JOHANNES KEPLER
UNIVERSITY LINZ

Author
Stefan Kempinger, BSc
11908714

Submission
**Institute of
Networks and Security**

Thesis Supervisor
Prof. Dr. **René Mayrhofer**

Assistant Thesis
Supervisor
Dr. **Michael Roland**

July 2024

Assessing the Feasibility of Developing a Secure Digital Identity Wallet for Android

Master's Thesis

to confer the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

This master thesis explores the feasibility and security aspects of implementing a digital identity wallet on Android smartphones. With the increasing prominence of digital wallets in various domains, the security of these wallets, which store and manage sensitive data, is of paramount importance. The research project Digidow aims to develop decentralized digital identity systems for the physical world, with the digital wallet being a crucial component of this system. This thesis assesses the current state of protection capabilities on Android smartphones and aims to define a pathway for implementing a secure digital identity wallet. The research involves redefining the requirements for and threats to a digital identity wallet, analyzing best practice advice and theoretical capabilities, dissecting actual wallets to understand their implementation, and refining the list of theoretical capabilities based on the evaluation. The findings of this research could potentially contribute to the development of more secure digital identity wallets and enhance the overall security of digital identification systems.

Kurzfassung

Diese Masterarbeit untersucht die Machbarkeit und Sicherheitsaspekte der Implementierung einer digitalen Identitätswallet auf Android-Smartphones. Mit der zunehmenden Bedeutung von digitalen Wallets in verschiedenen Bereichen ist die Sicherheit dieser Wallets, die sensible Daten speichern und verwalten, von größter Bedeutung. Das Forschungsprojekt Digidow zielt darauf ab, dezentrale digitale Identifikationssysteme für die physische Welt zu entwickeln, wobei die digitale Wallet eine entscheidende Komponente dieses Systems ist. Diese Arbeit evaluiert den aktuellen Stand der Sicherheitsfunktionen auf Android-Smartphones und zielt darauf ab, einen Weg zur Implementierung einer sicheren digitalen Identitätswallet zu definieren. Die Forschung beinhaltet die Neudefinition der Anforderungen an und Bedrohungen für eine digitale Identitätswallet, die Analyse von Best-Practice-Ratschlägen und theoretischen Fähigkeiten, die Dekompilierung und Auswertung tatsächlicher Wallets, um deren Implementierung zu verstehen, und die Verfeinerung der Liste der theoretischen Fähigkeiten auf der Grundlage der Bewertung. Die Ergebnisse dieser Forschung könnten potenziell zur Entwicklung sichererer digitaler Identitätswallets beitragen und die allgemeine Sicherheit digitaler Identifikationssysteme verbessern.

Acknowledgements

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World and has partially been supported by ONCE (FFG grant FO999887054 in the program "IKT der Zukunft") and the LIT Secure and Correct Systems Lab. We gratefully acknowledge financial support by the Austrian Federal Ministry of Labour and Economy (BMAW), the Austrian Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, Österreichische Staatsdruckerei GmbH, and the State of Upper Austria.

Contents

List of Acronyms

API	Application Programming Interface
APK	Android Application Package
AOSP	Android Open Source Project
ART	Android Runtime
CRUD	Create, Read, Update, Delete
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DAC	Discretionary Access Control
DAO	Data Access Object
DEX	Dalvik Executable
DID	Decentralized Identifier
eIDAS	electronic Identification, Authentication and Trust Services
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JAR	Java Archive
JDK	Java Development Kit
JSON	JavaScript Object Notation
JSON-LD	JSON for Linking Data
JRE	Java Runtime Environment
JVM	Java Virtual Machine
NFC	Near Field Communication
NPM	Node Package Manager
OEM	Original Equipment Manufacturer
OS	Operating System
OTP	One Time Password
PIA	Personal Identity Agent
PIN	Personal Identification Number
ROM	Read-Only Memory
SDK	Software Development Kit
SQL	Structured Query Language
SSI	Self-Sovereign Identity
SSL	Secure Sockets Layer

TEE	Trusted Execution Environment
TLS	Transport Layer Security
UID	Unique Identifier
UI	User Interface
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
UX	User Experience
VPN	Virtual Private Network
W3C	World Wide Web Consortium
WORA	Write Once, Run Anywhere
XML	Extensible Markup Language
ZKP	Zero-knowledge proofs

Chapter 1

Introduction

Due to their convenience, digital wallets are enhancing or even replacing physical wallets in many aspects of our lives, encompassing various domains such as digital driver's licenses, mobile banking and payments, event tickets or cryptocurrency wallets. With the increasing prominence of digital wallets, the question arises:

Is it even possible to develop a secure Digital Wallet for a common mobile phone?

The security of these wallets is of utmost importance, as they are used to store and manage sensitive data, but proofing that such a system is secure is a difficult to impossible task.

1.1 Larger Project Context

Digidow[1] is a research project that aims to develop decentralized digital identity systems for the physical world. The ideal outcome of this project would be a system that allows users to securely identify themselves in physical scenarios, without any form of physical identity document or trusted mobile device, and without privacy concerns [87].

The wallet is part of the PIA, the Personal Identity Agent, which is a software that runs either on a device owned by the user or on a device owned by a trusted third party. The PIA is responsible for storing and managing the user's identity documents, as well as providing them to verifiers upon verification request. In the case of a smartphone, the PIA would be an app installed on the user's device, and the wallet would be a part of this app. This is only possible if the smartphone is able to securely store the user's identity documents and provide them to the PIA upon request.

1.2 Objectives and Approach

With this thesis, we aim to explore the feasibility of implementing a secure digital identity wallet on Android smartphones.

More specifically, we aim to

- define the requirements for and threats to a digital identity wallet,
- assess the current state of protection capabilities on Android smartphones,
- dissect actual wallets to understand their implementation and how they use Android's capabilities, and

[1] https://digidow.eu/

- define a list of theoretical capabilities and best-practice usage based on the evaluation.

The steps to achieve this goal are:

1. Define the requirements for and threats to a digital identity wallet.
2. Assess the current state of platform security and privacy capabilities usable for digital wallets on Android smartphones by
   a) analyzing and categorizing best practice advice as theoretical capabilities,
   b) dissecting actual wallets to find out if and how they implement these capabilities,
   c) comparing the theoretical and actual capabilities, and
   d) refining the list of theoretical capabilities and best-practice usage based on the evaluation.

The process of discovering and evaluating security and privacy capabilities is repeated until a satisfactory level of detail is reached.

Chapter 2

Background and Related Work

2.1 Digital Wallets and Data Storage Applications

Digital wallets are applications that securely store blocks of data. The proposed use case, security level, and implementation effort of such wallets vary greatly, and this can also be seen in the number of existing applications.

2.1.1 Cryptocurrency Wallets

Cryptocurrency wallets are used to store cryptocurrency, such as Bitcoin, Ethereum, or Dogecoin. They usually only store the private key of the user, which is used to sign transactions. The public key is derived from the private key and can be used to receive cryptocurrency. The challenge with cryptocurrency wallets is that the private key needs to be stored securely, as it is the only way to access the cryptocurrency.

2.1.2 Payment Wallets

Apple Pay

Apple Pay[1] is Apple's mobile payment service. Users can add their credit or debit cards to the wallet and use their Apple device to make payments at contactless terminals. The data is stored and managed by the Apple Wallet app.

Google Pay

Google Pay[2] is a wallet platform by Google that allows users to make payments using their Android devices. It is integrated into the Google Wallet app, which also allows users to store credit cards, concert tickets, boarding passes, or loyalty cards.

2.1.3 Ticket Wallets

Ticket wallets are used to store tickets for events, such as concerts, movies, or public transport. Their functionality usually consists of a way to import tickets, store them securely, and display them when needed.

[1] https://www.apple.com/at/apple-pay
[2] https://pay.google.com/intl/de_de/about

2.1.4 Identity Wallets

When it comes to identity wallets, there are two main approaches: the centralized approach and the decentralized approach.

In the centralized approach, a central authority handles the management of the digital identity. When a user requests verification for a service, that request is sent to the central authority, which confirms the user's identity before sending the confirmation to the service provider.

In the decentralized method, it is up to the user to manage their identity wallet and confirm their identity. As a result, the user doesn't need a centralized authority to access services. A user usually feels more secure using a decentralized approach because they have control over their identity and no central authority can access it. To the service provider, however, the decentralized approach is less secure because the service provider cannot easily trust users are who they claim to be.

Apple Wallet

Apple started rolling out the support for digital identity documents in the Wallet app in selected US states in 2022. Users can add their ID to the wallet by downloading it from the state's website, optionally also adding a selfie or pictures of the front and back of the physical ID to verify the identity of the user. At the time of writing, only US driver's licenses and state IDs are supported, and the places where they can be used are limited to certain airports and TSA checkpoints.

Google Wallet

Similar to Apple, Google started rolling out support for digital identity documents in the Google Pay app in selected US states in 2023. The implementation depends on Googles IdentityCredential API, which is a part of the Android Jetpack Security library. At the time of writing, similar to Apple, only US driver's licenses and state IDs are supported, and the places where they can be used are limited to certain airports and TSA checkpoints.

ONCE Wallet

The ONCE Wallet is a decentralized identity wallet concept developed by the ONCE project [45]. It is meant to hold digital identity documents, such as driver's licenses, passports, or vaccination certificates. The ecosystem consists of multiple independent systems, such as the wallet, the lifecycle management system or a central ID gateway. By the end of the project, the concepts could not be fully implemented.

Digidow PIA

The Digidow PIA is meant to be, among other things, a storage for digital identity related documents. It is meant to use these documents dynamically for authentication and authorization, and to be able to selectively disclose parts of the documents to third parties. The PIA is meant to be used in a decentralized way, where the user has full control over their data, and the issuer of the data is not involved in the authentication process.

2.2 Android Architecture

The following section should give a brief overview of the Android architecture and the security features that are built into the system. The contents are based on the Android Open Source Project [24], GeeksforGeeks [69], Mayrhofer et al. [99], and Rohan Vaidya [124]. Figure 2.1 shows the elements of each layer of the Android software stack.

2.2.1 Linux Kernel

The Linux Kernel, the foundation of the Android platform, provides a level of abstraction between the device hardware and the rest of the software stack. In Android, the Linux Kernel is modified to include custom libraries and APIs that are not present in the standard Linux Kernel.

2.2.2 Hardware Abstraction Layer (HAL)

The Hardware Abstraction Layer (HAL) in Android provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework. The HAL allows for the encapsulation of vendor-specific implementations, ensuring that Android applications and frameworks can interact with hardware drivers in a uniform manner without needing to know the details of each hardware configuration.

2.2.3 Android Runtime (ART)

The Android Runtime (ART) is the managed runtime used by applications and some system services on Android, that enables developers to write Android apps using the Java programming language. Android applications are compiled into native machine code during installation, which is then executed by the ART. ART superseded the Dalvik Virtual Machine[3], which was used in earlier versions of Android, but still supports the Dalvik Executable (.dex) format.

2.2.4 Libraries and Android Framework

The Libraries and Android Framework layer provides many services including window management, view system, resource management, and lifecycle management. Noteworthy libraries include WebKit, which is used for rendering web content; SQLite, a lightweight database engine for data storage; OpenGL, which provides APIs for 2D and 3D graphics rendering; and libc, the standard C library offering a range of system functionalities.

2.2.5 Application Layer

The Application Layer is where applications run. Each application runs in its own process within its own instance of the Android Runtime (ART). This layer includes the applications that ship with the device, as well as third-party applications that are installed by the user.

[3]https://source.android.com/docs/core/runtime

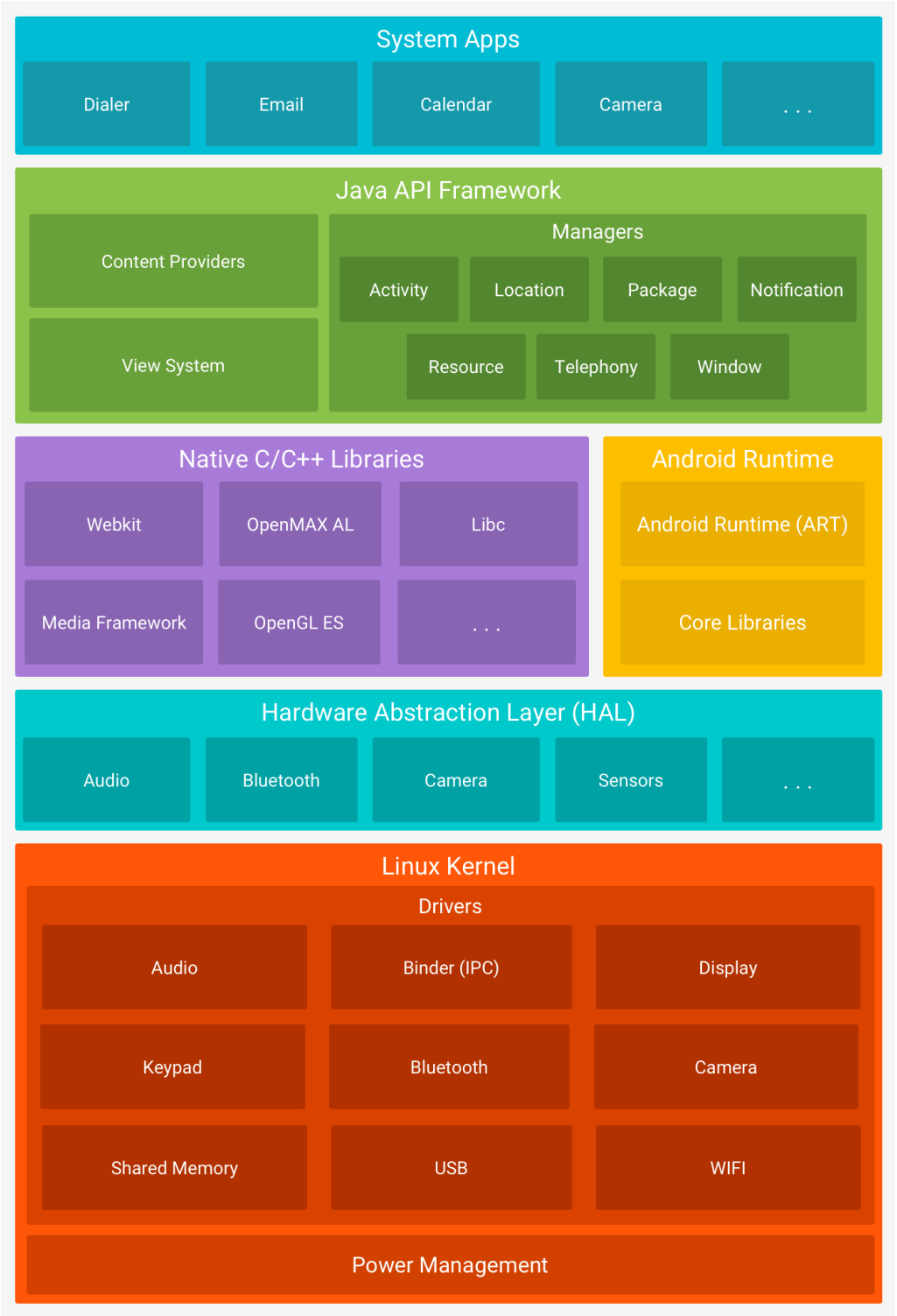

Figure 2.1: The Android software stack. (Source: [24], licensed under CC BY 2.5)

2.2.6 Security Features in Android Architecture

Android's security model is based on sandboxing on as many layers as possible, and on the principle of least privilege. Using verified boot, file-based encryption, and hardware-backed security, Android ensures that the device is secure from the moment it boots up, and via KeyStore it provides a secure way to attest the state of the device at runtime.

