HOCHSCHULE
MITTWEIDA
University of Applied Sciences

MASTERARBEIT

Frau
Sarah Otto

Identity-Wallets für Android: Datensicherheit und Entwicklung eines Flutter-basierten Prototypen

2022

Fakultät **Angewandte Computer- und Biowissenschaften**

MASTERARBEIT

Identity-Wallets für Android: Datensicherheit und Entwicklung eines Flutter-basierten Prototypen

Autorin:
Sarah Otto

Studiengang:
Cybercrime/Cybersecurity

Seminargruppe:
CY19wC-M

Erstprüfer:
Prof. Dr.-Ing. Andreas Ittner

Zweitprüfer:
Prof. Dipl.-Ing. (BA) Ronny Bodach

Mittweida, Februar 2022

Faculty of **Applied Computer and Life Sciences**

MASTER THESIS

Identity wallets for Android: data security and development of a Flutter-based Prototype

Author:
Sarah Otto

Study Programme:
Cybercrime/Cybersecurity

Seminar Group:
CY19wC-M

First Referee:
Prof. Dr.-Ing. Andreas Ittner

Second Referee:
Prof. Dipl.-Ing. (BA) Ronny Bodach

Mittweida, Februar 2022

Bibliografische Angaben

Otto, Sarah: Identity-Wallets für Android: Datensicherheit und Entwicklung eines Flutter-basierten Prototypen, 70 Seiten, 6 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Masterarbeit, 2022

Referat

Das Modell der Selbst-Souveränen digitalen Identität (SSI) ist ein seit 2015 diskutiertes und entwickeltes Konzept zum Identitätsmanagement, das dem Nutzer die volle Kontrolle über seine digitale Identität verspricht. Um dies gewährleisten zu können, müssen u.a. Anwendungen, zumeist Apps für mobile Betriebssysteme entwickelt werden, die dem Nutzer dies ermöglichen. Diese Arbeit untersucht mit esatus Wallet, Lissi Wallet, ID-Wallet, Connect.Me und Jolocom SmartWallet fünf bereits bestehende Wallet-Anwendungen für das Betriebssystem Android hinsichtlich ihrer Datensicherheit. Als Grundlage für die Sicherheitsbewertung dient der Mobile Application Security Verification Standard (MASVS) des gemeinnützigen Open Web Application Projects (OWASP). Dabei wurden bei allen Apps, vor allem Connect.Me, Mängel festgestellt. Diese beziehen sich vornehmlich auf die zum Teil fehlende Verschlüsselung von Daten aber auch auf den Umgang mit zur Verschlüsselung verwendeten Passwörtern.
Weiterhin wird gezeigt wie mittels dem Framework Flutter eine nach Stand der Technik sichere Wallet-Anwendung entwickelt werden kann.

I. Inhaltsverzeichnis

Inhaltsverzeichnis I
Abbildungsverzeichnis II
Abkürzungsverzeichnis III
1 Einführung 1
2 Grundlagen 3
2.1 Kryptografische Grundlagen 3
2.2 Decentralized Identifier 6
2.3 Verifiable Credentials 7
2.4 Identity Wallets 9
2.5 Android 10
2.5.1 Plattformsicherheit 10
2.5.2 Aufbau einer APK 12
2.5.3 App Sicherheit 15
2.6 OWASP Mobile Security Testing Guide und Mobile Application Security Verification Standard 16
3 Jolocom SmartWallet 19
3.1 Anwendersicht 19
3.2 Speichermodell 22
3.3 Netzwerkverkehr 24
4 Lissi / esatus / ID-Wallet / Connect.Me 26
4.1 Hyperledger Indy und Hyperledger Aries 27
4.2 Anwendersicht 27
4.3 Speichermodell 33
4.4 Netzwerkanalyse 36
5 Eigener Prototyp 40
5.1 Vorüberlegungen 41
5.2 Speichermöglichkeiten 43
5.3 Android Sicherheitsmechanismen in Flutter 46

5.4 (Beispiel-)Anwendung 49
6 Fazit 52
A App-Analyse nach MASVS 54
A.1 Anforderung an Datenspeicherung und Datenschutz 54
A.2 Anforderungen an Kryptografie 56
A.3 Anforderung an Netzwerkkommunikation 57
A.4 Anforderungen zur Plattform-Interaktion 58
A.5 Anforderungen an Manipulationssicherheit/Resilienz 59
Literatur 61

II. Abbildungsverzeichnis

2.1 Struktur einer APK-Datei 12

3.1 Mnemonics in Jolocom SmartWallet 20

3.2 Anzeige der Credentials in Jolocom SmartWallet 22

4.1 Credential-Übersicht der einzelnen Apps 31

4.2 Erstellen und Wiederherstellen eines Backups in esatus Wallet 32

5.1 Credential-Übersicht und Credential-Anfrage 50

III. Abkürzungsverzeichnis

ADB ..... Android Debug Bridge, Seite 18
AES ..... Advanced Encryption Standard, Seite 5
AGB ..... Allgemeine Geschäftsbedingungen, Seite 19
APK ..... Android Package Kit, Seite 12
BSI ..... Bundesamt für Sicherheit in der Informationstechnik, Seite 3
CBC ..... Cipher Block Chaining, Seite 5
CCM ..... Counter Mode mit Cipher Block Chaining Message Authentication, Seite 5
CL-Signatur ..... Camenisch-Lysyanskaya-Signatur, Seite 8
CTR ..... Counter, Seite 5
DID ..... Decentralized Identifier, Seite 1
DLT ..... Distributed Ledger, Seite 26
DSA ..... Digital Signature Algorithm, Seite 5
ECDSA ..... Elliptic Curve Digital Signature Algorithm, Seite 5
FBE ..... File-based Encryption, Seite 10
FDE ..... Full Disk Encryption, Seite 10
GCM ..... Galois/Counter Mode, Seite 5
HD-Wallet ..... Hierarchisch-Deterministisches Wallet, Seite 45
HSM ..... Hardware Security Module, Seite 16
IPC ..... Inter-Prozess-Kommunikation, Seite 14
IPFS ..... Inter-Planetary File System, Seite 25
IWCE ..... Inter-Wallet-Credential Exchange, Seite 40
JWT ..... JSON-Web-Token, Seite 21
MAC ..... Message Authentication Code, Seite 4
MASVS ..... Mobile Application Verification Standard, Seite 1
MSTG ..... Mobile Security Testing Guide, Seite 2
OWASP ..... Open Web Application Security Project, Seite 16
PKI ..... Public-Key Infrastruktur, Seite 4
SE ..... Secure Element, Seite 16
SHA ..... Secure Hash Algorithm, Seite 4
SSI ..... Self-Sovereign Identity, Seite 1
TEE ..... Trusted Execution Environment, Seite 16
URL ..... Uniform Resource Locator, Seite 6

VC ..... Verifiable Credential, Seite 1

W3C ..... World Wide Web Consortium, Seite 7

1 Einführung

Das Anmelden mit einem Nutzernamen und zugehörigem Passwort oder durch Klicken auf Buttons wie „Anmelden mit Google“ oder „Anmelden mit Facebook“ ist gängige Praxis für heutige Internetnutzer. Technisch lässt sich beides als das Nutzerinterface zweier Modelle zum Identitätsmanagement auffassen: Dem isolierten und dem zentralisierten Modell [1, S.17]. Beide haben gemein, dass die Daten des Nutzers, sei es sein Name oder seine Adresse, beim jeweiligen Anbieter der Login-Lösung hinterlegt sind, nachdem sie bei der Registrierung vom Nutzer eingegeben wurden. Daraus ergeben sich für diesen wie auch für den Anbieter verschiedene Nachteile. So sind zum einen die Nutzer-Datenbanken der Anbieter lohnenswerte Ziele für Angreifer. Zum anderen haben die Anbieter kaum eine Möglichkeit, die Korrektheit der Nutzerdaten zu verifizieren. Im Falle eines zentralisieren Ansatzes kommt hinzu, dass hier die jeweiligen Anbieter in der Lage sind, Login-Vorgänge ihrer Nutzer bei anderen Diensten zu protokollieren.

Lösungsansätze, die die genannten Probleme adressieren, werden seit dem Jahr 2015 unter dem Modell der Self-Sovereign Identity (SSI), zu deutsch Selbst-Souveräne Identität, entwickelt. Ein selbst-souveränes Identitätsmanagement soll den Nutzer in die Lage versetzen, seine Identitätsdaten selbst zu verwalten und darüber zu bestimmen, für wen welche Informationen freigegeben werden. Daneben werden an ein solches noch weitere Anforderungen gestellt, die Christopher Allen in „The Path to Self-Sovereign Identity“ [2] zusammenfasst.

Neben verschiedenen Standards für z.B. Decentralized Identifier (DID) und Verifiable Credentials (VC) und Kommunikationsprotokollen zum Austausch von Identitätsinformationen, entstand im Zuge der Entwicklungen zu SSI auch Endnutzer-Software. Diese gemeinhin als Wallet bezeichnete Software bietet für den Nutzer die Schnittstelle zu seiner digitalen Identität und ist damit die Grundvoraussetzung für diesen, der bei SSI geforderten Nutzerkontrolle nachzukommen. Entsprechend hohe Sicherheitsstandards sollten für diese Software gelten, da sie zumeist auch für die lokale Speicherung der Identitätsdaten auf dem Endgerät verantwortlich ist. Zumeist handelt es sich bei dem Endgerät um ein Smartphone; Endnutzer-freundliche SSI-Wallets für Desktop-Systeme sind nach derzeitigem Stand nicht verfügbar.

Für diese Arbeit wurden fünf bestehende Wallet-Implementierungen für eine Untersuchung ausgewählt. Es handelt sich dabei um die Anwendungen esatus Wallet [3], Lissi Wallet [4], ID-Wallet [5], Connect.Me [6] und Jolocom SmartWallet [7]. Konkret wird sich auf die Android-Versionen[1] konzentriert, da dieses System im Vergleich zu anderen mobilen Betriebssystemen den größten Marktanteil besitzt [8]. Grundlage der Untersuchung bilden der Mobile Application Security Verification Standard (MASVS) [9] und der

[1] Alle genannten Apps sind auch für iOS verfügbar.

eng damit verwandte Mobile Security Testing Guide (MSTG) [10] . Das Hauptaugenmerk liegt auf einer sicheren Speicherung der Identitätsinformationen sowie darauf, den Umgang mit kryptografischem Material zu überprüfen. Weiterhin wird Wert darauf gelegt, dass von Android zur Verfügung gestellte Schnittstellen und Bibliotheken in Bezug auf Datenspeicherung und Kryptografie genutzt werden.

Im Anschluss an die Untersuchung der Apps werden die gewonnen Erkenntnisse genutzt, eine eigene prototypische Wallet-Anwendung und ihr zugrunde liegende Software-Bibliothek zu entwickeln. Für diese wird das Framework Flutter genutzt. Auch hier steht die Sicherheit der Identitätsdaten an erster Stelle. Demnach gilt es ebenfalls zu evaluieren, inwieweit Flutter die Sicherheitsmaßnahmen Androids unterstützt. Dabei soll die Eigenschaft der plattformübergreifenden Funktionalität nicht aus den Augen gelassen werden.

Zusammenfassend lässt sich sagen, dass diese Arbeit zwei Ziele verfolgt: Zum einen die Untersuchung der fünf genannten Anwendungen hinsichtlich der Datensicherheit und zum anderen die Entwicklung eines sicheren Wallet-Prototypen mittels Flutter. Dazu werden im Anschluss an diese Einführung alle notwendigen technischen Grundlagen erläutert. Die beiden darauffolgenden Kapitel widmen sich der Untersuchung der Wallet-Anwendungen. Sie werden dabei hinsichtlich der verwendeten SSI-Technologien auf die Kapitel aufgeteilt. So untersucht Kapitel 3 lediglich Jolocom SmartWallet und das nachfolgende Kapitel 4 die übrigen vier Anwendungen. Beide Kapitel sind dabei vergleichbar aufgebaut: Zuerst werden die Apps aus Nutzersicht betrachtet. Anschließend werden die dabei erzeugten Daten vom Endgerät extrahiert und untersucht, wie die Identitätsdaten gespeichert wurden. Anschließend wird der Netzwerkverkehr der Apps überprüft. Das vorletzte Kapitel 5 widmet sich dem eigenen Prototyp und damit dem Framework Flutter. Abschließend wird in Kapitel 6 ein Gesamt-Fazit gezogen.

2 Grundlagen

Dieses Kapitel beschreibt alle notwendigen Grundlagen für das Verständnis der Arbeit. Hierzu gehören kryptografische Verfahren wie Hash-Funktionen, digitale Signaturen und Verschlüsselungsalgorithmen. Dabei wird neben dem Verständnis der Verfahren Wert darauf gelegt, für jedes Algorithmen zu benennen, die nach heutigem Stand der Technik als sicher anzusehen sind. Im Anschluss werden die für SSI wichtigen Standards Decentralized Identifier (DID) und Verifiable Credentials (VC) erläutert. Weiterhin wird definiert, was unter einem Wallet zu verstehen ist. Anschließend wird das Betriebssystem Android betrachtet bevor abschließend auf MASVS und MSTG und damit das Vorgehen bei den Untersuchungen der Wallet-Apps eingegangen wird.

2.1 Kryptografische Grundlagen

Kryptografische Verfahren bilden die Grundlage Integrität, Authentizität und Vertraulichkeit von Daten sicherzustellen. Neben diesen drei grundlegenden Schutzzielen der IT-Sicherheit wird in der Kryptografie noch ein viertes Ziel, die Nicht-Abstreitbarkeit, betrachtet. Darunter ist zu verstehen, dass der Ersteller/Sender einer Nachricht[2] gegenüber Dritten nicht leugnen kann, diese erstellt/gesendet zu haben [11, S.5]. Keines der nachfolgend vorgestellten, grundlegenden Verfahren erfüllt alle Ziele alleine. Hierzu ist stets eine Kombination selbiger notwendig.

Neben der Funktionsweise der grundlegenden kryptografischen Verfahren soll aufgezeigt werden, welche jeweils bekannten Algorithmen als sicher anzusehen sind. Für diese Bewertung wird zum einen die Technische Richtlinie 02102-1 „Kryptografische Verfahren: Empfehlungen und Schlüssellängen“ des Bundesamts für Sicherheit in der Informationstechnik (BSI) [12] herangezogen. Zum anderen werden die Forschungsergebnisse eines von der Europäischen Union geförderten Forschungsbündnisses aus vier Universitäten und einem Unternehmen (ECRYPT) [13] herangezogen.

Hash-Funktionen sind Funktionen die Nachrichten beliebiger Länge auf Bit-Folgen fester Länge abbilden. Sie sind Einweg-Funktionen, d.h. das Berechnen des Hashwertes aus einer Nachricht ist einfach möglich; der Rückweg, also von einem Hash-Wert auf die eingegebene Nachricht schließen, ist rechenaufwendig. Anders gesagt wird gefordert, dass es keinen effizienteren Weg als stupides Ausprobieren geben soll, von einem Hash auf die Nachricht zu schließen.
Hash-Funktionen dienen vornehmlich dazu, die Integrität von Daten sicherzustellen,

[2] Der Begriff Nachricht beschreibt hier und im folgenden jedwede Daten, die einem kryptografischen Verfahren zugeführt werden können. Dabei kann es sich um einfachen Text, Dateien, ganze Festplattenpartitionen, u.ä. handeln.

d.h. mit ihrer Hilfe kann gezeigt werden, dass eine Nachricht z.B. während ihre Übertragung über ein Netzwerk, nicht verändert wurde. Dies ist der Fall wenn Sender und Empfänger einen identischen Hash-Wert erhalten. Um eben diesen Zweck sicher erfüllen zu können, soll es rechnerisch nicht möglich sein, zu einer bestehenden Nachricht eine weitere zur ersten verschiedene Nachricht zu finden, die den gleichen Hash produziert (schwache Kollisionsresistenz). Weiterhin soll es rechnerisch nicht möglich sein, zwei verschiedene Nachrichten zu finden, die den gleichen Hash erzeugen (starke Kollisionsresistenz) [11, S.267].
Zu den am weitesten verbreiteten Hash-Algorithmen gehört der Secure Hash Algorithmus (SHA). Von diesem existieren drei Versionen, wobei Version eins (SHA1) als unsicher gilt. SHA2 wie auch SHA3 gelten derzeit als sicher. Für diese Algorithmen existieren weitere Untergliederung nach den erzeugten Hash-Längen. Hier sind jeweils 256, 324 und 512 Bit möglich und sicher [12, S.42, 13, S.41].

Sehr ähnlich zu Hash-Werten sind Message Authentication Codes (MAC). MAC-Verfahren haben ebenfalls die Eigenschaft, aus einer Nachricht beliebiger Länge eine Ausgabe fester Länge zu erzeugen. Allerdings benötigen sie hierfür nicht nur die Nachricht selbst, sondern zusätzlich einen kryptografischen Schlüssel, der Sender wie Empfänger der Nachricht bekannt ist. Sie können somit neben der Integrität auch die Authentizität einer Nachricht sicher stellen. Auch für MACs existieren verschiedene Algorithmen zur Erzeugung. Am verbreitetsten ist dabei HMAC. Dieser basiert auf einer Hash-Funktion, die grundsätzlich frei wählbar ist und entsprechend sicher sein sollte. Neben Hash-Funktionen eignen sich auch bestimmte Verschlüsselungsalgorithmen, wie der später vorgestellte Advanced Encryption Standard (AES) als Grundlage. Auf diesem basieren die Algorithmen GMAC und CMAC [14]. Alle hier genannten Verfahren (HMAC, GMAC, CMAC) werden vom BSI als sicher eingestuft [12, S.45], ECRYPT empfiehlt allerdings GMAC nicht für zukünftige Anwendungen [13, S.57].

Digitale Signaturen gehen noch einen Schritt weiter: Sie können neben Integrität und Authentizität auch Nicht-Abstreitbarkeit sicherstellen. Dies liegt daran, dass für sie ein asymmetrisches Schlüsselpaar bestehend aus einem Öffentlichen und Privaten Schlüssel, zwischen denen ein mathematischer Zusammenhang besteht, benötigt wird. Dabei wird der Private Teil zur Erzeugung der Signatur verwendet und der Öffentliche zur Überprüfung selbiger. Zu diesem Zweck sollte dieser uneingeschränkt zur Verfügung gestellt werden, z.B. mittels einer Public-Key-Infrastruktur (PKI) [11, S.145ff].
Grundlegend ist zu beachten, dass stets der Hash einer Nachricht unterschrieben wird. Somit setzt eine sichere digitale Signatur die Verwendung einer sicheren Hash-Funktion voraus. Trotz dessen sind Signaturen rechenaufwendiger und damit langsamer als MAC-Verfahren.
Derzeitig erachtet das BSI sowohl die Algorithmen RSA und DSA (Digital Signature Al-

gorithm), dessen auf elliptischen Kurven[3] basierende Variante ECDSA (Elliptc Curve DSA) und sogenannte Merklesignaturen als sicher, vorausgesetzt, die Schlüssel haben die geforderten Mindestlängen [12, S.46ff]. ECRYPT trifft vergleichbare Aussagen zu den empfohlenen Algorithmen [13, S.68ff].

Das letzte Schutzziel, das bisher noch von keinem der vorgestellten Verfahren erreicht werden konnte, die Vertraulichkeit, lässt sich durch Verschlüsselung erzielen. Ähnlich wie eben dargestellt, kann auch hier zwischen Verfahren unterschieden werden, bei denen allen beteiligten Parteien ein gemeinsamer Schlüssel zum Ver- und Entschlüsseln bekannt sein muss (symmetrische Verfahren) und welchen, für die ein Schlüsselpaar notwendig ist (asymmetrische Verfahren). Hier dient der Öffentliche Schlüssel der Verschlüsselung und der Private der Entschlüsselung.
Asymmetrische Verfahren lösen weiterhin eines der Hauptprobleme symmetrischer: Es entfällt das Austauschen des Schlüssels. Allerdings sind sie deutlich rechenaufwendiger und aufgrund der verwendeten Berechnungen nicht ohne Erweiterung für Nachrichten, die länger als der verwendete Schlüssel sind, geeignet. Eine Lösung dieses Problems bieten sogenannte hybride Verfahren. Dabei wird ein Text selbst symmetrisch verschlüsselt und der dafür verwendete symmetrische Schlüssel asymmetrisch [13, S.67].
Damit lässt sich festhalten, dass symmetrische Verfahren für die Verschlüsselung großer Datenmengen (Dateien/ganzer Festplatten) besser geeignet sind. Der dabei am weitesten verbreitet Algorithmus ist der Advanced Encryption Standard (AES) [15]. Dieser verwendet Schlüssellängen von 128, 192 oder 256 Bit und sieht vor, die zu verschlüsselnden Daten in einzelne Blöcke von 128 Bit zu zerlegen. Damit gehört AES zur Gruppe der Blockchiffren. Auf jeden dieser Blöcke wird der AES Algorithmus mit dem gleichen Schlüssel angewendet. Inwiefern die Blöcke bzw. der Schlüssel dabei noch in einer gewissen Art und Weise verändert werden, bestimmt der Betriebsmodus. Insgesamt beschreibt das National Institute of Technology (NIST) 14 dieser Betriebsmodi [16, 17, 18, 19, 20, 21, 22]. Drei davon sind allerdings stark zweckgebunden, nämlich zum Schutz von Vertraulichkeit und Integrität kryptografischer Schlüssel [20] und einer ist vielmehr ein MAC [21], nämlich der bereits erwähnte CMAC. Von den übrigen Modi empfiehlt das BSI vier zur Nutzung. Dazu gehören der Cipher Block Chaning Mode (CBC), der Counter Mode (CTR), der Galois/Counter Mode (GCM) und der Counter Mode mit Cipher Block Chaining Message Authentication (CCM) [12, S.23]. Dabei ist anzumerken, dass GCM wie auch CCM zusätzlich die Integrität des verschlüsselten Textes sicherstellen können. D.h. es handelt sich um Verschlüsselungsmodi, die mit einem MAC-Verfahren kombiniert wurden. Für die übrigen Modi sollte ein eigenes Authentisierungsverfahren eingesetzt werden. ECRYPT hingegen empfiehlt die hier genannten Modi mit Ausnahme von GCM lediglich für bestehende Anwendungen. Dafür werden weitere Blockmodi aufgelistet und beschrieben, die verwendet werden können. Weiterhin ist anzumerken, dass neben AES ein weiterer symmetrischer Algorithmus mit

[3] Zum Verständnis dieser Arbeit genügt es, sich eine elliptische Kurve stark vereinfacht als Menge von Zahlen vorzustellen, innerhalb derer eigens definierte Rechenoperationen durchgeführt werden können.

zugehörigem MAC-Verfahren als nutzbar angesehen wird. Dabei handelt es sich um ChaCha20 kombiniert mit dem MAC Poly1305. Im Gegensatz zu AES handelt sich bei ChaCha20 um eine Stromverschlüsselung. D.h. der Algorithmus generiert aus einem Anfangs-Schlüssel einen fortlaufenden Schlüsselstrom, der Bitweise mit der Nachricht XOR verknüpft wird [13, S.54]. Nach Ansicht des BSI gibt es derzeitig keine zu empfehlende Stromverschlüsselung [12, S.25].

Gerade im Bereich der symmetrischen Datenverschlüsselung sieht sich ein Nutzer in den wenigsten Fällen damit konfrontiert, einem für die Ver- und Entschlüsselung verantwortlichen Programm einen Schlüssel zu präsentieren. Vielmehr wird das Festlegen eines Passwortes nötig. Entsprechend existieren Funktionen, die aus einem Passwort einen kryptografischen Schlüssel ableiten können. Diese werden als Password-Based Key Derivation Functions (dt. Passwort-basierte Schlüsselableitungsfunktionen) bezeichnet. Verbreitete Vertreter sind PBKDF2, Scrypt, bcrypt und Argon2. Das BSI empfiehlt direkt keine der eben genannten Funktionen. An ihrer Stelle soll ein MAC-Verfahren (HMAC oder CMAC), das in Hardware implementiert ist, auf das Passwort angewendet werden. Selbiges sollte vorher mit einer weiteren zufällig erzeugten Zeichenkette (Salt) verknüpft werden. Wenn dies nicht möglich ist, kann auf die Funktion Argon2 zurückgegriffen werden [12, S.74f]. Auch ECRYPT ist zurückhaltend, was die Empfehlung derartiger Funktionen angeht. Hier werden PBKDF2, Scrypt und bcrypt als derzeitig verwendbar angesehen. Inwiefern sie für zukünftige Anwendungen als sicher gelten, kann aufgrund fehlender Sicherheitsbeweise nicht gesagt werden. Argon2 ist nicht Teil der Empfehlung, da Untersuchungen zur Sicherheit der Funktion zum Zeitpunkt der Erstellung des ECRYPT Reports noch nicht abgeschlossen waren [13, S.71f].

2.2 Decentralized Identifier

Ein Identifier ist ein eindeutiger Name bzw. Bezeichner, der einer Person, einem Gegenstand oder Sonstigem zugeordnet ist und über den dieses Objekt identifiziert werden kann. Dies können beispielsweise für eine Person in Deutschland ihre Steueridentifikationsnummer oder für ein Auto sein KFZ-Kennzeichen sein. In technischen Systemen ist ein bekannter Identifier ein URL (Uniform Resource Locator), der eine Website lokalisiert. Diesen Beispielen ist gemein, dass die Identifier von einer Kontrollinstanz vergeben werden, um ihre Einzigartigkeit zu garantieren.
Demgegenüber handelt es sich bei einem Decentralized Identifier (DID) [23] um einen vom zu identifizierenden Objekt (DID-Subject) selbst gewählten bzw. generierten Identifier, der aufgrund des Generationsprozesses einzigartig ist. Der Aufbau eines DID erinnert an den eines URL. So beginnt er stets mit der Angabe eines Schemas, das allerdings nicht variabel ist, sondern immer `did` lautet. Danach wird die DID-Methode angegeben. Diese spezifiziert, nach welchen Regeln der DID generiert, abgerufen, aktualisiert und gelöscht werden kann. Darauf folgt eine Methodenspezifische ID. Dies ist eine

alphanumerische Zeichenketten, die nach den Regeln der DID-Methode erzeugt wird. Beispielhaft kann eine gültige DID folgendermaßen aussehen: `did:example:1234689` Der DID selbst verweist dabei auf ein ihm zugehöriges DID-Document, zu welchem er mithilfe einer speziellen Software, dem DID-Resolver aufgelöst werden kann. Dieses DID-Document enthält u.a. Öffentliche kryptografische Schlüssel; allerdings niemals (persönliche) Informationen, die das DID-Subject näher beschreiben.
Aufgrund dieser Eigenschaft, dass einem DID stets Schlüsselmaterial zugeordnet ist, bieten sie ihrem Subject die Möglichkeit mittels digitaler Signaturen nachzuweisen, den DID zu kontrollieren.

