnis speichern zu müssen. Über die Funktion `primaryKey() -> String?` wird festgelegt, welche Variable die ID für das jeweilige Objekt enthält. Somit ist sichergestellt, dass immer nur ein *Buddy* mit der jeweiligen ID existiert und bei Bedarf aktualisiert werden kann.

```
1 class KeyPartSSSS: Object {
2     override static func primaryKey() -> String? {
3         return "keyPartID"
4     }
5
6     @objc dynamic var keyPartID:String!
7     @objc dynamic var partNumber:Int = 0
8     @objc dynamic var keyPart:Data!
9     @objc dynamic var alreadyShared = 0
10    @objc dynamic var usage:String!
11
12    let buddies = List<Buddy>()
13    let keyContainer = LinkingObjects(fromType: KeyPartContainer.self,
14                                     property: "includingKeys")
15 }
```

Quellcodebeispiel 5.2: Definition der *KeyPart* Klasse in Realm



5 Implementierung

Im Codebeispiel 5.2 ist ein Teil der Klasse KeyPart dargestellt. Diese enthält im Vergleich zu 5.1 eine zusätzliche Art von Objekten, das `List<T>`. Es funktioniert ähnlich wie ein Array, wobei `T` den jeweiligen Typ angibt, welcher in diesem Array gespeichert wird. Hier enthält es Elemente vom Typ `Buddy`. Es ist wiederum das Objekt, über das in den Buddies durch das `LinkingObjects` auf die Instanz des `KeyPartSSSS` referenziert wird.

```
1 class KeyPartContainer: Object {
2     override static func primaryKey() -> String? {
3         return "containerID"
4     }
5
6     @objc dynamic var containerID:String!
7     @objc dynamic var keyUsage:String!
8     @objc dynamic var totalKeyParts:Int = 0
9     @objc dynamic var thresholdKeyParts:Int = 0
10    @objc dynamic var KeySharingCounter:Int = 0
11
12    let includingKeys = List<keyPartSSSS>()
13 }
```

Quellcodebeispiel 5.3: Definition der `KeyPartContainer` Klasse in Realm

Die Klasse `KeyPartContainer` ist die übergeordnete Klasse zu `KeyPartSSSS` und bietet Zugriff auf zusammengehörige Schlüsselteile, welche in `includingKeys` gespeichert sind. Über die Variable `KeySharingCounter` in Verbindung mit dem Faktor `alreadyShard` der `KeyPartSSSS` Klasse wird sichergestellt, dass jedes Teil zuerst einmal verteilt wird, bevor eines zweimal ausgewählt wird.

Lesend zugegriffen wird auf die Datenbank stets über das Datenbankobjekt selbst (vgl. Codebeispiel 5.4), wohingegen der Schreibzugriff über eine Schreibfunktion ausgelöst wird. Dies stellt einen asynchronen Prozess dar, da das Schreiben großer Datensätze einige Zeit in Anspruch nehmen kann und das Smartphone derweil weiterhin auf Eingaben des Nutzers reagieren können soll (vgl. 5.5). Durch das Codewort `try` wird dem Betriebssystem suggeriert, dass dies ein Prozess ist, welcher aus verschiedenen Gründen fehlschlagen könnte.

```
1 buddies = database.objects(Buddy.self)
```

Quellcodebeispiel 5.4: Lesezugriff zum Erhalt aller `Buddies`-Instanzen

```
1 //add updateable instance
2 try! database.write {
3     database.add(actualBuddy,
4                 update:true)
5 }
6 //change property
7 try! database.write {
8     actualBuddy.nickname = NiknameTextfield.text
9 }
```



5 Implementierung

```
10
11 //Delete instance
12 try! database.write {
13     database.delete(self.actualBuddy)
14 }
```

Quellcodebeispiel 5.5: Verschiedene Möglichkeiten des Realm Schreibzugriffs

Einen Vorteil, den die Verwendung von Datenbanken mit sich bringt, ist das gefilterte und sortierte Auslesen von Datensätzen. Dadurch kann sehr komfortabel auf exakt die Daten zugegriffen werden, die benötigt werden, ohne explizit Funktionen zur Sortierung oder Filterung zu implementieren. Dazu müssen die gewünschten Parameter dem Aufruf in Form eines NSPredicate mitgegeben werden. Das Codebeispiel 5.6 zeigt ein Beispiel für gefilterten und sortierten Datenbankzugriff.

```
1 //Filter with predicate
2 var predicate = NSPredicate(format: "alreadyShared == %@", 
3                             KeySharingCounter as NSObject)
4 var keyparts = database.objects(KeyPartSSSS.self).filter(predicate)
5
6 //sort for nickname
7 buddies = database.objects(Buddy.self).sorted(byKeyPath: "nickname",
8                                              ascending: true)
```

Quellcodebeispiel 5.6: Beispiele für einen gefilterten und sortierten Datenbankzugriff

5.1.2 SSSS in Swift

Für die Realisierung der SSSS-Funktionalität wird die *SecretShare.swift* Bibliothek [Grinman, 2018] verwendet. Sie stellt die Funktionen zum Erstellen und Kombinieren der Schlüsselteile zur Verfügung. Diese werden anschließend in den vorgesehenen Datenbankobjekten gespeichert.

```
1 //needed informations
2 let totalNumberParts = 5
3 let thresholdNumber = 3
4 let secretMessage = "SecretMessage"
5 let keyUsage = "MyLittleSecret"
6 //generate SSSS secret
7 let secret = try Secret(data: secretMessage.data(using: .utf8)!, 
8                         threshold: thresholdNumber,
9                         shares: totalNumberParts)
10 let parts = try secret.split()
11 //generate database objects
12 let keyParts = createPartsAndStore(secrets: parts, usage: keyUsage)
13 let keyContainer = KeyPartContainer(keyUsage: keyUsage,
14                                     keys: keyParts,
15                                     totalKeyParts: totalNumberParts,
```



5 Implementierung

```
16                                         thresholdKeyParts: thresholdNumber)
17 //write parts to database
18 try database.write {
19     database.add(keycontaier,
20                 update: true)
21 }
```

Quellcodebeispiel 5.7: Erstellung der Schlüsselteile und Speicherung in der Datenbank in den vorgesehenen Objekten

Im Codebeispiel 5.7 wird der Zerteilungsprozess beispielhaft für die Parameter

- Anzahl an Personen = 5
- Schwellenwert = 3
- Geheimnis = SecretMessage

programmtechnisch aufgeführt.

Im nachfolgenden Programmcodebeispiel 5.8 ist der Ablauf dargestellt, um das Geheimnis aus den importierten Schlüsselteilen wiederherzustellen.

```
1 //imported keyparts
2 let keyParts = [KeypartSSSS]()
3 let shares = [Secret.Share]()
4 // restore each share from data representation
5 actualKeyContainer.includingKeys.forEach { (keyPart) in
6     let newKeyPart = try! Secret.Share.init(data: keyPart.keyPart!)
7     if !(shares.contains(secret: newKeyPart)) {
8         shares.append(newKeyPart)
9     }
10 }
11 //combine shares
12 let data = try? Secret.combine(shares: fewShares)
13 //print restored Secret
14 print(data)
```

Quellcodebeispiel 5.8: Rekonstruktion des Geheimnisses aus den importierten Schlüsselteilen

5.1.3 MultipeerConnectivity

Das von Apple zur Verfügung gestellte Framework *MultipeerConnectivity* ermöglicht die direkte Kommunikation zwischen zwei oder mehreren iOS-Geräten auf Basis von Bluetooth und WLAN. [Apple, 2013] Kommuniziert wird generell über eine `MCSession`, wobei jeder Teilnehmer anfänglich eine `MCPeerID` bekommt und sich selbst einen `displayName` zuweist. Grundsätzlich kann eine Session mehr als zwei Teilnehmer verarbeiten. Durch die zusätzliche



5 Implementierung

Schicht der RSA-Verschlüsselung ist jedoch zum momentanen Zeitpunkt nur die Kommunikation zwischen zwei Geräten möglich. Grundsätzlich besitzt das verwendete Netzwerkprotokoll auch eine Möglichkeit der Verschlüsselung, durch den RSA-Zusatz ist aber ein Mitlesen der Nachrichten seitens einer dritten Instanz komplett unterbunden. Die Session kann von jedem der beiden Teilnehmer initiiert werden. Der jeweils andere muss die Session suchen und ihr beitreten. Der Initiator muss diesen Verbindungsauflauf bestätigen. Dies ist ein weiterer Sicherheitsmechanismus, da alle Teilnehmer wissen, wer gerade verbunden ist.