2.2.7 Inter-Process Communication (IPC) and Intent Messaging

Interprocess Communication (IPC) on Android allows applications and components to communicate with each other and share data in a secure and efficient manner. Android provides several mechanisms for IPC, including the use of Binder, which is the underlying framework facilitating direct method calls between processes. Through Binder, developers can implement services that can be accessed from other applications. Intents are another IPC mechanism, enabling communication between components like activities and services. Content Providers allow for the sharing of structured data between applications, while Messenger and Broadcast Receivers facilitate message passing and broadcasting system-wide events, respectively.

2.3 Used Programming Languages/Technologies

2.3.1 Java Programming Language

Java[4] is an object-oriented, class-based, and concurrent programming language [114]. The Language is meant to be platform-independent, and is compiled to Java bytecode, which is then run on the Java Virtual Machine (JVM), which is platform-dependent.

In the context of Android, Java is the main programming language used to write Android applications. It is used to write the business logic of the application, and to access the Android SDK, which is written for Java. Android applications are compiled to Dalvik bytecode, which is then run on the ART.

2.3.2 Kotlin Programming Language

Kotlin[5] is a JVM-based programming language developed by JetBrains, the company behind the IntelliJ IDEA IDE (and in turn, also Android Studio, which is a fork of that) [89]. Since Kotlin also runs on the JVM, it is fully interoperable with Java, can be used in the same project as Java, and Java code can be called from Kotlin, and vice versa.

Over the years, Kotlin has become the preferred language for Android development, and is now the official language for Android development.

[4] https://www.oracle.com/java/technologies/introduction-to-java.html
[5] https://kotlinlang.org/

2.3.3 Java Class File

The .class file is the output of the Java compiler, and contains lower-level code that can be run on the JVM [115]. It defines the constants, the access flags, the fully qualified name of the class, the fully qualified name of the superclass, the interfaces that the class implements, the fields, and the methods of the class.

2.3.4 Java Archive

The .jar file is a specialized .zip archive that contains the .class files needed in an application or library, and a manifest file that describes the contents of the archive [110, 117]. It is the main way to distribute Java applications, and can be run on any platform that supports a JVM if the version of the JVM is compatible with the version of the .class files in the .jar archive.

2.3.5 Android Package

An .apk file format is a modified .jar file that contains the compiled Android application, a manifest file, resources like images, sounds, and layouts and signatures that are used to verify the integrity of the application [117]. There are multiple versions of the .apk file format, which are explained in section 4.10.

2.3.6 Application Bundles

The most recent app format on Android is the Android App Bundle format[6], which is a more efficient way to distribute Android applications, as it only contains the resources that are needed for the target device [71]. These app bundles are split into multiple .apk files, and on the target device, only the .apk files that are needed are downloaded and installed.

2.3.7 Dalvik Executable

These files are the Android equivalent of .class files, and contain the compiled application code [117]. In the actual implementation, the JVM-compatible source code is compiled into larger .dex files, which contain the compiled code for the entire application. There might be multiple .dex files in an .apk though, as there is a method limit of 65,536 methods per .dex file.

2.3.8 Smali Assembly Language

Smali[7] is an assembly language for the Dalvik Virtual Machine and ART [88]. It is a common output format for Android decompilers, as it is a human-readable representation of the .dex code. It can be used to decompile and recompile Android applications, and to modify functionality, but it is not the most user-friendly way to do so.

[6] https://developer.android.com/guide/components/fundamentals
[7] https://github.com/JesusFreke/smali/wiki

2.3.9 Javascript Programming Language

JavaScript is an interpreted functional programming language that conforms to the ECMAScript language specification [106]. JavaScript is mainly used to write web applications, but there are also implementations of JavaScript runtimes for other platforms, like Node.js, which runs on Googles Chromium engine, and can be used to write server-side applications. There are also implementations of JavaScript runtimes for mobile platforms, which can be used to write mobile applications. These applications often run in a native webview, which is an element that can render web content.

2.3.10 Dart Programming Language

Dart[8] is a programming language developed by Google, and is mainly used to write web applications using the Flutter framework[73]. It is a JavaScript-like language that looks like a more organic implementation of React Native, with a syntax that can natively provide a similar functionality as React Native.

2.3.11 App Development Frameworks

Third-party frameworks are often used to simplify the development of apps, and to provide a unified API for multiple platforms. They are often open-source, and can be audited by the community, but they can also introduce new security vulnerabilities, and can be hard to maintain, since they are often developed by volunteers. There are two main types of frameworks: those that compile to native code, and hybrid apps that run in a virtual machine, usually a JavaScript engine in a native webview element. This section lists some of the most popular frameworks and their security implications. Generally, it can be said that more high-level frameworks make using secure best practices easier, but they often lack the flexibility to easily implement custom lower level security measures.

Flutter

Flutter[9] is a cross-platform app development toolkit developed by Google [75]. It compiles to platform-specific code and can be used to build apps that run on iOS, Android, Windows, macOS, Linux, and the web, with the main target being mobile apps.

In the case of Android, Flutter apps are compiled to native ARM code, which means that the result of a compilation is a libflutter.so file, that contains the compiled flutter engine, and a libapp.so file, that contains the compiled Dart code (see Figure 2.2 for a more detailed overview). There is also a Java wrapper that is used to start the Flutter engine, and to provide an interface between the Flutter app and platform-specific libraries.

Reverse engineering a Flutter app is not trivial [7, 43], but it is possible, especially since Android-specific code is written in Java or Kotlin, and can be decompiled using existing tools.

[8] https://dart.dev/
[9] https://flutter.dev/

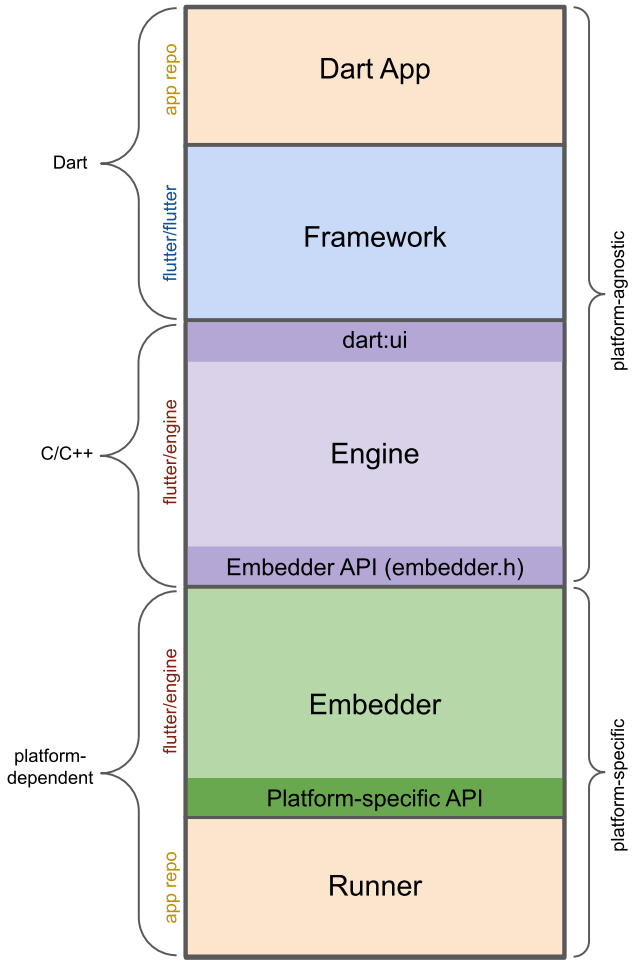

Figure 2.2: A stylized diagram of the elements present in a Flutter application (Source: [74], licensed under CC BY 4.0)

React Native

React Native[10] is a cross-platform app development toolkit developed by Facebook. It allows developers to abstract the UI and parts of the business logic of their app as a single codebase, and only write platform-specific code when necessary.

React Native is interesting, as the development is made easier by its ability to leverage the vast JavaScript ecosystem, including npm packages. Moreover, React Native supports hot reloading, which means developers can see the changes they make in real-time without having to rebuild the application.

A React Native app is run in two separate threads, one that interacts with the host UI, and one that runs the JavaScript code. Platform-specific code can be written in Java or Kotlin for Android and in Objective-C or Swift for iOS, and can be called from the JavaScript code using a bridge. This native code is also always run in a single thread, unless the code itself spawns new threads. This is particularly useful when the developer needs to optimize certain aspects of the application, access platform-specific APIs, or write high-performance, multithreaded code. In the case of a digital identity wallet, the access to Keystore

[10] https://reactnative.dev/

functionality, especially Keystore attestation, would have to be implemented manually for each platform.

Despite its many advantages, React Native also presents certain challenges. For instance, achieving high performance for complex animations can be challenging due to the asynchronous nature of the React Native bridge, which can lead to delays. Reverse engineering a React Native app is made more difficult by the fact that the engine is compiled to native code, which leads to many more files that need to be decompiled to get a full picture of the app. Furthermore, as React Native is still evolving, developers may encounter issues related to compatibility and stability.

Given the complexity of the build process, it is not trivial to set up a reproducible build environment, but since Node.js apps can be built reproducibly, it is possible to build pure React Native apps reproducibly.

Xamarin / .NET MAUI

Xamarin[11] / .NET MAUI (Multi-platform App UI)[12] is a cross-platform app development framework developed by Microsoft. Similar to other cross-platform technologies, it allows developers to write a single codebase, this time in C# and .NET, and compile it into native applications for iOS and Android.

Microsoft provides a set of libraries that abstract the platform-specific APIs, and allows developers to access them from their shared codebase. The code is then compiled into a native executable, and can be run on the target platform via an added Mono runtime, which is shipped with every app.

If the developer needs to access platform-specific APIs, they can do so by writing platform-specific code, and calling it from the shared codebase via JNI or Objective Sharpie [116].

Apache Cordova

Apache Cordova[13] (formerly known as Adobe PhoneGap[14]) is an open-source platform that enables developers to build mobile applications using web technologies. It acts as a bridge between the web application and the native device APIs, allowing developers to access device APIs like the camera, contacts, and geolocation, while the interface is rendered in a native WebView (see Figure 2.3 for a more detailed overview).

Cordova plugins are used to access native device features from the web application. They are usually wrappers for native code that is executed in the native runtime, and can be accessed from the web application using JavaScript. Cordova itself comes with a set of core plugins that are included in every Cordova app, and there are a lot of third-party plugins that can be used to access more advanced device features, like the fingerprint sensor, or the NFC reader.

An example of a third-party plugin is the cordova-plugin-fingerprint-aio [107], which is a wrapper for the Android Fingerprint API and the iOS equivalent, that allows developers to use the fingerprint sensor from their Cordova apps.

[11] https://dotnet.microsoft.com/en-us/apps/xamarin
[12] https://dotnet.microsoft.com/en-us/apps/maui
[13] https://cordova.apache.org/
[14] https://cordova.apache.org/announcements/2020/08/14/goodbye-phonegap.html

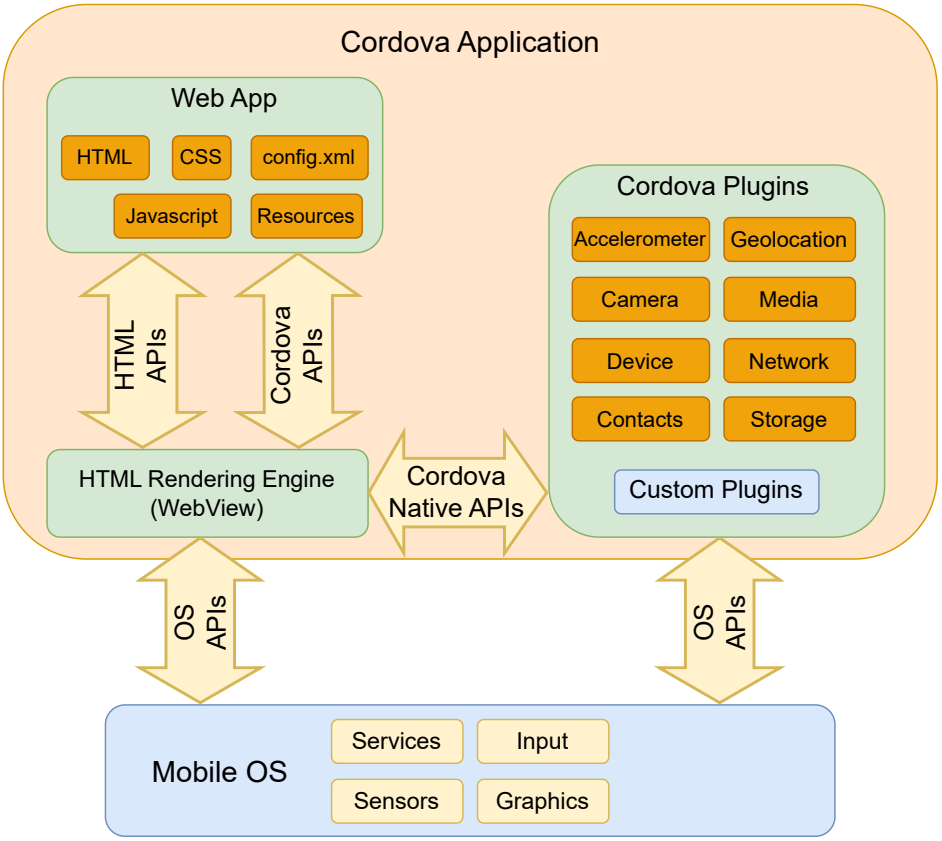

Figure 2.3: A stylized diagram of the Cordova application architecture [134]

Capacitor by Ionic

Ionic[15] is a framework of UI components for web apps that are optimized for mobile devices. Ionic Native (nowadays Awesome Cordova Plugins [56]) is a set of wrappers for Cordova plugins that allows developers to access native device features from their Ionic apps. Capacitor is a cross-platform runtime that allows developers to build web apps that run natively on iOS, Android, Electron, and the web as Progressive Web Apps (PWAs).