Es ist allerdings anzumerken, dass es sich bei dem vom World Wide Web Consortium (W3C) entwickelten Standard zu DIDs [23] bisher nur um einen Vorschlag zur Standardisierung handelt, der derzeit kontrovers diskutiert wird [24, 25].

2.3 Verifiable Credentials

Credentials (dt. Berechtigungsnachweise) sind Dokumente wie ein Führerschein, Personalausweis oder Hochschulzeugnis, die einer Person von einer Organisation (oder anderen Person) ausgestellt werden. Sie attestieren ihrem Besitzer verschiedene Eigenschaften, z.B. die Erlaubnis zum Führen eines Fahrzeugs oder einen entsprechenden Bildungsabschluss. Zur digitalen Abbildung Credentials jeglicher Art entwickelte eine Arbeitsgruppe des W3C das Verifiable Credentials Data Model [26]. Diese Spezifikation beschreibt den Lebenszyklus und mögliche Datenformate eines Credentials. Wie auch ihre analogen Vorbilder werden Verifiable Credentials von einer (vertrauenswürdigen) Organisation oder Einzelperson, nachfolgend als Issuer bezeichnet, ihrem Besitzer (Holder) ausgestellt. Holder wie auch Issuer sind mittels einer ID, zumeist einer DID, innerhalb eines Verifiable Credentials eindeutig identifiziert. Der Nachweis, dass ein bestimmter Issuer ein Credential ausgestellt hat, erfolgt mittels digitaler Signatur.
Nach abgeschlossenem Ausstellprozess ist der Holder in die Lage versetzt, seine Credentials jederzeit ohne Zutun des Issuer einer Dritten Partei, dem Verfier vorzuzeigen. Dazu generiert er aus einem (oder mehreren) seiner Verifiable Credentials eine Verifiable Presentation, die er wiederum digital signiert. Mittels der enthaltenen Signaturen kann der Verifier prüfen, dass die Credentials selbst von den ihnen zugeordneten Issuern ausgestellt wurden, die Presentation vom Holder persönlich zusammengestellt wurde (Authentizität) und die Inhalte der Credentials nicht verändert wurden (Integrität).

Abhängig von der konkreten Implementierung besitzen Verifiable Credentials datenschutzrechtlich wünschenswerte Eigenschaften, die mit analogen Credentials nicht umgesetzt werden können. Dazu gehört Selective Disclosure, eine Eigenschaft, die es dem Holder erlaubt, nur bestimmte Teile (Attribute) eines Credentials zu zeigen. Bildlich vergleichbar ist dies mit dem Vorgehen beim Vorzeigen seines Ausweises beispielsweise

den Namen mit dem Finger zu verdecken.
Implementieren lässt sich diese Eigenschaft auf drei verschiedene Wege: Der trivialste besteht darin, dass der Issuer für jedes einzelne Attribut ein eigenes Credential ausstellt. Weiterhin besteht die Möglichkeit, auf alle Attributwerte einzeln eine Hashfunktion anzuwenden und schlussendlich ein Credential zu signieren, dass lediglich die Hashwerte der einzelnen Attribute oder einen Hash über alle Attribut-Hashes enthält. Die dritte Umsetzungsmöglichkeit ist der Einsatz spezieller Signaturverfahren wie Camenisch-Lysyanskaya (CL)- oder BBS+-Signaturen [27]. Beide Verfahren sind derart gestaltet, dass sie es dem Holder im Nachhinein ermöglichen, aus denen ihm ausgestellten Credentials (mathematische) Beweise zusammenzustellen, dass er ein Credential besitzt, das ihm ein gewisses Attribut attestiert. Es wird bei CL- bzw. BBS+-signierten Credentials also nicht wie bei mittels bekannter asymmetrischer Verfahren (RSA, ECDSA) unterschriebenen Credentials, das Credential als solches in die Verifiable Presentation eingefügt, sondern lediglich ein Beweis darüber, dass sich ein Credential mit dem entsprechenden Attribut im Besitz des Holders befindet. Allgemein ist ein solches Beweisverfahren als Zero-Knowledge-Proof of Knowledge bekannt.
Neben Selective Disclosure ermöglicht letztere Art der Credentials sogenannte Predicate Proofs. D.h. sie bieten die Möglichkeit nachzuweisen, dass ein bestimmter Attributwert (z.B. das Alter einer Person) größer oder kleiner als ein geforderter Wert ist. Ebenso bieten sie ein Feature, dass als Multishow-Unlinkabiltity bekannt ist. Dies bedeutet, dass mehrere Presentations mit den gleichen Attributwerten aus Credentials zusammengestellt werden können, ohne dass es möglich ist, eine Verbindung zwischen diesen Presentations herzustellen. Daher wird diese Art der Credentials auch als Anonyme Credentials bezeichnet [28].

Neben ihren verschiedenen Eigenschaften bezogen auf die Erstellung von Presentations aus ihnen, unterscheiden sich die beiden Arten der Credentials - Anonyme Credentials und klassisch signierte Credentials - auch in den Daten, die ein Holder vorhalten muss, um den Besitz eines solchen zu beweisen.
Für den Besitznachweis eines klassisch signierten Credentials muss der Holder das Credential selbst, also die vom Issuer signierte Datenstruktur, vorhalten. Weiterhin ist der private Schlüssel nötig, der mit dem Identifier assoziiert ist, auf den das Credential ausgestellt wurde.
Auch bei Anonymen Credentials ist das Credential selbst mit der Signatur des Issuers nötig. Diese Credentials werden allerdings nicht an einen klassischen Identifier gebunden, sondern an ein sogenanntes Master- bzw. Link-Secret. Technisch gesehen handelt es sich dabei um eine große Zufallszahl. Diese ist in jedem Credential eines Holders enthalten. In einer Presentation muss nachgewiesen werden, dass der Holder das Mastersecret kennt und dass alle in der Presentation enthaltenen Credentials auf dasselbe Mastersecret ausgestellt sind [28]. Es kann von seiner Funktion also mit dem privaten Schlüssel der klassisch signierten Credentials gleichgesetzt werden.

2.4 Identity Wallets

Das Modell der Selbst-souveränen (digitalen) Identität sieht vor, dass Nutzer in die Lage versetzt werden, eigenständig Entscheidungen zu treffen, wofür ihre Identitätsdaten verwendet werden und allen voran, ihre Identitätsdaten selbst zu verwalten. Für diese Aufgabe ist entsprechende Software nötig. Diese Software wird zumeist als Wallet bzw. genauer Identity Wallet bezeichnet. Was dabei konkret unter einer Wallet zu verstehen ist, dafür gibt Andreas Antonopoulos in seinem Buch „Mastering Bitcoin“ [29] eine treffende Definition:

> „At a high level, a wallet is an application that serves as the primary user interface. The wallet controls access to a user's money, managing keys and addresses, tracking the balance, and creating and signing transactions.
>
> More narrowly, from a programmer's perspective, the word ‚wallet' refers to the data structure used to store and manage a user's keys.“

Beide Definitionen lassen sich mit der Anpassung, dass zusätzlich Identitätsdaten in Form von Verifiable Credentials verwaltet werden müssen, auf Identity Wallets übertragen. So trifft auf alle in dieser Arbeit untersuchten Anwendung erstere Definition zu. Die Anwendungen stellen dem Nutzer ein grafisches Interface bereit, mit dem er seine Identität (seine Credentials) verwalten kann. Dazu gehört auch der Austausch der Daten mit anderen Diensten. Das Herzstück aller Anwendung ist die Speicherung von Daten, allen voran Schlüssel und Credentials, in lokalen Datenstrukturen.
Im Umfeld von SSI wurde mit Agent für erstere Definition ein eigener Begriff eingeführt. Der Begriff Wallet wird wie Antonopoulos im zweiten Teil seiner Definition anspricht, für Speicherstrukturen verwendet. Entsprechend nutzt jeder Agent ein Wallet als Datengrundlage. Streng genommen sind also alle in dieser Arbeit untersuchten Anwendungen, wie auch die eigens entwickelte App (Kapitel 5), Agents. Da sich die Apps selbst als Wallets bezeichnen, wird dieser Begriff im Laufe der Arbeit synonym mit Agent verwendet.

Wie bereits in Abschnitt 2.3 zu Verifiable Credentials angedeutet, unterliegen Identity Wallets der Herausforderung, Daten unterschiedlicher Funktion und Schutzstufen zu verwalten. So folgt aus der Extraktion eines Credentials aus dem Wallet durch einen Angreifer in erster Linie ein Problem in Bezug auf den Datenschutz. Der Angreifer ist in der Lage, persönliche Daten des Nutzers zu lesen. Er kann das Credential allerdings nicht für sich selbst einsetzen und sich als der Nutzer ausgeben. Dazu ist der private Schlüssel bzw. das Mastersecret notwendig. Gelingt es dem Angreifer, auch an letzteres zu gelangen, ist die Übernahme der digitalen Identität (Impersonation) möglich.

Nach derzeitigem Stand existieren zwei allgemein gehaltene Spezifikationen, wie ein Wallet strukturierbar ist. Die W3C beschreibt unter dem Titel Universal Wallet [30] zum

einen JSON-basierte Datenmodelle, wie bestimmte Datengruppen z.B. Schlüssel oder Credentials in einem Wallet abgelegt werden. Zum anderen wird aber auch ein abstraktes Interface nach außen beschreiben, wie verschiedene Aktionen z.B. das Hinzufügen oder Abfragen eines Credentials angeregt werden. Allerdings werden für diese Interface-Funktionen keine notwendigen Übergabeparameter angegeben. Auch wenn angedeutet wird, dass es eine Verschlüsselte (locked) Version eines Wallets geben muss, wird keine Empfehlung für den zu verwendenden Verschlüsselungsalgorithmus gegeben. Es wird lediglich angedeutet, dass die für diese Verschlüsselung zu verwendenden Schlüssel aus einem Passwort abgeleitet werden, da die Interface-Funktionen zum Ver- und Entschlüsseln die Kenntnis eines solchen erfordern.

Konkreter ist die Wallet Spezifikation des Hyperledger Aries Projektes [31]. Hier wird ein Wallet im technischen Sinne - also lediglich der Aufbau der Datenstruktur beschrieben. Dabei wird definiert, wie die Daten zu verschlüsseln sind, damit sie dennoch durchsuchbar bleiben. Der verwendetet Verschlüsselungsalgorithmus ist ChaCha20Poly1305. Die Standard-Implementierung beruht auf einer SQLite-Datenbank.

2.5 Android

Android ist mit 71% weltweit bzw. 59% Marktanteil Deutschlandweit (Stand Oktober 2021) das meistgenutzte mobile Betriebssystem [8]. Bei Android handelt es sich um ein quelloffenes System, das federführend von Google entwickelt wird [32].
Nachfolgend werden die wichtigsten Punkte zur Datensicherheit unter diesem System vorgestellt, sowie der Aufbau einer App beschrieben. Weiterhin werden Aspekte zur Sicherheit der App-eigenen Daten diskutiert. Dies soll einen grundlegenden Überblick darüber geben, auf was bei der Untersuchung der Wallet-Apps und der Entwicklung der eigene App geachtet wird.
Der Fokus liegt dabei auf Funktionen und Merkmalen, die ab Android Version 5.0 verfügbar sind, da die zu untersuchenden Anwendungen minimal ab dieser bzw. ab Version 6.0 nutzbar sind.

2.5.1 Plattformsicherheit

Ein wichtiger Punkt der Datensicherheit ist die Verschlüsselung. Android unterstützt hierfür die vollständige Verschlüsselung des internen Speichers, was die Extraktion von Daten aus einem ausgeschalteten Gerät deutlich erschwert. In Android 5.0 wird Full Disk Encryption (FDE) angewendet, d.h der gesamte Speicher wird blockweise mit dem gleichen Schlüssel verschlüsselt. Ab Android 6.0 ist eine solche Verschlüsselung Pflicht. Ausgenommen sind dabei Geräte mit sehr wenig Rechenleistung und solche, die mit Android 5.0 ausgeliefert wurden. Ab Version 7.0 ist zusätzlich die sogenannte File-based Encryption (FBE) möglich. Hierbei werden einzelne Dateien mit verschiedenen

Schlüsseln verschlüsselt und können demnach unabhängig voneinander entschlüsselt werden. Diese Art der Verschlüsselung ist ab Android 10 verpflichtend.
Anzumerken ist, dass externer Speicher wie SD-Karten nicht immer mit verschlüsselt sind. Dies ist nur möglich, wenn sie als sogenannter Adoptable Storage eingebunden werden. Dies bedeutet, dass die Speichererweiterung genauso behandelt wird wie der interne Speicher, nur mit einem Gerät funktioniert und sowohl Apps als auch Dateien speichern kann [33].
Für beide Arten der Verschlüsselung gilt, dass zum Entschlüsseln das Passwort/PIN/ Muster des Nutzers notwendig ist. Aus diesem wird ein sogenannter Key Encryption Key abgeleitet, mit dem der eigentliche Disk Encryption Key entschlüsselt werden kann. Mit letzterem wird dann der Speicher (FDE) entschlüsselt oder aus ihm Schlüssel für die Entschlüsselung einzelner Dateien abgeleitet (FBE). Im Falle von FBE sind nicht alle Dateien auf die eben beschrieben Weise verschlüsselt. Es existieren Dateien/Speicherbereiche, deren Disk Encryption Key mittels eines an Gerätemerkmale gebunden Schlüssels entschlüsselt werden kann. Damit ist es möglich, Zugriff auf bestimmte Dateien zu gewährleisten, ohne dass der Nutzer seine Daten (Passwort/PIN/Muster) eingeben muss. Dies bietet den Vorteil, dass grundlegende Telefonfunktionalitäten (Anrufe/SMS/Wecker) ohne Nutzerinteraktion gestartet werden können [34].
FBE unterliegt bis Android 9 einer Schwachstelle: Die Metadaten wie Zeitstempel für Erstellung, Veränderung und Zugriff der einzelnen Dateien werden nicht verschlüsselt. Damit ist es möglich, aus dem verschlüsselten Speicherabbild eine Liste installierter Apps zu extrahieren sowie auf Interaktionen mit einer App zu schließen, wie Groß et.al. in [35] zeigen. Verpflichtend muss diese Metadaten-Verschlüsselung ab Android 11 für den internen Speicher aktiviert sein [36].

Während des Boot-Vorgangs eines Android Systems wird standardmäßig die Integrität der gestarteten Komponenten geprüft. Damit kann verhindert werden, dass eine manipulierte Version des Betriebssystems gestartet wird. Konsequent wird dies allerdings erst ab Android 7.0 durchgesetzt. D.h. ab dieser Version warnt das System nicht nur, wenn die Integritätsprüfung fehlschlägt, sondern verweigert das Starten gänzlich [37].
Weiterhin hängt das Durchführen der Integritätsprüfung vom Zustand des Bootloaders ab. Selbige findet nur statt, wenn sich der Bootloader im gesperrten (locked) Zustand befindet, was standardmäßig der Fall ist. Ein entsperrter Bootloader ist für den Fall notwendig, wenn der Nutzer die vom Gerätehersteller mitgelieferte Android-Version durch einen eigene (Custom-ROM) ersetzen oder Root-Rechte erlangen will. Dabei ist zu beachten, dass jeder Zustands-Wechsel alle Nutzerdaten löscht [38].

Wie bereits angedeutet, gibt es auf einem Android System keinen Root-Benutzer bzw. keine Möglichkeit für einen normalen Benutzer Root-Rechte zu erlangen. Damit kann sichergestellt werden, dass das Linux Benutzer- und Berechtigungskonzept genutzt werden kann, um Daten einer App privat, d.h nur für diese App nutzbar, zu halten. Auch dem Geräte-Nutzer bleiben App-Daten damit verborgen.
Trotz dass die Abwesenheit von Root ein wichtiges Sicherheitskonzept von Android dar-

stellt, entscheiden sich Nutzer, ihr Gerät zu „rooten“, also entsprechende Berechtigungen zu erlangen. Damit wird es beispielsweise möglich, Systemapps, die sonst nicht deinstallierbar sind, zu löschen, individuelle, lokale Backups anzulegen oder eine systemweite Firewall einzurichten.
Für bestimmte Anwendungen beispielsweise aus dem Finanzwesen ist es demgegenüber entscheidend, auf einem unveränderten und damit möglichst sicheren Android System zu laufen. Dementsprechend sollten sie in der Lage sein, gerootete Geräte zu erkennen. Hierzu existieren verschiedene Techniken wie das Suchen nach gewissen Apps und Anwendungen, die bekannt dafür sind, Rootrechte zu benötigen oder zum rooten verwendete werden zu können [39]. Auch Google selbst stellt mit der SafetyNet Attestation-API seit Android 4.4 eine entsprechende API bereit, die dazu genutzt werden kann, die Integrität des Gerätes un der eigenen App zu prüfen. Eine korrekte Nutzung dieser API sieht vor, dass das Ergebnis, das von dieser an die App zurückgeliefert wird, nicht lokal auf den Gerät ausgewertet wird, sondern an die jeweiligen Backend-Server der App zur Auswertung weitergeleitet werden [40]. Andernfalls ergeben sich für Angreifer mehrere Möglichkeiten, die Prüfung zu umgehen, wie Forscher in [41] zeigen.

2.5.2 Aufbau einer APK

Android Apps werden in Form einer Android Package Kit (APK)-Datei an ein Endgerät ausgeliefert. Dabei handelt es sich um eine Archiv-Datei, die alle für den Betrieb der App notwendigen Dateien enthält. Das Archiv weist dabei die in nachfolgender Abbildung 2.1 dargestellte grundlegende Struktur auf.

```
apk/
├── AndroidManifest.xml
├── ressources.arsc
├── classes.dex
├── META-INF/
│   ├── *.RSA
│   ├── *.SF
│   └── MANIFEST.MF
├── res/
│   ├── fonts/
│   └── drawable/
├── assets/
└── lib/
    ├── armeabi
    └── x86
```

Abbildung 2.1: Struktur einer APK-Datei

Hier enthalten die `.dex` Dateien den für die in Android verwendete Laufzeitumgebung (Android Runtime) kompilierten Code. Im Falle einer mittels Java bzw. Kotlin entwickelten Anwendung fördert das Dekompilieren dieser Dateien den Quellcode der App zu-

tage. Demgegenüber enthält das `lib`-Verzeichnis für die jeweilige Prozessorarchitektur des Endgerätes kompilierte Bibliotheken; entsprechend auf Unterverzeichnisse für die infrage kommenden Architekturen aufgeteilt.
In `META-INF` sind alle Informationen zu finden, die für die Integritätsprüfung einer App notwendig sind. D.h. alle SHA256-Hashes der enthalten Dateien sowie die digitale Signatur des Erstellers.
Die Verzeichnisse `res` und `assets` enthalten nicht kompilierbare Dateien, wie Bilder, Videos oder Schriftartendefinitionen. Sie unterschieden sich darin, dass `res` eine fest vorgegebene Ordnerstruktur besitzt und darin, mittels welcher Android APIs jeweils auf sie zugegriffen werden kann. Die unter `assets` gespeicherten Dateien werden über ihren Dateinamen abgerufen, für die unter `res` gespeicherten Daten existiert in der Datei `resources.arsc` ein Mapping auf eine numerische ID [42, S.52].
Die `AndroidManifest.xml`-Datei enthält alle für die App wichtigen Meta-Informationen. Anzumerken ist, dass die Datei innerhalb der APK in einem Binär-Format vorliegt, auch wenn ihre Dateiendung etwas anderes vermuten lässt. Entsprechend ist spezielle Software, wie sie beispielsweise in der Entwicklungsumgebung Android Studio integriert ist, nötig, um die Datei in ein lesbares XML-Format zu überführen [43]. Zu den dann sichtbaren Informationen gehören unter anderem der Paket-Name, die Version und der Autor der App, aber auch ob beispielsweise ein automatisiertes Backup erlaubt ist. Daneben werden alle für die App erforderlichen Berechtigungen (engl. Permissions) angegeben, sogenannte Intent-Filter definiert und alle Komponenten der App wie Activities oder Hintergrund-Services gelistet [44]. Bei den Berechtigungen wird zwischen „normalen“ bzw. zur Installationszeit vergebenen und „gefährlichen“ Berechtigungen, für die die Zustimmung des Nutzers notwendig ist, unterschieden. Diese Unterscheidung findet allerdings erst seit Android 6 statt. In allen vorherigen Versionen wurden alle Berechtigungen zur Installation erteilt. Beispiele für „normale“ Berechtigungen sind der Zugriff auf das Internet, genauer die Möglichkeit Netzwerksockets zu erstellen und das Verbinden mit bekannten Bluetooth-Geräten. Zu den „gefährlichen“ zählt das Benutzen der Kamera und das Lesen bzw. Schreiben im externen Speicher. Weiterhin existieren noch „spezielle“ und „Signatur“ Berechtigungen. Erstere können nur von Systemapps, also Apps die Google bzw. die Smartphonehersteller mit den Systemen ausliefern, angefordert werden [45]. Zweitere Kategorie ist üblich für von App-Entwicklern selbst definierte Berechtigungen, um z.B. Teile der App anderen Apps zur Verfügung zu stellen. Diese selbst definierten Berechtigungen der Kategorie „Signatur“ zuzuordnen, bewirkt, dass sie nur Apps erteilt wird, die mit dem selben Schlüssel signiert wurden, wie die App, die die Berechtigung erstellt hat [46].
Neben dem Definieren von Berechtigungen ist das Beschreiben aller Komponenten einer App die Hauptaufgabe der AndroidManifest.xml. Zu diesen gehören Activities, Services, Broadcast-Receiver und Content-Provider. Activities sind dabei die einzelnen Ansichten einer App, die ein Nutzer angezeigt bekommen kann, beispielsweise bei einer Mail-App eine Sicht für alle empfangenen Mails und eine zum Verfassen einer neuen. Sind sie nicht in der Manifest-Datei definiert, können sie nicht gestartet werden [47]. Demgegenüber bezeichnen Services alle Hintergrunddienste einer App, z.B. zum Emp-

fangen neuer Nachrichten. Diese können aktiv sein, auch wenn ein Nutzer gerade nicht mit der App interagiert [48]. Broadcast-Receiver dienen dazu, auf Nachrichten/Events der eigenen App oder dem Android System zu reagieren. So wird beispielsweise vom System der Abschluss des Boot-Vorgangs mittels Broadcast bekannt geben. Innerhalb der eigenen App ist vorstellbar, dass ein Hintergrundservice das eintreffen einer E-Mail registriert, meldet und darauf mittels eines Broadcast-Receivers reagiert wird, indem dem Nutzer eine Benachrichtigung angezeigt wird. Receiver müssen dabei nicht zwingend in der Manifest-Datei bekannt geben werden. Sie können auch aus Quellcode heraus gestartet und beim System registriert werden. Sie sind dann allerdings nur solange aktiv, wie die App bzw. deren Activity, die sie registriert hat [49]. Content-Provider dienen dazu, Daten der eigenen App für andere Apps zur Verfügung zu stellen. Die Datenbereitstellung ist für die nutzende App mit einer relationalen Datenbank vergleichbar [50].
Während der Analyse der in einer Manifest Datei definierten Komponenten ist entscheidend, ob diese exportiert sind oder nicht. Dies ist damit zu begründen, dass nur exportierte Komponenten von anderen Apps und dem Android-System genutzt bzw. aufgerufen werden können und damit einen potentiellen Angriffspunkt darstellen können. Dabei ist anzumerken, dass in Bezug auf Activities mindestens eine exportiert sein muss. Diese stellt den standardmäßigen Start-Punkt der App dar und muss damit systemseitig aufrufbar sein, wenn der Nutzer das App-Icon antippt. Als exportiert gilt eine Komponente dann, wenn entweder die Eigenschaft `android:exported="true"` gesetzt ist oder mindestens ein Intent-Filter für sie definiert ist. Unter Intents sind dabei Objekte zur Inter-Prozess-Kommunikation (IPC) in Android zu verstehen und Intent-Filter beschreiben entsprechend, auf welche dieser Intents eine Komponente reagiert. Beispielsweise lassen sich Intents und Intent-Filter dafür nutzen, gezielt Activities mittels Aufruf eines URL zu starten.

Dieser Mechanismus wird als Deep-Linking bzw. App-Linking bezeichnet. Listing 2.1 zeigt eine beispielhafte Definitionen für Intent-Filter beider Link-Typen. Deep-Links unterscheiden sich dabei von App-Links darin, dass bei ihnen ein vom Entwickler definiertes URL-Schema verwendet wird. Außerdem bekommt der Nutzer beim Anklicken eines Links mit diesem Schema einen Systemdialog zu sehen, der ihm alle Apps anzeigt, die das Schema unterstützen. Aus diesen muss der Nutzer wählen, welche App schlussendlich den Link öffnen soll. App-Links setzen hingegen `http(s)` voraus und überlassen bei korrekter Implementierung die Wahl der App nicht dem Nutzer, sondern sind exakt für eine App registriert. Unter korrekter Implementierung ist dabei zu verstehen, dass zum einen die Option `android:autoVerify="true"` gesetzt ist und zum anderen, dass auf dem zur angegebenen Domain gehörigen Webserver eine sogenannte Digital Asset Links Datei hinterlegt ist. Genauer muss diese sich im `.well-known` Verzeichnis befinden, zwingend via `https` abrufbar sein und `assetlinks.json` heißen. Bezogen auf das Beispiel aus Listing 2.1 lautet die korrekte URL dieser Datei `https://example.com/.well-known/assetlinks.json`. Diese Datei enthält den Paket-Namen der App sowie den Fingerprint (SHA256-Hash) des Öffentlichen Schlüssels, mit dem die App

signiert wurde. Während des Installationsprozesses einer App prüft Android entsprechend auf das Vorhandensein und die Korrektheit dieser Datei. Es können mehrere App-Links für eine App registriert sein. Allerdings muss für jeden dieser Links die Prüfung der Digital Asset Link Datei erfolgreich ausfallen; andernfalls wird keiner ohne Nutzerinteraktion geöffnet [51]. Studien zeigen, dass eine korrekte Implementierung von App-Links nicht selbstverständlich ist und damit oftmals dem Nutzer die Entscheidung obliegt, welche App für welchen Link zuständig ist [52, 53].

```
<!-- Deep-Link -->
<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
<category android:name="android.intent.category.BROWSABLE" />
<data android:scheme="example" />
<data android:host="example.com" />
</intent-filter>
<!-- App-Link -->
<intent-filter android:autoVerify="true">
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
<category android:name="android.intent.category.BROWSABLE" />
<data android:scheme="https" />
<data android:host="example.com" />
</intent-filter>
```

Listing 2.1: Intent-Filter für App-Links und Deep-Links

2.5.3 App Sicherheit

Aus Sicht einer Android-App ist zur Laufzeit die Isolation des App-Prozesses gegenüber anderen Prozessen von Bedeutung. Dies wird dadurch realisiert, dass jeder App bei Installation ein eigener Linux-Benutzer, technisch gesehen also eine Nutzer-ID (UID), zugeordnet wird. Ebenfalls wird für jede App ein eigenes Daten-Verzeichnis im Ordner `/data/data` angelegt, das bei Deinstallation der App gelöscht wird. In diesem sind alle App-Daten in standardisierten Ordnern abgelegt. So liegen verwendete SQLite-Datenbanken im Ordner `databases`, sogenannte Shared Preferences - ein simpler key-value-Speicher im XML-Format - im Ordner `shared_prefs` und sonstige Dateien im Ordner `files`. Das Datenverzeichnis der Apps ist standardmäßig nicht für den Geräte-Nutzer zugreifbar. Dies ist lediglich mit Root-Berechtigungen möglich.
Weiterhin kann eine App wie bereits angedeutet Lese- und Schreibberechtigungen für den Externen Speicher eines Gerätes anfordern. Dieser ist grundsätzlich für den Nutzer einsehbar. Auch hier gibt es Bereiche, genauer die Unterordner von `Android/data`, in denen nur die jeweilige App Schreibberechtigungen hat. Dieser sogenannte Scoped-

Storage ist ab Android 10 verfügbar und ohne Deklaration zusätzlicher Berechtigungen nutzbar [54].