Um eine Verbindung erfolgreich aufzubauen, müssen zuerst die Parameter gesetzt und die benötigten Verbindungsstrukturen initialisiert werden (vgl. Codebeispiel 5.9).

```
1 //import Framework
2 import MultipeerConnectivity
3 // set necessary parameters
4 var peerID = MCPeerID(displayName: UIDevice.current.name)
5 var mcSession = MCSession(peer: peerID,
6                           securityIdentity: nil,
7                           encryptionPreference: .required)
8 var mcAdvertiserAssistant = MCAdvertiserAssistant(serviceType: "adorsys",
9                                                   discoveryInfo: nil,
10                                                 session: mcSession)
```

Quellcodebeispiel 5.9: Initialisierung der benötigten Verbindungsparameter

Anschließend muss eine Partei die Session mit `mcAdvertiserAssistant.start()` starten. Es wird eine Session erstellt, die von allen Geräten in der näheren Umgebung, welche einen MCAdvertiserAssistant mit denselben serviceType besitzen, gefunden werden kann. Alle anderen Teilnehmer müssen dieser Session beitreten. Dafür gibt es einen von Apple bereitgestellten Auswahlbildschirm, der diese Funktionalität abbildet. Er wird in Abbildung 5.1 a) dargestellt. Die Funktionalität der manuellen Bestätigung durch den Initiator zum Aufbau der Verbindung, dargestellt in Abbildung 5.1 b), ist ebenfalls im Framework enthalten. Abbildung 5.1 c) zeigt die Oberfläche zum Versenden von Schlüsselteilen nach erfolgreicher Verbindung zweier Geräte.

```
1 //host session
2 mcAdvertiserAssistant.start()
3 //show session searching viewcontroller
4 let mcBrowser = MCBrowserViewController(serviceType: "adorsys",
5                                         session: mcSession)
6 mcBrowser.delegate = self
7 present(mcBrowser,
8         animated: true)
```

Quellcodebeispiel 5.10: Programmcode zum Initialisieren einer Session bzw. Aufrufen des Auswahlfensters



5 Implementierung

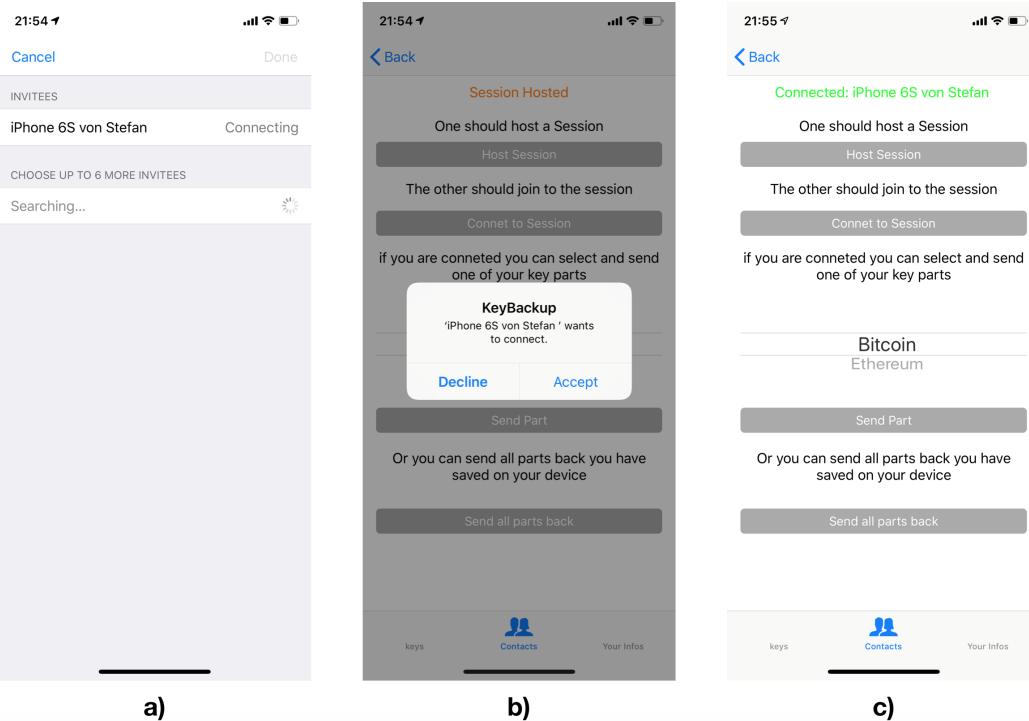


Abbildung 5.1: Beispiel der verschiedenen Status der MultipeerConnection: (a) Gerätesuchvorgang, (b) Verbindungsbestätigung und (c) Ready-To-Send-State

Quelle: eigene Darstellung

Gesendet werden Daten mit Hilfe der Instanz der `McSession`. Durch die Übergabevariable `toPeers` wird festgelegt, wem die Daten übermittelt werden sollen. Dadurch wäre theoretisch die Verbindung mit mehr als zwei Personen zeitgleich möglich, wobei alle Kommunikationsschlüssel dem Sender vorliegen müssen, beziehungsweise eine exakte Verknüpfung zwischen der `MCPeerID` und dem Schlüssel bestehen muss, um die korrekte Übertragung zu gewährleisten.

```
1 let data = "Message".data(.utf8)
2 //send data
3 do {
4     try mcSession.send(data,
5                         toPeers: mcSession.connectedPeers,
6                         with: .reliable)
7 } catch{
8     fatalError("Could not send keypart")
9 }
10 //receive data
11 func session(_ session: MCSession,
12              didReceive data: Data,
13              fromPeer peerID: MCPeerID) {
14     //connect to main thread for UI access
15     DispatchQueue.main.async {
16         //print received data
17     }
18 }
```



5 Implementierung

```
17     print(data)
18 }
19
20 }
```

Quellcodebeispiel 5.11: Programmcode zum Senden und Empfangen von Daten

Da ungewiss ist, wann und ob Daten eintreffen, ist es nicht sinnvoll, auf eine Nachricht zu warten und es wäre sehr leistungsintensiv, ständig zu überprüfen, ob eine Nachricht vorliegt. Daher wird das Empfangen von Daten durch einen *callback* verarbeitet. Dies ist ein vom Hauptprogrammablauf unabhängiger Programmcode, der dann aufgerufen wird, wenn ein spezielles Ereignis eintritt. In diesem Fall ist dies das Eintreffen eines Datensatzes. Da in iOS nur der Hauptprogrammstrang, der *main thread*, die Möglichkeit besitzt, das User Interface (UI) zu aktualisieren, muss zuerst aus der Nebenläufigkeit auf diesen *main thread* gewechselt werden, um dem Nutzer das Eintreffen mitzuteilen und das UI anzupassen.

Durch den direkten Zugriff auf die Sende- und Empfangsfunktionen, ist es in diesem Kommunikationsmodus möglich, automatisch eine Rückmeldung an das Gerät des Senders zu übertragen und auf dem UI in Form eines *toast*¹ anzuzeigen. Diese Statusdaten werden nicht zusätzlich durch RSA verschlüsselt. Einerseits enthalten sie keine sensiblen Informationen und andererseits könnte bei fehlerhafter Verschlüsselung der Statuscode nicht gelesen werden. Dies ist ein großer Vorteil im Gegensatz zum Versenden der Daten über das dokumentbasierte Verfahren, denn der Sender kann durch die Rückmeldung mit Gewissheit sagen, dass die Informationen richtig übertragen und verarbeitet wurde. Diese Art der Rückmeldung ist bei der dateigestützen Übertragung nicht denkbar, da es das iOS-Betriebssystem nicht zulässt, E-Mails, Nachrichten oder ähnliches ohne die explizite Zustimmung des Nutzers zu versenden. Der einzige Weg wäre, dass die Backupapplikation über die bereits erwähnten Push-Nachrichten die App auf dem Gerät des Senders informiert und diese wiederum den Geheimnisinhaber. Dies wäre aber erneut eine dokumentlose Datenübertragung.