Ionic started as a set of AngularJS components for Cordova, until the team behind it decided to create their own runtime, called Capacitor, which is now the default runtime for Ionic apps. Ionic and the Capacitor runtime are open-source, and can be audited by the community, but they are mainly developed by engineers at Ionic, Inc.

Kivy

Kivy[16] is a niche framework for building cross-platform apps using Python [92]. Kivy, similar to Ionic, is a collection of multiple related Projects, such as Kivy, python-for-android, Buildozer, and PyJNIus. Kivy is the frontend framework, python-for-android is a toolchain for building Android apps using

[15] https://ionicframework.com/
[16] https://kivy.org/

Python, Buildozer is a tool for automating the build process, and PyJNIus is a library for accessing Java classes from Python. The interesting part is PyJNIus, which allows developers to access the Android SDK from Python, which means that it is possible to use the Android SDK to implement security measures, and to use the Android Keystore to store cryptographic keys, but in reality this just adds extra steps to the usage, since no specific libraries exist that abstract exiting security measures, and the developer has to implement everything manually. This is only useful if a large Python codebase already exists, and the developer wants to port it to Android, but even then there is a high probability that a rewrite of the code in Java or Kotlin is easier.

2.4 Hyperledger Project

The Hyperledger Project[17], supported by the Linux Foundation, is a collective open-source initiative aimed at promoting the development and application of blockchain technologies across various industries [67]. It is the umbrella for many open source blockchain and distributed ledger technologies, including Hyperledger Fabric, Hyperledger Indy, Hyperledger Aries, and Hyperledger Ursa. There are also a number of other projects and systems that depend on and sometimes are closely intertwined with these projects, such as the Sovrin Network.

These blockchain technologies are used in digital identity context to provide a decentralized, secure, and verifiable way to manage digital identities. This is achieved by storing identity data and/or issuer verification data on a blockchain to make it immutable and by providing ecosystems to surround this functionality.

- Hyperledger Fabric[18] is a modular blockchain framework that separates roles between nodes, runs smart contracts (chaincode), and allows for customizable consensus and membership services.
- Hyperledger Indy[19] is a distributed ledger designed for decentralized identity.
- Hyperledger Aries[20] is a toolkit for building blockchain-based identity management solutions.
- Hyperledger Ursa[21] is a shared cryptographic library, providing a modular, flexible, and standardized cryptographic library that can be used in a variety of distributed ledger and identity systems.

[17] https://www.hyperledger.org/
[18] https://www.hyperledger.org/projects/fabric
[19] https://www.hyperledger.org/projects/hyperledger-indy
[20] https://www.hyperledger.org/projects/aries
[21] https://www.hyperledger.org/blog/2018/12/04/welcome-hyperledger-ursa

Chapter 3

Requirements and Threats

These requirements and threats are based on the considerations of the W3C Verifiable Credentials Data Model [130], and contain further refinements based on the evaluation of digital identity wallets.

There are also several other sources that provide insights into the requirements and threats of digital identity wallets. A comprehensive overview of usable security is presented by Theofanos [137], while a metastudy on usable privacy and security is explored by Distler et al. [58]. Implementation decisions concerning OpenID and Verifiable Credentials (VC) are discussed by Hoops and Matthes [80], alongside explanations of DIDs and VC use cases by Mazzocca et al. [100]. Additionally, considerations regarding data exchange in the context of an eIDAS PoC are outlined by Gonçalves et al. [70], while privacy threats within the eIDAS framework are examined by Kutylowski et al. [94]. A resource discussing digital identities is provided in [96], and insights into usable digital wallets are presented by Krauß et al. [93].

3.1 Requirements

The following requirements are split into three generic parties, Identity Holder, Identity Verifier, and Identity Issuer, which all have their respective role in the W3C Verifiable Credentials data model [130], as each party has different requirements for the Digital Identity Wallet.

3.1.1 Affected Party: Identity Holder

As the owner of the identity, the user is the most affected party.

The wallet should be designed to minimize complexity, minimize loading times, and avoid other inconveniences for the user. The user should be able to use the wallet without having to learn a new interface or a new way of thinking. In the worst case, a user of a Digital Identity Wallet also has to face the consequences of a compromised identity, which should be avoided at all costs.

- **User-friendly design**
  The wallet should be user-friendly and intuitive, with clear and concise instructions provided to guide users through the process of creating and managing their digital identities.
- **Privacy protection**
  The wallet should be designed to protect the privacy and confidentiality of the users' personal information.

- **Multi-layered security**
  The wallet should provide multiple layers of security to prevent unauthorized access or modification. Neither a phone thief nor a hacker should be able to access or use the users' digital identities.
- **Scalability**
  The wallet should be designed to be scalable and flexible, able to accommodate a large and growing number of digital identities and adapt to evolving technology and regulatory requirements.
- **Availability**
  The wallet should provide the digital identities which are always available, even in the case of system outages or other disruptions.
- **Recovery**
  The wallet should provide reliable backup and recovery options, allowing users to restore their digital identities in case of device loss, theft, or other incidents.
- **Non-discriminatory**
  The wallet should be designed to be non-discriminatory, ensuring that all users have equal access to the service.

3.1.2 Affected Party: Identity Verifier

A Verifier is any entity responsible for the verification of an identity.

The following user stories for digital identity wallets were constructed from the aforementioned sources, Domeyer et al. [61], European Commission [63] and PwC Luxembourg [121].

- Airport security personnel may require a mobile identity wallet that can quickly and securely verify the identity of travelers. Travelers may come from different jurisdictions with different issuing authorities, travel for an extended period of time, and may have different requirements for entry and exit visas. In addition to visa requirements, there may be other entry requirements such as proof of vaccination, negative COVID-19 test results, or travel insurance required. The provided identity may need to contain information such as name, date of birth, address, photo or other biometric identification as well as potentially contain information about the traveler's routes. The provided identity must also not be invalidated by the time of verification.
- Police officers may require a mobile identity wallet that can quickly and accurately verify a persons' identity. The provided identity may need to contain information such as name, date of birth, address, photo identification and potentially biometric information. The provided identity may also need to integrate with law enforcement databases to provide additional information on criminal records or outstanding warrants.
- Banks may require a mobile identity wallet that can verify the identity of customers for opening accounts, applying for loans, or conducting financial transactions. The provided identity may need to contain information such as name, address, or photo identification. The provided identity may also need to integrate with banking systems to verify the customer's credit history and financial standing.
- Government agencies may require a mobile identity wallet that can verify the identity of citizens for access to government services or benefits. The

provided identity may need to contain information such as name, date of birth, or photo identification. The provided identity may also need to integrate with government databases to verify citizenship or immigration status.

- Employers or human resources personnel may desire a mobile identity wallet that can verify the identity of job applicants or employees for background checks, employment verification, or access to company facilities. The provided identity may need to contain information such as name, date of birth, or photo identification. The provided identity (or an additional identities) may also be able to provide employment history or educational background.
- Bouncers may require a mobile identity wallet that can quickly verify the age and identity of patrons entering a venue. The provided identity may need to contain information such as name, to check for house bans, date of birth, and photo identification, or a similar biometric check, as well as be able to quickly and easily verify the authenticity of the ID. A bouncer will probably only need to verify that a person is over a certain age, and not need to know the exact date of birth.
- Ticket inspectors or event organizers may require a mobile identity wallet that can verify the identity and ticket information of attendees entering an event. The provided identity may need to contain information such as name and photo identification, as well as be able to quickly and easily verify the authenticity and uniqueness of the ticket. The provided identity (or additional identities) may also be able to provide ticket information, such as the event name, date, location, ticket type, and if the ticket has been used before.
- Online service providers may require a mobile identity wallet that can verify the identity of users for account creation or access to sensitive information. The provided identity may need to contain information such as name, address, e-mail, and photo identification. The provided identity also needs to integrate with online systems for verification purposes such as two-factor authentication [44, 85]. Most online services, however, should not require a full digital identity, and instead require unique pseudonymous identities, as this enhances privacy and reduces the risk of data breaches.

These individual requirements can thus be summarized into the following general requirements for a gatekeeping verifier:

- **Identity verification**
  The mobile identity wallet should provide a secure and reliable method for the gatekeeper to verify the identity of the user.
- **Ease of use**
  The mobile identity wallet should be user-friendly, making it easy for the gatekeeper to verify the user's identity quickly and efficiently.
- **Privacy protection compliance**
  The mobile identity wallet should ensure the privacy of the user's personal information. The gatekeeper should only be able to access the necessary information for identity verification purposes, and no other personal data should be disclosed without the user's consent as this could interfere with the user's privacy rights.
- **Data integrity**
  The mobile identity wallet should ensure the integrity of the user's personal data. The gatekeeper should be confident that the information presented to them is accurate and has not been tampered with.

- **Interoperability**
  The mobile identity wallet should be interoperable across various gatekeepers, platforms, and organizations, ensuring that the user's identity can be verified seamlessly across different contexts.
- **Handle multiple identity related documents**
  The mobile identity wallet should be able to provide multiple identity related documents and information, such as:
  - Biometric data (fingerprints, face, etc.)
  - Identifying data (name, date of birth, etc.)
  - Vehicle permissions (car, airplane, etc.)
  - Tickets (concert, train, etc.)
  - Educational information (school, university, etc.)

3.1.3 Affected Party: Identity Issuer

An identity provider, in this context, is a third party that provides the user with a digital identity. This can be a government agency, a bank, or any other entity that is trusted by the user and the gatekeeper to issue a digital identity. These authorities also face risks when issuing identities, as they are responsible for the availability of the service they provide. For example, if a user relies on digital identities, and the system is not fully functional, the user might not be legally able to enter a country. The documents they authored need to be always available, correct, and up to date. This means a wallet should be able to:

- **Accurate information / Data integrity**
  Ensure that the digital identities are up-to-date and accurately reflect the information provided by the identity provider.
- **Provisioning**
  Provide a means for the identity provider to issue digital identities to users.
- **Revocation**
  Provide a means for the identity provider to revoke or invalidate a digital identity if necessary.
- **Access control / User authorization**
  Provide appropriate access controls to ensure that only authorized parties can access the assigned digital identities.
- **Ensure trust**
  The identity provider needs a wallet to protect the security of the digital identities they issue, as users and verifiers only trust and use services that they know are secure. This includes the protection of the digital identities from unauthorized use.

3.2 Threats

As a general starting point, the inverse of the requirements from section 3.1 can be used to identify threats, as non-fulfillment of a requirement is a threat to the security, the functionality, the usability and in general the acceptance of a system.

User-friendly design

T.1 Malicious actors may create phishing wallets with similar designs to deceive users and steal their digital identities [49, 68].

UI imitation/phishing attacks are a common threat to any system that is used by a large number of people. The wallet's design can and will be imitated by malicious actors to deceive users and steal their digital identities. In most cases, this attack will fail, but attackers only need a small percentage of users to fall for the attack to make it profitable. Especially with AI-based UI generation [1], it is becoming increasingly difficult to distinguish between a real and a fake UI.

T.2 The wallet's design may not be intuitive enough, leading to user errors that compromise the security of their digital identities [68].

- Users may accidentally share their digital identities with unauthorized parties. In our case, this could be a malicious sensor that might or might not be known to be malicious.
- Users may accidentally share more information than intended or necessary with other parties. In our case, this could mean that a user might share a full facial embedding with a sensor, while a multi-party computation would be sufficient.
- Users may misuse the wallet's features, leading to deanonymization or other privacy violations. In our case, this could mean that a user might share a full digital identity with a sensor that is not allowed to have certain information.

Privacy protection

T.3 The wallet's privacy protection measures may be insufficient, making it easier for hackers to access and steal user's personal information [2].

In our case, this includes the network traffic, the data stored on the device, and the data shared with other parties. This is an implementation issue for both the wallet and the overall system, as the wallet might be forced to allow privacy violations due to the system's design.

Holder verification

T.4 On the device, the method used to verify the identity of the user may be vulnerable, allowing malicious actors to impersonate legitimate users and gain unauthorized access to their digital identities [140].

Multi-layered security

T.5 Malicious actors could exploit weaknesses in any layer of security to gain unauthorized access to user's digital identities [108].

This includes the physical security of the device, the security of the wallet, and the security of the overall system. We can only effectively tackle security threats in our own implementations, but we can still be affected by security breaches in other parts of the system. Within the wallet it is also

important to note that the wallet is only as secure as its weakest link, so any security measure that is not fully implemented or not fully functional is a potential threat or overlooks a potential threat.

T.6 A security breach could compromise multiple layers of security, making it easier for hackers to steal user's digital identities [108].

This is especially relevant for attacks on the system the wallet is running on, because issues within elements that multiple layers of security rely on can lead to a broad compromise of the wallet.

T.7 The wallet's security measures may be too complex or time-consuming, discouraging users from using the wallet and potentially reducing its security [137].

This is a common issue with security measures, as users will often try to bypass them if they are too complex or time-consuming.

T.8 A thief might try to bypass security measures by using brute force attacks, social engineering, or other methods [54].

It boils down to the weakest link argument, and too similar layers of security. If, for example, the wallet is secured by two factors, a fingerprint and a local authenticator app, any person with a stored fingerprint to unlock the device can bypass the wallet's entire security. This means the layers of security need to be different enough to not be bypassed by the same attack.

Scalability

T.9 The wallet may not be scalable enough to handle a large number of digital identities, leading to slow performance and reduced security [48].