In Bezug auf kryptografische Schlüssel werden Entwicklern mittels des Keystore-Systems Möglichkeiten bereitgestellt, selbige sicher zu erzeugen, zu speichern und zu nutzen. Systemseitig können die Keystore-Funktionen auf drei Wegen implementiert sein: zum Ersten lediglich mittels Software, zum Zweiten unter Verwendung eines Trusted Execution Environments (TEE) und zum Dritten mittels eines speziellen Kryptografie-Chips, im Kontext von Android meist als Secure Element (SE) bezeichnet, aber auch unter der Bezeichnung Hardware Security Module (HSM) bekannt. Betrachtet man die Sicherheit der Schlüssel, nimmt diese in der genannten Reihenfolge der Implementierungen zu. Zu beachten ist, dass ab Android 7.0 eine hardwaregestützte Implementierung, sei es durch ein TEE oder SE, verpflichtend ist; für Android 6 wird dies nur streng empfohlen. Anzumerken ist weiterhin, dass bei einer hardwaregestützten Implementierung die Extraktion der Schlüssel auch für einen Root-Benutzer nicht möglich ist. Allerdings ist es mit Root-Rechten kein Problem, dafür zu sorgen, dass der Schlüssel einer App von einer anderen genutzt werden kann. Dies liegt daran, dass im Ordner `/data/misc/keystore/user_0` Dateien mit Verweisen auf die Schlüssel im SE bzw. TEE gespeichert werden. Die Dateinamen in diesem Ordner sind folgendermaßen aufgebaut: `<App-UID><Typ><Schlüssel-Name>`. Die App-UID bezieht sich dabei auf die App, die den Schlüssel angelegt hat und legitim nutzen darf. Über die Typ-Bezeichnung wird zwischen Nutzerzertifikat (USRCERT), Privatem Schlüssel (USRPKEY) und CA-Zertifikat (CACERT) unterschieden [42, S.174]. Durch entsprechendes Kopieren und Umbenennen lässt sich also der Schlüssel einer App mit einer anderen nutzen. Weiterhin zeigen Untersuchungen der finnischen Sicherheitsfirma F-Secure, dass dies nicht der einzige Angriffsvektor gegen die Keystore-API ist und Nutzungsfehler seitens der App-Entwickler das Entschlüsseln von Daten für einen Angreifer einfach gestalten. Die Forscher leiten aus ihren Erkenntnissen Best-Practices zur Nutzung des Keystores und damit in Beziehung stehender APIs, wie der zur Biometrischen Authentifizierung, ab. So sollten erstellte Schlüssel stets an die Bedingung gekoppelt sein, dass das Gerät entsperrt ist und vor Schlüsselbenutzung ebenfalls eine Authentifizierung stattfindet. Entwickler sollten dabei aber auch beachten, dass an Biometrische Daten gekoppelte Schlüssel ungültig werden, sobald z.B. ein neuer Fingerabdruck hinterlegt wird. Allen voran sollte der beim Keystore angefragte Schlüssel dann auch verwendet werden, um für die App wichtige Daten zu entschlüsseln [55].

2.6 OWASP Mobile Security Testing Guide und Mobile Application Security Verification Standard

Das Open Web Application Security Project (OWASP) ist eine nicht profitorientierte Stiftung, die sich mit der Sicherheit von Anwendung, Web-Diensten u.ä. beschäftigt und am 01. Dezember 2001 gegründet wurde. Alle Projekte der OWASP Foundation wer-

den quelloffen auf GitHub veröffentlicht. Zu einem der bekanntesten gehört dabei die OWASP Top Ten, eine alle vier Jahre (zuletzt 2021) veröffentliche Liste über die häufigsten Sicherheitslücken in Web-Anwendungen [56].

In Bezug auf mobile Anwendungen entwickelt das Mobile Security Project, ein Unterprojekt des OWASP, zum einen den Mobile Application Security Verification Standard (MASVS) [9] und zum anderen den Mobile Security Testing Guide (MSTG) [10]. Ersterer gibt dabei die Eckpunkte vor, die von einer sicheren mobilen App erfüllt werden sollten und zweiterer beschreibt ausführlich, auf was beim Test einer Anwendung in Abhängigkeit ihrer Plattform (Android oder iOS) zu achten ist und welche Tools für einen Test verwendet werden können. Der MASVS ist weiterhin Grundlage anderer Bewertungssysteme zur Anwendungssicherheit. Beispielsweise testet das BSI derzeit eine Technische Richtlinie für „Sicherheitsanforderungen an digitale Gesundheitsanwendungen“ [57]. Auch die Europäische Union hat 2016 eine Richtlinie zur sicheren Entwicklung von Smartphone-Anwendungen herausgegeben [58], die den MASVS als Grundlage nutzt.
Diese Arbeit stützt sich alleinig auf den MASVS, da mit Connect.Me auch eine Anwendung einer US-amerikanischen Firma in den untersuchten Anwendungen vertreten ist und es somit sinnvoll ist einen offenen, internationalen Standard heranzuziehen.

Der MASVS unterteilt seine Kriterien in acht Kategorien, wobei die achte „Anforderungen an Manipulationssicherheit/Resilienz“ eine Sonderstellung einnimmt. Sie kann je nach Sicherheitsanspruch weggelassen werden. Weiterhin wird in den Kategorien zwischen Punkten unterschieden, die von allen Apps zwingend erfüllt werden sollten und solchen, die nur für besonders sicherheitskritische Anwendungen, wie solche aus dem Finanz- oder Gesundheitswesen, relevant sind. Auch die hier untersuchten Wallet-Anwendungen sind in letztere Kategorie einzuordnen, da ihr Hauptzweck darin besteht, persönliche, der Identifikation dienenden Daten zu speichern und zu verarbeiten.
Weiterhin empfiehlt der MASVS einen offenen Review, d.h. Quellcode sowie Architektur- und Design-Dokumente liegen beim Test vor. Dies ist für die ausgewählten Anwendung nicht vollständig gegeben. Es handelt sich nur bei drei der fünf Apps um Open-Source Anwendungen. Da das Hauptaugenmerk dieser Arbeit auf der Untersuchung der Datensicherheit der Apps und damit den Punkten zwei „Anforderung an Datenspeicherung und Datenschutz“ und drei “Anforderungen an Kryptografie“ liegt, stellt dies kein schwerwiegendes Problem dar. Beide lassen sich nur mittels vorliegender APK prüfen.
Zusätzlich zu diesen beiden Schwerpunkten werden die Kategorien fünf „Anforderungen an Netzwerkkommunikation“ und sechs „Anforderungen zur Plattform-Interaktion“ behandelt. Auch Punkte aus Kategorie acht, wie die Forderung eine Root-Detektion durchzuführen, werden angewandt. Dies wird als sinnvoll erachtet, da das Rooten eines Gerätes gerade in Bezug auf Wallet-Anwendungen die Sicherheit massiv beeinträchtigt. Dies gilt v.a. durch die Möglichkeit, auf die privaten Speicherbereiche einer App zuzugreifen, sobald das Smartphone entsperrt und USB-Debugging aktiviert ist. Dennoch wird rooten als freie Entscheidung des Nutzers gesehen und somit sollte die Nutzung der App mit entsprechenden Sicherheitshinweisen hinsichtlich eines sicheren Sperr-

Passworts möglich bleiben.
Die übrigen Kategorien eins „Anforderungen an Architektur, Design und Bedrohungsanalyse“ , vier „Anforderungen an Authentifizierung und Session Management“ und sieben „Anforderungen an Code Qualität und Build-Einstellungen“ werden nicht geprüft, da sie aufgrund fehlender Dokumente nicht prüfbar sind (eins und sieben) oder für die Anwendungen keine Rolle spielen (vier).
Die Ergebnisse der Untersuchung sind in Anhang A tabellarisch zusammengefasst. Methodiken und ausführliche Beschreibungen der entsprechenden Funde sind den nachfolgenden Kapiteln 3 und 4 zu entnehmen.

Demgegenüber gibt der MSTG für die meisten Kategorien und Kriterien des MASVS an, auf was konkret geachtete werden sollte. Hierzu gehören auch Code-Ausschnitte, die zeigen, wie etwas (nicht) umgesetzt werden sollte. Diese beziehen sich allerdings nur auf Java bzw. Kotlin und sind damit nur nutzbar, wenn React Native/Xamarin/Flutter-Bibliotheken betrachtete werden, die Android-APIs für mit diesen Frameworks erstellte Apps verfügbar machen.
Weiterhin werden Tools aufgelistet, die für eine Untersuchung der Apps genutzt werden können. Für diese Arbeit wird vornehmlich auf Frida [59] zurückgegriffen. Dieses Tool dient der Analyse einer App zu ihrer Laufzeit (dynamische Analyse). Mit Frida ist es möglich, herauszufinden, ob gewisse Funktionen aufgerufen werden, ohne den Quellcode einer Anwendung zu kennen. Dies kann mittels Javascript-Skripten automatisiert werden. Solche wurden beispielsweise während der Analyse der Nutzung der Keystore-API von F-Secure entwickelt und zur Verfügung gestellt [55].
Insgesamt wird die dynamische Analyse auf einem Google Pixel 4a mit Android 11 durchgeführt, das mittels Magisk [60] gerootet wurde. Die Extraktion der App-Daten wird über die Android Debug Bridge (ADB), einem Kommandozeilen-Tool zur Interaktion mit über USB oder Netzwerk verbundenen Android-Geräten, vorgenommen. Die weitere Untersuchung der extrahierten Dateien wird auf einem Notebook mit Linux-Betriebssystem (Linux Mint 20), nachfolgend als Arbeitsrechner bezeichnet, durchgeführt.

3 Jolocom SmartWallet

Jolocom ist eine Berliner Firma, die sich seit ihrer Gründung 2014 mit digitalen Identitäten befasst. Seit 2017 liegt der Fokus der Firma dabei auf Selbst-Souveränen bzw. dezentralen Identitäten. Jolocom entwickelt dabei ein eigenes Protokoll zum Austausch ihrer Verifiable Credentials sowie eigene DID-Methoden (`did:jolo` und `did:jun`) für ihre Identifier. Die seit März 2018 in den App-Stores verfügbare Jolocom SmartWallet ist dabei die Anwendung für Endnutzer zum Verwalten ihrer Identitäten (Credentials) [61]. Diese wurde mittels React Native entwickelt und ist quelloffen auf Github verfügbar [62]. Bei der hier untersuchten Version handelt es sich um Version 2.2.0, die am 23.09.2021 veröffentlicht wurde.

3.1 Anwendersicht

Beim ersten Start der App muss diese eingerichtet werden. Hierzu wird der Nutzer anfänglich aufgefordert, Datenschutzerklärung und Allgemeine Geschäftsbedingungen (AGB) zu akzeptieren. Diese klären den Nutzer darüber auf, dass alle Identitätsdaten lokal und verschlüsselt auf dem Gerät hinterlegt werden. Entsprechend ist der Nutzer selbständig für Backups und ein aktuelles Betriebssystem verantwortlich. Weiterhin wird der Nutzer verpflichtet, seine Identitätsdaten wahrheitsgemäß anzugeben.

Im nächsten Schritt soll der Nutzer willkürlich auf dem Bildschirm zeichnen, vermutlich um Zufall für kryptografische Schlüssel zu erzeugen. Eine Analyse des Quellcodes ergibt, dass diese Vermutung zwar grundlegend korrekt ist, die „Zeichnung“ in der vorliegenden Version aber noch nicht verwendet wird. Stattdessen wird auf die React-Native Bibliothek `react-native-randombytes` zurückgegriffen. Diese bildet auf die Android/Java-Klasse `SecureRandom` ab [63]. Konkret wird ein 16 Byte langes Zufalls-Array erzeugt.

Im Anschluss daran werden dem Nutzer zwölf englischsprachige Wörter angezeigt, die er sich notieren und an einem sicheren Ort ablegen soll. Aus dem Umfeld von Kryptowährungen sind diese zwölf Wörter als Mnemonic bekannt. Dieser lässt sich aus einem Seed, also einer Zufallszahl wie sie im vorhergehenden Schritt erzeugt wurde, ableiten und umgekehrt wieder in einen solchen verwandeln. Der Seed kann dann schlussendlich genutzt werden, um deterministisch kryptografische Schlüssel zu erstellen [64]. Zusammenfassend ist ein Mnemonic also eine Möglichkeit, ein einfaches Backup kryptografischer Schlüssel zu erstellen. Jolocom schreibt allerdings in der Erklärung zu diesen Wörtern, dass sie verwendet werden können, um die Identität wiederherzustellen. Da diese nicht nur aus kryptografischen Schlüsseln sondern auch aus Credentials besteht, ist diese Aussage als irreführend für unbedarfte Nutzer zu bewerten. Erst wenn

ein Nutzer versucht, die Wallet mittels des Mnemonic wiederherzustellen und sich dabei eine Information anzeigen lässt, wie mit einem verlorenen Mnemonic erfolgt ein entsprechender Hinweis. Dieser informiert, dass zusätzlich eine Backup-Datei nötig ist, um alle Daten wiederzuerlangen. Abbildung 3.1 zeigt den angesprochenen Bildschirm mit dem Mnemonic sowie die beiden erwähnten Hinweis-Texte. Anzumerken ist dabei, dass die gezeigten Screenshots nicht direkt auf dem Testgerät aufgenommen werden konnten, da Jolocom SmartWallet Screenshots der App verbietet. Aus sicherheitstechnischer Sicht ist dieses Verhalten erwünscht. Mit der entsprechenden Einstellung der App wird so auch verhindert, dass in Androids Übersicht geöffneter, im Hintergrund befindlicher Anwendungen, keine App-Inhalte zu sehen sind.
Um dennoch Bildschirmfotos aufnehmen zu können, wurde der Handy-Bildschirm mittels des Tools Scrcpy [65] auf den Arbeitsrechner gespiegelt und die Bildschirmfotos mit der dort verfügbaren Software erstellt.

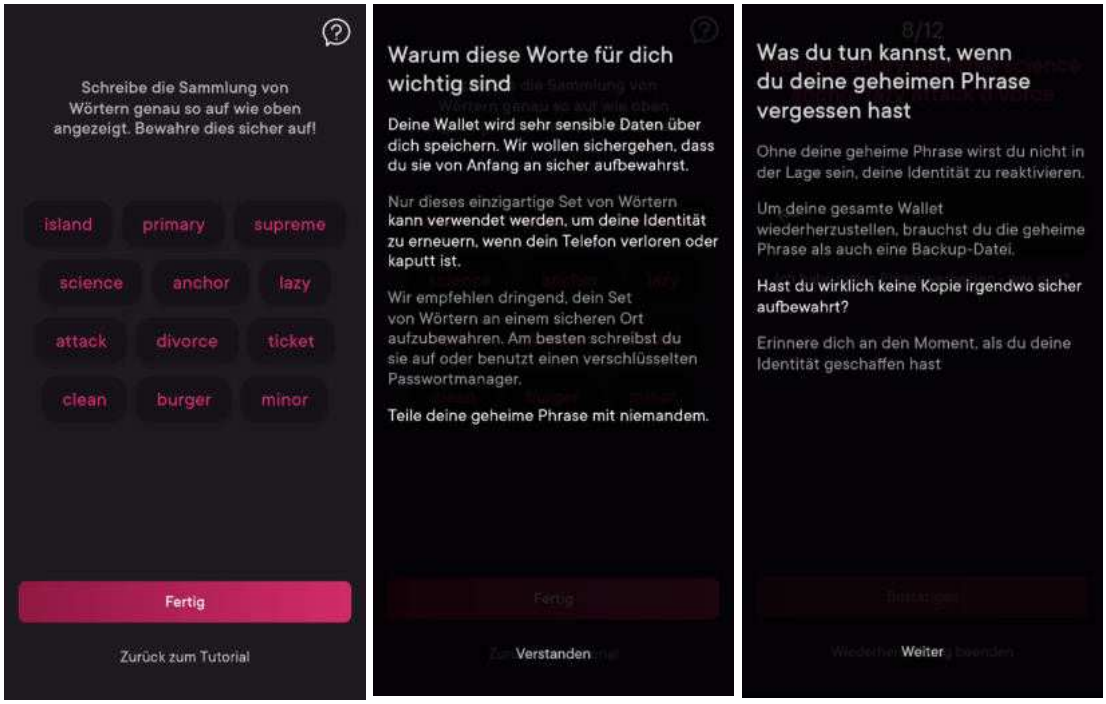


Abbildung 3.1: Mnemonics in Jolocom SmartWallet

Nachdem der Nutzer sich (bestenfalls) den Mnemonic notiert hat, muss er entweder die ersten oder letzten sechs Wörter in die richtige Reihenfolge bringen. Schlussendlich wird ein vierstelliger PIN vergeben, mit dem die App entsperrt werden kann. Sollten beim Android-System biometrische Daten zur Authentifizierung hinterlegt sein, kann auf diese zurückgegriffen werden. Eine Authentifizierung mittels PIN oder Biometrie findet bei jedem App-Start und wenn die App in den Vordergrund geholt wird, statt. Schlägt die biometrische Authentifizierung zu oft fehl, muss der PIN eingeben werden. Dieser darf fünfmal falsch sein, bevor die Wallet für eine Minute gesperrt wird. Während der Countdown herunterläuft, sollte sich die App im Vordergrund befinden, da dieser sonst angehalten wird. Anschließend bekommt der Nutzer noch zweimal fünf Versuche mit fünf Minuten Sperrung dazwischen, den PIN korrekt einzugeben. Danach ist der Mne-

monic nötig, um die Wallet wieder freizuschalten. Auch wenn der Nutzer direkt angibt, seinen PIN vergessen zu haben, wird er zur Eingabe des Mnemonic aufgefordert.

Am Hauptbildschirm der App angelangt, kann der Nutzer beginnen, seine Identität zu erstellen. Hierbei rückt Jolocom nicht den Ausstellprozess eines Credentials in den Vordergrund, sondern fordert den Nutzer auf, seine Daten selbst anzugeben. Dazu gehören Name, Telefonnummer, E-Mail-Adresse und Adresse. Technisch gesehen stellt sich der Nutzer also Credentials mit diesen Angaben selbst aus. Dies erklärt, warum die AGB zur wahrheitsgemäßen Angabe von Daten auffordern.

Um weitere Funktionalitäten der App testen zu können, stellt Jolocom eine Demo-Webseite (`https://interxns.jolocom.io/`) bereit. Über diese kann sich ein Nutzer Credentials ausstellen lassen, sie einem Verifier präsentieren oder sich für einen Dateizugriff autorisieren. Jede Aktion wird ausgelöst, indem mit der SmartWallet-App der angezeigte QR-Code gescannt wird. Dieser enthält ein JSON-Web-Token (JWT), das wiederum der App mitteilt, was zu tun ist.
Die Webseite bietet die Möglichkeit, einen zweiten QR-Code zu generieren, der einen App-Link codiert. Dieser enthält den JWT als Query-Parameter. Demnach sollte es möglich sein, diesen Code mit jeder anderen App zu scannen, die zum Lesen von QR-Codes in der Lage ist und zur SmartWallet-App geleitet zu werden. Allerdings musste festgestellt werden, dass selbst bei installierter SmartWallet lediglich die PlayStore-Seite der Anwendung geöffnet wird. Eine Analyse der AndroidManifest.xml gibt Aufschluss über die Ursache: In der untersuchten Version ist kein App-Link definiert. Die entsprechende `assetlink.json` Datei wurde aber bereits auf dem Webserver hinterlegt (`https://jolocom.app.link/.well-known/assetlinks.json`).

Wurde nun beispielsweise ein QR-Code gescannt, der das Ausstellen eines Credentials anregt, werden dem Nutzer die enthaltenen Attribute angezeigt. Anzumerken ist dabei, dass die Attributwerte stets mit einem grauen Balken verdeckt sind und auch nur die ersten drei Attribute angezeigt werden. Dem Nutzer bleibt also keine Chance, die eingetragenen Werte zu prüfen, bevor er das Credential akzeptiert.
Alle einmal akzeptierten Credentials werden als Dokumente in der App nach Kategorien oder Herausgebern geordnet, angezeigt. Auch hier sind lediglich die ersten drei Attribute, diesmal mit Werten, zu sehen. Erst eine Detailansicht offenbart weitere. Abbildung 3.1 zeigt entsprechend die drei erwähnten Ansichten.

Wird nachfolgend eine Aktion zum Vorzeigen des eben ausgestellten Credentials und z.B. des anfänglich eingegebenen eigenen Namens angeregt, werden zwar alle angefragten Credentials angezeigt, allerdings nicht all ihre Attribute. Ein Abwählen einzelner Attribute (Selective Disclosure) wird nicht unterstützt. Die zu sendenden Credentials müssen dabei noch einzeln ausgewählt werden. Erst wenn alle geforderten Credentials angewählt sind, ist ein Absenden der Antwort an den Verifier möglich.

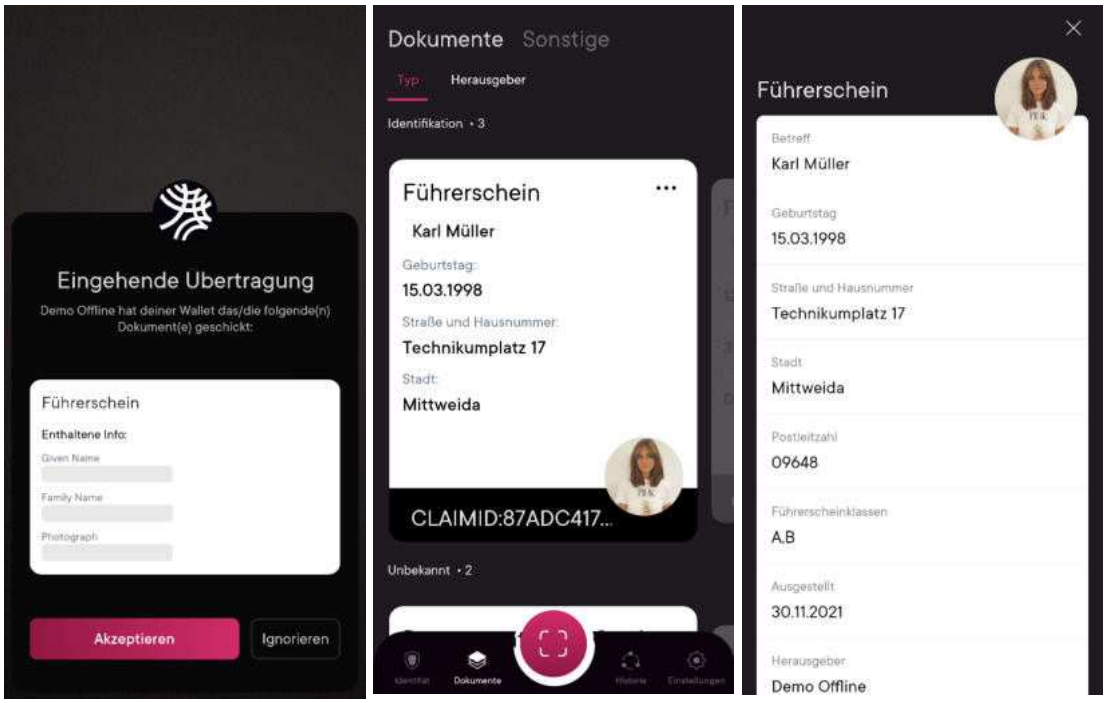


Abbildung 3.2: Anzeige der Credentials in Jolocom SmartWallet

Alle Aktionen, die über das Wallet ausgelöst wurden, werden dem Nutzer in einer Historie angezeigt. Dazu gehört, wann welches Credentials vom wem ausgestellt wurde und wann Anfragen für welche Credentials gemacht wurden.

In den Einstellungen der App ist es möglich, die Sprache einzustellen (Englisch oder Deutsch), den PIN zu ändern und die biometrische Authentifizierung zu aktivieren. Daneben werden Informationen zum App-Ersteller gegeben, die Möglichkeit, die App zu bewerten und ein Kontaktformular. Weiterhin werden Antworten auf beliebte Fragen gegeben. Hierzu gehört auch, was passiert, wenn das Smartphone verloren geht. An dieser Stelle wird drauf hingewiesen, dass der Anfangs angezeigte und sicher aufzubewahrende Mnemonic keine Credentials oder anderweitige Identitätsinformationen wiederherstellt. Um diese Daten zu sichern soll die Backup-Funktionalität im Einstellungsmenü genutzt werden. In der untersuchten Version der App war dieser Punkt allerdings nicht auffindbar.
Zuletzt wird dem Nutzer noch die Möglichkeit gegeben, alle Identitätsdaten zu löschen, die in seiner Wallet hinterlegt sind.

3.2 Speichermodell

Nachdem vorangehend Daten durch Nutzung der Demo-Website erzeugt wurden, werden diese nun vom Smartphone mittels ADB extrahiert. Nach einem Sichten der Daten stellt sich heraus, dass die für diese Arbeit interessanten in einer unverschlüsselten SQLite-Datenbank abgelegt werden. Es ist also möglich, alle ausgestellten Credentials

und damit private Daten zu lesen. Weiterhin werden einige Einstellungen in der Datenbank abgelegt. Hierzu gehört u.a. die Anzahl der Versuche, die ein Nutzer noch übrig hat, seinen PIN richtig einzugeben. Durch Manipulation dieses Wertes ist es möglich, ausreichend Versuche zu erlangen, alle möglichen vierstelligen PINs durchzuprobieren. Es findet augenscheinlich keine Code-seitige Prüfung auf einen Maximalwert statt.