5.1.4 iCloud Drive Synchronisation

Dieses Kapitel widmet sich der Umsetzung der in Kapitel 4.2.2 erwähnten iCloud-Funktionalität. Die Synchronisation über den Apple eigenen Clouddienst besitzt den Vorteil, dass der Netzwerkordner ähnlich einem lokalen Dateipfad verwendet werden kann. Die Synchronisation erfolgt automatisch durch das Betriebssystem. Allerdings muss im Apple Developer Portal [Apple, 2019] erst ein Datencontainer angelegt und mit der App-ID verknüpft werden. Anschließend muss der Name des Containers in der *info.plist*² der App eingetragen werden. Erst dann kann,

¹Ein *toast* ist ein kleiner Textoverlay, der nach kurzer Zeit wieder ausgeblendet wird

²Die *info.plist* ist eine Datei, die Umgebungsvariablen der App, wie die AppID und verwendete Hardwarekomponenten enthält



5 Implementierung

nach erfolgreicher Anmeldung im Betriebssystem, ein Ordner in iCloud Drive erstellt und hier Daten gelesen beziehungsweise geschrieben werden. Abbildung 5.2 zeigt einen beispielhaften Inhalt von *iCloud Drive* mit dem erstellten und verwendeten Ordner *p2pCryptoContainer*.

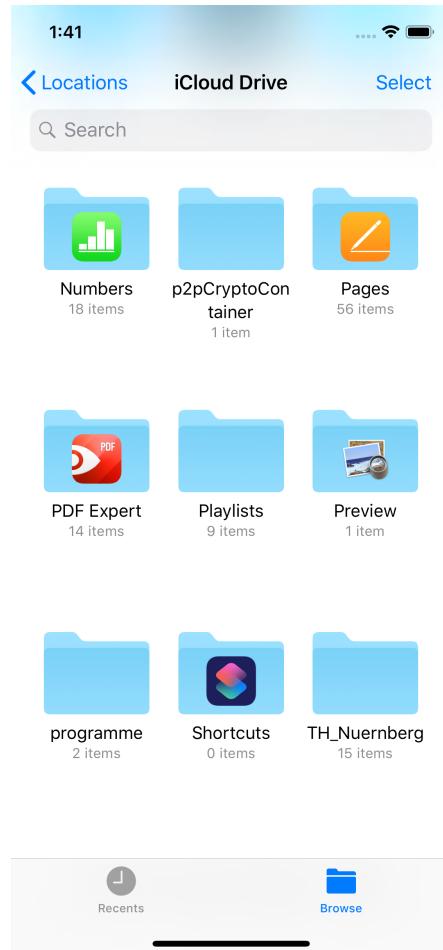


Abbildung 5.2: Beispielhafter Inhalt von iCloud Drive mit dem erstellten *p2pCryptoContainer*

Quelle: eigene Darstellung

```
1 {
2     "iCloudKeyPartBuddyUsages": [
3         {
4             "linkedBuddies": [
5                 {
6                     "nickname": "Simulator",
7                     "keyNumber": 2
8                 },
9                 {
10                     "nickname": "iPhone XS",
11                     "keyNumber": 5
12                 }
13             ]
14         }
15     ]
16 }
```



5 Implementierung

```
12     }
13     ],
14     "thresholdParts": 3,
15     "usage": "Bitcoin",
16     "keyContainerID":
17     "KeyPartContainer_9324E9B3-3435-412A-9430-2AE729AF1695"
18   }
19 ],
20 "creationDate": 567782045.89666903
21 }
```

Quellcodebeispiel 5.12: Beispielhafter Inhalt der *Buddies* Datei

Codebeispiel 5.12 zeigt einen beispielhaften Inhalt der in iCloud Drive gespeicherten Datei. Sie enthält die *ContainerID*, den Verwendungszweck, den Schwellenwert und ein Array von *Buddies*, die von diesem Container einen Teil besitzen.

```
1 // needed constances
2 let filemgr = FileManager.default
3 let iCloudDocumentsURL = filemgr
4   .url(forUbiquityContainerIdentifier:nil)?
5   .appendingPathComponent("Documents")
6
7 func writeBuddyFileToCloud(fileName:String) throws {
8   //create Folder if it does not exists
9   if !(filemgr.fileExists(atPath: iCloudDocumentsURL.path)) {
10     try! filemgr.createDirectory(at: iCloudDocumentsURL,
11                               withIntermediateDirectories: true,
12                               attributes: nil)
13   }
14   // create Path
15   let iCloudBuddyFile = iCloudDocumentsURL
16     .appendingPathComponent(fileName)
17     .appendingPathExtension("buddies")
18   //create data representation for storing
19   let data = "data that should be stored in iCloud folder"
20     .data(using: .utf8)
21   // try to write data to file
22   try data.write(to:iCloudBuddyFile,
23                 options: [.completeFileProtection])
24 }
```

Quellcodebeispiel 5.13: Programmcode für einen Schreibzugriff auf den iCloud Ordner

Durch die Benutzung eines iCloudContainers kann ein Schreibzugriff wie auf einen lokal verfügbaren Ordner erfolgen. Es muss nur der iCloudpfad des Containers angegeben werden. Das Betriebssystem synchronisiert den Inhalt des Ordners automatisch. Der eigentliche Schreibvorgang passiert durch `try data.write(to:...)`, (vgl Programmcode 5.13). Das `try` wird benötigt, da ein derartiger Programmaufruf, etwa durch einen inkorrekten Pfad, fehlschlagen könnte.



Der Lesevorgang ist geringfügig komplizierter, denn hier ist es möglich, dass die eigentliche Datei noch nicht vorhanden ist, sondern nur ein Verweis. Die Datei müsste erst noch heruntergeladen werden. Dies ist vor allem dann möglich, wenn die Datei lange Zeit nicht benötigt wurde. Sie wird dann zur Optimierung des Speicherverbrauchs gelöscht und muss bei Bedarf neu heruntergeladen werden.

```
1 // needed constances
2 let filemgr = FileManager.default
3 let iCloudDocumentsURL = filemgr
4     .url(forUbiquityContainerIdentifier:nil)?
5     .AppendingPathComponent("Documents")
6
7 // cancel if iCloudPath doesn't exists
8 if !(filemgr.fileExists(atPath: iCloudDocumentsURL.path)) {
9     return
10 }
11 // create path for reading
12 iCloudBuddyFile = iCloudDocumentsURL
13     .AppendingPathComponent(fileName)
14     .AppendingPathExtension("buddies")
15
16
17 if !(filemgr.fileExists(atPath: iCloudBuddyFile.path)) {
18     //if file doesn't exist but the link does try to download the file
19     let iCloudFileForDownload = iCloudDocumentsURL
20         .AppendingPathComponent("." + fileName)
21         .AppendingPathExtension("buddies")
22         .AppendingPathExtension("icloud")
23
24     if filemgr.fileExists(atPath: iCloudFileForDownload.path) {
25         if filemgr
26             .changeCurrentDirectoryPath((localDocumentsURL?.path)!) {
27                 try filemgr
28                     .startDownloadingUbiquitousItem(at:
29                         iCloudFileForDownload)
30             }
31         }
32     }
33 // acutal read data from file
34 guard let iCloudData = try readFromFile(url: iCloudBuddyFile)
35     else {return nil}
```

Quellcodebeispiel 5.14: Programmablauf zum Lesen von Daten aus einer Datei in iCloud Drive