This is a common issue with any system that is not designed to be scalable. In our case, slow performance can lead to users not using the wallet, users trying to bypass security measures, or users trying to speed up the process by using unsafe methods, such as sharing their digital identities with other parties or using third party tools/add-ons.

T.10 The wallet may not be able to adapt to evolving technology and regulatory requirements, making it obsolete and potentially insecure [48].

In any case, the wallet needs to be able to adapt to new technology and regulatory requirements. This is especially important for a wallet that is used for digital identities, as the requirements for digital identities are constantly changing. Scalability is affected by the ability to adapt to new requirements, as the wallet needs to be able to handle new types of digital identities, new security measures, or new privacy protection measures ideally without a complete overhaul of the system.

Availability

T.11 System outages or other disruptions could make user's digital identities unavailable, potentially causing inconveniences, financial losses or even health risks [95].

In time critical scenarios, such as public transport, the availability of the digital identities is crucial. If the wallet is unable to provide the digital

identities when needed, or if the response time is too slow, users might be unable to enter trains, buses, or other public transport on time or at all.

Similarly, in health care, the availability of patient data is crucial. Here, patients might be unable to receive necessary treatment in time or at all.

Recovery

T.12 Backup and recovery options may be insufficient or unreliable, leading to loss of user's digital identities and potentially causing inconveniences, financial losses or even health risks [142, 143].

Users are expected to rely on the wallet to store their digital identities, and if the wallet is unable to provide a reliable backup and recovery option, users might lose access to their digital identities. Recovery options are important in the case of a lost, stolen or damaged device, as users should be able to continue using their digital identities without any issues. If a recovery option is not available, users might be unable to access critical resources, such as health care, financial services, or travel documents.

T.13 Hackers could exploit recovery options to gain unauthorized access to user's digital identities [139].

Recovery options are a potential security risk, as they are often used to bypass security measures. The existence of a recovery option implies that there is data stored somewhere that can be used to recover the digital identities, and this data is a potential target for hackers. This means if the recovery option is not as secure as the wallet itself, it will be a prime target for hackers.

Identity verification

T.14 The method the gatekeepers use to verify the identity of the user may be vulnerable, allowing malicious actors to impersonate legitimate users and gain their privileges [140].

Ease of use

T.15 The identity verification process may be too complicated or time-consuming, discouraging gatekeepers from verifying identities correctly (as observed with COVID-19 Certificates[1]) [60].

This overlaps with the user-friendly design and the multi-layered security threats, but here the focus is broader, as the wallet ecosystem in its entirety needs to be easy to use, not just specific components. The gatekeeper needs to be able to verify the identity of the user quickly and efficiently, and if the process is too complicated or time-consuming, the gatekeeper might try to bypass the process.

[1]https://twitter.com/fabian_schmid/status/1443193052521275400/photo/1

Privacy protection compliance

T.16 The Gatekeeper may misuse or mishandle user's personal information, leading to reputational damage for the identity provider [91].

This is a potential threat for the identity provider, as they are responsible for the privacy of the user's data. Users rely on the identity provider to protect their privacy, and if the identity provider fails to do so, the user might lose trust in the identity provider and the wallet.

T.17 The wallet may fail to comply with privacy protection laws and regulations, violating their privacy rights and potentially leading to legal liability [84].

Legal compliance is the very basis of a digital identity system, as the system needs to comply with the laws and regulations of the jurisdictions it operates in to be legally valid. If the wallet fails to comply with privacy protection laws and regulations, the identity provider might face legal liability. In the worst case, certain elements of the system might be banned from certain jurisdictions, making the system unusable for a large number of users.

Interoperability

T.18 Incompatible systems or protocols could result in errors or delays in the identity verification process, making it difficult or impossible for users to access resources [60].

All the elements in the system need to be able to communicate with each other, and if they are unable to do so, the system will not work properly. This entails the need for proper standards and protocols, as well as the need for proper error handling and recovery options. The standards need to define exact APIs and data formats, potentially including fallback options for older versions.

T.19 Sharing personal data across multiple platforms could result in information leakage or the creation of a larger attack surface for hackers [96].

Interoperability brings a large attack surface, as every implementation of a standard is a potential target for hackers. This means that the components need to trust each other whilst at the same time not knowing each other, which is a challenging task.

Provide multiple identity related documents

T.20 Featuring documents from more providers could create a larger attack surface for hackers [6, 66].

Combining multiple credentials from different providers into a single wallet increases the attack surface, as every provider is a potential target for hackers. The system therefore needs to make sure that identities from different providers can only be issued by their respective providers, and leaked data from one provider does not affect the other providers.

T.21 The wallet could be hacked, resulting in the theft of user data and the exposure of larger amounts of sensitive information [6].

If multiple documents or credentials are stored in a single wallet, this wallet becomes a single point of failure for all the documents and credentials.

T.22 If a wallet features wrong documents from problematic providers, all parties may lose trust in the wallet and the correct identities [68, 79].

If the wallet features documents from problematic providers, the entire wallet might be considered untrustworthy, and all the documents and credentials in the wallet might be considered untrustworthy. This could lead to a loss of trust in the wallet and the digital identity system as a whole.

Accurate information

T.23 Users may use or select incorrect or outdated information, resulting in errors or delays in the identity verification process [59].

A digital identity system must be able to handle outdated, incorrect or invalid data, as there will be such occurrences in any system.

T.24 Employees or contractors with access to the identity provider could modify user data [8].

The authorized personnel of the identity provider are a potential threat to the system, as they have access to the data and could modify it. Completely preventing this is impossible, as the identity provider needs to be able to modify the data, but the system needs to be able to detect modifications and prevent unauthorized changes.

Revocation

T.25 Revocation requests may not be processed in a timely manner, allowing unauthorized access to resources [86].

Revocation usually has a time delay, as the revocation request needs to be processed by the identity provider. This delay is a potential threat, as any revoked identity is still valid until the revocation request is processed, and there is usually a reason behind the revocation.

T.26 Employees or contractors with access to the wallet could revoke digital identities without proper authorization, resulting in the denial of legitimate user access [8].

The revocation process needs to be secure, as any unauthorized revocation could lead to the denial of legitimate user access. Employees or contractors with access to the wallet are a potential threat, as they can potentially revoke digital identities without proper authorization.

Ensure trust

T.27 If the mobile identity wallet fails to provide adequate security measures, users may lose trust in the identity provider, leading to decreased usage and adoption of the digital identity service [59, 79].

The usage of a digital identity system is directly linked to the trust in the system, and if the system fails to provide adequate security measures, users might lose trust in the system. This includes the security of the wallet, the security of the overall system, and the security of the identity provider.

Chapter 4

Security Measures in Android

This chapter will delve into Android's security measures, encompassing features integrated into the operating system as well as those available for incorporation within apps. These measures aim to safeguard user data and privacy effectively.

4.1 Tapjacking / UI-Redressing

A user interface element on Android can theoretically be layered on top of another element, which can be used to trick the user into clicking on elements in ways that are not intended by app developers or users. This is called *tapjacking*. Tapjacking has been an issue on many frontend platforms, and has been used to trick users into clicking on ads, installing malware, or even giving permissions to apps. Android classifies multiple types of tapjacking attack vectors, which are described in the following sections [32].

4.1.1 Full Occlusion

A full occlusion attack is when a malicious app overlays a transparent button on top of a legitimate button, which can be used to trick the user into clicking on the malicious button (see figure 4.1). This attack vector is mitigated by Android by setting the *android:filterTouchesWhenObscured* attribute to *true*, which prevents the malicious app from receiving touch events when the legitimate button is being touched.

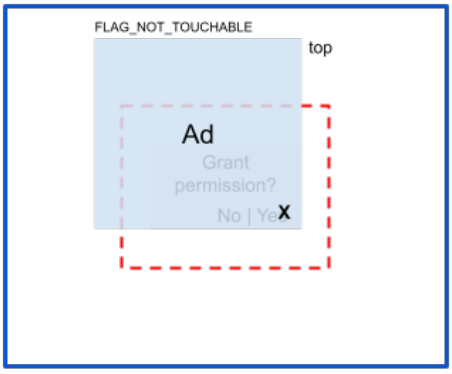

Figure 4.1: A full occlusion attack. The malicious app overlays a transparent button on top of a legitimate button, which can be used to trick the user into clicking on the malicious button (Source: [32], licensed under CC BY 2.5)

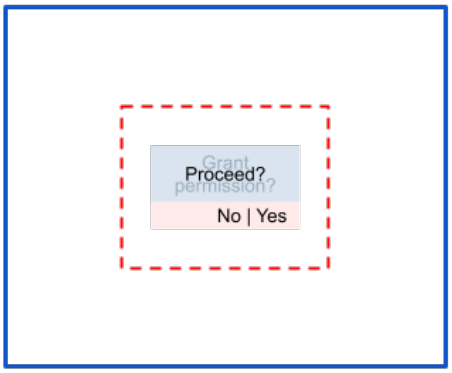

Figure 4.2: A partial occlusion attack. The malicious app overlays fake information over a legitimate interface (Source: [32], licensed under CC BY 2.5)

On Android 12, API 31, and higher this behavior is enforced by default, except for System Alert Windows and window animations, where only touches from layers with opacity greater than 0.8 are blocked, as this might negatively impact the user experience.

4.1.2 Partial Occlusion

In partial occlusion attacks only parts of an interface are covered by a malicious app, which can be used to trick the user into clicking on/submitting information to fields they did not intend to (see figure 4.2).

A developer can mitigate this attack vector by manually ignoring touch events that have the *FLAG_WINDOW_IS_PARTIALLY_OBSCURED* flag set.

4.1.3 Custom Toasts

A Toast message is a small pop-up that appears on the screen for a short period of time, typically to display a message to the user. On Android API 29 and lower it was possible to create a custom Toast message, set its appearance with *Toast.setView()* and display it on top of other apps while the app itself was running in the background.

4.1.4 Notification Bubbles

Even though these types of interface elements overlay other apps, they are not considered to be a tapjacking attack vector, as they are integrated with the Android notification system, together with its chat features. They might overlay other apps, and even overlay permission request dialogs, but they are in their own little sandbox, unable to influence or get information from anything outside their sandbox, not even their own absolute coordinates. The only way the process in a bubble can interact with the rest of the system is through shortcuts the parent app provides.

4.1.5 System Alert Windows

System Alert Windows are a special type of window that can be displayed on top of other apps. This was used by apps like Facebook Messenger to display chat heads, which were small bubbles that could be used to interact with the chat without having to open the app. Apps have to request the *SYSTEM_ALERT_WINDOW* permission to be able to display System Alert Windows, which from Android 6.0 (API 23) onwards was granted by default on apps that were installed from the Play Store, but had to be granted manually for apps that were installed from other sources.

4.1.6 Cross-Application Embedding

Since Android API 33 it is possible to embed an Activity from one app into another app, to allow for a more seamless user experience on devices with multiple or larger screens. This feature has to be explicitly enabled by the developer, and the developer can also specify which apps are allowed to embed their activities by specifying the *android:knownActivityEmbeddingCerts* attribute in the *<activity>* tag in the manifest file.

There is also the possibility to set *android:allowUntrustedActivityEmbedding* to *true*, which allows any app to embed the activity, but this is an obvious security risk, and should only be used if the developer is sure that the activity does not contain any sensitive information.

The Android Jetpack library also provides the *ActivityEmbeddingController#isActivityEmbedded(android.app.Activity)* method, which can be used to check if an activity is being embedded by another app.

The security measures are in place to prevent clickjacking and other UI-redressing attacks by ensuring that only trusted hosts can embed and control the presentation of activities, thereby protecting sensitive information, UI controls, and input fields from malicious misuse.

4.2 Sandboxing

Android application sandboxes are protected on a Linux kernel level, where each app has its own user, with its own UID, and runs in its own process.

These features underwent gradual hardening over Android version updates by implementing more SELinux policies, by limiting the filesystem and the process permissions, and by limiting the access to the underlying system, like with syscalls.

The following sections describe the most important changes to the Android sandboxing system up to the time of writing. The changes are paraphrased from the Android Source Docs, and are grouped by API level [14].

4.2.1 Android 5.0 - API 21

SELinux enforces mandatory access control (MAC) to separate the system from apps, but all third-party apps run within the same SELinux context, so inter-app isolation is mainly achieved through UID-based DAC.

4.2.2 Android 6.0 - API 23

The SELinux sandbox was expanded to separate apps based on physical user boundaries. Additionally, Android implemented safer default settings for app data by changing the default DAC permissions on an app's home directory from 751 (owner: read, write, execute; group: read, execute; others: execute) to 700 (owner: read, write, execute; group: no permissions; others: no permissions) for apps with a targetSdkVersion of 24 or higher. This improved the security of private app data, although apps could still choose to override these defaults.

4.2.3 Android 8.0 - API 26

A seccomp-bpf [136] filter was implemented for all apps, which restricted the number of syscalls that apps could use, thus enhancing the security at the app/kernel boundary.

4.2.4 Android 9.0 - API 28

Apps that do not have privileged access and have a targetSdkVersion of 28 or higher must operate within their own SELinux sandboxes, which provide mandatory access control (MAC) on an app-by-app basis. This enhances app isolation, prevents the alteration of secure defaults, and most importantly, prevents apps from making their data globally accessible.

4.2.5 Android 10.0 - API 29

Apps have restricted access to the filesystem and cannot directly access certain paths, such as /sdcard/DCIM. However, they do have full access to paths specific to their own package, as provided by methods like Context.getExternalFilesDir().

4.3 KeyStore System

In Android, key material should never enter the app process. During cryptographic operations in an Android app, the plaintext and messages are transmitted to a system process responsible for executing the cryptographic procedures. This process can even be supported by specific hardware like a physically isolated TEE module. On a somewhat current phone it can be safely assumed a trusted environment with hardware support is existent, but it can be checked with the method *KeyInfo.isInsideSecurityHardware()* [10].

Again, the following sections describe the most important changes to the Android sandboxing system up to the time of writing. The changes are paraphrased from the Android Source Docs, and are grouped by API level [10].

4.3.1 Android 4.0 - API 14

The KeyStore system was introduced in Android 4.0 (API level 14) and is based on the Java KeyStore API.

4.3.2 Android 4.3 - API 18

The KeyStore system was enhanced to support hardware-backed keys using the KeyStore provider.