Private Schlüssel fallen auf den ersten Blick nicht ins Auge. Allerdings erscheint die Tabelle `encrypted-wallet` diesbezüglich auffällig. Diese ordnet der DID, auf die alle Credentials ausgestellt werden, einen Zeitstempel und einen als encryptedWallet bezeichneten String zu. Eine Analyse des Quelltextes ergibt, dass dieser String mit einer rudimentären Implementierung der Universal Wallet-Spezifikation erzeugt wurde. Diese ist in Rust geschrieben und bietet lediglich Funktionen zum Verwalten und Nutzen kryptografischer Schlüssel (Ver- und Entschlüsseln, Signieren). Das Wallet selbst wird dabei mit XChaCha20Poly1305, einer Weiterentwicklung von ChaCha20Poly1305, verschlüsselt. Der Schlüssel ist der SHA3-256 Hash eines Passwortes [66, 67].

Es gilt nun herauszufinden, ob es möglich ist, besagtes Passwort ausfindig zu machen. In der Datenbank selbst deutet nichts auf ein solches hin. Auch der PIN der App ist hier nicht abgelegt. Da ein weiterer Ort der Datenspeicherung unter Android Shared Preferences sind, wird der entsprechende Ordner für diese als nächstes untersucht. Hier fällt die Datei `RN_ENCRYPTED_STORAGE_SHARED_PREF.xml` auf. Auf den ersten Blick lassen sich dieser keine Klardaten entnehmen. Es wird aber deutlich, dass zwei Datenwerte hier abgelegt sind, mutmaßlich der PIN für die Anwendung und das Passwort für encryptedWallet. Eine Suche im Quellcode bestätigt diese Vermutung. Weiterhin wird dabei deutlich, dass das Passwort ebenfalls mittels `react-native-randombytes` erzeugt wurde, 32 Byte lang und base64-codiert abgelegt ist. Auch lässt sich die React-Native Bibliothek finden, die für die Erstellung und Nutzung der Shared Preference Datei verantwortlich ist. `react-native-encrypted-storage` macht die Android-API für verschlüsselte Shared Preferences für eine React-Native Anwendung nutzbar. Entsprechend wird für die Speicherung der Verschlüsselungs-Schlüssel der Android Keystore verwendet. Allerdings bietet die React-Native Bibliothek keine Möglichkeit, die Schlüssel mit entsprechenden Attributen zu versehen, beispielsweise dass für die Nutzung eine Authentifizierung notwendig ist. Auch eine Auswahl des Verschlüsselungsalgorithmus ist nicht möglich. Es wurde aber mit AES im GCM Modus eine sichere Standard-Einstellung gewählt [68].

Alle weiteren gesicherten Ordner verbergen in Bezug auf die Ablage von Kryptografischen Schlüsseln und Verifiable Credentials keine interessanten Daten. Also wird nachfolgend die App mittels Frida direkt zur Laufzeit untersucht. Insbesondere wird dabei ein Script von F-Secure verwendet, dass den Aufruf von Android-Keystore Funktionen analysiert. Entsprechend ist das Script in der Lage, verwendete Algorithmen sowie die Eingabe und Ausgabewerte der Funktionen anzuzeigen. Damit wird sichtbar, was wann verschlüsselt und entschlüsselt wird. Erwartet wird, dass zumindest das Passwort für

das encryptedWallet erst nach erfolgreicher PIN-Eingabe im Klartext zu sehen ist. Es muss allerdings festgestellt werden, dass diese Erwartung nicht erfüllt wird. Sämtliche Keystore-Operationen finden beim Start der App statt, ohne dass der Nutzer mit dieser interagieren musste. Dementsprechend lassen sich der App-PIN und das Wallet-Passwort ablesen.

Abschließend wird geprüft, was mit der Datenbank passiert, wenn der Nutzer sein Passwort zu oft falsch eingegeben hat und die App entsprechend gesperrt ist. Nach Forderungen des MASVS sollten sämtliche App Daten in diesem Fall gelöscht werden. Ein Kopieren und Analysieren der App-Daten einer gesperrten Version ergab, dass dies nicht geschieht.

3.3 Netzwerkverkehr

Nachdem sich mit der Speicherung von Credentials und kryptografischen Schlüsseln detailliert beschäftigt wurde, wird nun der Netzwerkverkehr der App untersucht. Um eine Untersuchung dieses erfolgreich durchzuführen und entsprechend mittels TLS verschlüsselten Verkehr analysieren zu können, ist es nötig, selbigen über einen Proxy-Server auf dem Arbeitsrechner zu leiten. Weiterhin muss das Zertifikat des Server erfolgreich auf dem Endgerät installiert werden.
Grundsätzlich lässt sich unter Android ein Proxy-Server manuell für jedes gespeicherte WLAN-Netzwerk hinterlegen. Allerdings nutzen nicht alle Apps diese systemweite Proxy-Einstellung. Um dieses Problem zu umgehen, wird der Arbeitsrechner als Standard-Gateway des genutzten WLAN-Netzwerkes gesetzt. Aus Sicht des Smartphones muss nun also der Arbeitsrechner wie ein Router fungieren und jeglichen Netzwerkverkehr ins Internet weiterleiten. Alle Pakete werden vorher allerdings durch den Proxy-Server geleitet, um ihren Inhalt einzusehen. Die gewählte Proxy-Software ist MITM-Proxy, konkret wurde also der Anleitung unter [69] gefolgt, um das System entsprechend zu konfigurieren. Listing 3.1 zeigt die verwendete Befehlsfolge.

```
sudo sysctl -w net.ipv4.ip_forward=1 &&
sudo sysctl -w net.ipv6.conf.all.forwarding=1 &&
sudo sysctl -w net.ipv4.conf.all.send_redirects=0 &&
sudo iptables -t nat -A PREROUTING -p tcp -j REDIRECT --to-port 8080 &&
sudo ip6tables -t nat -A PREROUTING -p tcp -j REDIRECT --to-port 8080
```

Listing 3.1: Befehle zur Konfiguration des Arbeitsrechners

Das Zertifikat des Proxy-Servers wurde anfänglich über die Android-Sicherheitseinstellungen als CA-Zertifikat installiert. Allerdings werden Selbst-Installierte CA-Zertifikate ab Android 7 nicht mehr als vertrauenswürdig eingestuft. Entsprechend werden HTTPS-Anfragen über korrekt verwendete Systembibliotheken nicht durchgeführt. Demzufolge gelang es mit diesen Konfigurationen noch nicht, Netzwerkverkehr der Jolocom Smart-

Wallet mitzulesen. Genauer verweigerte die App HTTPS-Anfragen, die z.B. nötig sind, um mit der Demo-Webseite für die Ausstellung von Credentials zu kommunizieren. Auch wenn es aktuell als Problem dargestellt wird, den Netzwerkverkehr nicht mitlesen können, ist dies aus Sicht der App-Sicherheit wünschenswertes Verhalten.

Dennoch besteht das Ziel weiterhin, einen vollständigen Einblick zu bekommen. Daher soll auch dieser Schutz umgangen werden. Dies lässt sich zum einen mittels eines Frida-Scripts wie [70] bewerkstelligen, welches in der Lage ist, zahlreiche Funktionen zum sogenannten Zertifikats-Pinning ausfindig zu machen sowie zu überschreiben. Unter Zertifikats-Pinning ist zu verstehen, dass sich eine App merkt, welchen TLS-Zertifikaten sie vertrauen kann bzw. konsequent prüft, ob das vorgelegte Zertifikat vom Betriebssystem als vertrauenswürdig eingestuft ist.
Unter Nutzung dieses Frida-Scripts war es möglich, Einblick in den Netzwerkverkehr der Jolocom SmartWallet zu erhalten. Der Ausgabe des Scriptes ist zu entnehmen, dass hierzu Pinning Methoden der von Jolocom bzw. allgemein React-Native verwendeten Netzwerk-Bibliothek OkHttp umgangen werden mussten.

Bei der Analyse des Netzwerkverkehrs konnten keine Überraschungen festgestellt werden. Nachdem beispielsweise auf der Demo-Webseite ein QR-Code gescannt wurde, der die Ausstellung eines Credentials veranlasst, wird selbiges angefragt und zum Server zurückgesendet. Ist die DID des Ausstellers dabei unbekannt, wird diese vorher in ihr DID-Document aufgelöst. Da Aussteller die Methode `did:jolo` nutzen, sind hierfür zwei Schritte nötig. Diese DID-Methode sieht vor, das DID-Document im verteilten Dateisystem IPFS (Inter-Planetary File System) abzulegen. In einem solchen sind Dateien anhand ihres SHA256-Hashes auffindbar. Die Zuordnung zwischen diesem Hash und der DID wird in einem Blockchain-Netzwerk, hier dem Ethereum-Testnetzwerk Rinkeby abgelegt. Entsprechend muss zuerst eine Anfrage an dieses Netzwerk stattfinden. Hierzu wird ein Knoten des Dienstleisters Infura (`https://rinkeby.infura.io/v3/64fa85ca0b28483ea90919a83630d5d8`) verwendet. Anschließend erfolgt eine Abfrage in IPFS. Hierfür stellt Jolocom einen eigenen Zugangspunkt (`ipfs.jolocom.com`).

Weiterhin ist zu sehen, dass alle Anfragen TLS in der aktuellsten Version 1.3 benutzen und zur Datenverschlüsselung die Algorithmen-Kombination `TLS_AES_256_GCM_SHA384` verwendet wird.

4 Lissi / esatus / ID-Wallet / Connect.Me

Nachdem sich im vorangegangenen Kapitel auf eine App konzentriert wurde, untersucht dieses die vier verbleibenden Wallet-Anwendungen. Dies ist damit zu begründen, dass diesen vier Wallets dieselbe SSI-Technologie zugrunde liegt. D.h. sie alle Nutzen dieselben Open-Source Bibliotheken zur Erstellung, Speicherung und Austausch von Credentials. Konkret sind dies Bibliotheken aus den Projekten Hyperledger Indy und Hyperledger Aries. Indy bildet dabei die Codebasis für ein Distributed Ledger (DLT) System zur Verwaltung von für Credentials notwendigen Metadaten. Es sind derzeit mehrere auf Indy basierende DLT-Netzwerke in Betrieb. Für internationale Anwendungsfälle sei hier das Sovrin-Netzwerk genannt. Im deutschsprachigen Raum spielt vermehrt das IDUnion-(Test-)netzwerk eine Rolle.
Hyperledger Aries stellt demgegenüber die Protokolle und Bibliotheken für den Austausch von Credentials, wie auch Formalien zum Aufbau eines Wallets (siehe Abschnitt 2.4).

Connect.Me ist das Produkt der US-Amerikanischen Firma Evernym, die zu den Initiatoren des Hyperledger Indy Projektes gehört. Die App selbst wurde mit React-Native entwickelt und ihr Code ist auf GitLab verfügbar [71]. Die untersuchte Version ist Version 1.6.2 veröffentlicht am 24.08.2021.
Die übrigen drei Apps wurden mittels Xamarin, einem Open-Source App-Framework, das vornehmlich von Microsoft vorangetrieben wird, entwickelt. Die Lissi Wallet ist dabei die Wallet-Anwendung des Lissi[4]-Projekts, das seit 2019 läuft. Beteiligt an dem Projekt sind u.a. die Main Incubator GmbH, die Bundesdruckerei GmbH, die Commerzbank AG und die TU Berlin [4]. Für die Untersuchung liegt die Lissi Wallet in Version 1.3.3 vor, die am 05.11.2021 veröffentlicht wurde.
Die esatus Wallet wird von der in Hessen und Hamburg ansässigen IT-Firma esatus AG entwickelt. Sie liegt derzeit in Version 1.11 vom 05.12.2020 vor. Entsprechend wird vermutet, dass diese App nicht mehr aktiv weiterentwickelt wird. Ihr Nachfolger ist offenbar die ID-Wallet. Diese wird von der Digital Enabeling GmbH, einer Tochterfirma der esatus AG, bereitgestellt. Sie ist die Wallet-App des vom Bundeskanzleramt geführten Projekts „Ökosystem digitale Identitäten“. Anfangs nur für den prototypischen Anwendungsfall eines Check-Ins von Geschäftsreisenden in Hotels vorgesehen, kam im September 2021 noch die Funktion hinzu, ein digitales Abbild des Führerscheins zu speichern. Mit dieser Funktion wurde die App der Öffentlichkeit präsentiert. Aufgrund von Konfigurationsfehlern in Back-End Systemen und von Sicherheitsforschern kritisierten Mängeln in Aries Protokollen wurde die App einige Tage später aus den App-Stores entfernt [72]. Daher liegt zur Untersuchung nur Version 1.2 vor. Weiterhin ist der Quellcode der App offen verfügbar [73]. Aus diesem geht hervor, dass seit der vorliegenden Version keine tiefgreifenden Änderungen am Speichermodell vorgenommen wurden.

[4] Lissi ist ein Akronym für Let’s initiate Self-Sovereign Identity

4.1 Hyperledger Indy und Hyperledger Aries

Um nachfolgend die Funktionsweise der Apps besser verstehen zu können, soll hier zunächst auf die Hintergrundsysteme und Protokolle eingegangen werden, die notwendig sind, um Credentials in die vier vorgestellten Apps auszustellen und mit Verifiern auszutauschen. Das verwendete und im Aries Projekt spezifizierte und implementierte Kommunikationsprotokoll heißt DIDComm. Dies ist ein Nachrichten-basiertes Protokoll, das unabhängig vom darunter genutzten Übertragungsprotokoll verwendbar sein soll. Ein Beispiel für ein derzeit nutzbares Übertragungsprotokoll ist http(s).

Die Grundlage einer Interaktion zweier Parteien über DIDComm ist eine bestehende Verbindung. Darunter ist zu verstehen, dass beide Parteien das Öffentliche Schlüsselmaterial, meist assoziiert mit einer DID, des jeweils anderen besitzen. Dies wird dafür genutzt, die einzelnen Nachrichten zu verschlüsseln. Für jede Verbindung sollte eine eigene DID / eigenes Schlüsselmaterial verwendet werden.

Über eine solche Verbindung kann beispielsweise ein Issuer ein Credential ausstellen. Voraussetzung ist, dass der Issuer in einem Hyperledger Indy-Netzwerk ein Credential Schema und eine Credential Definition hinterlegt hat. Ersteres beschreibt den Aufbau des Credentials, also welche Attribute sind in welcher Reihenfolge enthalten und zweiteres gibt den Öffentlichen Schlüssel des Issuers für Credentials dieses Schemas an. Die Notwendigkeit dieser beiden Datenstrukturen ist der Verwendung von CL-Signaturen geschuldet. Demnach sind alle in Indy-Wallets ausgestellte Credentials anonyme Credentials (s. Abschnitt 2.3). Die aktuell verwendeten CL-Signaturen basieren auf RSA-Prinzipien mit einer Schlüssellänge von 2048 Bit und sind demnach mit dessen Sicherheitsniveau vergleichbar.

Hyperledger Indy bringt nicht nur Software-Bibliotheken zum Aufbau und Betrieb eines verteilten Netzwerkes mit, sondern auch eine Implementierung eines Wallets. Dabei wurde die in Abschnitt 2.4 vorgestellte Spezifikation aus dem Aries Projekt umgesetzt.

4.2 Anwendersicht

Auch bei diesen vier Apps sieht sich der Nutzer zuerst mit der Einrichtung selbiger konfrontiert. Diese besteht im Falle von esatus und ID-Wallet lediglich aus dem Festlegen einer fünf- bzw. sechsstelligen PIN. Connect.Me und Lissi verlangen neben dem Angeben einer sechsstelligen PIN noch die Zustimmung zu den Nutzungsbedingungen und Datenschutzbestimmungen. Im Fall von Lissi klären diese darüber auf, dass die App sich noch in der Entwicklung befindet und noch nicht für einen produktiv-Einsatz verwendet werden sollte. Weiterhin wird über die Funktionsweise der App aufgeklärt und erläutert, wie Credentials empfangen und ausgetauscht werden können. Hierbei wird besonders auf einen Mediator-Service hingewiesen, der von der Main Incubator GmbH

betrieben wird und dem Weiterleiten aller Nachrichten dient, die im Zusammenhang mit dem Credential-Austausch stehen.
Connect.Me weist lediglich auf die lokale Speicherung der Daten hin und fordert von den Nutzern, dass sie grundlegende Kenntnisse über Public-Key Kryptografie und Blockchain-Systeme haben.
In Bezug auf eine Datenschutzerklärung verweisen esatus und ID-Wallet auf die Webseiten ihrer jeweiligen Hersteller. Die hier auffindbaren Erklärungen enthalten jeweils kurze Absätze zur App selbst. Auch hier wird auf einen Mediator-Dienst hingewiesen, der von esatus auf Infrastruktur von Amazon Web Services betrieben wird. esatus gibt weiterhin vergleichbar mit Lissi einige Hinweise zur Funktion der App und des Credential Austauschs.

Bei Connect.Me fällt vor der eigentlichen Einrichtung der App auf, dass geprüft wird, ob das Smartphone gerootet ist. Ist dies der Fall, wird die Einrichtung nicht gestartet. Aus dem Quellcode geht hervor, dass hierfür die SafetyNet-Api von Google verwendet wird. estaus und ID-Wallet geben in ihren Datenschutzbestimmungen zwar auch an, SafetyNet zu benutzen, allerdings ist dies bei der Nutzung der Apps auf dem Testgerät nicht aufgefallen. Dies kommt nur zum Tragen, wenn versucht wird, die Apps auf einem emulierten Android-Gerät zu starten. Auf einem solchen startete keine der drei verbleibenden Apps.

Um Connect.Me auf dem Testgerät nutzen zu können, erwies sich eine Information aus der AndroidManifest.xml Datei als hilfreich: Diese verriet, dass die App Backups über Google-Dienste und ADB erlaubt. Entsprechend wurde die App zuerst auf einen unveränderten Android-System (gesperrter Bootloader, kein Root) installiert und eingerichtet. Nachfolgend wurde mittels ADB ein Backup erstellt und die App-Daten aus diesem auf dem gerooteten Gerät wiederhergestellt. Connect.Me konnte nachfolgend problemlos auf dem Testgerät genutzt werden. Eine Überprüfung mittels SafetyNet findet also lediglich einmalig vor der Einrichtung der App statt.

Zur Nutzung der Apps muss nach der Einrichtung bei jedem Start der festgelegte PIN eingeben werden. Mit Ausnahme von ID-Wallet erlauben die übrigen Apps, dies durch Abfrage biometrischer Daten (Fingerabdruck) zu ersetzen.
Bei Lissi und esatus hat der Nutzer anfänglich fünf Versuche zur PIN-Eingabe, bevor eine gewisse Zeit gewartet werden muss, bis der nächste Versuch möglich ist. esatus lässt den Nutzer dabei stets fünf Minuten warten, bis er genau einen nächsten Versuch bekommt. Lissi hingegen erhöht die Zeit in drei Schritten auf 10 Minuten. Ist biometrische Authentifizierung aktiviert gehen dem fünf Versuche mit einem Fingerabdruck voraus, bevor selbige Authentifizierungsmethode deaktiviert wird.
ID-Wallet erlaubt maximal fünf Eingabe-Versuche, bevor kein Zugriff auf die Wallet gewährt wird. Nach vier Versuchen erfolgt eine entsprechende Warnung.
Connect.Me nutzt ebenfalls die Strategie, die Zeit zwischen einzelnen, falschen Eingabeversuche zu erhöhen. Dies beginnt nach vier Fehlversuchen mit einer Minute Sper-

rung. Die Sperrzeit erhöht sich Schrittweise. Nach 10 Fehlversuchen ist die App bereits 24 Stunden gesperrt. Nach elf gescheiterten Eingaben wird überhaupt kein Zugriff mehr gewährt. Ist eine Authentifizierung mittels Fingerabdruck aktiv, darf der Nutzer fünfmal einen falschen Abdruck präsentieren, bevor eine 30-sekündige Sperrung erfolgt. Dieser Zyklus (fünf falsche Abdrücke mit 30 Sekunden Sperrung) kann dreimal durchlaufen werden, bevor laut Hinweistext die Eingabe eines Passwortes notwendig ist. Da in diesem Zustand keine Tastatur zur Eingabe angezeigt wird, kann zum einen nicht überprüft werden ob mit Passwort der PIN der App gemeint ist und zum anderen die App nicht weiter genutzt werden. Grundsätzlich ist anzumerken, dass bei aktiver Fingerabdruck-Authentifizierung keine Möglichkeit besteht, auf eine PIN-Eingabe zu wechseln, sollte der entsprechende Sensor nicht mehr funktionieren.

Um nachfolgend die Wallets mit Credentials zu füllen, werden entsprechende Demo-Webseiten genutzt. Connect.Me wie auch Lissi stellen solche zur Verfügung (`https://try.connect.me/#/Home` und `https://lissi.id/demo`). Beide Demos bilden verschiedene Anwendungsfälle ab, beispielsweise das Ausstellen eines Personalausweises durch das Bürgeramt oder das Einrichten eines Bankkontos. Auch esatus stellt eine Demo bereit (`https://wallet-demo.esatus.com/`). Da diese allerdings nur in Lage ist, ein einziges Credential auszustellen, wird sie nicht weiter in Betracht gezogen.

Lissis Demo ist dabei mit allen vier untersuchten Anwendungen kompatibel. Anzumerken ist dabei, dass im Falle von esatus- und ID-Wallet noch das richtige Indy-Netzwerk eingestellt werden muss, da für die Demo alle Credential Schemas und Definitionen der Beispiel-Credentials im IDUnion-Testnetzwerk hinterlegt sind. Dieses wird nicht standardmäßig von den beiden genannten Apps genutzt. Die Apps warnen dabei, dass „existierende Verbindungen und Credentials nicht mit einem anderen Ledger kompatibel sind“. Dies liegt daran, dass für die Überprüfung eines Credentials eine Credential Definition notwendig ist. Diese ist zwar über eine ID eindeutig im Credential spezifiziert, ist aber netzwerkspezifisch. Das Netzwerk selbst geht weder aus dem Credential noch der genannten ID hervor. Für unbedarfte Nutzer ist dieser Sachverhalt nicht trivial durchschaubar. Gleichzeitig ist dies ein Problem der dahinterliegenden Protokolle und Spezifikationen von Indy und Aries. Da sich aktuell eine Spezifikation zu einer `did:indy`-Methode in Arbeit befindet, die auch für Credential Definitionen und Schemas nutzbar ist [74], ist davon auszugehen, dass das genannte Problem in Zukunft gelöst ist.
Connect.Me und Lissi erlauben derzeit prinzipiell keinen Wechsel des Netzwerkes. Es bleibt also zu diesem Zeitpunkt noch fraglich, wie Connect.Me sowohl mit Lissis Demo und damit IDUnion sowie der eigenen Demo, die Credentials auf Basis des Sovrin Netzwerkes ausstellt, zusammenarbeiten kann.

Beide Demos lassen sich dem Anschein nach sowohl mit dem Browser des Test-Smartphones als auch auf einem anderen Gerät, z.B. einem Laptop benutzen. In letzterem Fall wird vergleichbar zu Jolocom ein QR-Code angezeigt, der abgescannt werden muss. Selbiger enthält im Fall der Lissi-Demo einen Deep-Link beginnend mit dem

Schema `didcomm` und zum Verbindungsaufbau notwendige Informationen. Er kann also mit jedem beliebigen QR-Code Reader gescannt werden. Im Falle der Connect.Me Demo enthält der Code lediglich einen https-Link zu einer JSON-Struktur, die dann wiederum alle nötigen Informationen zum Verbindungsaufbau enthält. Hier kann der Link also nur mit der Connect.Me-App gescannt werden. Dies begründet auch, warum diese Demo nicht mit den übrigen drei Apps kompatibel ist.
Im Falle der Nutzung der Demos im Mobilbrowser werden entsprechende Buttons angezeigt, die zu den Apps führen sollen. Im Falle von Lissi wird der auch im QR-Code codierte Deep-Link aufgerufen. Im Falle von Connect.me funktioniert die Weiterleitung in die App nicht. Es wird grundsätzlich nur die Play-Store-Seite der App angezeigt. Eine Auswertung der AndroidManifest.xml ergibt, dass für Connect.Me zwei App-Links und mehrere Deep-links, u.a. mit dem Schema `didcomm`, registriert sind. Für den ersten App-Link `https://connectme.app.link` ist eine korrekte `assetlinks.json` an der erwarteten Stelle auffindbar. Dies ist bei dem zweiten Link `https://link.comect.me` nicht der Fall.
Auch für esatus Wallet sind App-Links definiert wurden, ohne `assetlinks.json` Dateien zu hinterlegen. Auch die Links selbst (`https://transaction` , `https://aries_connection_invitation` und `https://invite`) lassen die Frage aufkommen, ob die Entwickler den Grundgedanken von App-Links, die Assoziierung einer App mit der eigenen Webseite, verstanden haben.

Ist die Verbindung zur jeweiligen Demo-Institution einmal aufgebaut, passiert das Ausstellen eines Credentials oder das Abfragen von Identitätsinformationen auf Grundlage bereits ausgestellter Credentials automatisch. Die Apps zeigen Benachrichtigung an, dass entweder ein neues Credential vorliegt oder eine Anwendung Informationen anfragt. Bei letzterem werden die angefragten Informationen dargestellt. Connect.Me zeigt dabei vorrangig diese an und markiert, aus welchem Credential sie jeweils stammen. Die übrigen zeigen eine Liste der Credentials an, die in der Freigabe enthalten sein werden und geben jeweils die Attribute mit an. Dass hierbei nicht alle Attribute eines Credentials freigegeben werden, ist auf den ersten Blick nicht ersichtlich, da lediglich die angefragten Attribute zu sehen sind. Bei ID-Wallet und Connect.Me muss das Senden der freigegebenen Daten durch die Eingabe der PIN bestätigt werden.

Nachfolgend stellen alle Wallets auf geeignete Weise die Credentials dar und zeigen auf, wann welches mit wem geteilt wurde und welche Attribute dabei freigegeben wurden. Kritikpunkt hierbei ist, dass alle Attributnamen unabhängig der eingestellten Sprache auf Englisch angezeigt werden. Mit Ausnahme von ID-Wallet können auch alle bestehenden Verbindungen eingesehen werden. Abbildung 4.1 verdeutlicht dies. Im Gegensatz zu Jolocom SmartWallet konnten diese Screenshots direkt auf dem Testgerät erstellt werden. Keine der Apps verbietet das Aufnehmen solcher. Entsprechend sind auch in Androids Übersicht der geöffneten Apps potentiell sensible App-Inhalte sichtbar.