5.2 Verschlüsselung in Swift

Das iOS-Betriebssystem stellt Funktionen zur Erstellung und sicheren Aufbewahrung von zufälligen Kryptographieschlüsseln bereit. Es werden zur Verschlüsselung und Signierung von Daten eine Vielzahl von Algorithmen unterstützt. Dabei sind symmetrische und asymmetrische Verfahren möglich und auch solche, bei denen der Schlüssel auf elliptischen Kurven ba-



5 Implementierung

siert. In dieser Arbeit wird der `SecKeyAlgorithm.rsaEncryptionOAEP SHA512AESGCM` Algorithmus verwendet. Dieser verwendet eine Kombination aus asymmetrischer und symmetrischer Verschlüsselung, um jede Art und Größe an Datensätzen zu verschlüsseln. Den einen Teil bietet der RSA-Algorithmus mit der Erweiterung Optimal Asymmetric Encryption Padding (OAEP). Dies verbessert die Sicherheit von RSA hinsichtlich Angriffsszenarien, bei denen der Angreifer durch einen bekannten Klar- und dem zugehörigen Chiffre-Text versucht, den Schlüssel zu rekonstruieren. Den Teil der symmetrischen Verschlüsselung übernimmt der AES Algorithmus im Galois/Counter Mode (GCM). Dieser Betriebsmodus ist auf einen hohen Datendurchsatz sowie eine hohe Performance ausgelegt. Außerdem ist es durch die Anwendung einer Blockchiffre möglich, Daten zu verschlüsseln, welche größer als der eigentliche Schlüssel sind. Dabei werden die Daten in Teile zerlegt, einzeln verschlüsselt und nummeriert, um die Rekonstruktion zu ermöglichen. Daher auch der Name *Counter*-, zu deutsch Zähler-, Modus. Symmetrische Verschlüsselung ist sehr performant und kann nahezu jede Datengröße verarbeiten. Die asymmetrische Art der Verschlüsselung hingegen hat das Problem des sicheren Schlüsselaustausch größtenteils gelöst. Aus diesen Gründen werden diese beiden Arten zu dem verwendeten Hybridverschlüsselungsverfahren kombiniert. Dabei besteht die verschlüsselte Datei aus zwei Abschnitten. Als erstes wird ein *session key* erstellt. Das ist ein Schlüssel für die symmetrische Verschlüsselung. Dieser wird ausschließlich zur Verschlüsselung dieser einen Datei verwendet. Dann wird der eigentliche Inhalt mit diesem Schlüssel unlesbar gemacht. Anschließend wird der *session key* durch den RSA-Algorithmus mit dem öffentlichen Kommunikationsschlüssel des Empfängers verschlüsselt und an die Nachricht angehängt. Der Empfänger entschlüsselt mit seinem *private key* den *session key* und kann dadurch den Inhalt lesen. Dieser Algorithmus verarbeitet den Datensatz direkt und erstellt jeweils bei Ver- und Entschlüsselung einen neuen. Aus diesem Grund kann diese Art der Verschlüsselung für die beiden in Kapitel 2.4 erwähnten Übertragungsmethoden verwendet werden. Denn der neue Datensatz kann sowohl direkt übertragen als auch in einer Datei gespeichert werden, die dann über beispielsweise E-Mail übertragen wird. Im Beispiel 5.15 wird der Inhalt einer Schlüsselteil im Klartext dargestellt. Beispiel 5.16 zeigt den selben Inhalt nach der Verschlüsselung.

```
1 {
2   "KeyPartContainerID": "KeyPartContainer_C09E3A6A-D25F-45EB-B774-DE4DB3B2AAC4",
3   "SenderCommKey": "MIICCg ... CAwEAQ==",
4   "partsThreshold": 3,
5   "partNumber": 2,
6   "fromOwner": true,
7   "transferDate": 568118479.16380095
8   "keyPart": "AoOZqb24im9Lg/GpXn/JFyB71/o1+WHvia5QgXnPts/i"
9 }
```

Quellcodebeispiel 5.15: Darstellung einer .abf Datei im Klartext



5 Implementierung

```
1 00000000 0d 9b 6f b6 a4 02 97 69 4c f5 95 8d 5f b5 16 64 | ..o....iL...._..d
2 0000010 51 b9 c8 76 2a f3 03 d4 77 35 cf c6 61 39 b0 47 | Q..v*....w5..a9.G
3 0000020 c8 2a 1c 59 44 f2 7b 8b 9c ba 64 21 ea 3d f4 0c | .*..YD.{....d!..=..
4 0000030 e2 8b 65 cc 17 b7 43 f8 ae a4 32 78 ff e4 73 33 | ..e....C....2x...s3
5 0000040 35 a6 a9 38 75 41 5f a8 8b 8e f2 79 7d 68 55 ce | 5..8uA.....y}hU.
6 0000050 6f 9a 28 b7 4f e2 b9 74 30 8d 6e d0 9f 9d 20 a7 | o.(.0..t0.n.... .
7 0000060 ff 3e 58 35 66 56 a5 9a 62 ab 9c 51 d4 52 61 24 | .>X5fV..b..Q.Ra$.
8 0000070 06 eb 99 9e 13 c4 3a 29 7a 90 8c bf e3 00 f4 e6 | .....:)z.....
9 0000080 48 29 68 32 f3 76 87 23 fe 07 78 a3 1b 5b a3 1d | H)h2.v.#..x...[...
10 ...
11 ...
12 00005c0 a5 96 83 4f 39 a6 97 2d ee cb 6a f5 6f f5 e9 84 | ...09....j.o...
13 00005d0 e6
14 00005d1
```

Quellcodebeispiel 5.16: Ausschnitt einer verschlüsselten .abf Datei

Vor der Kommunikation muss das erwähnte Kommunikationsschlüsselpaar erstellt werden. Dabei ist es gängige Praxis, ausschließlich den privaten Schlüssel abzuspeichern und den öffentlichen Schlüssel aus dem privaten zu generieren, sobald er benötigt wird. Erstellt wird der private Teil, indem man der Funktion `SecKeyCreateRandomKey(...)` einen bestimmten Satz an Parametern übergibt. Diese bestehen unter anderem aus der Schlüsselart, der Länge und dem *Tag*³ des Schlüssels. Für den Ladevorgang aus der *KeyChain* wird ein ähnlicher Satz an Schlüsselparametern benötigt. Beispiel 5.17 zeigt die in dieser App verwendeten Parametersätze zum Erstellen und Laden des Kommunikationsschlüssels.

```
1 //parameter set for creating a key
2 var keychainCreateDictionary: [String:Any] {
3
4     let privateKeyAttr: [String : Any] = [
5         kSecAttrIsPermanent as String: true , // Store in KeyChain
6         kSecAttrApplicationTag as String: tagPrivateKey] // KeyChain Tag
7
8     let keyPairAttr: [String : Any] = [
9         kSecAttrKeyType as String: kSecAttrKeyTypeRSA,
10        kSecAttrKeySizeInBits as String: 4096,
11        kSecPrivateKeyAttrs as String: privateKeyAttr]
12
13     return keyPairAttr
14 }
15 //parameter set for loading a key from the KeyChain
16 var keychainLoadDictionary: [String:Any] {
17
18     let privateKeyAttr: [String : Any] = [
19         kSecAttrKeyType as String: kSecAttrKeyTypeRSA,
20         kSecClass as String: kSecClassKey,
21         kSecReturnRef as String: true,
22         kSecAttrApplicationTag as String: tagPrivateKey]
23
24     return privateKeyAttr
25 }
```

³eine alphanumerische Zeichenkette, die das Laden des Schlüssels aus der *KeyChain* ermöglicht



5 Implementierung

Quellcodebeispiel 5.17: Parametersätze zum Erstellen und Laden des Kommunikationsschlüssels