4.3.3 Android 7.0 - API 24

Since there was no way to check if a key was hardware-backed, key attestation (section 4.11.5) was introduced. This allows remote servers to verify the state of the key material.

4.3.4 Android 10 - API 29

The *KeyInfo* class now provides information about the security level of the key, such as if it is hardware-backed, and if so, if it is StrongBox backed.

4.4 Cryptography

Android provides a standard Java Cryptography Architecture implementation, which is based on the Bouncy Castle library [15]. There is hardware support for the following cryptographic objects: Ciphers, Message Digests, Message Authentication Codes (MAC) and Signature.

4.5 Biometrics

The following sections are based on the Android developer training materials for biometric authentication [28].

4.5.1 APIs

android.hardware:FingerprintManager

The FingerprintManager was the first API to be introduced to support generic biometric authentication. It was introduced in Android 6.0 (API level 23) and was deprecated in Android 9.0 (API level 28).

It enabled the authentication against the KeyStore system, and similar to the more modern APIs, you could specify a CryptoObject, which would be unlocked if the authentication was successful. The biggest drawback of this API was that it only supported fingerprint authentication, and developers had to implement their own user interface.

android.hardware.biometrics:BiometricPrompt

To address the drawbacks of the FingerprintManager, the BiometricPrompt was introduced in Android 9.0 (API level 28). It is a system-provided dialog that prompts the user to authenticate using biometric authentication. The OEM is able to customize the dialog to support different biometric modalities, like fingerprint, iris, or face, and to better incorporate in-display fingerprint sensors.

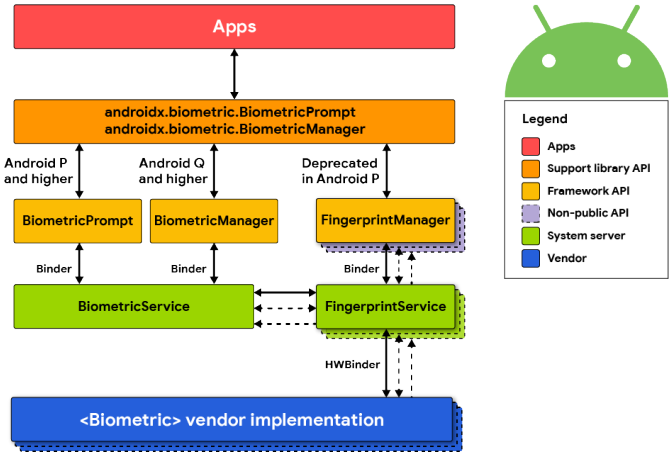

Figure 4.3: How the different fingerprint APIs are related (Source: [17], licensed under CC BY 4.0)

androidx.biometric:BiometricManager

The androidx.biometric library was introduced in Android 10.0 (API level 29) and provides a single API to authenticate a user with biometrics. It is a wrapper around the existing biometric authentication APIs, and provides a simple interface to authenticate a user with biometrics [55]. It also provides a way to check if the device supports biometric authentication, and if so, what type of biometric authentication. For apps that support Android 10.0 (API level 29) and above, the androidx.biometric library uses android.hardware.biometrics. For apps that support API 28 to API 23, the androidx.biometric library uses android.hardware.FingerprintManager, and for apps on API 22 and 21 it uses the Confirm Credential API, which is a system-provided dialog that prompts the user to authenticate using a device credential (PIN, pattern, or password).

These different implementations shouldn't matter to a developer, as a properly implemented app authenticates the credential with the KeyStore system, which is the same for all APIs (see figure 4.3). Similar to this, the access permissions to hardware-backed keys are also the same for all APIs, which means even though a device without biometric sensors might support Confirm Credential, the hardware-backed keys can never show to be unlocked by biometrics.

4.5.2 Usage

The management class typically requires a PromptInfo parameter, and optionally a CryptoObject parameter, which undergoes authentication. The PromptInfo contains information about the prompt, such as the displayed text and a callback function to be executed once the user has interacted with the prompt. The CryptoObject can be accessed from the callback function, where it is unlocked if the biometric prompt meets the cryptographic key requirements. These key requirements are set during key generation, and can be viewed in the key attestation certificate, which can be retrieved with KeyAttestation.

4.5.3 Security

To ensure compatibility with Android, device implementations must adhere to the Android Compatibility Definition Document (CDD) [9], which evaluates

biometric security through assessments of architectural security and spoofability [21]. The metrics used, including Spoof Acceptance Rate (SAR), Imposter Acceptance Rate (IAR), and False Acceptance Rate (FAR), help classify biometric implementations into three classes (Class 3, Class 2, and Class 1) based on their security performance [21].

4.6 Databases

4.6.1 android.database.sqlite

This is the standard Android database API, which is based on the SQLite database engine [26]. The versions used in the Android system are from 3.4.0 (API 1), released 2007-06-18 [132], currently up to 3.32.2 (API 31), released 2020-06-04 [131]. The versioning is usually the same across devices in the same Android version, but manufactures can choose to include a different version of SQLite in their shipped OS.

4.6.2 androidx.room

The Room Persistence Library is a wrapper around the standard SQLite API [25]. It provides compile-time checks for SQL statements, generates boilerplate code for common database operations and features standardized support for database migrations. It uses

- DAOs to access the database, which are annotated with SQL statements, for CRUD operations,
- Data entities to represent tables in the database, which are annotated with the table name and column names, and
- Database classes to represent the database, which are annotated with the list of entities and the database version.

4.7 Identity Credential API

To aid the development of apps that use identity credentials Google is developing an API that allows apps to interact with identity credentials securely. In August 2020 the first alpha version of the Jetpack Identity Credential API was released [20].

4.8 Network

In an Android app, network communication is usually done using the URLConnection class, which is a standard Java class. This class wraps around different protocols, such as HTTP, HTTPS, FTP, and can be used to send and receive data. This way the app developer doesn't have to implement any part of the protocol itself, but can use the standard Java API to communicate with a server.

The Android API implements Certificate Pinning by default for all HTTPS connections [27]. Even manually added certificates are not trusted by the system by

default, which means the app has to implement its own TrustManager to trust a non-standard certificate. This makes it harder for an attacker to intercept the communication between the app and the server, as the attacker has to either compromise the server or the device to be able to intercept the communication, and can't just use a self-signed certificate that is installed as a user-certificate in the system.

4.9 Permissions

Permissions expand the functionality that an app is allowed through the Android sandbox [22]. In general, a default Android app can only do computations, draw on the screen, and read and write to its own private storage. To extend the capabilities of an app, it can request permissions to access certain resources, such as the network, sensors or elements of the OS. Some permissions are granted implicitly, some are granted explicitly by the user, and some OS and device functionality is locked away from apps completely.

4.9.1 Normal Permissions

Normal permissions are permissions that, according to the Android development team, don't pose a risk to the user's privacy or the device's operation. They are automatically granted to the app when it requests them, so the user doesn't have to explicitly grant them.

4.9.2 Signature Permissions

Signature permissions are permissions that are only granted to apps that are signed with the same certificate as the app or the Android system that declared the permission.

4.9.3 Dangerous Permissions

Dangerous permissions, or Runtime permissions, are permissions that can pose a risk to the user's privacy or the device's operation. Before an app wants to use functionality that requires a dangerous permission, it has to request the permission from the user, which is done via the Android permission prompt, which the app can call, but not influence in any way. The user can then choose to grant or deny the permission, which means the app has to account for the permission being denied.

4.9.4 Special Permissions

Special permissions are permissions that are granted by the user, but can only be granted through the system settings, and not through the Android permission prompt.

4.9.5 Recent Changes

The Android permission system has evolved quite a lot over the years, with the goal of giving users more control over their data and privacy [5]. This also meant that the permission system has become more complex, which can lead to users not understanding the permission system and just blindly granting permissions to apps.

Since Android 11, the user can now choose to only grant the permission for the current session for certain permissions related to location, microphone and camera. Additionally, if the user hasn't used an app for a few months, the system will automatically reset all sensitive runtime permissions for that app [23].

In Android 12, a privacy dashboard was added to Android, which gives the user an overview of which apps have accessed which permissions, when, and even gives developers a possibility to show an app specific rationale screen when a user wants to know more about the need for certain permissions [16]. Android 12 also added microphone and camera indicators, which show the user when an app is using the microphone or camera [16].

4.10 App Signing

For an app to be installed on an Android device, it has to be signed with a certificate by the developer [30].

Since an Android app is stored as an APK file, which is based on the JAR file format, which in turn is a ZIP archive, an app can be unzipped, and the files can be modified. To prevent this, the APK file is signed with a certificate, which is then verified by the Android system when the app is installed (see figure 4.4). This certificate is used to identify the developer of the app, and to verify that the app hasn't been tampered with.

4.10.1 APK Signature Scheme v1

The APK Signing Scheme v1 is based on the JAR Signing Scheme [112], which is used to sign JAR files [113]. The *jarsigner* tool [111] is used to sign each file in the APK individually, and then the signature is stored in the *META-INF* folder of the APK.

4.10.2 APK Signature Scheme v2

The APK Signing Scheme v2 adds a signature block in front of the ZIP central directory, which contains signed digest of digests of 1MB chunks of the ZIP archive and signatures for each signer (see figure 4.5). This removes some of the flaws of v1, such as the fact that only indexed files were signed, and that all contained files had to be uncompressed before verification [11].

4.10.3 APK Signature Scheme v3

This version is similar to v2, but it adds information on the target SDK version of the app and a proof-of-rotation structure [12].

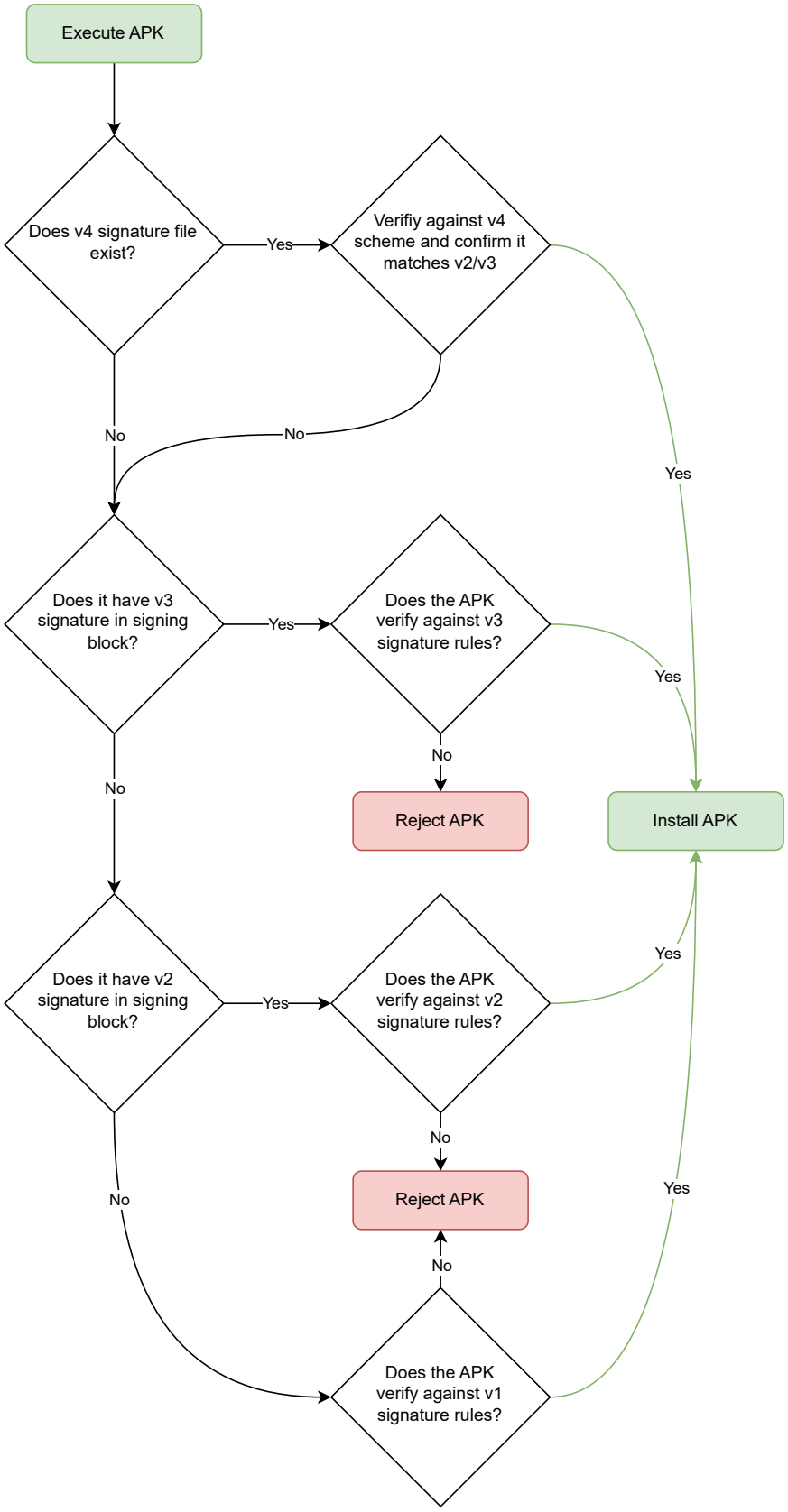

Figure 4.4: APK Signature Verification Mechanism (Source: [30])

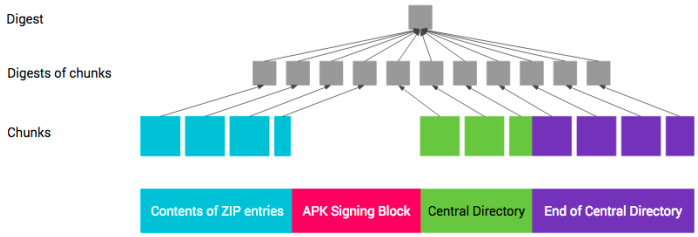

Figure 4.5: How signing with APK Signing Scheme v2 creates the digest for the ZIP archive (Source: [11], licensed under CC BY 4.0)

This proof-of-rotation can be used to prove that the signing key of the app has been rotated, which can be used to migrate to a new signing key without having to re-sign all APKs that have been signed with the old key.

4.10.4 APK Signature Scheme v4

The v4 signature scheme also generates hashes over the ZIP archive, but instead of storing them in the APK, it stores them in a separate file [13]. For the generation of the hashes, the fs-verity merkle tree algorithm is used, which gives a standard way to verify the integrity of files on a block device [135].