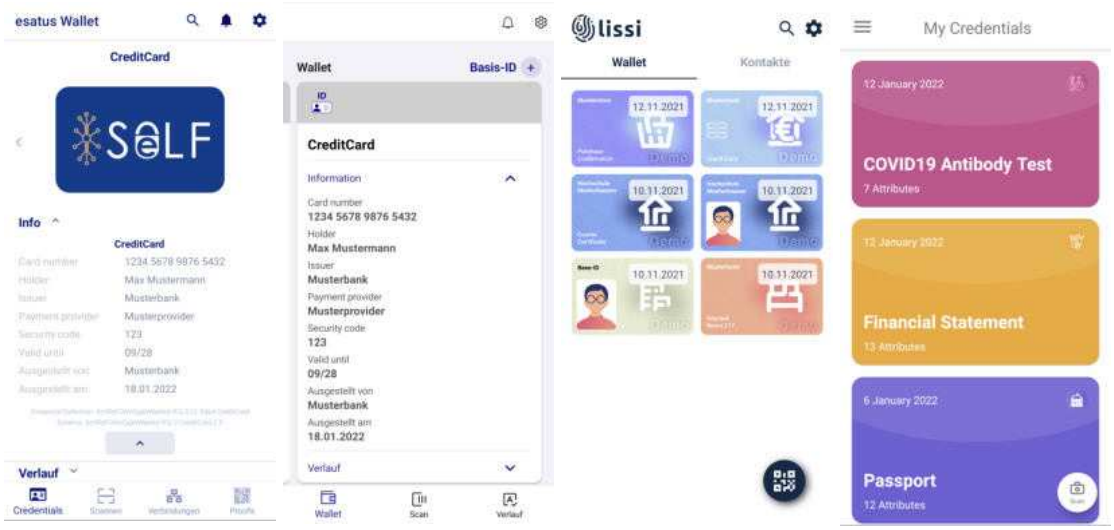


Abbildung 4.1: Credential-Übersicht der einzelnen Apps

Alle Apps stellen weiterhin ein Einstellungs-Menü zur Verfügung, hier kann z.B. biometrische Authentifizierung (außer bei ID-Wallet) aktiviert werden oder der PIN gewechselt werden. Weiterhin können hier nochmals Datenschutzbestimmungen und Nutzungsbedingungen sowie die Lizenzen verwendeter Softwarebibliotheken eingesehen werden. Im Fall von Connect.Me steht auch ein Kontaktformular zu Verfügung. Interessant ist, dass esatus Wallet die einzige App ist, die an dieser Stelle eine Backup-Funktion anbietet. Möchte ein Nutzer ein Backup erstellen, werden ihm zwölf Wörter angezeigt, die er notieren soll. Die Vermutung liegt nahe, dass auch dies ein Menmonic ist, der in einen Seed verwandelt werden kann, aus dem sich wiederum Schlüssel zur Verschlüsselung der Datei ableiten lassen. Laut Erklärung sind sie auch der Schlüssel für die Wiederherstellung. Nachfolgend müssen diese zwölf Wörter in der richtigen Reihenfolge aus einer Liste ausgewählt werden. Schlussendlich kann ein Dateiname gewählt und die Datei geteilt werden (E-Mail, GoogleDrive, Nearby-Share,...). Eine lokale Speicherung auf dem Gerät wird nicht angeboten.
Bei Betrachtung der Datei konnte festgestellt werden, dass diese verschlüsselt ist. Soll nun das Backup wiederhergestellt werden, muss die entsprechende Datei aus dem lokalen Speicher oder Google Drive ausgewählt werden. Anschließend muss der Schlüssel, also die zwölf notierten Wörtern, mit Leerzeichen getrennt, eingegeben werden. Der Nutzer sollte beachten, vor der Wiederherstellung auf das richtige Indy-Netzwerk zu wechseln, zu dem die im Backup gespeicherten Credentials gehören, da esatus selbige strikt Netzwerk-abhängig speichert. Ein Hinweis seitens der App zu diesem Sachverhalt erfolgt im Wiederherstellungsprozess nicht. Lediglich die Ansicht zum Wechseln des Netzwerks zeigt einen entsprechenden Hinweis. Abbildung 4.2 zeigt Bildschirmfotos des gesamten Vorgangs.

Um konkret überprüfen zu können, was beim Backup-Prozess passiert und herauszufinden, wie die Datei verschlüsselt wird, muss der Quellcode der App betrachtet werden. Allerdings liegt dieser nicht vor. Xamarin Apps liefern ihn aber indirekt in ihrer APK mit.

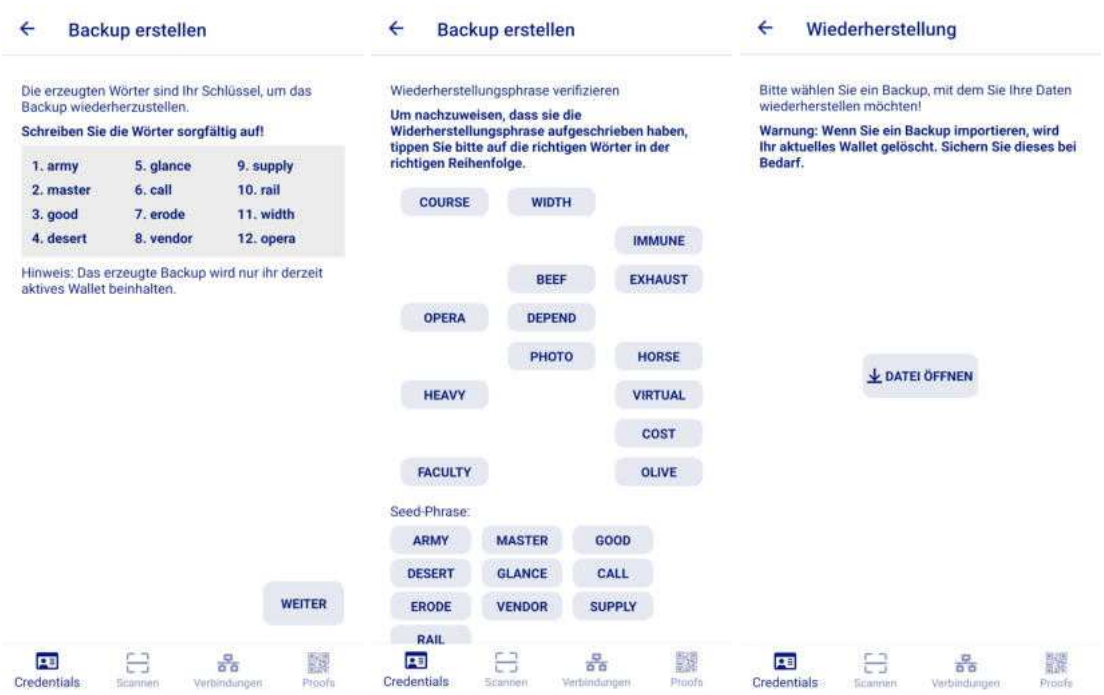


Abbildung 4.2: Erstellen und Wiederherstellen eines Backups in esatus Wallet

Im `assemblies`-Ordner selbiger liegt dieser in kompilierter Form als `.dll`-Datei. Derartige Dateien lassen sich mit Tools wie ILSpy problemlos dekompilieren. Einstiegspunkt ist dabei stetes die Datei, die den gleichen Namen wie die App trägt, hier `esatus.dll`. Das Dekompilat weist erwartungsgemäß eine mit dem Quellcode von ID-Wallet vergleichbare Struktur auf.

Es zeigt sich, dass zum Exportieren die Indy-Funktion `indy_export_wallet` genutzt wird. Dieser muss ein Dateipfad und eine Passphrase, also ein bestenfalls sehr langes Passwort übergeben werden. Als solche wird der zum angezeigten Menmonic gehörige Seed genutzt. Aus diesem wird mittels der Schlüsselableitungsfunktion Argon2 ein Kryptografischer Schlüssel abgeleitet. Dieser und der Algorithmus XChaCha20Poly1305 werden zur Verschlüsselung des Exports verwendet, bevor dieser in eine Datei geschrieben wird [75]. Diese Datei liegt im `cache`-Ordner im Arbeitsverzeichnis der App. Aus dem Quellcode der App geht hervor, dass diese nicht explizit gelöscht wird. Entsprechend ist die Datei dort nach dem Backup Vorgang auffindbar.

Auch bei Lissi listet das Einstellungsmenü eine Export-Funktion. Es wird aber angegeben, dass diese erst in einer nächsten Version verfügbar ist. Dennoch wird ein Blick in den dekompilierten Quellcode der App geworfen. Bei Lissi ist dieser nicht direkt dekompilierbar, da hier die von Xamarin angebotene Kompressionsfunktion für dlls genutzt wurde. Da das Kompressionsverfahren bekannt ist, existiert bereits ein Python-Skript [76], mit dem selbige rückgängig gemacht werden kann.
Im dekompilierten Code lassen sich bereits Spuren der zukünftigen Funktionalität finden. Es ist ersichtlich, dass das Exportieren ähnlich zur esatus Wallet funktionieren

wird. Es wird also auch ein Mnemonic mit Seed generiert, der der Indy-Funktion `indy_export_wallet` als Passphrase übergeben wird.

4.3 Speichermodell

Auch hier soll es, vergleichbar zu Jolocom, das Ziel sein, herauszufinden, wo und wie die soeben erzeugten Daten gespeichert werden. Es werden also die Datenverzeichnisse aller Apps auf den Arbeitsrechner kopiert und untersucht. Dabei fällt auf, dass bei keiner der Apps im Verzeichnis `databases` eine Datenbank vorhanden ist, die entsprechend Credentials enthält. Dagegen offenbart der `files` Ordner das Unterverzeichnis `.indy-client`. Dies ist der Standard-Name für ein Verzeichnis in dem Hyperledger-Indy Bibliotheken Wallets und Netzwerkkonfigurationen speichern. Entsprechend lassen sich in nach dem jeweils verwendeten Indy-Netzwerk benannten Unterordnern SQLite-Datenbanken finden, die wie in 2.4 beschrieben, verschlüsselte Daten enthalten. Entsprechend ist der nächste Schritt, die Shared Preference Dateien nach Schlüsseln zu durchsuchen. Um zu prüfen, ob sich die entsprechende Wallet-Datei tatsächlich mit dem gefundenen Schlüssel öffnen lässt, wird die JavaScript-Version der Indy Bibliothek verwendet. Der in Listing 4.1 dargestellte Code öffnet das angegeben Wallet und gibt alle darin enthaltenen Credentials aus. Um den Code einfach zu halten, wird darauf verzichtet, für jede Wallet-Datei den Datei-Pfad anzugeben. Stattdessen werden die Dateien in das Verzeichnis auf dem Arbeitsrechner kopiert, in dem Wallets standardmäßig von Indy erwartete werden. Konkret ist dies der Ordner `$HOME/.indy-client/wallet`.

```
var indy = require('indy-sdk')
var config = '{"id" : "esatus"}'
var credentials =
   '{"key" : "4XVuMLCDvEy1PcVegiE61PHjZWubb72rHph2vYxTr8wc"}'

indy.openWallet(config, credentials).then(wh => {
 indy.proverGetCredentials(wh, null).then(list => {
  console.log(list)
  indy.closeWallet(wh)
 })
})
```

Listing 4.1: Öffnen eines Indy-Wallets

Im Falle von Connect.Me gestaltet sich die Suche nach dem Wallet-Passwort erstaunlich einfach. In der Shared Preferences Datei `ConnectMeSharedPref.xml` findet sich nicht nur das gesuchte im Klartext, sondern auch alle Credentials, alle Anfragen nach solchen mit entsprechenden Antworten, alle Verbindungen und weitere Metadaten. Kurz gesagt: Bis auf kryptografische Schlüssel steht her alles, was die Wallet-App zum funktionieren braucht in unverschlüsselter Form. Zumindest der PIN für die App wurde nicht

im Klartext sondern als Hash hinterlegt. Bzw. genauer gesagt sind es die ersten acht Byte des Ergebnisse von PBKDF2 angewendete mit einem ebenfalls in der Datei befindlichen Salt auf den PIN.
Ursächlich für die unverschlüsselte Speicherung der Daten ist die Verwendung einer veralteten Version einer React-Native Bibliothek, die zum schreiben von Shared Preference Dateien genutzt werden kann. `react-native-sensitive-info` verschlüsselt unter Android erst ab Version 6 alle Daten unter Nutzung von Schlüsseln aus dem Android Keystore [77]. Connect.Me verwendet allerdings Version 5.5.5.
Die genannte Shared Preference Datei ist auch nicht die einzige Quelle, aus der relevante Daten im Klartext ausgelesen werden können. Auch in der Log-Datei `connectme.rotating.<id>.log` im `files`-Ordner sind alle Verbindungsanfragen Credentials usw. auffindbar. Entsprechend lassen sich die Daten auch mittels `adb logcat` zur Laufzeit der App anzeigen. Bei keiner anderen App konnte ein derart intensives Logging festgestellt werden.
Weiterhin existiert im `databases` Ordner die Datei `RKStorage`, die Konfigurationsinformationen der App enthält. Hierzu gehört, wie oft der PIN falsch eingeben wurde oder ob biometrische Authentifizierung aktiviert ist. Durch Manipulation ersteren Wertes gelang es im Fall von Connect.Me allerdings nicht, sich unendlich viele Versuche zur PIN-Eingabe zu verschaffen. Die App sperrt sich und verweigert die PIN-Eingabe, wenn der Wert außerhalb eines erwarteten Bereichs liegt, also z.B. negativ ist.

Bei Lissi wirkt der Inhalt der Datei `io.lissi.mobile.android.xamarinessantial.xml` zumindest, als seien die jeweiligen Werte verschlüsselt und anschließend Base64-codiert. Insgesamt lassen sich zwei interessante Eintrage in dieser Datei ausmachen: `pinId` und `walletKeyId`.
Der Dateiname gibt entsprechend Aufschluss, welche Xamarin-Standardbibliothek verwendet wurde, um die Datei zu erstellen. `SecureStorage` verschlüsselt die Werte der key-value-Paare in der Shared Prefence Datei unter Nutzung von Schlüsseln aus dem Android Keystore. Die entsprechenden Schlüssel der Paare bleiben im Klartext[5]. Da damit bekannt ist, dass Funktionen der Android-Keystore-API verwendet werden, wird in der weiteren Untersuchung das bereits bei Jolocom verwendete Frida-Script zum Auffinden selbiger eingesetzt. Es ist dabei festzustellen, dass auch Lissi wichtige Daten wie das Wallet-Passwort bereits während des App-Starts entschlüsselt. Dies führt dazu, dass der PIN/Fingerprint-Screen der App umgangen werden kann, ohne dass mit Einschränkungen bei der App-Nutzung gerechnet werden muss. Der ebenfalls in der erwähnten Shared Preference Datei gespeicherte App-PIN wird allerdings erst nach einmaliger Eingabe eines PINs in die App entschlüsselt, entsprechend um die Eingabe mit dem gespeicherten Wert zu vergleichen.
Neben der genannten Datei lässt sich mit `io.lissi.mobile.android_preferences.xml` noch eine weitere Shared Preferences Datei finden, die Konfigurationen der App

[5] Auch wenn die Dokumentation zu `SecureStorage` behauptet, von den Schlüsseln wird nur der MD5-Hash gespeichert, passiert dies in aktuellen Versionen nicht mehr. Der Quellcode deutet aber darauf hin, dass dies in vergangen Versionen geschehen ist.

speichert, beispielsweise wie oft der PIN bereits falsch eingegeben wurde oder ob biometrische Authentifizierung aktiviert ist. Listing 4.2 zeigt exemplarisch den Inhalt einer solchen Datei. In dieser wird nicht verschlüsselt. Entsprechend problemlos lassen sich die enthaltenen Werte manipulieren, womit es möglich wird, sich beliebig viele Versuche zur Eingabe der Wallet-PIN zu verschaffen. Hierzu muss der Wert von `LoginFailedAttemptsId` auf eine negative Zahl gesetzt werden.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <long name="SleepTimeStampId" value="-8585650628763484578" />
    <string name="TermsOfServiceVersionId">1.0</string>
    <boolean name="IsLoggedInId" value="true" />
    <int name="AcceptedConnectionsCountForAutoAcceptId" value="4" />
    <boolean name="AutoAcceptConnectionsId" value="false" />
    <int name="LoginFailedAttemptsId" value="0" />
    <boolean name="ShowLissiDemoAdvertisementId" value="false" />
    <boolean name="IsAutoAcceptConnectionsDeclinedId" value="true" />
    <boolean name="IsWalletProvisionedId" value="true" />
</map>
```

Listing 4.2: Auszug aus der Konfiguration der Lissi Wallet

Auch ID-Wallet und esatus Wallet nutzen Xamarins `SecureStorage`-Bibliothek zur Speicherung von Shared Preferences. Hier lassen sich in den entsprechenden Dateien `com.essatus.wallet.xamarinessentials.xml` und `com.digitalenabling.idw.xamarinessentials.xml` deutlich mehr Einträge finden als bei Lissi. Bei beiden deutet allerdings auf den ersten Blick kein Eintrag darauf hin, das gesuchte Wallet-Passwort zu enthalten. Da zur ID-Wallet der Quellcode vorliegt, wird dieser genauer unter die Lupe genommen, um herauszufinden, wo das Passwort hinterlegt ist bzw. wie es zusammengesetzt wird. Dies ergibt, dass das Passwort aus dem PIN und zwei in der Shared Preference Datei gespeicherten Werten (`WalletSalt` und `WalletPreKey`) generiert wird. Dies erklärt auch, warum in der Shared Preference Datei nichts darauf hindeutet, dass in ihr der Wallet-PIN gespeichert wird. Er wird, so wie es wünschenswert ist, stets benutzt um das Wallet-Passwort zu generieren und damit für die App wichtige Daten zu entschlüsseln. Weiterhin erklärt dieser Fund, warum diese App keine Authentifizierung mittels Biometrie anbietet.
Beide Werte, die ID-Wallet zur Generierung des Passortes benutzt, lassen sich bei esatus allerdings nicht finden. Dies deutet darauf hin, dass hier das Passwort anders erzeugt wird oder doch irgendwo gespeichert sein muss. Entsprechend wird der dekompilierte Code der App untersucht. Aufgrund der änlichen Struktur dessen zu ID-Wallet konnten die entsprechende Code-Teile, die für das Speichern und Auslesen des Passwortes verantwortlich sind, leicht gefunden werden. Es wurde dabei klar, dass sich das gesuchte in einer Struktur namens `AgentConfig` befindet, die in der Shared Preference Datei unter `ActiveAgent` abgelegt ist. Im Anschluss wird also Frida und das entspre-

chende Script genutzt, um zu schauen, wann diese Struktur entschlüsselt wird und zu prüfen, ob sie das Passwort enthält. Die Entschlüsselung findet direkt beim Start der App statt. Listing 4.3 zeigt Auszugsweise die entsprechende JSON-Struktur mit dem Passwort.