Das eigentliche Erstellen beziehungsweise das Laden des Schlüssels aus der *KeyChain* und das anschließende Berechnen des öffentlichen Schlüsselteils wird in Codebeispiel 5.18 gezeigt.

```
1 let privateKey:SecKey?
2 //Read Keychain if there is already a Key with this tag
3 var resultPrivateKey: CFTyperef?
4 let statusPrivateKey = SecItemCopyMatching(keychainLoadDictionary
5                                     as CFDictionary,
6                                     &resultPrivateKey)
7 var error: Unmanaged<CFError>?
8     //if loading was sucessful
9 if(statusPrivateKey == noErr){
10     privateKey = resultPrivateKey as! SecKey
11 } else {
12     //if loading wasn't successful create a key
13     privateKey = SecKeyCreateRandomKey(keychainCreateDictionary
14                                         as CFDictionary,
15                                         &error) ?? nil
16 }
17 //create publicKey from privateKey
18 let publicKey = SecKeyCopyPublicKey(privateKey) ?? nil
```

Quellcodebeispiel 5.18: Erstellen und Laden des *private keys* aus der *KeyChain* und anschließendes Generieren des *public keys*

Anschließend wird die jeweilige Verschlüsselungsfunktion beziehungsweise Entschlüsselungsfunktion, wie in Beispiel 5.19 dargestellt, aufgerufen.

```
1 //encrypt data
2 var error: Unmanaged<CFError>?
3 let cipherText = SecKeyCreateEncryptedData(publicKey,
4                                             algorithm,
5                                             dataToEncrypt as CFData,
6                                             &error) as Data?
7
8 //decrypt data
9 let clearText = SecKeyCreateDecryptedData(privateKey,
10                                            algorithm,
11                                            dataToDecrypt as CFData,
12                                            &error) as Data?
```

Quellcodebeispiel 5.19: Ver- und Entschlüsselungsfunktion



5.3 Import und Export von Dateien

Sollen bestimmte Datentypen in eine iOS-Applikation importiert oder daraus exportiert werden, müssen diese zuerst in der bereits genannten *info.plist* eingetragen werden. Dabei sind die zu importierenden Datentypen unabhängig von den zu exportierenden. Angegeben werden die Typen als *Uniform Type Identifier (UTI)*. Dies sind standardisierte Dateiarten, die sehr häufig verwendet werden und meistens an der Dateiendung zu erkennen sind wie beispielsweise *.pdf* und *.jpg*. Werden diese UTIs gesetzt, suggeriert die Applikation dem Betriebssystem, dass es diese Dateiart verarbeiten kann. Es ist auch möglich, eigene UTIs zu definieren. Dazu müssen vier Dinge definiert werden:

- **Description**

Sie gibt an, wofür die darin enthaltenen Daten verwendet werden.

- **Identifier**

Der Identifier ist eine Zeichenkette, die den Dateityp eindeutig definiert und für jeden Datentyp einzigartig ist.

- **Conforms To**

Dieser Parameter gibt an welche Art an Daten enthalten sind und wie die Daten gelesen bzw. geschrieben werden sollen.

- **Dateiendung**

Dieser Wert legt die Dateiendung fest, die dieser Dateityp trägt.

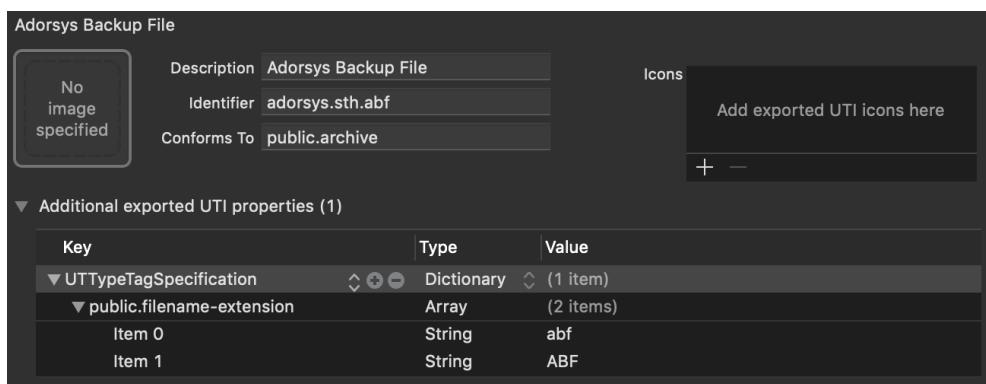


Abbildung 5.3: Definition der .abf-Datei als UTI in Xcode

Quelle: eigene Darstellung

Der **Export** von Dateien findet nahezu ausschließlich über den in Kapitel 4.3.1 und in Abbildung 4.4 gezeigten, iOS internen Exportcontroller statt. Dabei kann annähernd jede Datei exportiert werden. Zusätzlich ist der Export nicht auf eine einzelne Datei beschränkt, sondern es können ganze Datensätze verteilt werden. Durch das Exportfenster gibt es eine Vielzahl an



Möglichkeiten, mit dieser Datei weiter zu verfahren wie beispielsweise das Versenden auf diverse Arten, die Speicherung oder der Import in andere Programme. Das Betriebssystem stellt diese Methoden zur Verfügung und sorgt gleichzeitig für deren korrekte Durchführung. Der `UIActivityViewController` kann mit einer Reihe von Parametern an die Bedürfnisse des Programmierers angepasst werden. Dabei müssen die unerwünschten Exportmethoden explizit verboten werden. Alle ungenannten sind standardmäßig aktiviert. In dem hier vorgestellten Programm können sowohl Bilddateien als auch `.abf`-Dateien exportiert werden. Dabei können die Bilddateien zusätzlich zu der Behandlung als reines Dokument in der Fotobibliothek des iPhones gespeichert werden. Mit den `.abf` Dateien ist das nicht möglich.

```
1 //prepare data to share
2 let message = "Data to share via iOS sharing screen".data(using: .utf8)
3
4 //write data to file
5 path = try? writeToFile(data: message, fileName: "file.abf")
6 //instantiate the sharing view
7 let activityViewController =
8     UIActivityViewController(activityItems: [path],
9                             applicationActivities: nil)
10 // iPad compatibility
11 activityViewController.popoverPresentationController?.sourceView = self.view
12
13 // exclude some activity types from the list
14 activityViewController.excludedActivityTypes = [.postToFacebook,
15                                               .postToTwitter,]
16 // present the view controller
17 self.present(activityViewController, animated: true, completion: nil)
```

Quellcodebeispiel 5.20: Aufruf des iOS-Exportfensters

Die zwei verwendeten Datentypen können wiederum auf verschiedene Art und Weisen **importiert** werden. Um Dateien in Programmen zu öffnen, bietet iOS zwei grundsätzliche Optionen. Bei der *copy to*-Methode wird die Applikation gestartet und die Dateien werden als Parameter übergeben. Dabei müssen sie während des Startvorgangs in einer speziellen Funktion verarbeitet werden. Diese Vorgehensweise wird in Codebeispiel 5.21 dargestellt. Ansonsten ist der Zugriff zu einem späteren Zeitpunkt nicht möglich. Die zweite Möglichkeit zum Import von Dateien ist die *sharing extension*. Sie stellt ein quasi eigenständiges Programm dar, welches sehr beschränkte Möglichkeiten zur Verarbeitung von Daten bietet, jedoch losgelöst von der eigentlichen Applikation lauffähig ist. Sie eignet sich hervorragend, um beispielsweise eine Datei in einem speziellen Speicherpfad auf einer Onlineplattform abzulegen. Soll beispielsweise ein Bild in einem sozialen Netzwerk geteilt werden, können so sehr einfach und schnell Parameter hinzugefügt werden, ohne die eigentliche App zu starten und alle Beiträge zu laden. Diese Art des Imports wird in der vorliegenden Applikation nicht verwendet. Beide Verfahren werden durch dasselbe Auswahlfenster des Exports getätigter. Abbildung 4.6 zeigt eine beispielhafte Darstellung.