The v4 signature scheme requires a v2 or v3 signature to be present in the APK, so it can't be used on its own.

4.10.5 APK Signature Issues and Vulnerabilities

The Janus vulnerability (CVE-2017-13156) was a huge issue with the APK Signing Scheme v1, where an attacker could append malicious code to the end of the APK, which would be executed by the system, but not be detected by the signature verification [77]. This is possible due to a combination of factors.

1. The APK Signing Scheme v1 signs the files in a ZIP folder, but there could be arbitrarily large gaps between the files, which are not signed.
   This situation is even worse when you consider that such a gap could also be in the beginning of the APK, which would allow an attacker to prepend data to the APK.
2. The Android runtime can execute both DEX and APK files, and it determines the type of the file based on the magic bytes at the beginning of the file.

This allows for a malicious actor to prepend a malicious DEX file to the APK, which would still be signed by the original author of the app, but would be executed by the system. The vulnerability is not present in the APK Signing Scheme v2, since the signature digest is computed over the entirety of the APK data.

Another issue with multiple versions of increasing security is rollback attacks, where an attacker tries to trick the system into using a more unsafe (older) version of signatures, which can be used to exploit vulnerabilities in older versions of the signature schemes. This is mitigated by including the preferred version of the signature scheme in the older version of signatures.

4.11 Device Protection

The topic of device protection is very broad, and includes many different security measures. Since an Android phone is based on a Linux kernel, it inherits many of the security features of Linux. Usually, modern devices also feature a Trusted Execution Environment (TEE), which runs its own operating system, potentially using a separate processor, and is used to store sensitive data, such as cryptographic keys, and to perform cryptographic operations.

4.11.1 SELinux

SELinux is a Linux kernel security module that provides a mechanism for supporting access control security policies. It is used to enforce the principle of least privilege, by restricting the access of processes to resources, such as files, devices, networks, etc. In a standard Android device, SELinux is set to *enforcing* mode, which means that all access requests are denied by default, and only allowed if they are explicitly allowed by the policy.

4.11.2 Verified Boot

Verified boot is a security feature that ensures that the device boots using only trusted software. This is done by verifying the signature of the bootloader, kernel, and the system partition.

There are 5 different states that the device can be in:

- **Green**: The device is locked, and the OS is verified using a root-of-trust from the OEM.
- **Yellow**: The device is locked, and the OS is verified using a custom root-of-trust that is also set in the device by a user.
- **Orange**: The device is unlocked, and any OS can be booted.
- **Red (eio)**: The device is unlocked, and the OS seems to be corrupted according to dm-verity [19].
- **Red (no OS)**: The device is unlocked, and there was no bootable OS found.

These states can be attested by the TEE (see section 4.11.5), and this way a remote server can somewhat reliably determine the state of a device.

4.11.3 File-Based Encryption/Full-Disk Encryption

File-based encryption is a feature that was introduced in Android 7.0 (API level 24) [18]. Android encrypts the data partition using a key that is stored in the TEE, and the key is encrypted using a key derived from the user's lock screen PIN, pattern, or password. This means that the data partition can only be decrypted if the user enters their PIN, pattern, or password, which is verified by the TEE.

4.11.4 Operating System Modifications

OEMs usually modify the Android operating system to add their own features and to differentiate their products from the competition. These modifications, for the most part, don't affect the security of the device.

There are, however, some groups of developers that provide modified versions of Android, for the purpose of adding features that are not present in the official OEM version of Android on a certain device. These modified versions of Android are called custom ROMs, and they are usually based on AOSP.

Custom ROMs usually require the bootloader to be unlocked, which allows the user to boot a non-OEM signed OS image. They can be used to bypass security measures, and the OEM can't prevent them from doing so. This is why it is important to have security measures that are not affected by the OS.

There are some parts of a device that are usually not designed to be modifiable, even when a bootloader is unlocked, specifically TEEs. This means that even if the OS is modified, the TEE can still be used to store sensitive data, and to perform cryptographic operations.

A whole different topic is the security of custom ROMs themselves. Since they are mostly developed by volunteers, there is no guarantee that they are secure, and that the code is audited. Some groups try to provide additional code security, for example strict coding guidelines, and code reviews, but in the end, it is up to the user to decide if they trust the authors of a custom ROM. A similar argument could be made for the official OEM versions of Android, especially since they are usually closed-source.

4.11.5 Key Attestation

An Android device can, using the aforementioned cryptographic libraries (see section 4.4 and section 4.5), generate cryptographic keys, and store them in the TEE. This key material can then be used to perform cryptographic operations, such as encrypting data, or signing messages. The TEE can also be used to attest the key material, which means that it can generate a certificate that contains information about the key, such as the key's purpose, its protection level and custom attestation challenge data to guarantee that no replay attacks are possible. This certificate is the last in a chain of certificates, which starts with the root certificate, which is stored in and cannot be extracted from the TEE.

This way the TEE can be used to attest the state of the device, and to prove that the device is in a certain state, such as the bootloader being locked, or the device being encrypted. It can also be used to attest the state of the key material, and to prove that the key material is stored in the TEE, and whether it is only accessible after the user has authenticated themselves with a certain biometric modality, PIN, pattern, or password.

4.11.6 Root Detection

The term *root* refers to the user with the highest level of privileges on a Linux system. A *rooted* device is a device that has been modified to allow the user to gain root privileges. Rooted devices provide users with administrative access to their devices, allowing them to modify the operating system and use otherwise

restricted features. This is often used to bypass security measures, such as the ones described in this document.

Detecting if a device is rooted is not a trivial task, since there are many different ways to root a device, and they all have different effects on the system.

The modern way, as implemented by Magisk [90] to root a device is to modify the boot image, which is the first image that is executed by the bootloader, and edit the zygote process, which is the parent process of all Android applications. This way the necessary libraries are loaded into the zygote process, and all applications that are started from that process will have root access, without the need to modify the system partition. Magisk also provides a way to deny root access to certain applications, where it cleans the environment of the application before it is started. Such a setup alone makes it tough to detect if a device is rooted, since no additional libraries are in the environment and the zygote process looks completely normal.

Additionally, since the user has root access, they can modify an app dynamically, by injecting code into the process, and thus bypassing any root detection implemented in the app. This leads to the conclusion that it is not possible to reliably detect if a device is rooted, and that it is not possible to prevent a user from using a rooted device to bypass security measures [47, 64].

4.11.7 Google Security APIs

Google provides a set of APIs to detect certain features of the device, and to perform certain security-related operations. These APIs are not part of the Android Open Source Project, and are only available on devices that have Google Play Services installed.

SafetyNet Attestation API

The SafetyNet API consists of multiple parts, such as:

- **SafetyNet Safe Browsing API**: Allows the app to check if a URL is safe to open.
- **SafetyNet reCAPTCHA API**: Allows the app to use reCAPTCHA to verify that the user is human.
- **SafetyNet Verify Apps API**: Allows the app to check if the device is protected by Google Play Protect.
- **SafetyNet Attestation API**: Allows the app to check if the device integrity has been compromised.

In our case, the SafetyNet Attestation API is the most interesting one, since it allows the app to check if the device's integrity has been compromised. This is done by evaluating the runtime environment of the device and certain device identifiers, sending them to the Google servers, together with a nonce, where they are evaluated, and a response is sent back to the device.

This process can be somewhat easily tricked, since good root hiding techniques can be used to hide the fact that the device is rooted, and the device identifiers can be spoofed. This means that the SafetyNet Attestation API can't be used to reliably detect if a device is rooted.

Play Integrity API

The Play Integrity API is the successor of the SafetyNet Attestation API, and is set to replace it completely by 2025. This API, similar to the SafetyNet Attestation API, also transmits an attested KeyStore certificate chain to the Google servers, which can be used to verify the boot state of the device.

This means that, as long as the TEE displays the correct boot state (which is not always the case [127]), the Play Integrity API can be used to (somewhat) reliably detect if a device could be modified to bypass security measures, but there are still no absolute guarantees.

4.12 Third-Party Libraries

4.12.1 SQLCipher

SQLCipher[1] is a widely recognized open-source fork of SQLite, providing transparent 256-bit AES encryption of database files [31]. On Android, it can be used to secure Room databases or SQLite databases. When integrated with Room Databases, SQLCipher adds a layer of security by encrypting the database content. This is particularly useful when storing sensitive data, as it ensures that even if the database file is somehow accessed, the content remains secure and unreadable without the encryption key. SQLCipher performs encryption at a page level, meaning that data is encrypted before it's written to disk and decrypted when read into memory. The encryption key is required to initialize the database and can be changed without losing data. This makes SQLCipher a valuable tool for enhancing data security in Android applications.

4.12.2 Google Tink

Google Tink[2] is a multi-platform cryptographic library that aims to provide unified cryptographic APIs for developers. It is designed to implement key management using system-specific hardware keystores, enhancing the security of cryptographic operations. In the context of Android, Tink integrates with the Android KeyStore System through the AndroidKeysetManager class. This class handles the storage of the keyset in a SharedPreferences file and uses the Android KeyStore System to secure the master key. However, a notable concern with Google Tink is its deviation from the base functionality provided by Android, particularly its inability to require user authentication for key usage by default, and instead leaving this responsibility to the developer again. This could potentially limit its applicability in scenarios where user authentication is a crucial aspect of the security protocol.

4.12.3 Themis

Themis[3], developed by Cossack Labs Limited, is a high-level cryptographic services library designed to provide secure storage and transport of data across multiple platforms. It is designed to address 90% of typical data protection

[1] https://www.zetetic.net/sqlcipher
[2] https://developers.google.com/tink
[3] https://www.cossacklabs.com/themis

use cases, making it a versatile tool for app developers. Themis uses libcrypto from a variety of open source providers for its cryptographic operations, ensuring robust and secure data encryption. However, it leaves the responsibility of key management to the developer, allowing for greater flexibility and control. Themis is capable of encrypting stored secrets in apps and backends, supporting searchable encryption, maintaining real-time secure sessions, and even comparing secrets between parties without revealing them using zero-knowledge proofs.

Chapter 5

Security Measure Evaluation Criteria

The evaluation criteria are used to assess the security measures implemented in an app. The criteria are designed to help identify the strengths and weaknesses of an app's security measures, and to provide a structured way to objectively evaluate the app's security posture. The criteria are divided into different levels, with each level representing a different degree of security, which allows for easy and compact classification. The levels are based on the iterative evaluation process and roughly show the assumed capabilities of an attacker needed to bypass the security measures, from attackers with no special capabilities to nation-state actors. This, of course, does not include vulnerabilities to the security measures themselves, as these are not predictable and can be found at any level.

5.1 Storage Protection

These criteria are used to evaluate the protection of the storage of sensitive data. Sensitive data can be anything from a user's biometric, financial or other private data. It can also include login credentials or private API tokens that further unlock secrets a user might want to protect. The levels are based on the time and effort required for an attacker to access the data, assuming other security measures have already failed.

Level 0 - Unencrypted storage of sensitive data in publicly available folders

In this level, sensitive data is stored in publicly available folders that are accessible for every app on a device without any encryption.

Level 1 - Unencrypted storage of sensitive data or decryption keys in program private folders

In level 1, sensitive data or decryption keys are stored in program private folders without any encryption. While this offers some protection compared to level 0, it is still vulnerable to attacks by malware or malicious apps that can gain access to these folders.

These risks are somewhat mitigated by the Android OS, but especially on modified devices, these mitigations are easily circumvented.

Level 2 - Encrypted storage of sensitive data in publicly available or program private folders

In level 2, the focus is on ensuring that sensitive data is stored securely in both publicly available and program private folders by encrypting the data. This adds an extra layer of protection to the data, making it more difficult to access it as an unauthorized party. It is important to note that while encryption can help protect data, it's not foolproof, and relies heavily on the integrity of the underlying KeyStore and encryption algorithms used.

Level 3 - Biometrically encrypted storage of sensitive data in publicly available or program private folders

In level 3, sensitive data is biometrically encrypted, which means that the data is not only encrypted but also requires biometric authentication to access it. This provides a higher level of security compared to level 2 as biometric authentication ensures that only authorized users can access the data, and user presence is required to decrypt the data. However, it is important to note that biometric authentication methods are not foolproof and can be bypassed.

Level 4 - Distributed, encrypted storage of sensitive data in physically different machines

In level 4, sensitive data is stored on physically different machines in a distributed manner. This provides the highest level of protection for sensitive data as it ensures that even if one machine is compromised, the sensitive data is still safe on the other machines. The data is also encrypted to ensure that even if a machine is compromised, the sensitive data cannot be accessed without the encryption key.

Distributed, encrypted storage is particularly useful for organizations that deal with highly sensitive data, such as financial institutions, healthcare organizations, and government agencies.

5.2 Network Communication Protection

Level 0 - Plain text

At this level, network communication is done in plain text, which means that the data transmitted over the network is not encrypted or scrambled. This makes it straightforward for an attacker to intercept and view the data being transmitted.

Level 1 - Bad encryption - e.g. key publicly available

Level 1 is an improvement over level 0 in that it introduces encryption. However, the encryption used is weak in any way, e.g. when the key is publicly available.

This makes it relatively easy for an attacker to intercept and view the encrypted data.

Level 2 - Encryption using state-of-the-art methods

In level 2, network communication is encrypted using state-of-the-art encryption methods, which are designed to be challenging to crack. This means that even if an attacker intercepts the data being transmitted, they will not be able to make sense of it without the encryption key.

Level 3 - End-to-end encryption using state-of-the-art methods

End-to-end encryption is a communication system where data is encrypted on one device and decrypted on another, so that the data is only readable by the sender and the intended recipient. Level 3 introduces end-to-end encryption using state-of-the-art encryption methods, which provides an additional layer of security. This is different from level 2, where the data is encrypted during transmission, but could be decrypted at the server.

Extra

Since the next levels are not necessarily better than the previous ones, they are listed as extra levels.

VPN - Usage of a VPN

Using a Virtual Private Network (VPN) is a way to add a layer of security to network communication. It can provide anonymity and help to protect against attacks. However, it's important to note that using a VPN does not necessarily improve security, and can sometimes create new vulnerabilities.

Offline - No network communication

The only 100% secure network connection is no connection. By disconnecting from the internet and other networks, data can be kept completely secure. This, however, is obviously only possible in apps that do not require a network connection.