```
{
  "WalletConfiguration" : {
        ...
  },
  "WalletCredentials" :{
        "key" : "8EeDFynRTaRrjyE1qkLfcHV7UkKsz1aPR5Xx1vueF2GU"
  },
  ...
}
```

Listing 4.3: Wallet-Passwort der esatus Wallet

Weiterhin fordern alle Apps Lese- und Schreibberechtigungen für den externen Speicher an. Es konnte festgestellt werden, dass sich hier im Verzeichnis der jeweiligen Apps keine Daten finden. Lediglich Lissi speichert an dieser Stelle Informationen zu allen Indy-Netzwerken, mit denen die App in der Lage ist zu arbeiten. Insgesamt sind dies elf verschiedene, unter anderem auch Sovrin. Die übrigen Apps legen diese Daten in ihren Verzeichnissen im internen Speicher ab.

4.4 Netzwerkanalyse

Vergleichbar zu Abschnitt 3.3 soll sich auch hier mit den Netzwerkverbindungen der einzelnen Apps näher beschäftigt werden. Es wird dabei die gleiche Strategie genutzt, wie bei der Untersuchung der Jolocom SmartWallet. Auch hier wird versucht, die Apps zu benutzen, ohne Zertifikats-Pinning zu umgehen. Dabei musste festgestellt werden, dass es mit esatus- wie auch ID-Wallet, kein Problem ist, Verbindungen zu Issuern aufzubauen und sich Credentials ausstellen zu lassen. Bei einem erneuten Test musste allerdings festgestellt werden, dass beide Apps das Starten in dieser Netzwerkumgebung verweigern. Ursächlich ist laut angezeigter Fehlermeldung, dass eine initiale Verbindung zum Push-Benachrichtigungsdienst nicht möglich ist. Diese Initiale Verbindung muss nicht bei jedem App-Start aufgebaut werden, sondern besteht im Hintergrund eine gewisse Zeit fort. Das erklärt warum dieses Verhalten beim ersten Test nicht beobachtet wurde.

Lissi und Connect.Me verweigern unter dieser Konfiguration den Verbindungsaufbau zu Issuer- bzw. Verifierdiensten. Entsprechend werden alle Apps nochmals unter Verwendung des bereits bei Jolocom vorgestellten Frida-Scripts zur Umgehung von Zertifikats-Pinning verwendet. Dies führt dazu, dass es nun auch mit Lissi Wallet möglich ist, Verbindung zu Issuern/Verifiern aufzunehmen. Die Ausgabe des Script teilt mit, dass

Pinning-Funktionen der Android-TrustManger-API umgangen werden mussten. Diese ist dafür verantwortlich Zertifikate aus dem Speicher des Betriebssystems zu überprüfen. Zusammen mit dem Fakt, dass selbst installierte Zertifikate ab Android 7 nicht als vertrauenswürdig angesehen werden, erklärt dies, warum vorher die Verbindungen fehlschlugen. Trotzdem zeigt das Log des Proxies an, dass noch ein paar Verbindungsversuche, speziell zu Google-Diensten, aufgrund eines nicht vertrauenswürdigen Zertifikats abgelehnt werden.
Connect.Me konnte allerdings nicht zum funktionieren bewegt werden.
Auch esatus- und ID-Wallet werden mit diesem Script getestet. Hier zeigt die Ausgabe des Scripts erwartungsgemäß an, dass keine Pinner umgangen werden mussten. Das Problem bezüglich der Push-Benachrichtigungen ist damit allerdings noch nicht gelöst.

Entsprechend der eben gewonnen Erkenntnisse muss ein weiterer Weg gefunden werden, das Zertifikat auf dem Gerät zu installieren. Hierbei kann ausgenutzt werden, dass das Testgerät gerootet wurde. Android speichert alle vertrauenswürdigen Zertifikate im Ordner `/system/etc/security/cacerts` und vom Nutzer installierte CA-Zertifikate im Ordner `/data/misc/user/0/cacerts-added/`. Entsprechend genügt es, das Zertifikat des Proxies von letzterem in den zuerst genannten Ordner zu verschieben. Nachfolgend ließ sich der Netzwerkverkehr von Lissi Wallet, esatus- und ID-Wallet ohne Einschränkungen, d.h. ohne Fehlermeldungen im Proxy, mitlesen. Dabei kann das in den AGBs angedeutet und in Aries Protokollen [78] vorgesehene Verhalten, dass die Apps sich zu einem Relay-Server verbinden, bestätigt werden. Dieser wird im Zusammenhang mit Aries als Mediation Service bezeichnet und dient dem Weiterleiten von Nachrichten, wenn ein Endgerät nicht direkt erreicht werden kann. Bei Lissi ist dieser unter der Domäne `router.lissi.io` zu finden, die beiden anderen Apps nutzen `ssi-mediator.esatus.com`. Dabei ist zu beobachten, dass ID-Wallet diesen in regelmäßigen, kurzen (10 Sekunden) Abständen nach neuen Nachrichten anfragt, während Lissi und esatus den Server nur während eines aktiven Credential-Austausches kontaktieren.
Weiterhin ist die Kommunikation mit `ssi-mediator.esatus.com` durch TLS in Version 1.2 und `ECDHE-RSA-AES128-GCM-SHA256` geschützt, während die Verbindungen zu `router.lissi.io` TLS Version 1.3 und `TLS_AES_256_GCM_SHA384` nutzen. Auch wenn der Proxy in der Lage ist, über TLS verschlüsselte Daten im Klartext anzuzeigen, lässt sich die Kommunikation zwischen Relay und App bzw. mit dem Demo-Issuer nicht mitlesen. Diese ist zusätzlich Ende-zu-Ende verschlüsselt, wie in 4.1 angedeutet. Es ist ersichtlich, dass sich dabei an das in Aries RFC 0019 [79] beschriebene Format gehalten wird.

Neben diesen für die Funktionalität notwendigen Netzwerkverkehr lässt sich weiterhin beobachten, dass die drei Apps zum Ausspielen von Push-Benachrichtigung Googles Dienst Firebase Cloud Messaging nutzen. Lissi registriert sich dabei direkt bei Firebase, ID-Wallet und esatus Wallet indirekt über Microsofts Dienst Azure Service Bus.

Weiterhin lässt sich bei Lissi bei jedem Start der App eine POST Anfrage zur Domain `in.appcenter.ms` beobachten. Bei Appcenter handelt es sich um eine von Microsoft angebotene Logging und Analytics Plattform. Den von Lissi gemachten Anfragen an diese Domäne ist zu entnehmen, dass hier Appcenter für Analytics und dem Melden von Fehlern genutzt wird. Auch ist sichtbar, dass entsprechend das Starten der App geloggt wird und dabei Daten zum genutzten Gerät übertragen werden. Anhand von im HTTP-Header übermittelten Daten (`Install-ID` und `App-Secret`) ist eine Verknüpfung der gesendeten Daten mit einer App-Instanz möglich. Das Nutzen von Appcenter und die Art der Übermittelten Daten werden in der Datenschutzerklärung und den Allgemeinen Geschäftsbedingungen der App genannt.
Zusätzlich lassen sich seitens Lissi Verbindungen zu Googles Logging API unter `https://firebaselogging.googleapis.com/` beobachten. Auch dieser werden Informationen zum genutzten Gerät gegeben. Hierauf findet sich allerdings kein Hinweis in Datenschutzerklärung und AGB.

Bei esatus und ID-Wallet konnte im Laufenden Betrieb keine Verbindung zu Logging/ Analytics-APIs ausgemacht werden. Lediglich beim ersten Start nach Installation kontaktiert ID-Wallet `https://firebaseinstallations.googleapis.com/v1/projects/digital-enabling-idw/installations`. Diese API dient dazu, eine eindeutige ID für die eben installierte Instanz der App zu generieren.
Von den in der Datenschutzerklärung beider Apps angedeuteten Nutzung der SafetyNet API zur Integritätsprüfung des Gerätes konnte in den Netzwerkmitschnitten nichts gefunden werden, was den Verdacht erhärtet, dass eine derartige Prüfung nicht stattfindet.

Zusammenfassend lässt sich für Lissi Wallet, esatus Wallet und ID-Wallet sagen, dass die Analyse des Netzwerkverkehrs kaum überraschend verlaufen ist. Bei allen drei Apps waren nur Verbindungen zu beobachten, die technisch notwendig sind bzw. mit der Datenschutzerklärung vereinbar sind. Lediglich die mangelhafte Prüfung der SSL-Zertifikate durch esatus und ID-Wallet ist negativ zu bewerten. Selbiges gilt für Lissis gesendete Informationen an die Firebase Logging API.

Im Gegensatz zu diesen drei Apps ist Connect.Me deutlich mitteilungsfreudiger gegenüber Logging / Analyticts-Diensten. Sie teilt das Starten der App sowohl `https://api2.branch.io/v1/open` - hier inklusive Daten zum verwendeten Gerät- wie auch `https://api.apptentive.com/` mit. Letzterer API wird zusätzlich jede Nutzerinteraktion mit der App mitgeteilt, z.B. dass das Menü geöffnet wurde oder das auf den Bildschirm zur Credential Anzeige gewechselt wurde. Connect.Me beschreibt dieses Logging auch in seinen Datenschutzbestimmungen.
Auch wenn der Kontakt zu diesen APIs inzwischen funktioniert, ist ein Ausstellen bzw. Vorzeigen der Credentials nach wie vor nicht möglich. Das Log des Proxies meldet, dass ein Verbindungsaufbau zu `agency.evernym.com` aufgrund eines nicht vertrauenswürdigen Zertifikats nicht hergestellt werden kann. Eine genaue Betrachtung der

Zertifikate in `/system/etc/security/cacerts` offenbart eine mögliche Ursache. Die aus `/data/misc/user/0/cacerts-added/` durch Kopieren hinzugefügte Datei weißt ein anderes Dateiformat auf als die übrigen. Laut Ausgabe des Linux Befehls `file` lässt dieser sich bei der hinzugefügten Datei nicht genau bestimmen (Typ: `data`), während alle übrigen Dateien als `PEM certificate` beschrieben werden. Es wird nun also direkt das von MITMProxy bereitgestellte Zertifikatsdatei in den Zertifikatsspeicher des Android Systems kopiert. Nachfolgend gelingt Connect.Me die Verbindung zu `agency.evernym.com` und ein Teilen von Credentials mit der bereitgestellten Demo-Webseite wird möglich. Dabei wird deutlich, dass der Dienst hinter diesem URL die gleiche Funktion übernimmt, wie z.B. `router.lissi.io` für Lissi. Auch Connect.Me kontaktiert diesen Dienst regelmäßig, genauer im Abstand von 30 Sekunden, um auf neue Anfragen zu prüfen. Weiterhin kann bei der Interaktion mit der Demo-Webseite beobachtet werden, dass mit `vas.evernym.com` ein weiterer vergleichbarer Dienst kontaktiert wird. Dieser übernimmt die Relay-Funktion für die Demo-Webseite.
Ein weiterer möglicher Grund für die anfängliche Verweigerung, sich mit dem Proxy zu verbinden, wurde im `files`-Ordner im App-Verzeichnis gefunden. Hier befindet sich die Datei `cacert.pem`, die die Zertifikate bekannter Root-CAs enthält. Allerdings ist es in Anbetracht dieser Datei auch fragwürdig, weshalb überhaupt eine Verbindung zum Proxy hergestellt werden konnte. Dies ist nur damit zu begründen, dass die App ihre gespeicherten Zertifikate nicht für den Verbindungsaufbau verwendet.

Während der Analyse des Netzwerkverkehrs konnte bei allen Apps auch die Verbindung zu den jeweils genutzten Indy Netzwerken beobachtet werden. Während esatus- und ID-Wallet sich stets nur mit Knoten aus dem eingestellten verbanden, konnte bei Connect.Me beobachtet werden, wie die App Verbindungen zu Knoten verschiedener ihr bekannter Netzwerke aufnahm. Damit ist erklärbar, wie es die App schafft, ohne explizite Netzwerkangabe Credentials verschiedener zu akzeptieren. Es werden einfach alle bekannten Netzwerke angefragt, bis die gewünschten Daten gefunden wurden. Da auch in den Einstellungen von Lissi keine Möglichkeit gefunden wurde, das Netzwerk entsprechend anzugeben und die im Speicher gefundenen Daten aber darauf hindeuten, dass auch diese App mit mehreren interagieren kann, soll getestet werden, ob hier ein ähnliches Verhalten beobachtete werden kann. Mangels weiterer Auswahl wird hierzu die bereits erwähnte Demo von esatus (`https://wallet-demo.esatus.com/`) verwendet. Aus den Daten von Lissi und den Angaben auf der Webseite geht zwar hervor, dass das hier genutzte Netzwerk von Lissi nicht unterstützt wird, aber um prüfen zu können, ob Lissi prinzipiell mehrere Netzwerke anfragt, genügt die Demo. Die vom Proxy angezeigte Verbindungen bestätigen die Vermutung. Auch Lissi prüft mehrere Netzwerke, um die gewünschten Daten zu finden.

5 Eigener Prototyp

Die Analyse der ausgewählten Wallet-Apps in den vorhergehenden Kapiteln hat gezeigt, dass diese Best-Practices zur Entwicklung einer sicheren App nicht vollumfänglich umsetzen. Dieses Kapitel soll zeigen, wie sich mit einem Framework zur plattformübergreifenden Entwicklung von Anwendungen ein prototypisches Wallet und eine diesem zugrundeliegende Bibliothek zur Handhabung von Verifiable Credentials (VCs) unter Beachtung von MASVS und MSTG entwickeln lässt. Genauer wird das von Google entwickelte, quelloffene Framework Flutter und die damit verbundene Programmiersprache Dart genutzt. Dart ist dabei eine objektorientierte Sprache mit syntaktischer Ähnlichkeit zu Java. Flutter/ Dart bietet die Möglichkeit, Anwendungen für alle gängigen Betriebssysteme (Windows, Linux, MacOS, Android und iOS) sowie Browser-basierte Apps zu entwickeln. Weiterhin besteht im Falle von Android und iOS Anwendungen die Möglichkeit, einen mit Flutter entwickelten App-Bestandteil mit überschaubaren Aufwand in eine bestehende native Android/iOS-App zu integrieren. Somit besteht zum einen die Möglichkeit, zukünftig Wallet-Anwendungen für Desktop-Systeme bereitzustellen. Zum anderen wird ermöglicht, ein Wallet konzeptionell nicht wie bisher gesehen als alleinstehenden Anwendung zu betrachten, sondern Wallet-Funktionalitäten in bestehende Anwendungen einzelner Issuer zu integrieren.

Die hier beschriebene Dart-Bibliothek und prototypische Wallet-Anwendung sind Teil des Forschungsprojektes „Blockchain-basierte digitale Identität - BCS-ID“, einem Teilprojekt der Blockchain-Schaufensterregion Mittweida. Ziel des Projektes ist es, eine Identitätslösung für ebenjene Schaufensterregion zu entwickeln. Neben der hier beschriebenen Wallet- und Credential Bibliothek gehört auch ein einfach gehaltenes Austausch-Protokoll für VCs zu den Ergebnissen des Projekts. Dieses Austausch-Protokoll, genannt Inter-Wallet-Credential Exchange (IWCE), basiert auf dem bereits angedeuteten Grundgedanken, dass jede App eines Issuers ein Wallet enthält. In diesem sind die jeweils ausgestellten Credentials gespeichert. Auf dem Smartphone eines Holders existieren nun also mehrere Wallets, die in der Lage sein sollen, Credentials miteinander austauschen. Hierfür definiert IWCE entsprechende Datenstrukturen. Als Übertragungsweg sind Android App-Links bzw. iOS Universal-Links vorgesehen [80, 81].

Weiterhin bietet dieser Ansatz der verteilten Speicherung von Credentials in verschiedenen Apps die Möglichkeit, App- und damit Credentialspezifische Risikobewertungen zu erstellen um daraus Sicherheitsmaßnahmen abzuleiten und umzusetzen. So muss eine App, die den Mitgliedsausweis einer Bibliothek enthält, nicht die gleichen Sicherheitsrichtlinien umsetzen wie eine, die Bankdaten oder den Personalausweis speichert. Um eine Vergleichbarkeit zu den übrigen vorgestellten Apps zu wahren, wird auch bei der prototypischen Wallet-Anwendung und der ihr zugrunde liegenden Bibliothek höchstmögliche Datensicherheit angestrebt.

5.1 Vorüberlegungen

Neben dem verwendeten Austausch-Protokoll für VCs sollte vor Beginn der Entwicklung einer Wallet-Anwendung klar sein, welche DID-Methode (s. Abschnitt 2.2) anfänglich unterstützt werden soll und wie die verwendeten Credentials aussehen. Genauer bezieht sich letzterer Punkt auf die Frage, ob anonyme Credentials unterstützt werden sollen oder lediglich klassisch signierte (s. Abschnitt 2.3). Dies ist eine Abwägung zwischen unterstützter Funktionen, wie Selective Disclosure und Predicate Proofs gegen Implementierungsaufwand. Predicate Proofs sind dabei bisher nur mit CL-Signaturen und entsprechenden Zero-Knowledge Proofs möglich. Dafür existieren zum derzeitigen Zeitpunkt keine Dart/Flutter-Bibliotheken, was den Aufwand der Entwicklung wie auch die Fehleranfälligkeit erhöht. Demgegenüber steht die geringe Anwendungsbreite. In Bezug auf VCs wird eine sinnvolle Anwendung lediglich im Bereich der Altersüberprüfung gesehen, beispielsweise in Bezug auf die Fragestellung, ob eine Person Älter als 18 ist. Entsprechend wird sich entschieden, auf Predicate Proofs zu verzichten.

Selective Disclosure lässt sich dagegen auf verschiedene Arten umsetzen, wie in Abschnitt 2.3 diskutiert. Den besten Kompromiss bietet hierbei die Lösung, die einzelnen Attributwerte zu hashen und entsprechend vom Issuer ein Credential über diese Hashwerte signieren zu lassen. Damit bliebt es möglich, bestehende, weit verbreitete Signaturverfahren zu verwenden, die entsprechend empfohlen werden (s. Abschnitt 2.1) und für die Software-Bibliotheken zur Verfügung stehen.

Entsprechend setzt sich ein Credential aus zwei Datenstrukturen zusammen. Als Datenformat wird das JSON-Format (Javascript Object Notation) verwendet. Eine Datenstruktur entspricht dem W3C-Standard für VCs und ist damit von einem Issuer signiert. Sie enthält aber keine konkreten Attributwerte, sondern nur deren Hashwerte, wie in Listing 5.1 gezeigt wird. Die eigentlichen Attributwerte werden in einer zweiten Datenstruktur abgelegt. Diese wird nachfolgend als Plaintext-Credential bezeichnet. Wie das Beispiel für ein Plaintext-Credential in Listing 5.2 zeigt, enthält es weiterhin für jedes Attribut einen Salt, also eine individuelle, zufällige Zeichenkette. An diesen wird der eigentlichen Attributwert angehangen, bevor gehasht wird. Damit wird verhindert, dass Werte von Attributen, für die es nur wenige überhaupt mögliche Ausprägungen gibt, erraten werden können. Einfaches Beispiel ist das Alter einer Person. Hierfür ist die Anzahl der möglichen Werte überschaubar. Wird nur auf diese allein die Hash-Funktion angewandt, ist es ein Leichtes, aus dem Hash den eigentlichen Attributwert, also das Alter, zu ermitteln. Die Verknüpfung mit einer weiteren Zeichenkette, dem Salt, verhindert dies.

```
{
  "@context":["https://www.w3.org/2018/credentials/v1"],
  "type":["VerifiableCredential"],
  "credentialSubject":{
    "id":"did:ethr:ropsten:0xEFf19D411ac59ccb8FF4ea65CD58A859C2188e9D",
```

```
    "name":"0x095c54d898a992e66f506be16ffac7cf8fba607b86763df5c355bd87b1ed6aa3",
    "age":"0xedcf0e29c4b6264799c9ee92b37c2d04c75fd01c74ca23a3102315d812053204"
  },
  "issuer":"did:ethr:0x08047b8ad77Cc446a9D8268b3748b2a035393E67",
  "issuanceDate":"2022-01-26T12:00:42.772913Z",
  "proof":{
    "type":"EcdsaSecp256k1RecoverySignature2020",
    "proofPurpose":"assertionMethod",
    "verificationMethod":"did:ethr:0x08047b8ad77Cc446a9D8268b3748b2a035393E67",
    "created":"2022-01-26T12:00:42.774782Z",
    "jws":"eyJh...wAE="
  }
}
```


Listing 5.1: VC