5 Implementierung

```
1 //os function called when imported a File
2 func application(_ app: UIApplication,
3                   open inputURL: URL,
4                   options: [UIApplication.OpenURLOptionsKey : Any] = [:])
5                   -> Bool {
6
7     // set initial viewcontroller
8     guard let viewController = window?.rootViewController
9         as? RootViewController else {
10         fatalError("The root view is not a document browser!")
11     }
12
13    if(inputURL.pathExtension == "png"){
14        print("Picture input found")
15        //read file content
16        let data = try? Data(contentsOf: inputURL)
17        // decode picture data and extract key
18        let image = UIImage(data: data!)
19        let incomingKeyData = parseImage(image: image)
20        return true
21    }
22
23    if(inputURL.pathExtension == "abf"){
24        print("Backup File found")
25
26        let fileData = try? readFromFile(url: inputURL)
27        // decrypt Data with private key and parse incoming Data
28        if let decryptedData = decryptData(dataToDecrypt: fileData!) {
29            parseIncomingKeyData(fileData: decryptedData)
30        }
31    }
32    return false
33 }
```

Quellcodebeispiel 5.21: Programmcode zur Verarbeitung von importierten Dateien

Für die in QR-Codes dargestellten öffentlichen Kommunikationsschlüssel stehen aufgrund der Speicherung als Bilddatei weitere Funktionen zur Verfügung. Die Daten können zusätzlich in dem Fotoalbum des iOS Geräts gespeichert und über die für Bilder typischen Verteilungsmethoden wie *geteilte Fotoalben* oder *iCloud Sharing Links* geteilt werden. Ebenso kann das hier beschriebene Programm die Bilder aus der Bibliothek laden und weiterverarbeiten. Einen Spezialfall stellt das Scannen eines solchen QR-Codes durch die smartphoneinterne Kamera dar. Hierbei müssen keine Daten gespeichert oder versendet werden, da das Betriebssystem die Daten während des Scannens dekodieren kann. Dies macht die Übertragung einfach, eindeutig und sicher. Abbildung 4.5 zeigt die programminterne QR-Codescanningoberfläche. Das Scannen des QR-Codes aus einem Livebild der Kamera kann in mehrere Schritte aufgeteilt werden. Zuerst muss mittels einer `AVCaptureSession` Zugriff auf die Kamera erlangt werden. Anschließend wird das Bild in Verbindung mit einem `AVCaptureMetadataOutput` ausgewertet.



5 Implementierung

```
1 // build and show scanning view
2 generateScanningView()
3 // instantiate session for video caption
4 let avCaptureSession = AVCaptureSession()
5 // check if device has a camera and app can access it
6 guard let avCaptureDevice = AVCaptureDevice.default(for:
7                                     AVMediaType.video)
8     else {
9         print ("no camera")
10    return
11 }
12
13 guard let avCaptureInput = try? AVCaptureDeviceInput(device:
14                                     avCaptureDevice)
15     else {
16         print("Failed to init camera")
17     return
18 }
19 // instance for search for meta data during scan
20 let avCaptureMetadataOutput = AVCaptureMetadataOutput()
21
22 //select thread and delegate
23 avCaptureMetadataOutput
24     .setMetadataObjectsDelegate(self,
25         queue: DispatchQueue.main)
26 //connect camera with metadata searching action
27 avCaptureSession.addInput(avCaptureInput)
28 avCaptureSession.addOutput(avCaptureMetadataOutput)
29 //looking for QR-Codes
30 avCaptureMetadataOutput.metadataObjectTypes =
31 [AVMetadataObject.ObjectType.qr]
32 //set preeview frame
33 let avCaptureVideoPreviewLayer = AVCaptureVideoPreviewLayer(session:
34                                     avCaptureSession)
```

Quellcodebeispiel 5.22: Zugriff auf die Smartphonecamera zum Scannen eines QR-Codes

Wird ein QR-Code erkannt und die Information erfolgreich dekodiert, werden die Daten als *public key* interpretiert. Dann kann entweder ein neuer *Buddy* angelegt oder der Schlüssel eines bestehenden erneuert werden.

```
1 //os callback when metadata found in live camera image
2 func metadataOutput(_ output: AVCaptureMetadataOutput,
3                     didOutput metadataObjects: [AVMetadataObject],
4                     from connection: AVCaptureConnection) {
5     // Get the metadata object.
6     let metadataObj = metadataObjects[0]
7         as! AVMetadataMachineReadableCodeObject
8     //if metadata is from type QR
9     if metadataObj.type == AVMetadataObject.ObjectType.qr,
10         metadataObj.stringValue != nil {
11         //decode String to data
12         if let publicKeyIncomingdata = Data(base64Encoded:
13                                         metadataObj.stringValue!) {
14             //send decoded key to caller
```



5 Implementierung

```
15         self.dismiss(animated: true) {
16             self.delegate.
17             receiveKeyDataFromImportView(key: publicKeyIncomingdata)
18         }
19     }
20     avCaptureSession.stopRunning()
21 }
22
23 }
```

Quellcodebeispiel 5.23: *Callback* Funktion zur Verarbeitung der erkannten Metadaten



6 User Interface

In diesem Kapitel soll das User Interface des Sicherungsprogramms kurz vorgestellt werden. Sie ist generell in drei Segmente unterteilt, durch die mittels der *Tabbar* am unteren Bildschirmrand navigiert werden kann. Die drei Abschnitte sind:

- **Key Management**

Hier werden die gespeicherten Schlüssel aufgelistet und deren Details können eingesehen werden.

- **Buddies**

An dieser Stelle können *Buddies* hinzugefügt beziehungsweise mit gespeicherten Profilen interagiert werden.

- **eigene Informationen**

Hier wird der persönliche Kommunikationsschlüssel als QR-Code angezeigt.

Im Folgenden wird etwas genauer auf die jeweiligen Abschnitte eingegangen. Abbildung 6.1 zeigt das Icon für das Programm. Der sich aus vielen Einzelteilen zusammensetzende stilisierte Schlüssel ist eine Hommage an die zentrale Funktionalität des Programms, den Shamir's Secret Sharing Scheme (SSSS).



Abbildung 6.1: Icon für die Backupapplikation

Quelle: eigene Darstellung

6.1 Key Management

Abbildung 6.2 zeigt eine beispielhafte Darstellung der *Key Management Section* des Programms. Dieser Screen stellt den Eintrittsbildschirm dar. Die Tabelle ist in zwei Abschnitte unterteilt. Die obere zeigt die eingegebenen Verwendungszwecke der lokal gespeicherten



6 User Interface

Schlüssel an, wobei die Graphik auf der rechten Seite den Status des Backups anzeigt. Dabei sind die drei gezeigten Fälle möglich. Hierbei steht grün dafür, dass alle Teile dieses Schlüssels an Vertrauenspersonen verteilt wurden. Sind zwei Kreise orange, übersteigt die Zahl der aktuell verteilten *shares* den gesetzten Schwellenwert, jedoch sind noch Teile übrig, die nicht verteilt wurden. Sollte ein Kreis rot dargestellt werden, ist der Schwellenwert noch nicht erreicht. Das Backup ist noch nicht wiederherstellbar. Der zweite Abschnitt der Tabelle stellt die Verwendungszecke der Geheimnisse dar, deren Teile sich nicht oder noch nicht auf diesem Gerät befinden. Hier sollen im Fall, dass eine Wiederherstellung nötig ist, die in iCloud gespeicherten Daten angezeigt werden. Über die einzelnen Zellen hat der Benutzer Zugriff auf die gesammelten Schlüsselteile des betreffenden Schlüssels. Solange die Anzahl der importierten Schlüsselfragmente unterhalb des Schwellenwerts liegen, wird ein roter Kreis angezeigt. Wird er überschritten, färben sich die Kreise grün und das Geheimnis kann wiederhergestellt werden. Die Farbe orange tritt in diesem Abschnitt der Tabelle nie auf. Durch das Tippen auf einen Eintrag gelangt man auf die jeweilige Detailansicht mit näheren Informationen. Durch Antippen des Plussymbols in der oberen rechten Ecke bekommt man die Möglichkeit ein Geheimnis hinzuzufügen.