5.3 User Interface Protection

Level 0 - No protection - can be embedded, used without user interaction

At this level, the app can behave like a library to other apps; it can be called and used without any user interface and is not confined in its sandbox.

This could mean that methods can be invoked from other apps, the app reacts to unverified deep links, or it exposes data using IPC mechanisms.

Level 1 - Stand alone - uses OS methods to be a standalone program - interaction without user presence possible

In level 1, the app is a standalone program that uses the operating system's methods to run independently of other programs. This means that the user interface can be used without the presence of an authenticated user, but it does not allow for any unauthorized actions to be performed from outside the app sandbox.

Level 2 - Uses confirmation of user presence

Level 2 introduces a higher level of protection for the user interface by requiring confirmation of user presence. This means that the user must be actively present and interact with the user interface to perform any actions. This can be achieved through various means, such as entering a password, using biometric authentication, or responding to a prompt.

However, it's important to note that this does not provide complete protection and should be combined with other security measures.

5.4 Permissions

Users should carefully review the permissions requested by an app before granting them because a misbehaving app is only capable of performing actions that align with the permissions granted to it. It is irrelevant whether such unwanted behavior stems from an attack or poor programming practices.

Level 0 - App wants ALL permissions possible

In level 0, an app requests all possible permissions, regardless of whether they are necessary for the app's functionality. This is a significant security risk, as it can allow the app to access sensitive data or perform unwanted actions on the device.

Level 1 - App wants more permissions than needed

In level 1, an app requests more permissions than are strictly necessary for its functionality. While this may not be as extreme as level 0, it can still pose a security risk.

Level 2 - App wants more permissions than needed for necessary functionality

In level 2, an app requests some additional permissions beyond those strictly necessary for its functionality. This may be done to enable additional features or improve the user experience. However, these additional permissions should be carefully reviewed to ensure that they do not pose a security risk.

Level 3 - App wants only those permissions needed for necessary functionality

In level 3, an app requests only the permissions that are strictly necessary for its functionality. This is the most secure approach to app permissions, as it minimizes the potential for security risks.

5.5 Update Policy

Level 0 - No updates/patches, even after publicized security issues

In level 0, an application does not get updated or patched, even after security issues are publicized. This is a significant security risk. Users should be cautious when using such applications.

There is also the chance that development for this application has finished, and there are no further updates possible or necessary, but in such cases the same caution should be applied by users, as there are constantly new threats being found.

The only real examples for apps that don't need updates would be those that display static information that is not going to change and not interacting with the user, like an app that displays the Declaration of Independence of the USA, and nothing else.

Level 1 - Irregular/incomplete updates/patches

In level 1, updates and patches are developed for the application, but they are irregular or incomplete. This means that the security of the application is improved periodically, but not necessarily in a timely or comprehensive manner.

Level 2 - Regular updates/patches

In level 2, updates and patches are regularly released for the application. This means that the security of the application is continuously improved, and any known vulnerabilities are addressed in a timely and comprehensive manner.

5.6 Root Detection

Extra - Additional functionality on rooted devices

In this extra level, additional functionality is provided to users who have rooted their devices. This is sometimes used as a powerful way to import data from other apps, like backups, or to provide additional features that are not possible on non-rooted devices. We need to declare this as an extra level, as it is not necessarily better or worse than the other levels, and it can be combined with some of the other levels.

Level 1 - No apparent root detection

In level 1, there is no clear root detection implemented in the app. This means that the app does not check or does not care if the device is rooted, and provides the same functionality to all users, regardless of whether their device is rooted or not.

Level 2 - Warning on rooted devices

At this level, the user gets notified that an app is aware of the root status of the device. This could be an alert to the user, a notification banner or anything else visibly notifying the user of the root status.

These messages usually contain some text on the possible risks of using the app on a rooted device, and disclaimers on the possible reduced stability due to the modifications that had to be made to obtain that status.

Level 3 - No functionality on rooted devices

In level 3, the app checks if the device is rooted, and if so, denies access to the app or at least certain features. This means that the app or those features cannot be used on rooted devices, which helps to prevent potential security risks associated with rooted devices.

However, this level of protection may not be foolproof, as there are ways to bypass root detection and gain access to the app on rooted devices. Additionally, this practice of denying access to rooted devices may also have unintended consequences. The (usually) small percentage of users who have rooted their devices will be unable to use the app, and some may resort to installing potentially dangerous software to bypass these checks.

5.7 Reproducibility

This criterion should answer whether a user can build their own version of the app that is the same as the version provided in app stores. This is important for ensuring the integrity and security of the app, as it allows users to verify that the code in the app has not been tampered with or modified. A reproducible build means that anyone can verify that the app was built from the same source code as the official app, providing greater confidence in the app's security and trustworthiness.

In our case, we are not looking to build a bit-by-bit identical copy, but a version that behaves exactly the same way as the official version. This leaves out things like timestamps, build numbers, and other metadata that can be different between builds, and signing keys, which should not be available to the user.

5.8 Business Logic in Native Code

Here, we are looking at whether the app has parts of its business logic written in native code. Having business logic written in native code can provide better performance and security for the app. However, it also means that the code is

more challenging to analyze. It can introduce additional security risks if the code is not properly audited, if there are vulnerabilities in the native libraries themselves or if the library developer itself is not trustworthy [4]. There are also potential issues with memory management, as native code can introduce memory leaks and other memory-related vulnerabilities. In this criterion, UI code or auxiliary libraries are not considered business logic, but differentiating them can be challenging, especially in closed-source apps.

Chapter 6

Test Methodology and Tools

An issue that we encountered was the fact that most published apps leave little to no traces of their implementation details, especially when it comes to libraries, because R8 [72], a compiler that can obfuscate and shrink Java bytecode, is included in Android Studio [29]. This shrinking removes unused code from libraries, and obfuscation renames classes, methods, and variables, which makes determining libraries and their versions difficult. Our research was aided by the fact that access to system APIs is usually not obfuscated, so we could at least determine which Android APIs were used by an app.

Most of our testing was done using static analysis, by decompiling the APK files of the wallets, and then analyzing the code.

6.1 Test Environment

The hardware used to test the wallets was a recent Laptop with an Intel Core i7-8565U CPU, 16GB of RAM and a 512GB NVMe SSD, running an up-to-date version of Ubuntu 22.04 LTS, together with 2 rooted Android smartphones, a Xiaomi Redmi Note 7 (lavender), running the PixelExperience 13 ROM and a Google Pixel 6a (bluejay), running the stock ROM, both on Android 13.

6.2 Toolchain

There are several tools that we used to test the wallets, and to analyze their code and behavior. Some of these tools provide redundant functionality and different tools can produce different results.

6.2.1 Raccoon

Raccoon [119] is a tool that allows to download APK files from Google Play Store, without the need to use the Google Play Store app. It uses a Google account to authenticate with the Play Store API and then allows to download the APK files of any app that is available in Play Store. This is useful for testing, as it allows downloading the most recent APK files of the wallets, without having to install them on the test devices, and without having to rely on 3rd party re-uploads. The tool itself is open source and can be found on GitHub. There are several issues with the tool, such as the fact that it does not work with 2FA enabled Google accounts, and the author recommends using a dedicated Google account for this purpose.

In our case, we used the tool to gather all the APK files of the wallets that we tested.

6.2.2 Apktool

Apktool [52] can be used to decompile APK files into more understandable formats, such as .smali for the compiled code, and it can also decompile the resources, such as images, layouts, etc. into their original formats. If the files are in the right configuration, it can also be used to recompile the APK files, which is useful for testing, as it allows modifying the code and resources of the apps, and then recompile them to test the changes.

6.2.3 Jadx

Jadx [129] can be used to decompile dex files into Java source code. It also has a GUI that displays the decompiled code nicely formatted, and allows for jumping to the definition of a class or method, which is useful for navigating the code.

There is no easy way to recompile the code, so it is only useful for viewing the code behind a program.

6.2.4 JD-GUI

JD-GUI [78] is similar to Jadx, but it can be used to decompile .class files into Java source code, and view the result.

6.2.5 Java Class File Editor

The Java Class File Editor can be used to view and edit .class files. It gives a great overview of the structure of the class files, and allows for byte-level understanding of the files.

Java Class File Editor [50] is an old tool, with its last release in 2004. Since it has not been updated for newer versions, it does not correctly decompile newer Java features.

6.2.6 dex2jar

dex2jar [122] is a collection of tools that can be used to convert dex files to zipped class files, dex files to smali files, and the other way around.

6.2.7 MITM Proxy

Man in the Middle Proxy [104] is a tool that can be used to intercept and modify network traffic. It works by setting up a middle point on a gateway device, and then intercepting all traffic that goes through it. If the traffic is HTTPS, a user needs to install a certificate on the device, that allows the proxy to decrypt the traffic. This does not work with apps that use certificate pinning, as they will refuse to connect to the proxy.

6.2.8 Xposed

The Xposed Framework [125] allows hooking into the execution of an app and to modify its behavior. It can be used to change the return value of a function, force an error, or to modify the parameters of a function call. This is useful for testing, as it allows modifying the behavior of an app, without having to modify the source code, and without having to recompile and re-sign the app. To use it, however, the app needs to be installed on a rooted device, and a fitting implementation of the Xposed Framework needs to be installed, which might get complicated, depending on the Android version and the device. In our case, LSposed [97] was used, which is an open source implementation of the Xposed Framework, that works as a Magisk module [97].

6.2.9 apk.sh

apk.sh [35] is a tool that can be used to decompile APK files. It is a wrapper around several other tools, such as Apktool, adb and jarsigner, and can be used to decompile, modify and recompile APK files. The most interesting feature of apk.sh is that it can be used to automatically collect application bundles from a local device, decompile them into the format that Apktool uses, and then recompile them into a single APK file. This way, the gathering of APK files does not involve unstable third-party Play Store scrapers, and modifications to application bundles are as easy as with regular APK files.

6.2.10 Mobile Security Framework (MobSF)

The Mobile Security Framework [105] is a tool that automatically decompiles and scans APK files for security issues. It is open source and uses some of the tools mentioned above to automatically decompile the APK files and then scan them for potential security or privacy issues, report native libraries, and give an overview of the permissions that the app requests. In our case, we used it to do initial scans of the APK files. This helped us to quickly identify potential issues, such as the use of insecure libraries, or the use of permissions that are not necessary for the app to function. The integration of the tracker database was also interesting, as it allowed us to quickly see if/which trackers were used by the wallets. However, this information was omitted from the final report, as it was not the focus of our research.

Chapter 7

Evaluation of Existing Wallets and Data Storage Apps

This chapter is split into sections for different types of wallets and data storage apps. Each section contains a table with the evaluation results of the apps in that category. This division is made to make it easier to compare the apps in the same category, as they usually have similar requirements and features.

7.1 Evaluated Apps

7.1.1 ID Wallets

ID Wallets are apps that store digital identity documents, such as passports, drivers' licenses, or vaccination certificates. They are interesting to look at, as the data they store is critical to users, has to be authentic, and the apps are expected to be secure, whilst being able to display the verifiable, authentic data to third parties. They usually have to transmit a somewhat large amount of data, sometimes in the Megabytes, as they have to transmit the document and cryptographic proofs of its authenticity. Their implementation often depends on underlying SDKs, especially for SSI related functionality. The evaluation shows generally good results in this category (Table 7.1), with apps using secure network connections, the Android KeyStore to protect their keys and confirmation of user presence when the app is unlocked.

Learner Credential Wallet The Learner Credential Wallet (lcw.app) [57] is an open-source application designed to store credentials in accordance with the W3C Verifiable Credentials Data Model [130]. According to the developers, " *The wallet is based on the learner credential wallet specification developed by the Digital Credentials Consortium. The learner credential wallet specification is based on the draft W3C Universal Wallet interoperability specification and the draft W3C Verifiable Credentials data model*" [98].

This application leverages the React Native framework, enabling cross-platform development and availability on both the Google Play Store and the Apple App Store. It offers the ability to import and export these credentials as JSON-LD files. Given its open-source nature, the application can be verifiably built from source under specific conditions. However, auditing the native code can be challenging due to the presence of 106 native libraries for each architecture.

Google Wallet Google Wallet (com.google.android.apps.walletnfcrel) [76] is barely a freestanding app, as it uses the Google Play Services framework to provide its functionality. This makes it both interesting to look at, as it features best practice implementations, but also hard to analyze, as most of the functionality is not contained in the app itself. Interesting to note is that not all the data is end-to-end encrypted, as some of the data is stored in the Google cloud.

The app uses Play Integrity to detect and block rooted devices.

Evernym Connect.Me Evernym Connect.Me (me.connect) [65] is written in React Native and uses Blockchain technologies for SSI. It uses the Play Integrity API to detect and block rooted devices, and it uses the Android KeyStore to store its keys.

Its identity wallet functionality is based on an SSI model and uses multiple distributed ledgers to store the data.

7.1.2 Ticket Wallets

Ticket Wallets are functionally similar to ID Wallets, but they store tickets instead of identity documents. Tickets are usually not as problematic as identity documents, but they are still expected to be authentic, and the apps are expected to be secure, whilst being able to display the verifiable, authentic data to third parties.

A lot of the apps do not require additional confirmation of the user's identity when the app is unlocked. This is a security risk, as it allows any user who has access to the device to access the data. However, in this category, the apps are meant to provide quick access to the tickets, and the security risk is probably acceptable for the convenience that it provides (Table 7.1).

WalletPasses | Passbook Wallet WalletPasses | Passbook Wallet (io.walletpasses.android) [133] is a simple ticket wallet application, implemented in Java, that allows users to store and manage their passes. Previously, the application lacked any form of encryption and hadn't received updates for several years. However, recent versions have seen frequent updates, one of which notably introduced the use of SQLCipher for secure data storage. This app requires a large number of permissions, including access to the camera, location, internet, the ability to read and write to external storage and the Google advertising ID, which are not all necessary for its functionality.

Pass2U Wallet - digitize cards Pass2U Wallet - digitize cards (com.passesalliance.wallet) is another ticket wallet application [118], this time implemented in Kotlin, that allows users to store and manage their passes. The only thing that sets this app apart from other ticket wallets is that it has paid features, such as the ability to transfer passes between devices. This app wants even more permissions than WalletPasses, including access to the camera, location, internet, the ability to read and write to external storage, the Google advertising ID or the READ_PHONE_STATE permission.