```
{
  "id":"did:ethr:0xEFf19D411ac59ccb8FF4ea65CD58A859C2188e9D",
  "hashAlg":"keccak-256",
  "name":{
    "value":"Max Mustermann",
    "salt":"6982d07d-fc55-48d7-bd11-c6f217779c4e"
  },
  "age":{
    "value":24,
    "salt":"a377b823-e56b-4960-80c1-6dbf273a3604"
  }
}
```


Listing 5.2: Plaintext Credential

Die DID-Methode entscheidet zum einen darüber, welches Schlüsselmaterial verwendet wird und wo andere Parteien, z.B. Verifier das DID-Document finden können. Derzeitig existieren mehr als 100 verschiedene DID-Methoden [82]. Einige davon sind projektspezifisch (`did:jolo`), andere verweisen auf ein DLT-Netzwerk (`did:ethr`, `did:indy`), wieder andere verlangen den direkten Austausch des DID-Documents zwischen beteiligten Parteien (`did:peer`, `did:keri`).

Um die Wahl der DID-Methode zu verstehen, soll nochmals erwähnt sein, dass der hauptsächliche Vorteil eines DID gegenüber anderen Identifiern ist, dass sie untrennbar mit kryptografischen Schlüsseln verbunden sind und keine zentrale Instanz wie eine PKI zur Erstellung benötigen (s. Abschnitt 2.2). Entsprechend wird eine DID-Methode gesucht, die es auf einfache Art und Weise ermöglicht, DIDs anzulegen, zu einer DID gehörige Öffentliche Schlüssel abzufragen, sowie Schlüssel zu wechseln bzw. den DID für ungültig zu erklären. Diese Forderungen werden von der `did:ethr`-Methode am besten erfüllt. Die Methode sieht vor, das DID-Document im öffentlichen Blockchain-

Netzwerk Ethereum (bzw. einem diese Technologie nutzenden Testnetzwerk oder privaten Netzwerk) abzulegen. Genauer wird nicht das DID-Document selbst dort gespeichert, sondern lediglich Attribute, aus denen sich ein solches generieren lässt [83]. Das verwendete Schlüsselmaterial ist entsprechend das Gleiche wie es für Transaktionen in diesem Netzwerk nötig ist, also Schlüssel auf Basis der elliptischen Kurve secp256k1. Demzufolge ist der methodenspezifische Identifier des DID eine Ethereum-Adresse, die aus dem Öffentliche Schlüssel mittels Hashing abgeleitet wird. Beispiele für gültige DIDs sind Listing 5.1 zu entnehmen. Sie zeigen, dass die DIDs den grundlegenden Aufbau `did:ethr:<Netzwerkbezeichnung>:<Ethereum-Adresse>` besitzen. Lediglich wenn auf das öffentliche Hauptnetzwerk (Mainnet) von Ethereum verwiesen wird, kann die Netzwerkbezeichnung weggelassen werden.

Wie Ethereum-Adressen auch, lassen sich die DIDs ohne Kontakt zu einem Ethereum-Netzwerk erstellen. Es stellt sich also die Frage, wie eine Signatur, die mit den Schlüsseln einer solchen DID erzeugt wurde, prüfbar ist. Die Begründung liegt im verwendeten Signaturverfahren. ECDSA ermöglicht es, aus einer Signatur und der dazugehörigen Nachricht den Öffentlichen Schlüssel zu berechnen, mit dessen Privaten Schlüssel die Signatur erstellt wurde. Aus jenem lässt sich der DID rekonstruieren. Entsprechend genügen das signierte Credential bzw. eine Presentation selbst, um die Signaturen zu prüfen. Lassen sich die gegebenen DIDs wiederherstellen, ist die Signatur korrekt.
Eintragungen in Ethereum werden entsprechend nur nötig, wenn der Schlüssel gewechselt wird und damit der mathematische Zusammenhang zwischen DID und Schlüssel verloren geht. Es wird dabei nicht der neue Öffentliche Schlüssel abgelegt, sondern die neue Ethereum-Adresse, die rechnerisch zu diesem gehört. Diese sollte ein Verifier abfragen, wenn eine erste Signaturprüfung, wie sie eben beschrieben wurde, fehlschlägt.

5.2 Speichermöglichkeiten

Die Basis eines Wallets bildet die sichere Abspeicherung der Daten. Demnach besteht der erste Entwicklungsschritt darin, eine geeignete Bibliothek zu finden, die strukturierte (durchsuchbare), verschlüsselte und lokale Speicherungen von Daten ermöglicht. Insgesamt lassen sich fünf Bibliotheken ausmachen, die grundlegend in Frage kommen: sqflite [84], sembast [85], objectbox [86], isar [87] und hive [88]. Davon unterstützen allerdings objectbox und isar keine Datenverschlüsselung. Hier müssen die Daten also verschlüsselt werden, bevor sie abgelegt werden.

Unter sqflite ist eine Bibliothek zu verstehen, die die in Android, iOS und MacOS verfügbaren SQLite-APIs für Flutter Anwendungen verfügbar macht. Diese nativen APIs unterstützen allerdings keine Verschlüsselung der gesamten Datenbank. Hierfür sind zusätzliche/andere Bibliotheken auf System- wie auch auf Flutter-Seite nötig [89].
sqflite unterstützt außerdem Linux und Windows nicht, auch hierfür ist eine zusätzliche

Bibliothek notwendig [90]. Entsprechend wird SQLite für die weitere Entwicklung des Wallets nicht berücksichtigt.

hive und sembast verfolgen vergleichbare Konzepte. Beides sind komplett in Dart geschriebene NoSQL Datenbanklösungen. sembast speichert dabei die Daten in einer einzelnen Datei und hive erlaubt es, Daten auf mehrere sogenannte Boxen aufzuteilen, die speichertechnisch eigenen Dateien entsprechen. Um Daten aus einer Box lesen zu können bzw. in diese zu schreiben, muss sie „geöffnet“ sein. Daten geöffneter Boxen liegen vollständig im Arbeitsspeicher des Gerätes. Auch sembast lädt die komplette Datenbank vor Lese- und Schreibvorgängen in den Arbeitsspeicher.
Beide Bibliotheken unterstützen optional die Verschlüsselung der gespeicherten Daten. sembast setzt dabei zwingend voraus, dass ein entsprechender sogenannter Codec selbst implementiert wird. Dieser stellt den Ver- und Entschlüsselungsalgorithmus. Bei hive ist ein entsprechender Algorithmus Teil der Bibliothek. Genauer ist dies AES-256 im CBC-Modus. Entsprechend sollte überprüft werden, ob eine Manipulationsschutz der Daten vorgesehen ist. Tests ergeben, dass Hive selbst bei unverschlüsselten Daten Manipulationen in diesen erkennt und ein Öffnen der entsprechenden Box verweigert. Es ist also davon auszugehen, dass intern Prüfsummen bei verschlüsselten wie unverschlüsselten Daten berechnet werden. Somit kann AES-CBC als eine sichere Wahl angesehen werden. Dennoch ist das Hinzufügen weiterer Verschlüsselungsalgorithmen möglich. Anzumerken ist auch, dass hive ähnlich wie Shared Preferences in Android als key-value-Speicher aufgebaut ist und von der Verschlüsselung lediglich die values betroffen sind. Die zugehörigen keys bleiben im Klartext.
Aufgrund der einfacheren API und der Möglichkeit, Daten auf mehrere Boxen aufzuteilen, wird für die Wallet Implementierung hive gewählt. Weiterhin besteht die Möglichkeit, jeder Box ihren eigenen Datentyp zuzuweisen. Dabei werden alle in Dart vorhandenen Typen wie Zahlen, Strings, Arrays und Listen unterstützt. Zusätzlich ist das Definieren eigener Datentypen möglich.

Nachfolgend gilt es, einen geeigneten Aufbau der hive-Datenbank zu finden. Wie in Abschnitt 2.4 aufgezeigt, muss ein Wallet in der Lage sein, Daten unterschiedlicher Kategorie und Schutzniveau zu speichern. Dem kommt die Möglichkeit von hive entgegen, Daten auf mehrere Boxen aufzuteilen. So können für Schlüssel und Credentials zwei verschiedene Boxen verwendet werden, die unabhängig voneinander geöffnet und geschlossen werden können. Damit wird auch erreicht, dass Schlüsselmaterial nur so lange wie nötig im Arbeitsspeicher verbleibt.

Aufgrund der verwendeten DID-Methode ist bekannt, dass es sich bei dem Schlüsselmaterial um welches auf Basis der elliptischen Kurve secp256k1 handelt. Diese Kurve wird vom Android Keystore-System nicht unterstützt. Entsprechend ist begründet, weshalb es notwendig ist, eigene Überlegungen zur Schlüsselverwaltung anzustellen.

Die Schlüsselverwaltung sollte es weiterhin unterstützen, für jedes Credential ein eigenes Schlüsselpaar und damit einen eigenen Identifier verwenden zu können. Dadurch wird beispielsweise ein Trennen verschiedener Identitäten (z.B. privater und geschäftlicher) ermöglicht. Außerdem wird es Dritten (z.B. Verifiern) erschwert, verschiedene Credentials zu einer Person zurückzuverfolgen. Aus dem Bereich der Kryptowährungen, vor allem von Bitcoin, sind vergleichbare Probleme bekannt. Auch hier empfiehlt es sich, bei jeder Transaktionen eine neue Adresse, sei es für sein eigenes Wechselgeld oder empfangene Bitcoin, anzugeben. Demzufolge existieren auch Lösungen. Genauer handelt es sich um sogenannte Hierarchisch Deterministische (HD-) Wallets. Dieser Standard ermöglicht es, aus einem einzigen Seed eine größere Menge Schlüssel abzuleiten. Genauer wird aus dem Seed ein Master-Schlüssel generiert, aus welchen sich maximal $2^{32} - 1$ Schlüssel ableiten lassen. Aus jedem kann wiederum die gleiche Menge weiterer Schlüssel abgeleitete werden, usw. Es entsteht eine Art Schlüsselbaum [91]. Entsprechend müssen lediglich der Master-Schlüssel oder sein Seed bzw. ein Mnemonic und für jeden Schlüssel (hier: jedes Credential) der zugehörige Ableitungspfad in diesem Baum gespeichert werden. Weiterhin ermöglicht dieses Konstrukt auch das Ableiten Öffentlicher Schlüssel aus übergeordneten Öffentlichen Schlüsseln ohne die Kenntnis des Privaten. Dies bietet Vorteile im Ausstellprozess eines Credentials. Hier muss dem Issuer ein Identifier (DID) des Holders bekannt sein, auf den das Credential auszustellen ist. Da selbiger aus einem Öffentlichen Schlüssel abgeleitet ist, ist umgekehrt ein solcher nötig. Ohne genannte Möglichkeit, Öffentliche Schlüssel aus Öffentlichen abzuleiten, ist aus hive-Sicht stets das Öffnen der Box nötig, die schlussendlich für das Speichern des Master-Schlüssels verantwortlich sein wird, um ein neues Schlüsselpaar und damit einen neuen DID zu erhalten. Damit ist verbunden, dass vom Nutzer der entsprechende Sicherungsmechanismus z.B. sein Passwort für das Wallet verlangt werden muss. Ziel soll es aber sein, dies auf ein Minimum zu beschränken, heißt eine Nutzerauthentifizierung nur dann zu verlangen, wenn sie zwingend notwendig ist. Noch genauer gesagt, sie nur zu verlangen, wenn der Umgang mit Privaten Schlüsseln notwendig ist, sowie beim Start der Wallet-Anwendung. Ersteres ist vornehmlich bei der Signaturerstellung, also dem Vorzeigen eines Credentials der Fall. Dies soll dazu beitragen, die Akzeptanz bei Endnutzern zu erhöhen.

Zusammenfassend ist also bekannt, dass drei Datenwerte gespeichert werden müssen, damit ein Holder ein ihm ausgestelltes Credential später erfolgreich präsentieren kann: das signierte VC, das Plaintext-Credential und der Schlüssel-Ableitungspfad. Um der für Credentials verwendeten hive-Box einen Datentyp zuweisen zu können, wird also für Credentials ein eigener Typ (Klasse) angelegt, der die genannten Eigenschaften umfasst. Zudem wurde erwähnt, dass jedes Credential auf eine eigene DID ausgestellt werden soll. Entsprechend identifiziert diese ein Credential und kann als Speicherschlüssel verwendet werden.

Weiterhin geht aus den bisherigen Ausführungen hervor, dass das Wallet eine weitere, separat öffenbare Box für den Master-Schlüssel benötigt. Außerdem ist eine Box für

Konfigurationsinformationen notwendig. Diese enthält die angesprochenen Öffentlichen Schlüssel, die zum Ableiten neuer DIDs benutzt werden, sowie den Index, also den letzten Teil des Ableitungspfades des nächsten Schlüssels. Weiterhin können hier Informationen zum verwendeten Ethereum-Netzwerk, wie sie zum Erstellen korrekter DIDs nötig sind (s. Abschnitt 5.1), abgelegt werden.

Damit sind die grundlegenden Bestandteile der Wallet bekannt. Weiterhin wird es als sinnvoll erachtet, die Möglichkeit zu bieten, zu protokollieren, welche Ereignisse mit einem Credential durchgeführt wurden. Hier gehört z.B. dazu, wann es ausgestellt wurde oder wann es mit einer dritten Partei geteilt wurde. Vergleichbare Ansichten wurden bei allen untersuchten Wallet-Anwendungen ebenfalls gefunden. Entsprechend wird eine weitere hive-Box und ein Datentyp (Klasse) für derartige Einträge eingeführt. Dieser erlaubt es, die Art des Ereignisses (Ausstellen, Teilen,...), festzuhalten, sowie eine Bezeichnung der anderen Partei, mit der das Ereignis stattfand, zu speichern. Weiterhin können ein Zeitstempel und bei Bedarf die freigegeben Attribute eingetragen werden. Auch diese Box nutzt den eindeutigen DID eines Credentials als Speicherschlüssel.

Zusätzlich sollte die Dart-Bibliothek und ihr Wallet auch für Issuer verwendbar sein. Für diese kann es sinnvoll sein, auch die von ihnen ausgestellten Credentials zu hinterlegen. Entsprechend wird dafür eine eigene Box vorgesehen, die Werte vom bereits vorgestellten Typ für Credentials aufnehmen kann. Da Issuer aber nicht die Mehrheit der Anwender sind, wird diese Box nicht standardmäßig angelegt. Dafür wird eine separate Funktion bereitgestellt, die diese Aufgabe erfüllt.

Entsprechend den Ausführungen in diesem und dem vorhergehenden Abschnitt 5.1, umfasst die entwickelte Dart-Bibliothek folgende grundlegende Funktionen:

- Bereitstellung des eigentlichen Wallets auf Basis von Hive
- Funktionen zum Umgang mit Plaintext-Credentials, VCs und zugehörigen Presentations (Erstellen, Signieren, Prüfen)
- Funktionen zum Abfragen der aktuellen Ethereum-Adresse zu einer DID
- Bereitstellung von Klassen für IWCE Credential-Request und Credential-Response

5.3 Android Sicherheitsmechanismen in Flutter

Zu den relevanten Mechanismen zählen hier der Android Keystore und die (biometrische) Authentifizierung über Android-APIs. Entsprechend werden Flutter-Bibliothek untersucht, die diese Funktionalitäten anbieten. Ziel soll es sein, geeignete Bibliotheken zu finden, um in der Wallet-App zum einen eine Nutzerauthentifizierung über Betriebssystemmechanismen anbieten zu können sowie das Passwort/den Schlüssel zur Entschlüsselung der hive-Datenbank sicher zu speichern. Für letzteres wird die verschlüs-

selte Speicherung in Shared Preferences angestrebt. Im Idealfall werden alle drei geforderten Punkte von einer einzelnen Bibliothek abgedeckt, vergleichbar mit der Klasse `EncryptedSharedPreferences` [92] der Android-API. So lassen sich die in 2.5.3 vorgestellten Best-Practices zur Keystore-Nutzung ohne erhöhten Aufwand umsetzen.

Nach einer ersten Suche kommen folgende Bibliotheken in Frage: `flutter_secure_storage` [93], `encrypted_shared_preferences` [94], `biometric_storage` [95] und `local_auth` [96]. Hierbei bieten die drei zuerst genannten Bibliotheken Möglichkeiten zur verschlüsselten Speicherung kleinerer Datenmengen. `local_auth` hingegen dient der Authentifizierung des Nutzers und wird, im Gegensatz zu den übrigen, direkt von Googles Flutter-Entwicklern bereitgestellt. Alle Bibliotheken sind mindestens für iOS und Android verfügbar.

Nach genaueren Analysen des Quellcodes wird `encrypted_shared_preferences` aus der Auswahl ausgeschlossen. Trotz der namentlichen Ähnlichkeit zur entsprechenden Android-API wird diese hier nicht genutzt. Die Daten werden mittels Funktionen aus Dart-Bibliotheken verschlüsselt, denen entsprechende kryptografische Schlüssel übergeben werden müssen. Das macht eine Speicherung selbiger über den Android Keystore unmöglich, da hier keine Schlüssel extrahiert werden können. Anschließend legt die Bibliothek die Daten in einer Shared Preference Datei ab.

Auch von der Nutzung von `local_auth` wird vorerst abgesehen. Dies ist damit zu begründen, dass die Bibliothek lediglich wahr/falsch Aussagen über den Erfolg einer Authentifizierung liefert. Wie aus Abschnitt 2.5.3 bekannt, entspricht dies nicht Best-Practice Vorgaben und kann mithilfe von Frida umgangen werden.

`flutter_secure_storage` nutzt den Android Keystore für die Verschlüsselung einzelner Werte, die in einer Shared Preference Datei abgelegt werden. Es wird allerdings keine Möglichkeit gegeben, den Schlüsseln gewisse Eigenschaften zuzuweisen, beispielsweise dass zur Schlüsselnutzung zwingend eine Authentifizierung notwendig ist. Auch der Verschlüsselungsalgorithmus kann nicht gewählt werden. Allerdings wurde mit AES im GCM Modus seitens der Entwickler eine sichere Wahl getroffen. Entsprechend wird `flutter_secure_storage` lediglich für das Speichern von Konfigurationen, die keiner Authentifizierung bedürfen, in Betracht gezogen. Hierzu gehören z.B. die Anzahl fehlgeschlagener Authentifizierungsversuche. Eine verschlüsselte Speicherung derartiger Werte verhindert eine einfache Manipulation wie sie beispielsweise bei Jolocom SmartWallet möglich war.

`biometric_storage` hingegen vereint die eingangs gestellten Forderungen. Auch diese Bibliothek nutzt einen Schlüssel aus dem Android Keystore zum Verschlüsseln von in Shared Preferences gespeicherten Werten. Dem Schlüssel werden aber derart Parameter mitgegeben, dass dieser nur nach biometrischer Authentifizierung einmalig verwendet werden kann. Entsprechend wird sich an die in Abschnitt 2.5.3 vorgestellten

Best-Practices zur Keystore Nutzung gehalten: Ein Schlüssel wird freigegeben und damit werden wichtige Daten entschlüsselt. Dementsprechend kann diese Bibliothek zur Passwort-Speicherung verwendet werden.

Nachteilig an `biometric_storage` ist, dass nur biometrische Authentifizierungsmethoden unterstützt werden. Entsprechend sollte für Nutzer, die eben jene nicht aktiviert haben, eine andere Möglichkeit der Entsperrung angeboten werden. Auch sollte bedacht werden, dass an Biometrie geknüpfte Schlüssel ungültig werden, wenn ein neuer Fingerabdruck hinterlegt wird. Da ohnehin vorgesehen ist, das Öffnen des hive-Wallets an ein Passwort zu knüpfen, welches bei aktiver biometrischer Authentifizierung aus `biometric_storage` geholt wird, stellen beide genannten Fakten kein Problem dar. Das Passwort kann in jedem Fall als Rückfall-Möglichkeit genutzt werden, an die Daten im Wallet zu gelangen.

In Bezug auf das Passwort steht noch die Entscheidung aus, ob es vom Nutzer festgelegt werden soll oder zufällig generiert wird. Im Falle einer aktiven biometrischen Authentifizierung wird das Passwort nur benötigt, wenn der Nutzer einen neuen Fingerabdruck registrieren möchte. Entsprechend sollte dies lediglich für Notfälle an einem sicheren Ort hinterlegt sind. Um also die Passwortsicherheit zu erhöhen, empfiehlt sich in diesem Fall die zufällige Generierung mit Hinweis an den Nutzer, das Passwort für den genannten Fall an einem sichern Ort zu hinterlegen.
Für den Fall der regelmäßigen Verwendung, also bei deaktivierter biometrischer Authentifizierung, sollte der Nutzer selbst ein Passwort wählen können. Hierfür werden gewisse Mindestanforderungen vorgegeben, wie eine Länge von acht Zeichen, die Verwendung von Groß- und Kleinbuchstaben sowie Zahlen und Sonderzeichen.

Weiterhin ist zu klären, wie aus dem Passwort ein kryptografischer Schlüssel zur Verschlüsselung der hive-Datenbank abgeleitet wird. Auch wenn bisher nicht weiter diskutiert, wird die Verwendung eines Salt, der mittels `flutter_secure_storage` gespeichert werden kann, vorausgesetzt.
Anfänglich gilt es zu klären, ob die Ableitung des Schlüssels Teil der Wallet-Bibliothek wird und so in Dart implementiert ist oder ob sie von der auf der Bibliothek aufbauenden App übernommen wird. Erstere Variante bringt den Vorteil mit sich, eine sicheren Standard wie die Verwendung von Argon2 vorgeben zu können. Letztere Variante ermöglicht die Umsetzung der vom BSI empfohlenen Schlüsselableitung mittels in Hardware implementiertem MAC-Verfahren. Ein solches wird durch den Android Keystore ab Android 6.0 bereitgestellt. Dies bietet den weiteren Vorteil, dass eine Gerätebindung des Wallets sichergestellt werden kann. Dies liegt daran, dass durch Verwendung dieses MAC-Verfahrens die Schlüsselableitung nur auf einem spezifischen Gerät erfolgreich sein wird. Entsprechend wird sich vorrangig auf diese Variante konzentriert. Die Funktion aus der Wallet-Bibliothek, die entsprechend die hive-Boxen öffnet, wird also einen AES-Schlüssel erwarten. Eine Erweiterung dahingehen, dass eine zweite Funktion zum

Öffnen der Boxen bereitgestellt wird, die entsprechend ein Passwort erwartet und die Schlüsselableitung bibliotheksseitig übernimmt, wird nicht ausgeschlossen.

Die einzige Hürde an dem eben beschriebenen Vorgehen besteht darin, dass es derzeit keine Flutter-Bibliothek gibt, die eine MAC-Funktion aus Android für Flutter-Anwendungen nutzbar macht. Entsprechend muss diese selbst entwickelt werden. Genauer muss ein Flutter-Plugin geschrieben werden. Als Plugin werden Bibliotheken bezeichnet, die native Funktionen aufrufen und die Ergebnisse an die Futter-Anwendung zurückgeben.

5.4 (Beispiel-)Anwendung

Dieser Abschnitt stellt eine prototypische Anwendung vor, die die vorangehend beschriebene Bibliothek und das Plugin nutzt. Entsprechend ist die App selbst minimalistisch gehalten und beinhaltet vornehmlich eine prototypische Benutzeroberfläche. Es ist anzumerken, dass vorerst das Hauptaugenmerk auf Funktionalität und Sicherheit liegt und nicht auf dem Design der App. Speicherung und Austausch von Credentials wird über den Aufruf entsprechender Bibliotheksfunktionen umgesetzt. Entsprechend wird hier nicht näher darauf eingegangen. Vielmehr wird sich auf weitere im MASVS geforderte Sicherheitsaspekte aus den Kategorien 5, 6 und 8 konzentriert (s. Abschnitt 2.6).

Die vorgestellte App soll zunächst in der Lage sein, Studierendenausweise zu speichern und mittels IWCE anderen Anwendungen zur Verfügung zu stellen. Die Ausstellung eines solchen wird zu Demonstrationszwecken lokal durch die Anwendung selbst geschehen. Diese wird also in der Lage sein, dem Nutzer seinen Ausweis anzuzeigen, sowie auf Credential-Anfragen über einen App-Link reagieren. Zu dieser Reaktion gehört es, dem Nutzer anzuzeigen, welche Attribute zwingend verlangt werden und ihm die Möglichkeit zu bieten, zusätzliche Attribute hinzuzufügen. Abbildung 5.4 zeigt diese beiden Ansichten. Auch diese Screenshots wurden, wie die in Kapitel 4 direkt auf dem Testgerät erstellt. Hierzu wurde die Funktionalität, die das Erstellen von Screenshots verhindert, kurzzeitig durch Auskommentieren der entsprechenden Codezeile deaktiviert.

Zu den angesprochenen, noch zu diskutierenden Sicherheitsaspekten gehört der Umgang mit Netzwerkverbindungen und den für sichere Verbindungen notwendigen TLS-Zertifikaten. Hier gilt es auch zu klären, wie mit Zertifikats-Pinning verfahren wird. IWCE sieht vor, das Zertifikat der jeweils Anfragenden Partei, also eines Verifiers, zu prüfen. Dieses soll genutzt werden, um dem Nutzer Informationen über den Anfragenden zu liefern. Weiterhin ist die Übertragung von Credentials via https neben App-/Universal Links ein Weg, eine Presentation an einen Verifier zu übermitteln. Hierfür sollte sichergestellt sein, dass die Daten nicht mitgelesen werden können. Entsprechend sollte es nicht einfach möglich sein, Datenverkehr über einen Proxy zu lenken und mitzulesen. Hier ist zum einen nützlich, dass Flutter-Apps genauso wie Xamarin-Apps Proxy-Einstellungen

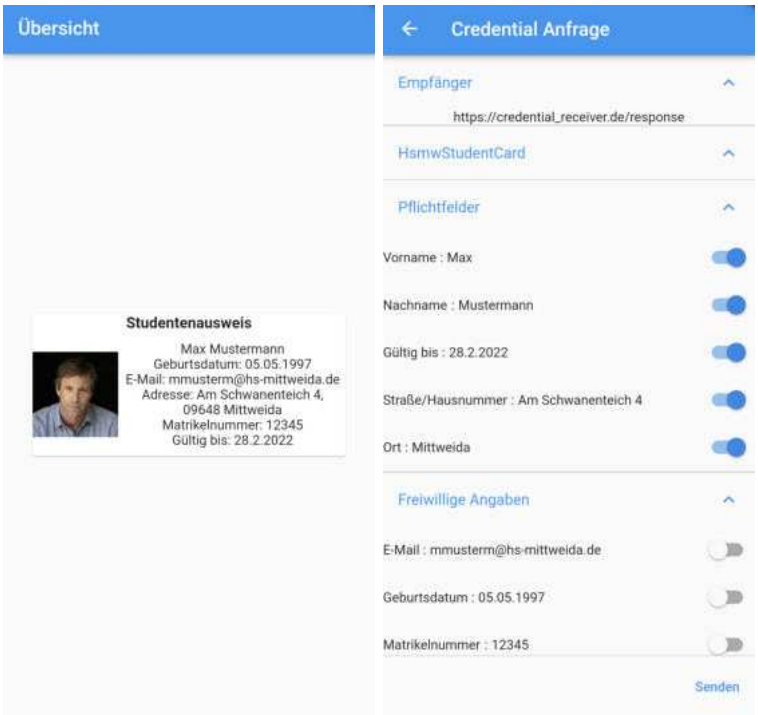


Abbildung 5.1: Credential-Übersicht und Credential-Anfrage

des Android Systems ignorieren. Zum anderen wird der Zertifikatsspeicher von Android nicht genutzt. Die Laufzeitumgebung von Dart/Flutter-Anwendung hat einen eigenen integriert. Genauer werden die von Mozilla bereitgestellten vertrauenswürdigen CA-Zertifikate genutzt. Somit führt der in Abschnitt 4.4 gezeigte Ansatz, das Zertifikat des Proxies nach `/system/etc/security/cacerts` zu kopieren, nicht zum Erfolg, wenn es darum geht, Datenverkehr am Proxy mitlesen zu wollen. In der Flutter-App wird ein Fehler geworfen, dass die Zertifikats-Prüfung fehlschlägt. Entsprechend zeigen sich damit einfache Wege zum Ausheben der Zertifikatsprüfung als erfolglos. Demzufolge wird diese als hinreichend sicher angesehen und auf ein zusätzliches Zertifikats-Pinning verzichtet.

Neben den Netzwerkverbindungen gilt es auch zu diskutieren, wie eine Root-Prüfung stattfinden soll. Hierbei gilt zu klären, inwiefern die SafetyNet Attestation-API von Google eingesetzt wird und was andere Flutter-Bibliotheken leisten. Wie in 2.5.1 aufgezeigt, ist eine Nutzung dieser API nur empfehlenswert, wenn die App die Möglichkeit hat, das Ergebnis an ihren Backend-Server zur Prüfung zu senden. Ein solcher ist im Falle der prototypischen Wallet-Anwendung nicht vorgesehen. Für produktiv betriebene Anwendungen, die ebenfalls nach den Prinzipien von IWCE funktionieren, kann dies anders Aussehen. Da hier vorgesehen ist, dass die jeweilige App eines Issuers die Credentials hält und damit auch der gesamte Issuing-Prozess in der Hand der App-Ersteller liegt, existiert mit dem Server für diese Aufgabe mindestens ein Backend-Dienst, der ebenfalls die Antworten der SafetyNet Attestation-API auswerten kann.

Für die hier beschrieben App wird aus genannten Grund auf eine lokale Root-Prüfung zurückgegriffen. Diese soll, wie in 2.6 erläutert, dazu dienen den Nutzer zu sensibilisieren, dass er mit dem Rooten wichtige Sicherungsmaßnahmen des Betriebssystems aushebelt und damit nun eigenständig für Geräte-Sicherheit sorgen muss (z.B. mittels starkem Geräte-Passwort). Eine Recherche ergab, dass zur Root-Erkennung mehrere Bibliotheken infrage kommen: `flutter_root_jailbreak` [97], `safe_device` [98], `root` [99] und `root_tester` [100]. Von diesen nutzt `root` den schwächsten Ansatz. Hier wird lediglich versucht, eine Shell mit Root-Rechten zu öffnen und anschließend mittels dieser geprüft, ob der Root-Nutzer verfügbar ist. Rooting Tools, wie das für das Testgerät verwendete Magisk, erlauben es aber, einzustellen, welchen Anwendungen Root-Rechte gewährt werden. Entsprechend kann die Shell davon ausgenommen werden und das Gerät wird mittels `root` nicht als gerootet erkannt.
Die übrigen Bibliotheken verfolgen vergleichbare Ansätze. Hier wird zum einen auf das Vorhandensein bekannter Rooting-Tools geprüft, sowie nach Apps gesucht, die Root-Rechte benötigen und solchen, die dafür bekannt sind Rooting-Tools zu verstecken. Weiterhin wird nach der ausführbaren Datei `su` gesucht. Ist eine der Prüfungen erfolgreich, wird das Gerät als gerootet erkannt. `root_tester` implementiert dabei die genannten Prüfungen selbst, die übrigen zwei greifen auf die Java-Bibliothek `rootbeer` [101] zurück. Ein Test dieser drei Bibliotheken auf dem gerooteten Testgerät ergibt, dass alle prinzipiell in der Lage sind, Root zu erkennen. Wird allerdings die Magisk-App umbenannt und ein Tool zum Verstecken der vorhandenen Root-Berechtigungen wie MagiskHide genutzt, schlägt bei allen Bibliotheken die Prüfung fehl. Tools wie MagiskHide erfordern allerdings, dass explizit angegeben wird, vor welchen Apps sie die Root-Berechtigungen verstecken sollen. Entsprechend ist davon auszugehen, dass bei einem Erstnutzer die Root-Erkennung mit hoher Wahrscheinlichkeit anschlägt.
Der Test ergab also, dass es prinzipiell egal ist, welche Bibliothek gewählt wird. Schlussendlich fällt die Wahl auf `safe_device`. Diese unterscheidet sich von den übrigen darin, dass zusätzlich geprüft werden kann, ob die App auf einem Emulator läuft.

6 Fazit

Kapitel 3 und 4 haben bereits bestehende Wallet-Anwendungen vorgestellt und ihre Datensicherheit untersucht. Dabei ist aufgefallen, dass alle Anwendungen diesbezüglich Mängel enthalten. Zuerst einmal muss festgehalten werden, dass diese mehrheitlich nur auf einem gerooteten Gerät zum Tragen kommen. Ausnahme ist hier Connect.Me, das mehrere Möglichkeiten bietet, lediglich mit aktivem USB-Debugging an Daten zu gelangen. Die Rede ist hier von Logging-Ausgaben und der Möglichkeit zur Erstellung eines lokalen Backups. Insgesamt lässt sich damit auch sagen, dass bei Connect.Me bei der Untersuchung die meisten Unzulänglichkeiten aufgefallen sind. Dennoch zeigt diese App Ansätze, die bei den übrigen vermisst wurden. Hierzu gehören eine funktionierende Root-Erkennung und der Ansatz eines eigenen Zertifikatsspeichers für TLS-Zertifikate.

Bei den übrigen Anwendungen sind sensible Daten nur mit Root-Rechten abrufbar, da sie jeweils im App-eigenen Datenverzeichnis des internen Speichers abgelegt wurden. Mit Ausnahme von Jolocom SmartWallet wurden die Daten hier sogar verschlüsselt. Allerdings ist der Umgang mit dem zur Entschlüsselung nötigen Passwort mit Ausnahme von ID-Wallet nicht gelungen. Es erschließt sich nicht, warum dieses bereits in den Arbeitsspeicher geladen und damit entschlüsselt wird, bevor sich der Nutzer authentifiziert hat. Lediglich ID-Wallet zeigt mit der Nutzung des eingegebenen PINs eine sicheren Ansatz zum Generieren des nötigen Passworts.