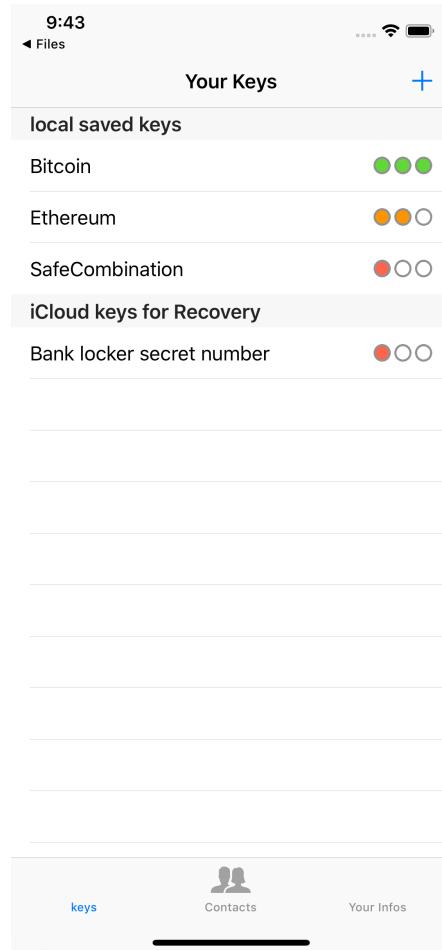


Abbildung 6.2: Beispielhafter *Key Management Screen* mit diversen Keys

Quelle: eigene Darstellung

6.2 Vertrauenspersonen

Der zweite Abschnitt wird in Abbildung 6.3 dargestellt und ist ebenfalls eine Tabelle. Sie beinhaltet alle Profile der Vertrauenspersonen. Jedes Individuum wird durch seinen *Nickname* und ein Profilfoto angezeigt. Sollte kein Profilfoto vergeben worden sein, tritt ein Platzhalter an dessen Stelle. Wird ein Eintrag berührt, öffnet sich die Seite mit den Details des jeweiligen *Buddies*. Hier können Informationen über an diesen *Buddy* ausgegebene beziehungsweise von diesem erhaltene Schlüsselteile abgerufen werden. Zusätzlich lassen sich die Parameter der Person ändern. Das Plus-Symbol auf der Hauptseite der Vertrauenspersonen bietet die Möglichkeit, neue Profile durch Scannen oder Import des Kommunikationsschlüssels anzulegen.

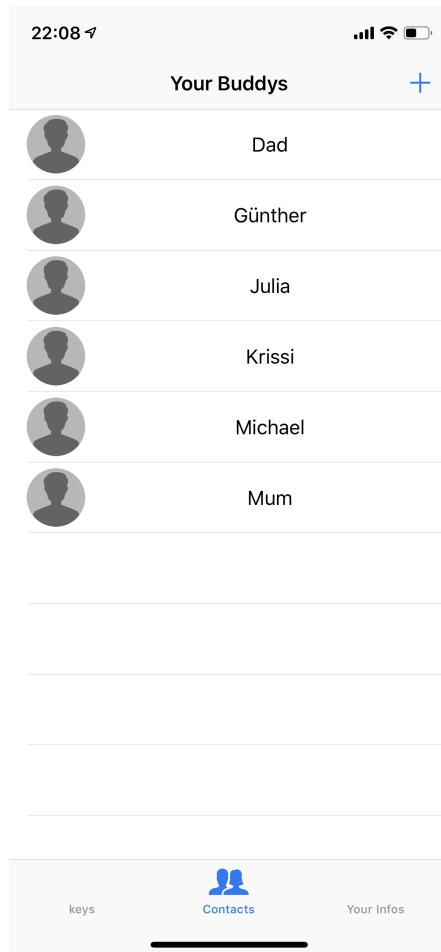


Abbildung 6.3: Beispielhafte Auflistung von Vertrauenspersonen

Quelle: eigene Darstellung

6.3 Persönliche Informationen

Auf der dritten Hauptseite des Programms, vgl. Abbildung 6.4, wird der eigene öffentliche Kommunikationsschlüssel als QR-Code dargestellt. Dieser kann entweder als Datei versendet oder mittels Kamera abgelesen werden. Zusätzlich besteht die Möglichkeit, das Schlüsselpaar für die Kommunikation zu erneuern. Diese Funktion sollte mit Bedacht genutzt werden, da alle Vertrauenspersonen, mit denen kommuniziert werden soll, den neuen Schlüssel importieren müssen, bevor eine erfolgreiche Kommunikation möglich ist.



6 User Interface

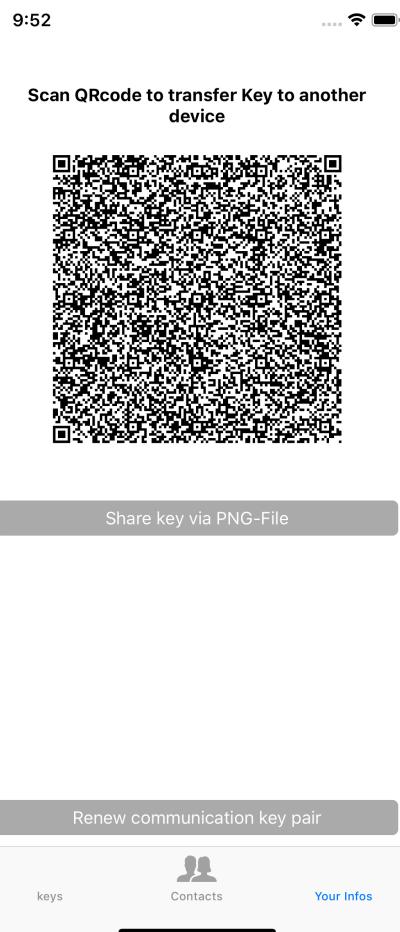


Abbildung 6.4: Persönliche Informationen

Quelle: eigene Darstellung



7 Fazit und Ausblick

Abschließend soll aus den Ergebnissen der vorangegangenen Kapitel ein Fazit gezogen und ein Ausblick auf zukünftige Erweiterungen der Applikation gegeben werden.

7.1 Fazit

Zusammenfassend lässt sich sagen, dass es möglich ist, den *private key* einer Kryptowährung beziehungsweise jedes datenbasierte Geheimnis dezentral zu speichern. Der Shamir's Secret Sharing Scheme-Algorithmus zeichnet sich durch Einfachheit, Sicherheit und Flexibilität aus und ist deshalb für die Erzeugung der Schlüsselteile und die anschließende Wiederherstellung des Geheimnisses hervorragend geeignet. Da die verwendete *library* mit grundlegenden Datenbausteinen (*bytes*) arbeitet, ist diese in der Lage, nahezu alle gängigen Datenformate zu verarbeiten. Somit ist der Algorithmus auch für die Verwendung in anderen Einsatzgebieten wie zum Beispiel der Verarbeitung von Bildern oder großen Datenmengen im Allgemeinen geeignet. Zusätzlich bietet die Applikation, durch die in Kapitel 2.3 und 3 dargestellten Mechanismen, die benötigte Sicherheit, die für ein Programm im Finanzsektor unabdingbar ist. Sollten jemals ein oder mehrere Teile kompromittiert werden, kann keinerlei Information über das Geheimnis erlangt werden, solange die Anzahl der Teile den Schwellenwert nicht übersteigt. In Kombination mit der AES verschlüsselten Datenbank und der Kommunikation über RSA bekleidet diese Backupsolution einen sehr hohen Sicherheitsstandard. Gepaart mit der Vielzahl an Übertragungsmechanismen, die das iOS-Betriebssystem zur Verfügung stellt, bietet das Programm ein komfortables Nutzererlebnis. Durch Speicherung der Metadaten auf iCloud kann der User aktiv durch die Wiederherstellung eines Geheimnisses geführt werden. Dieser Umstand entbindet den Benutzer von der Pflicht, sich an den Aufenthalt der jeweiligen Schlüsselteile zu erinnern. Zusätzlich kann durch die gespeicherten Beziehungen zwischen Vertrauensperson und Schlüsselteilen das Backup nachträglich komplettiert werden, sollte diese Person den Schlüsselteil verlieren. Da die Geheimnisfragmente als Dateien speicherbar sind, müssen diese nicht zwingend auf einem Gerät einer Vertrauensperson gespeichert werden. Dies bringt eine starke Flexibilität und Sicherheit mit sich, da es für einen Angreifer mit steigender Zahl an Teilen enorm aufwendig wird, Zugriff auf ausreichend Informationen zu bekommen. Auch eine Mischung zwischen der Speicherung auf Onlineplattformen und Geräten von Vertrauenspersonen



ist denkbar. Somit hat der User selbst eine nahezu unbegrenzte Anzahl an Speicherungsoptionen. Aus diesen Gründen bietet die in dieser Arbeit vorgestellte Art der Sicherungskopie auf Basis von Verteilen und Wiederherstellen von Datenfragmente eine echte Alternative zu den in Kapitel 2.1 genannten Backupmethoden. Verstärkt wird dies durch die Notwendigkeit der Erstellung eines Backups für jeden Nutzer von Kryptowährungen.

7.2 Ausblick

Im aktuellen Stand ist das Programm ist in der Lage, das Backup an sich zu erstellen und das Geheimnis zu rekonstruieren. Des Weiteren kann eine Vielzahl an Personen und Geheimnissen angelegt und verwaltet werden. Die Basisfunktionalität ist somit gegeben. Um das Nutzererlebnis zukünftig zu steigern, sind einige Erweiterungen und Zusatzfunktionen denkbar. Bereits in Kapitel 4.5 wurden einige denkbare Funktionserweiterungen dargelegt. Vor allem die Überprüfung der Schlüsselteile und das Entziehen dieser sind erstrebenswert. Dadurch würde die Sicherheit des Backups erneut steigen. Auch die Speicherung auf separaten Datenträgern oder mithilfe von Onlinespeicherservices ist eine wünschenswerte Weiterentwicklung. Zusätzlich ist die direkte Kommunikation über das *MultipeerConnectivity* Protokoll zwischen einer Gruppe von Personen denkbar. Für diese Funktionalität muss vor der eigentlichen Kommunikation allerdings eine Art Schlüsselaustausch mit allen Teilnehmern erfolgen. Um die Schlüsselteile mit einer weiteren Ebene der Sicherheit auszustatten, wäre es denkbar, das Geheimnis zuvor mit einem Passwort zu versehen, welches nach der Wiederherstellung aus dem Backup eingegeben werden müsste. Die Problematik dabei besteht darin, dass bei Verlust des Passworts, das Backup nutzlos wird. Um ein breiteres Publikum zu erreichen, können die dargelegten Mechanismen auch auf das Android Betriebssystem portiert werden. Die dargelegten Kommunikationsmethoden auf Dateibasis sind plattformunabhängig und das Datenbankframework ist ebenfalls für Android verfügbar.



Literaturverzeichnis

- [adorsys 2017] ADORSYS: *Adorsys Beschreibung Stellenpaket*. Version: 2017. <https://stellenpakete.de/vorschau/adorsys/2/default.htm>, Abruf: 26.01.2019
- [adorsys 2019] ADORSYS: *Logo der adorsys GmbH*. Version: 2019. https://wiki.adorsys.de/pages/viewpage.action?pageId=43942080&preview=/43942080/185761892/adorsys_logo_RGB_dunkelblau.jpg, Abruf: 26.01.2019
- [Apple 2013] APPLE: *MultipeerConnectivity Framework Development Portal*. Version: 2013. <https://developer.apple.com/documentation/multipeerconnectivity>, Abruf: 26.01.2019
- [Apple 2019] APPLE: *Apple Developer Side*. Version: 2019. <https://developer.apple.com>, Abruf: 26.1.2019
- [brokervergleich 2019] BROKERVERGLEICH: *Krypto Wallet Vergleich – Die e-Wallet Anbieter im Test*. Version: 2019. <https://www.brokervergleich.net/wallet-anbieter-vergleich>, Abruf: 26.01.2019
- [Buchmann 2016] BUCHMANN, Johannes: *Einführung in die Kryptographie (Springer-Lehrbuch) (German Edition)*. Springer Spektrum, 2016
- [bitcoin community 2019] COMMUNITY bitcoin: *BIP39 description github*. Version: 2019. <https://github.com/Bitcoin/bips/blob/master/bip-0039.mediawiki>, Abruf: 14.11.2018
- [digicert 2019] DIGICERT: *The Math Behind Estimations to Break a 2048-bit Certificate*. Version: 2019. <https://www.digicert.com/TimeTravel/math.htm>, Abruf: 26.01.2019
- [Grinman 2018] GRINMAN, Alex: *SSSS Library for Swift*. Version: 1 2018. <https://github.com/kryptco/SecretShare.swift>, Abruf: 26.01.2019. github.com
- [O’Sullivan 2018] O’SULLIVAN, Fergus: *How to Backup Your Bitcoin Wallet*. Version: 3 2018. <https://www.cloudwards.net/how-to-backup-your-bitcoin-wallet/>, Abruf: 26.01.2019
- [realm 2019a] REALM: *realm Documentation Side*. Version: 2019. <https://realm.io/docs/swift/latest>, Abruf: 26.1.2019. [web](https://realm.io)
- [realm 2019b] REALM: *realm Webside*. Version: 2019. <https://realm.io>, Abruf: 26.01.2019
- [Shamir 1979] SHAMIR, Adi: How to Share a Secret. In: *Commun. ACM* 22 (1979), November, Nr. 11, 2. <http://dx.doi.org/10.1145/359168.359176>. – DOI 10.1145/359168.359176. – ISSN 0001–0782



[Sixt 2016] SIXT, Elfriede: *Bitcoins und andere dezentrale Transaktionssysteme: Blockchains als Basis einer Kryptoökonomie (German Edition)*. Springer Gabler, 2016. – 37 S. – ISBN 9783658028435

[statista 2018] STATISTA: *Anzahl der monatlich aktiven Nutzer von WhatsApp weltweit in ausgewählten Monaten von April 2013 bis Januar 2018 (in Millionen)*. Version: 2018. <https://de.statista.com/statistik/daten/studie/285230/umfrage/aktive-nutzer-von-whatsapp-weltweit/>, Abruf: 26.01.2019

[steemit 2017] STEEMIT: *Security Alert: Encryption is not very hard to crack 1024-bit, 2048-bit, 4096-bit and NSA Quantum Resistant Algorithm*. Version: 2017. <https://steemit.com/encryption/@blockcodes/encryption-is-not-very-hard-to-crack-1024-bit-2048-bit-4096-bit-encryption-and-nsa-quantum-resistant-algorithm-encryption>, Abruf: 26.01.2019

[Wagner 2004] WAGNER, David: *Introduction Shamir Secret Sharing Sheme*. University of California, 2004

[hardware wallets.net 2017] WALLETS.NET hardware: *Cryptosteel*. Version: 12 2017. <https://www.hardware-wallets.net/wp-content/uploads/2017/04/Cryptosteel-BIP39-Recovery-Seed.jpg>, Abruf: 26.01.2019. web

[hardware wallets.net 2019] WALLETS.NET hardware: *What are Hardware Wallets?* Version: 2019. <https://www.hardware-wallets.de>, Abruf: 26.01.2019

[Wätjen 2018] WÄTJEN, Dietmar: *Kryptographie: Grundlagen, Algorithmen, Protokolle (German Edition)*. Springer Vieweg, 2018. – ISBN 3658224738

[Winter 2018] WINTER, Andreas: *Analyse der Nutzen und Herausforderungen von Crypto Wallets As A Service*. Hochschule Würzburg-Schweinfurth, 2018

[Wolfmankurd 2017] WOLFMANKURD: *Shamir's Secret Sharing*. Version: 11 2017. https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing#/media/File:Polynomial_over_a_finite_field_as_per_SSSS.png

[Zulauf 2018] ZULAUF, Daniel: *Die Nachwehen des Bitcoin-Booms*. Version: 9 2018. <https://www.aargauerzeitung.ch/wirtschaft/die-nachwehen-des-bitcoin-booms-133019769>, Abruf: 26.01.2019