Somit ist nach derzeitigem Stand von der Nutzung der vier noch verfügbaren Apps von den fünf untersuchten für die Speicherung wichtiger persönlicher Daten abzuraten. Generell ist anzumerken, dass diese noch keinen praktischen Nutzen für Endanwender haben, da es derzeitig noch keinen produktiv agierenden Issuer und Verifier für entsprechende Credentials gibt. So wie sich die Hintergrundtechnologien wie Hyperledger Indy/Aries oder die Jolocom Technologie noch in Entwicklung befinden, müssen auch die Wallet-Anwendungen weiterentwickelt werden. Bei dieser Weiterentwicklung sollte die Datensicherheit nicht aus den Augen verloren werden.

Demgegenüber konnte in Kapitel 5 gezeigt werden, wie sich mittels sorgfältiger Auswahl von Bibliotheken die Forderungen des MASVS und damit eine möglichst sichere Wallet-Anwendung umsetzen lässt. Diese hat einen prototypischen Charakter und soll vornehmlich den Umgang mit der ihr zugrundeliegenden Bibliothek zeigen, die ebenfalls nach den Maßgaben entwickelt wurde, eine bestmögliche Sicherheit der Wallet und der in ihr gespeicherten Daten zu gewährleisten.

Es ist zu bedenken, dass sich bei der Umsetzung der Sicherungsmaßnahmen des Wallets vornehmlich auf Mechanismen des Betriebssystems Android verlassen wurde. Dies trifft allen voran auf verwendete Kryptografische Schlüssel zu und tangiert damit das

Android Keystore System. Wie in Abschnitt 2.5.3 vorgestellt wurde, können die Sicherungsmaßnahmen dessen auf verschiedenen Wegen umgesetzt sein. Die jeweilige Implementierung entscheidet über das Maß der Sicherheit. Hier ist zu bedenken, dass eine hardwaregestützte Umsetzung mittels SE/HSM bisher nur in wenigen Geräten (Google Pixel Reihe ab Pixel 3 und Samsung Galaxy S Reihe ab S20) verfügbar ist. Seitens Google sind hier aber Ansätze zu sehen, dies zu ändern. So wurde 2021 die „Android Ready SE Alliance“ gegründet. Ziel dieser ist es, zusammen mit Herstellern von SEs Anwendungen/APIs zu schaffen, die selbiges direkt nutzen. Zu diesen Anwendungen kann z.B. auch das Speichern von Identitätsdaten, wie Ausweisen und Führerscheinen gehören [102]. In Bezug auf dieses konkrete Anwendungsbeispiel existiert für Android 11 und 12 bereits eine API für sogenannte Identity Credentials, deren Implementierung Geräteherstellern strengsten empfohlen wird. Diese ist allerdings für das Speichern von Identitätsdaten, die nach ISO 18013-5 „Personal identification - ISO-compliant driving licence - Part 5: Mobile driving licence (mDL) application“ aufgebaut sind, gedacht und damit nicht vollständig mit Verifiable Credentials der W3C kompatibel [103, 104]. Dennoch zeigt dies, dass zumindest seitens Google Anreize geschaffen werden, die sichere Speicherung von Identitätsdaten in Android voranzutreiben.

Auch nationale Projekte zeigen, dass das Vorhandensein von SEs in Smartphones für die Speicherung von Identitätsdaten eine wichtige Rolle spielt. Hier ist vor allem das 2021 abgeschlossene Projekt Optimos 2.0 zu nennen, das sich damit auseinandersetzt, deutsche Personalausweise mit online Funktion auf einem SE abzulegen und nutzbar zu machen (Smart-eID). Genauer wurde hier eine Anwendung für das im Samsung Galaxy S21 verbaute SE geschaffen [105, 106]. Im weitesten Sinne wurde also eine Anwendung vergleichbar zu denen entwickelt, die Google mit der „Android Ready SE Alliance“ plant.

Neben diesen beiden auf SEs und hoheitliche Dokumente spezialisierte Ansätze wird auch innerhalb der Community für SSI an Standards und Vorgaben für sichere Wallets gearbeitet. Hier ist die Arbeitsgruppe „Wallet Security“ der Decentralized Identity Foundation zu nennen. Deren Mitglieder diskutieren aktuell vornehmlich, wie sich einzelne Credentials an das jeweilige Smartphone binden lassen. Der vielversprechendes Vorschlag sieht dabei vor, in das jeweilige Credential den Öffentlichen Schlüssel eines im Keystore des Betriebssystem hinterlegten Schlüssels zu integrieren [107].

Abschließend lässt sich also festhalten, dass es für die Weiterentwicklung bzw. Neuentwicklung von Wallet-Anwendung Ansätze gibt, wie diese sicher gestaltet werden können. Außerdem konnte bereits mit der in dieser Arbeit entwickelten Wallet gezeigt werden, wie dies bereits mit dem derzeitigen Stand der Technik möglich ist. Natürlich bestehen auch hier Weiterentwicklungsmöglichkeiten hinsichtlich der Beachtung der eben genannten Projekte.

Anhang A: App-Analyse nach MASVS

Die nachfolgend Abschnitte fassen die Untersuchungsergebnisse für die einzelnen Apps tabellarisch zusammen. Der Übersicht halber werden die Prüfpunkte nur stichpunktartig genannt. Die konkreten Formulierungen sind dem MASVS zu entnehmen [9].

A.1 Anforderung an Datenspeicherung und Datenschutz

	Jolocom SmartWallet	Connect.Me	esatus Wallet	Lissi Wallet	ID-Wallet	eigene App
Betriebssystem-seitige Speichermechanismen	Zum Teil: Wichtige Daten liegen im Datenverzeichnis der App, aber nicht alle kryptografischen Schlüssel im Keystore					
keine sensiblen Daten im externen Speicher	ja	ja	ja	ja	ja	ja
keine sensiblen Daten in Logfiles	ja	nein	ja	ja	ja	ja
keine sensiblen Daten an Dritten	ja	ja	ja	ja	ja	ja
Tastatur-Cache	deaktiviert	deaktiviert	deaktiviert	deaktiviert	deaktiviert	deaktiviert
keine sensiblen Daten über IPC übertragen	ja	ja	ja	ja	ja	ja

	Jolocom SmartWallet	Connect.Me	esatus Wallet	Lissi Wallet	ID-Wallet	eigene App
keine sensiblen Daten über Benutzeroberfläche exponiert	ja	ja	ja	ja	ja	ja
Backup-Steuerung	deaktiviert	automatisch; via ADB	deaktiviert	deaktiviert	deaktiviert	deaktiviert
Im Hintergrundmodus keine sensiblen Daten sichtbar	ja	nein	nein	nein	nein	ja
sensible Daten im RAM löschen	nein	nein	nein	nein	nein	ansatzweise
Minimum an Geräteschutz-Richtlinien	nein	nein	nein	nein	nein	ja
Datenverarbeitungshinweise und Security-Best-Practices	geringfügig	versteckt in AGB	Ersteres ja; versteckt in Datenschutzbestimmungen; zweiteres nein			nein (Prototyp)
Sensible Daten nicht lokal speichern	Es ist Sinn und Zweck der Anwendungen personenbezogenen Daten lokal zu speichern. Daher kann dieser Punkt außer Acht gelassen werden					
Verschlüsselung lokaler Daten	nein	nein	teilweise: verschlüsselt, Schlüsselzugriff ohne Authentifizierung		teilweise: verschlüsselt; Schlüssel nicht im Keystore	ja (bei aktiver Biometrie)
Datenlöschung bei häufig scheiterndem Login	nein	nein	keine komplette Sperrung		nein	ja

A.2 Anforderungen an Kryptografie

	Jolocom SmartWallet	Connect.Me	esatus Wallet	Lissi Wallet	ID-Wallet	eigene App
keine hart-codierten Schlüsseln	ja	ja	ja	ja	ja	ja
bewährte Implementierungen für kryptografischer Primitive	ja	ja	ja	ja	ja	ja
kryptografische Primitive konfiguriert nach Stand der Technik	ja	ja	ja	ja	ja	ja
keine veralteten und unsicheren Algorithmen	ja	ja	ja	ja	ja	ja
kryptografische Schlüssel für genau einen Zweck	ja	ja	ja	ja	ja	ja
sicherer kryptografischer Zufallszahlengenerator	ja	ja	ja	ja	ja	ja

A.3 Anforderung an Netzwerkkommunikation

	Jolocom SmartWallet	Connect.Me	esatus Wallet	Lissi Wallet	ID-Wallet	eigene App
durchgängige TLS-Verschlüsselung	ja	ja	ja	ja	ja	ja
TLS-Einstellungen nach aktuellen Best-Practices	ja	ja	ja	ja	ja	ja
X.509-Zertifikat prüfen	ja	ja	nein	ja	nein	ja
Zertifikats-Pinning / eigener Zertifikatsspeicher	nein	nein, trotz gespeicherter Zertifikate	nein	nein	nein	via Flutter
keine unsicheren Kommunikationskanäle	ja	ja	ja	ja	ja	ja
Verwendung aktueller Bibliotheken	ja	ja	ja	ja	ja	ja

A.4 Anforderungen zur Plattform-Interaktion

	Jolocom SmartWallet	Connect.Me	esatus Wallet	Lissi Wallet	ID-Wallet	eigene App
nur unbedingt erforderliche App-Berechtigungen	ja	ja	ja	ja	ja	ja
Eingabe-Validierung	ja	ja	ja	ja	ja	ja
sensible Funktionalität über App-eigene URL-Schemas	noch nicht	Angebot und Anfrage von Credentials mit Zustimmung				Credential-Anfrage mit Zustimmung
keine sensible Funktionalität über IPC	ja	ja	ja	ja	ja	ja
JavaScript in WebViews	Es gibt keine WebViews					
Protokoll-Handler in WebViews	Es gibt keine WebViews					
WebView Zugriff auf native Methoden	Es gibt keine WebViews					
sichere Serialisierungs-API	ja	ja	ja	ja	ja	ja
Schutz vor Screen Overlay Angriffen	nein	nein	nein	nein	nein	nein
Ressourcen von WebViews löschen	Es gibt keine WebViews					

A.5 Anforderungen an Manipulationssicherheit/Resilienz

	Jolocom SmartWallet	Connect.Me	esatus Wallet	Lissi Wallet	ID-Wallet	eigene App
Root-Erkennung	nein	ja	nein	nein	nein	ja
Debug-Erkennung	nein	nein	nein	nein	nein	nein
Manipulationserkennung ausführbare Dateien	nein	nein	nein	nein	nein	nein
Erkennung von Reverse-Engeneering Tools/Frameworks	nein	nein	nein	nein	nein	nein
Emulator-Erkennung	nein	nein	ja	ja	ja	ja
Manipulationserkennung RAM	nein	nein	nein	nein	nein	nein
mehrere Mechanismen	nein	nein	nein	nein	nein	ja
Unterschiedliche Reaktionen	nein	nein	nein	nein	nein	nein
Obfuskierung	nein	nein	nein	nein	nein	nein
Gerätebindung	nein	nein	nein	nein	nein	ja
Verschlüsselung ausführbare Dateien	nein	nein	nein	nein	nein	nein

	Jolocom SmartWallet	Connect.Me	esatus Wallet	Lissi Wallet	ID-Wallet	eigene App
Obfuskierung nach Stand der Technik	nein	nein	nein	nein	nein	nein
Payload-Verschlüsselung	nein	Wird durch DIDComm geregelt				nein

Literatur

[1] Christian Tietz u. a. *Management digitaler Identitäten: aktueller Status und zukünftige Trends*. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam 114. Potsdam: Universitätsverlag Potsdam, 2017. 66 S. ISBN: 978-3-86956-395-4.

[2] Christoper Allen. *The Path to Self-Sovereign Identity*. 25. Apr. 2016. URL: http://www.lifewithalacrity.com/2016/04/the-path-to-self-soverereign-identity.html (besucht am 15. 05. 2020).

[3] *Self-Sovereign Identity – Esatus AG*. URL: https://esatus.com/loesungen/self-sovereign-identity/ (besucht am 23. 11. 2021).

[4] *Lissi*. lissi. URL: https://lissi.id/lissi.id (besucht am 23. 11. 2021).

[5] *My Digi ID*. URL: https://my-digi-id.github.io/index.html#mobile-app (besucht am 23. 11. 2021).

[6] *Connect.Me*. URL: https://www.connect.me/ (besucht am 23. 11. 2021).

[7] *Solution*. Jolocom. URL: https://jolocom.io/solution/ (besucht am 23. 11. 2021).

[8] *Mobile Operating System Market Share Worldwide*. StatCounter Global Stats. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (besucht am 17. 11. 2021).

[9] Carlos Holguera u. a. *MASVS - Mobile Application Security Verification Standard - Version 1.3*. 13. Mai 2021. URL: https://github.com/OWASP/owasp-masvs/releases (besucht am 23. 11. 2021).

[10] Bernhard Mueller u. a. *MSTG Mobile Security Testing Guide - Version 1.2*. URL: https://github.com/OWASP/owasp-mstg/releases/download/v1.2/OWASP_MSTG-1.2.pdf (besucht am 14. 09. 2021).

[11] Alasdair McAndrew. *Introduction to Cryptography with Open-Source Software*. 1st. USA: CRC Press, Inc., 2011. 461 S. ISBN: 978-1-4398-2570-9.

[12] Bundesamt für Sicherheit in der Informationstechnik (BSI). *BSI -Technische Richtlinie: Kryptografische Verfahren: Empfehlungen und Schlüssellängen*. 24. März 2021. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile&v=2 (besucht am 19. 10. 2021).

[13] Nigel P. Smart. *Algorithms, Key Size and Protocols Report (2018)*. 28. Feb. 2018. URL: https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf (besucht am 04. 11. 2021).

[14] Christof Paar und Jan Pelzl. „Message Authentication Codes (MACs)“. In: *Understanding Cryptography: A Textbook for Students and Practitioners*. Hrsg. von Christof Paar und Jan Pelzl. Berlin, Heidelberg: Springer, 2010, S. 319–330. ISBN: 978-3-642-04101-3. DOI: 10.1007/978-3-642-04101-3_12. URL: https://doi.org/10.1007/978-3-642-04101-3_12 (besucht am 03. 11. 2021).

[15] Morris J. Dworkin u. a. *Advanced Encryption Standard (AES)*. URL: https://www.nist.gov/publications/advanced-encryption-standard-aes (besucht am 28. 07. 2021).

[16] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. NIST Special Publication (SP) 800-38A. National Institute of Standards and Technology, 1. Dez. 2001. DOI: 10.6028/NIST.SP.800-38A. URL: https://csrc.nist.gov/publications/detail/sp/800-38a/final (besucht am 28. 07. 2021).

[17] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication (SP) 800-38D. National Institute of Standards and Technology, 28. Nov. 2007. DOI: 10.6028/NIST.SP.800-38D. URL: https://csrc.nist.gov/publications/detail/sp/800-38d/final (besucht am 28. 07. 2021).

[18] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. NIST Special Publication (SP) 800-38C. National Institute of Standards and Technology, 20. Juli 2007. DOI: 10.6028/NIST.SP.800-38C. URL: https://csrc.nist.gov/publications/detail/sp/800-38c/final (besucht am 19. 10. 2021).

[19] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices*. NIST Special Publication (SP) 800-38E. National Institute of Standards and Technology, 18. Jan. 2010. DOI: 10.6028/NIST.SP.800-38E. URL: https://csrc.nist.gov/publications/detail/sp/800-38e/final (besucht am 19. 10. 2021).

[20] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping*. NIST Special Publication (SP) 800-38F. National Institute of Standards and Technology, 13. Dez. 2012. DOI: 10.6028/NIST.SP.800-38F. URL: https://csrc.nist.gov/publications/detail/sp/800-38f/final (besucht am 19. 10. 2021).

[21] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. NIST Special Publication (SP) 800-38B. National Institute of Standards and Technology, 6. Okt. 2016. DOI: 10.6028/NIST.SP.800-38B. URL: https://csrc.nist.gov/publications/detail/sp/800-38b/final (besucht am 19. 10. 2021).

[22] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption*. NIST Special Publication (SP) 800-38G Rev. 1 (Draft). National Institute of Standards and Technology, 28. Feb. 2019. DOI: 10.6028/NIST.SP.800-38Gr1-draft. URL: https://csrc.nist.gov/publications/detail/sp/800-38g/rev-1/draft (besucht am 19. 10. 2021).

[23] Manu Sporny u. a. *Decentralized Identifiers (DIDs) v1.0*. 3. Aug. 2021. URL: https://www.w3.org/TR/did-core/ (besucht am 24. 11. 2021).

[24] Tantek Çelik. *[Wbs] Response to 'Call for Review: Decentralized Identifiers (DIDs) v1.0 Is a W3C Proposed Recommendation'*. 1. Sep. 2021. URL: https://lists.w3.org/Archives/Public/public-new-work/2021Sep/0000.html (besucht am 24. 11. 2021).

[25] Drummond Reed. *Does the W3C Still Believe in Berners-Lee's Vision of Decentralization?* Evernym. 12. Okt. 2021. URL: https://www.evernym.com/blog/w3c-vision-of-decentralization/ (besucht am 24. 11. 2021).

[26] Manu Sporny, Dave Longley und Dave Chadwick. *Verifiable Credentials Data Model v1.1*. 9. Nov. 2021. URL: https://www.w3.org/TR/vc-data-model/ (besucht am 24. 11. 2021).

[27] Dave Chadwick u. a. *Verifiable Credentials Implementation Guidelines 1.0*. 24. Sep. 2019. URL: https://www.w3.org/TR/vc-imp-guide/#progressive-trust (besucht am 20. 01. 2022).

[28] Jan Camenisch und Anna Lysyanskaya. „An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation". In: *Advances in Cryptology — EUROCRYPT 2001*. Hrsg. von Birgit Pfitzmann. Bearb. von Gerhard Goos, Juris Hartmanis und Jan van Leeuwen. Bd. 2045. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 93–118. ISBN: 978-3-540-42070-5 978-3-540-44987-4. DOI: 10.1007/3-540-44987-6_7. URL: http://link.springer.com/10.1007/3-540-44987-6_7 (besucht am 29. 04. 2020).

[29] Andreas Antonopolus. „Wallets". In: *Mastering Bitcoin*. O'Reilly, Juni 2017. ISBN: 978-1-4919-5438-6. URL: https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch05.asciidoc (besucht am 24. 11. 2021).

[30] *Universal Wallet 2020*. URL: https://w3c-ccg.github.io/universal-wallet-interop-spec/ (besucht am 21. 06. 2021).

[31] *Hyperledger/Aries-Rfcs*. Hyperledger, 2. Sep. 2021. URL: https://github.com/hyperledger/aries-rfcs/blob/08653f21a489bf4717b54e4d7fd2d0bdfe6b4d1a/concepts/0050-wallets/README.md (besucht am 02. 09. 2021).

[32] *Android Open Source Project*. Android Open Source Project. URL: https://source.android.com/ (besucht am 17. 11. 2021).

[33] *Adoptable Storage*. Android Open Source Project. URL: https://source.android.com/devices/storage/adoptable (besucht am 19. 01. 2022).

[34] Ronan Loftus und Marwin Baumann. *Android 7 File Based Encryption and the Attacks Against It*.

[35] Tobias Groß, Matanat Ahmadova und Tilo Müller. „Analyzing Android's File-Based Encryption: Information Leakage through Unencrypted Metadata". In: *Proceedings of the 14th International Conference on Availability, Reliability and Security*. ARES '19: 14th International Conference on Availability, Reliability and Security. Canterbury CA United Kingdom: ACM, 26. Aug. 2019, S. 1–7. ISBN: 978-1-4503-7164-3. DOI: 10.1145/3339252.3340340. URL: https://dl.acm.org/doi/10.1145/3339252.3340340 (besucht am 16. 09. 2021).

[36] *Metadata Encryption*. Android Open Source Project. URL: https://source.android.com/security/encryption/metadata (besucht am 13. 10. 2021).

[37] *Verified Boot | Android Open Source Project*. URL: https://source.android.com/security/verifiedboot (besucht am 17. 11. 2021).

[38] *Device State | Android Open Source Project*. URL: https://source.android.com/security/verifiedboot/device-state (besucht am 17. 11. 2021).

[39] Raphael Bialon. *On Root Detection Strategies for Android Devices*. 3. Dez. 2020. arXiv: 2012.01812 [cs]. URL: http://arxiv.org/abs/2012.01812 (besucht am 29. 12. 2021).

[40] *SafetyNet Attestation API | Android Developers*. URL: https://developer.android.com/training/safetynet/attestation?hl=de (besucht am 29. 12. 2021).

[41] Muhammad Ibrahim, Abdullah Imran und Antonio Bianchi. „SafetyNOT: On the Usage of the SafetyNet Attestation API in Android". In: *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '21. New York, NY, USA: Association for Computing Machinery, 24. Juni 2021, S. 150–162. ISBN: 978-1-4503-8443-8. DOI: 10.1145/3458864.3466627. URL: https://doi.org/10.1145/3458864.3466627 (besucht am 05. 02. 2022).

[42] Nikolay Elenkov. *Android Security Internals: An in-Depth Guide to Android's Security Architecture*. San Francisco: No Starch Press, 2015. 401 S. ISBN: 978-1-59327-581-5.

[43] *Analyze Your Build with APK Analyzer*. Android Developers. URL: https://developer.android.com/studio/debug/apk-analyzer?hl=de (besucht am 19. 01. 2022).

[44] *App Manifest Overview | Android Developers*. URL: https://developer.android.com/guide/topics/manifest/manifest-intro?hl=de (besucht am 20. 10. 2021).

[45] *Permissions on Android | Android Developers*. URL: https://developer.android.com/guide/topics/permissions/overview?hl=de (besucht am 13. 10. 2021).

[46] *Define a Custom App Permission*. Android Developers. URL: https://developer.android.com/guide/topics/permissions/defining?hl=de (besucht am 13. 10. 2021).

[47] *Introduction to Activities*. Android Developers. URL: https://developer.android.com/guide/components/activities/intro-activities?hl=de (besucht am 27. 10. 2021).

[48] *Services Overview*. Android Developers. URL: https://developer.android.com/guide/components/services?hl=de (besucht am 27. 10. 2021).

[49] *Broadcasts Overview*. Android Developers. URL: https://developer.android.com/guide/components/broadcasts?hl=de (besucht am 28. 10. 2021).

[50] *Content Provider Basics | Android Developers*. URL: https://developer.android.com/guide/topics/providers/content-provider-basics?hl=de (besucht am 28. 10. 2021).

[51] *Verify Android App Links | Android Developers*. URL: https://developer.android.com/training/app-links/verify-site-associations?hl=de (besucht am 26. 10. 2021).

[52] Fang Liu u. a. „Measuring the Insecurity of Mobile Deep Links of Android“. In: 26th USENIX Security Symposium (USENIX Security 17). 2017, S. 953–969. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/liu (besucht am 26. 10. 2021).

[53] Yutian Tang u. a. „All Your App Links Are Belong to Us: Understanding the Threats of Instant Apps Based Attacks“. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 8. Nov. 2020, S. 914–926. ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3409702. URL: https://doi.org/10.1145/3368089.3409702 (besucht am 26. 10. 2021).

[54] *Data and File Storage Overview*. Android Developers. URL: https://developer.android.com/training/data-storage?hl=de (besucht am 17. 11. 2021).

[55] Kamil Breński, Krzysztof Pranczk und Mateusz Fruba. *How Secure Is Your Android Keystore Authentication ?* F-Secure Labs. 21. Aug. 2019. URL: https://labs.f-secure.com/blog/how-secure-is-your-android-keystore-authentication/ (besucht am 30. 08. 2021).

[56] *OWASP Top 10:2021*. URL: https://owasp.org/Top10/ (besucht am 19. 11. 2021).

[57] Bundesamt für Sicherheit in der Informationstechnik (BSI). *Sicherheitsanforderungen an digitale Gesundheitsanwendungen - Technische Richtlinie BSI TR-03161*. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR03161/BSI-TR-03161.pdf?__blob=publicationFile&v=2 (besucht am 20. 01. 2022).

[58] European Union Agency for Network and Information Security. *Smartphone Secure Development Guidelines.* LU: Publications Office, 2016. URL: https://data.europa.eu/doi/10.2824/071102 (besucht am 20. 01. 2022).

[59] *Frida • A World-Class Dynamic Instrumentation Framework*. Frida • A world-class dynamic instrumentation framework. URL: https://frida.re/ (besucht am 24. 11. 2021).

[60] John Wu. *Topjohnwu/Magisk*. 24. Nov. 2021. URL: https://github.com/topjohnwu/Magisk (besucht am 24. 11. 2021).

[61] Jolocom GmbH. *About*. Jolocom. URL: https://jolocom.io/de/about/ (besucht am 29. 11. 2021).

[62] *Jolocom/Smartwallet-App*. JOLOCOM, 20. Okt. 2021. URL: https://github.com/jolocom/smartwallet-app (besucht am 29. 11. 2021).

[63] Mark Vayngrib. *React-Native-Randombytes*. 2. Jan. 2022. URL: https://github.com/mvayngrib/react-native-randombytes (besucht am 14. 01. 2022).

[64] Marek Paltinus u. a. *Mnemonic Code for Generating Deterministic Keysbitcoin/Bips*. Bitcoin, 10. Sep. 2013. URL: https://github.com/bitcoin/bips/blob/edffe529056f6dfd33d8f716fb871467c3c09263/bip-0039.mediawiki (besucht am 30. 11. 2021).

[65] *Scrcpy (v1.21)*. Genymobile, 14. Jan. 2022. URL: https://github.com/Genymobile/scrcpy (besucht am 14. 01. 2022).

[66] *Wallet-Rs*. JOLOCOM, 11. Okt. 2021. URL: https://github.com/jolocom/wallet-rs (besucht am 12. 10. 2021).

[67] *Vaulted Key Provider*. JOLOCOM, 9. Feb. 2021. URL: https://github.com/jolocom/vaulted-key-provider (besucht am 11. 10. 2021).

[68] Yanick Bélanger. *React Native Encrypted Storage*. 12. Jan. 2022. URL: https://github.com/emeraldsanto/react-native-encrypted-storage (besucht am 14. 01. 2022).

[69] *Transparent Proxying*. URL: https://docs.mitmproxy.org/stable/howto-transparent/ (besucht am 08. 01. 2022).

[70] *Frida CodeShare*. URL: https://codeshare.frida.re/@akabe1/frida-multiple-unpinning/ (besucht am 10. 01. 2022).

[71] *Evernym / Mobile / ConnectMe*. GitLab. URL: https://gitlab.com/evernym/mobile/connectme (besucht am 13. 12. 2021).

[72] Anna Biselli. *Interview zu ID Wallet: Konzeptionell kaputt und ein riesiger Rückschritt*. netzpolitik.org. 3. Okt. 2021. URL: https://netzpolitik.org/2021/interview-zu-id-wallet-konzeptionell-kaputt-und-ein-riesiger-rueckschritt/ (besucht am 10. 12. 2021).

[73] *ID Wallet*. My Digi ID, 29. Nov. 2021. URL: https://github.com/My-DIGI-ID/ID-Wallet (besucht am 13. 12. 2021).

[74] Stephan Curran, Paul Bastian und Daniel Hardman. *Indy DID Method Specification*. URL: https://hyperledger.github.io/indy-did-method/ (besucht am 21. 01. 2022).

[75] Hyperledger Community. *Indy SDK*. Hyperledger, 13. Jan. 2022. URL: https://github.com/hyperledger/indy-sdk (besucht am 14. 01. 2022).

[76] Christian Reitter. *X41sec/Tools*. X41 D-Sec, 22. Dez. 2021. URL: https://github.com/x41sec/tools/blob/c38bf086188ff965ab0b20c9f5a1abd2c0fa6e22/Mobile/Xamarin/Xamarin_XALZ_decompress.py (besucht am 14. 01. 2022).

[77] *Overview | React-Native-Sensitive-Info*. URL: https://mcodex.github.io/react-native-sensitive-info/docs/ (besucht am 15. 12. 2021).

[78] Daniel Hardman. *Aries RFC 0046: Mediators and Relays*. GitHub. 1. Feb. 2019. URL: https://github.com/hyperledger/aries-rfcs/tree/main/concepts/0046-mediators-and-relays (besucht am 11. 01. 2022).

[79] Kyle Den Hartog u. a. *Aries RFC 0019: Encryption Envelope*. GitHub. 4. Mai 2019. URL: https://github.com/hyperledger/aries-rfcs/tree/main/features/0019-encryption-envelope (besucht am 11. 01. 2022).

[80] Christoph Menzer. *Inter-Wallet Credential Exchange*. Praktikumsbericht. Hochschule Mittweida, 19. Feb. 2021.

[81] Christoph Menzer und Sarah Otto. *Inter-Wallet Credential Exchange*. 16. Aug. 2021. URL: https://b2cm.github.io/iwce/ (besucht am 27. 01. 2022).

[82] Orie Steele und Manu Sporny. *DID Specification Registries*. URL: https://www.w3.org/TR/did-spec-registries/#did-methods (besucht am 24. 01. 2022).

[83] Pelle Braendgaard und Joel Torstensson. *EIP-1056: Ethereum Lightweight Identity*. Ethereum Improvement Proposals. 3. Mai 2018. URL: https://eips.ethereum.org/EIPS/eip-1056 (besucht am 07. 02. 2022).

[84] *Sqflite | Flutter Package*. Dart packages. URL: https://pub.dev/packages/sqflite (besucht am 04. 01. 2022).

[85] *Sembast | Dart Package*. Dart packages. URL: https://pub.dev/packages/sembast (besucht am 04. 01. 2022).

[86] *Objectbox | Dart Package*. Dart packages. URL: https://pub.dev/packages/objectbox (besucht am 04. 01. 2022).

[87] *Isar | Dart Package*. Dart packages. URL: https://pub.dev/packages/isar (besucht am 04. 01. 2022).

[88] *Hive | Dart Package*. Dart packages. URL: https://pub.dev/packages/hive (besucht am 04. 01. 2022).

[89] *Sqflite_sqlcipher | Flutter Package*. Dart packages. URL: https://pub.dev/packages/sqflite_sqlcipher (besucht am 04. 01. 2022).

[90] *Sqflite_common_ffi | Dart Package*. Dart packages. URL: https://pub.dev/packages/sqflite_common_ffi (besucht am 04. 01. 2022).

[91] Peter Wuille. *BIP-32: Hierarchical Deterministic Wallets*. Bitcoin, 4. Nov. 2020. URL: https://github.com/bitcoin/bips/blob/02de475efc528058bd04a0c4ad31b6422aed5f5f/bip-0032.mediawiki (besucht am 24. 01. 2022).

[92] *EncryptedSharedPreferences | Android Developers*. URL: https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences?hl=de (besucht am 05. 01. 2022).

[93] *Flutter_secure_storage | Flutter Package*. URL: https://pub.dev/packages/flutter_secure_storage (besucht am 05. 01. 2022).

[94] *Encrypted_shared_preferences | Dart Package*. Dart packages. URL: https://pub.dev/packages/encrypted_shared_preferences (besucht am 05. 01. 2022).

[95] *Biometric_storage | Flutter Package*. Dart packages. URL: https://pub.dev/packages/biometric_storage (besucht am 05. 01. 2022).

[96] *Local_auth | Flutter Package*. Dart packages. URL: https://pub.dev/packages/local_auth (besucht am 05. 01. 2022).

[97] *Flutter_jailbreak_detection | Flutter Package*. Dart packages. URL: https://pub.dev/packages/flutter_jailbreak_detection (besucht am 05. 02. 2022).

[98] *Safe_device | Flutter Package*. Dart packages. URL: https://pub.dev/packages/safe_device (besucht am 05. 02. 2022).

[99] *Root | Flutter Package*. Dart packages. URL: https://pub.dev/packages/root (besucht am 05. 02. 2022).

[100] *Root_tester | Flutter Package*. Dart packages. URL: https://pub.dev/packages/root_tester (besucht am 05. 02. 2022).

[101] *Rootbeer/RootBeer.Java at Master · Scottyab/Rootbeer*. GitHub. URL: https://github.com/scottyab/rootbeer (besucht am 05. 02. 2022).

[102] *Android Security and Privacy*. Google Developers. URL: https://developers.google.com/android/security/android-ready-se (besucht am 04. 02. 2022).

[103] *Android 11 Compatibility Definition - Section 9.11.3 Identity Credential*. Android Open Source Project. URL: https://source.android.com/compatibility/11/android-11-cdd?hl=en#9_11_3_identity_credential (besucht am 04. 02. 2022).

[104] *IdentityCredentialStore | Android Developers*. URL: https://developer.android.com/reference/android/security/identity/IdentityCredentialStore (besucht am 04. 02. 2022).

[105] Olaf Clemens. *Optimos 2.0*. URL: https://www.landkreistag.de/images/stories/themen/ITSicherheit/siebenterKongress/2020-05-03_Optimos_20__7KITS.pdf (besucht am 04. 02. 2022).

[106] Bundesministerium für Wirtschaft und Klimaschutz. *Optimos 2.0 - Technologieprogramm Smart Service Welt II*. URL: https://www.digitale-technologien.de/DT/Redaktion/DE/Standardartikel/SmartServiceWeltProjekte/Wohnen_Leben/SSWII_Projekt_OPTIMOS_20.html (besucht am 04. 02. 2022).

[107] Paul Bastian und Sebastian Bickerle. *Wallet Security Working Group -Device Binding Work Item*. Decentralized Identity Foundation, 23. Dez. 2021. URL: https://github.com/decentralized-identity/wallet-security/blob/cdb23d3d996f30143a522bfae56ea7db749b624b/work_items/device_binding.md (besucht am 04. 02. 2022).

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.



Mittweida, 07. Februar 2022