PassWallet - mobile passes PassWallet - mobile passes (com.attidomobile.passwallet) [34] is another example of a ticket wallet, implemented in Kotlin, that, again, allows users to store and manage their passes. It's noteworthy that the implementation is straightforward, with no custom functionality added through native libraries and minimal permissions required, almost only those essential for the app's operation, with exceptions like access to the devices' advertising ID.

7.1.3 PDF Wallets

PDF Wallets are apps that store PDF files. They are interesting to look at, as the data they store is sometimes critical, but usually doesn't need to be transmitted to third parties. There is not a wide variety of features that these apps can and should have, and thus only one app was found that fits this category (Table 7.1).

Green Pass PDF Wallet Green Pass PDF Wallet (com.michaeltroger.gruenerpass) [101] is a pure and minimalistic PDF storage application. It does not rely on biometrics or any other KeyStore functionality for data protection. It uses the Jetpack RoomDatabase for storage. When it comes to retrieving data, the app only employs the query *SELECT * FROM certificate*. Additionally, the app is capable of decrypting password-protected PDFs and converting them into local, unprotected temporary files.

All in all, the app is basic and straightforward, lacks any kind of special encryption or security features, but it does its job and doesn't claim to be anything more than a PDF storage app.

7.1.4 Open Crypto Wallets

Crypto Wallets are apps that store cryptocurrencies. We made a distinction between open and closed source cryptocurrency wallets, as the closed ones are usually tough to decompile and analyze due to their complex and domain specific implementations and rather large codebases. They are interesting to look at, as the data they store is critical, and the apps are expected to be secure, whilst being able to transmit the verifiable, authentic data to third parties. They usually have multiple interesting components, such as their ability to store wallet keys, their ability to perform operations on the blockchain, and their ability to transmit data to third parties.

All the apps in this category use secure network connections, and most of them use the Android KeyStore to protect their keys (Table 7.1).

Electrum Electrum (org.electrum.electrum) [62] has both a desktop and Android client. It is built using Python and the Kivy framework. Despite the significant size of the codebase, there is minimal documentation available. Encryption hardware does not appear to be used in the application; instead, it only relies on a user password for authentication during each login to access the otherwise unencrypted wallet data files. Interesting to note is that the application uses three permissions, camera, internet and *WRITE_EXTERNAL_STORAGE*.

Bitpay Wallet Bitpay Wallet (com.bitpay.wallet) [39] by BitPay, Inc. is an app built using Apache Cordova. To protect its data, it uses the Fingerprint AIO plugin for Cordova [107], which uses system APIs of biometrics for both Android and iOS.

Bitcoin Wallet Bitcoin Wallet (de.schildbach.wallet) [38] is a simplistic Bitcoin wallet for Android, implemented in Java. It features no protection of data itself, and instead relies on the Android system to protect its data (Table 7.1).

Simple Bitcoin Wallet The Simple Bitcoin Wallet (com.btcontract.wallet) [33] is built using Scala. It uses the Tor network for communication, but it doesn't use the Android KeyStore for storing its keys, as it does all of its cryptography in software. Interestingly, this app has imported the source code of some libraries into its own codebase, instead of using the libraries as dependencies. This makes it harder to update the libraries, and makes it more likely that the app will use an outdated version of the library, but it also makes it easier to have a *good* version that is secure from supply chain attacks, and in the case of crypto wallets, modifying the libraries could also be a necessary step to include domain specific features.

Blockstream Green Blockstream Green (com.greenaddress.greenbits_android_wallet) [42] is a company backed Bitcoin wallet that uses a plain, unencrypted Jetpack RoomDatabase [40, 41]. It uses biometric authentication, but the CryptoObject is not used, and thus the biometric authentication is not backed by the Android KeyStore.

Unstoppable Crypto Wallet One key security feature of the Unstoppable Crypto Wallet (io.horizontalsystems.bankwallet) [81] is its ability to detect rooted devices. The wallet also utilizes the Tor network. The application employs biometric authentication through the biometricPrompt.authenticate(promptInfo.build()) method, but does not utilize the CryptoObject [83]. User data in the Unstoppable Crypto Wallet is stored in a plain Jetpack RoomDatabase [82]. To ensure data security, the application manually encrypts values within the database using an AES/CBC/PKCS7Padding cipher and a SecretKey that requires user authentication to access, but biometric authentication is optional.

Zap Android Zap Android (zapsolutions.zap) [144] is implemented in Java, can be built from source, and uses Tor for some of its network connections. For storing its preferences at rest, it uses encrypted shared preferences, together with an androidx.security.crypto.MasterKey [145]. Bypassing the biometric authentication is easily possible on rooted devices, as the app does not use the CryptoObject, and thus the access to the key is not backed by biometric authentication. This does, however, not mean that the app is less secure than other apps, as the data still is encrypted at rest.

AirGap Vault AirGap Vault (it.airgap.vault) [3] is a Crypto currency storage application. It is meant to be used together with the AirGap Wallet, which is a companion app that is used to sign transactions. Nevertheless, the AirGap Vault

is a standalone app used to store the keys, and the design decisions, as described by the authors [3], are interesting to look at.

Additionally, the app uses the RootBeer library [126] to detect rooted devices, and it uses the Android KeyStore to store its keys.

7.1.5 Closed Source Crypto Wallets

Even though the closed-source wallets are usually tough to decompile and analyze due to their domain specific implementations and rather large codebases, we were able to analyze some of them. Their results, as shown in Table 7.1, are generally good, but there are open source alternatives that offer better security features with more transparency.

Coinbase Wallet: NFTs & Crypto The Coinbase Wallet: NFTs & Crypto (org.toshi) [51], a React Native application, intriguingly appears to contain some debugging code, suggesting it might rather be a development build than a finalized release. This application has permissions to access user contacts and media folders, including images and videos, and is capable of media playback. The uses the Android KeyStore, and actually properly uses biometric authentication to protect its keys.

Trust: Crypto & Bitcoin Wallet Trust: Crypto & Bitcoin Wallet (com.wallet.crypto.trustapp) [141] is a Kotlin application with its core functionality encapsulated in a native shared object. This binary file is dynamically loaded into the app at runtime, providing the essential operations of the wallet.

Interestingly, the authors appear to have unintentionally left in DebugProbesKt.bin, a helper binary used for debugging Kotlin Coroutines in IntelliJ IDEA. This could potentially suggest that the app was in a development phase or that there was an oversight during the final stages of deployment.

In terms of security, the application employs the Android KeyStore to generate cryptographic keys, and it uses SQLCipher for encrypted databases, enhancing the security of stored data. The application also uses the RootBeer library to detect and block rooted devices, and it has been obfuscated in a way that significantly slows down the process of automatic decompilation and review.

On the data privacy side, the developers state in their Google Play Store listing that they collect user interaction data for analytics purposes, and they explicitly mention that they don't provide an option to delete this data.

7.1.6 Authenticator Apps

Aegis Authenticator Aegis Authenticator (com.beemdevelopment.aegis) [36] can "*Import from other authenticator apps: 2FAS Authenticator, Authenticator Plus, Authy, andOTP, FreeOTP, FreeOTP+, Google Authenticator, Microsoft Authenticator, Plain text, Steam, TOTP Authenticator and WinAuth (root access is required for some of these)*" [37].

This shows that the app is able to access the data of other apps, which shows a security risk in those apps if the extraction is possible without user interaction. The data that is extracted could be very well copied to a remote device if a malicious app wants to.

The app itself also employs a custom crypto library which is designed after the LUKS standard and protects its master key with the KeyStore [37].

Microsoft Authenticator Microsoft Authenticator (com.azure.authenticator) [102] stores its data in an SQLite database, which is not protected by the KeyStore. This allows for easy access to the data, as the database can be copied from the device and read with a SQLite database viewer, or the data can be extracted from the database using a rooted device, like Aegis Authenticator does. This app is part of the Microsoft Entra Verified ID [103] ecosystem.

7.1.7 Password Managers

Keepass2Android Keepass2Android (keepass2android.keepass2android) [120] is a password management application developed using the Xamarin framework. As an open-source project, it provides extensive documentation, making it accessible for developers and users alike to understand its functionality and structure. One of the unique features of Keepass2Android is that it offered some functionality that relies on the device being rooted. On rooted devices, the application was able to automatically switch the input method to a custom keyboard, which could be used to enter passwords securely. While this is not a common requirement for most users, it did provide additional capabilities for those who have rooted devices, but this feature was removed in a later version of the app. The application also incorporates biometric authentication as a security measure. This feature uses a symmetric cipher to secure the user's data. The biometric authentication can serve two purposes in the application:

- It can replace the QuickUnlock functionality, which allows users to reopen the database within a certain period of time using a different password, typically a portion of the master key. This provides a balance between security and convenience, as users can quickly access their data without needing to input the full master key each time.
- Alternatively, the biometric authentication can be used to store the master key on the device itself. The master key is then encrypted and decrypted using biometric data. This method is more convenient for users, as they only need to provide their biometric data to access the database, but in turn, it is less secure in certain scenarios, as the master key is stored on the device.

7.1.8 Banking

Banking Apps are an extreme example of wallets that need to protect their data. They need to be secure whilst maintaining a good user experience, as they are expected to be used frequently. Important required features are the ability to securely access the banking data, to perform transactions, and to display the data to the user. This has to happen in a secure way, as the data is critical to users. Security features have to include the protection of offline data, through key-store backed encryption, the protection of network traffic to avoid modification and tracking of transactions, and the protection of the user interface to deny screen captures and to deny tampering attempts with the data that is displayed to the user.

Raiffeisen Mein ELBA-App The Raiffeisen Mein ELBA-App (at.rsg.pfp) [123] is a banking application developed using the Kotlin programming language. It is designed with a high level of security and user-friendliness in mind. The application fetches all its data from remote servers, ensuring real-time access to banking information. Access to these servers is secured with a personal PIN or

a key protected by biometric authentication. To prevent unauthorized access or data leakage, the application disallows screenshots, ensuring that critical information remains within the app. It also features root detection capabilities to prevent accidental usage on compromised devices.

7.2 Evaluation Results

The evaluation results of the existing wallets and data storage apps are shown in Table 7.1. Interestingly, most apps in the same category have similar results.

7.2.1 Storage

In the realm of storage, many applications rely on the inherent security of the Android sandbox to safeguard their data. This is a common practice, especially among JavaScript-based development frameworks. These frameworks typically either have a custom KeyStore abstraction or depend on their ability to incorporate native Java code. When it comes to encryption, all the evaluated applications utilize either the encryption mechanisms provided by the Android KeyStore or implement well-known encryption algorithms themselves, an example of which is Keepass2Android. It's important to note that using a fingerprint without a CryptoObject does not ensure secure storage. Instead, it merely provides protection for the user interface, a fact confirmed by the development team behind Google's Tink library[1].

An example for how React Native apps can store their data securely is the Learner Credential Wallet, which uses Crypto-js and @microsoft/msr-crypto for cryptographic operations, and the interaction with the platform's hardware-backed KeyStore is managed by react-native-keychain.

7.2.2 Network

The default Android approach to establish network connections is using HTTPS, enhancing the overall security of applications. The complexity of establishing an HTTPS connection is abstracted away by the Android system, with added security features such as certificate pinning. Certificate pinning makes MITM attacks more difficult, as an attacker has to inject custom code into the app to bypass it. Additionally, Simple Bitcoin Wallet and Zap Android leverage the Tor network to anonymize their connections, further bolstering their security by ensuring the privacy of their network traffic. There were very few apps that did not use secure connections at some points, but those insecure connections were usually not critical to the app's functionality, but still can pose a security risk.

7.2.3 User Interface

The majority of applications, 81% (18/22), implement some form of user interface protection mechanism, but that still leaves 19% of the applications, which

[1] https://github.com/tink-crypto/tink-java/blob/6f0621038f828d965c3e416e836de761550b4549/src/main/java/com/google/crypto/tink/integration/android/AndroidKeysetManager.java

Table 7.1: Evaluation Results of Existing Wallets and Data Storage Apps

Name	Storage	Network	User Interface	Permissions	Updates	Root Detection	Reproducible	Native Code
Learner Credential Wallet	2	3	2	3	2	1	Yes	Yes
Google Wallet	3	2	2	2	2	2	No	Yes
Evernym Connect.Me	2	2	2	3	2	3	Yes	Yes
WalletPasses | Passbook Wallet	2	1	1	1	1	1	No	No
Pass2U Wallet - digitize cards	1	1	1	1	2	1	No	Yes
PassWallet - mobile passes	1	1	1	2	1	1	No	No
Green Pass PDF Wallet	1	Offline	2	3	2	1	Yes	No
Electrum	1	3	2	2	2	1	Yes	Yes
Bitpay Wallet	3	3	2	1	1	1	No	Yes
Bitcoin Wallet	1	3	1	2	2	1	Yes	No
Simple Bitcoin Wallet	2	TOR, 3	2	2	1	1	Yes	Yes
Blockstream Green	3	3	2	2	2	1	Yes	Yes
Unstoppable Crypto Wallet	2	VPN, 3	2	2	2	2	Yes	Yes
Zap Android	2	TOR, 3	2	2	2	1	Yes	Yes
AirGap Vault	2	Offline	2	3	2	3	Yes	Yes
Coinbase Wallet: NFTs & Crypto	3	3	2	1	2	2	No	Yes
Trust: Crypto & Bitcoin Wallet	3	3	2	1	2	2	No	Yes
Aegis Authenticator	3	3	2	2	2	0	Yes	No
Microsoft Authenticator	1	2	2	2	2	1	No	Yes
Keepass2Android	3	Offline	2	3	2	0	Yes	Yes
Raiffeisen Mein ELBA-App	4	3	2	2	2	2	No	Yes

```
public void onAuthenticationSucceeded(@NonNull BiometricPrompt.
    AuthenticationResult result) {
    super.onAuthenticationSucceeded(result);
    PrefsUtil.editPrefs().putBoolean(PrefsUtil.BIOMETRICS_PREFERRED, true).apply()
    ;
    TimeOutUtil.getInstance().restartTimer();
    PrefsUtil.editPrefs().putInt("numPINFails", 0).apply();
    Intent intent = new Intent(PinEntryActivity.this, HomeActivity.class);
    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK | Intent.
    FLAG_ACTIVITY_NEW_TASK);
    startActivity(intent);
}
```

Listing 7.1: Zap Android Biometric Authentication Callback

do not require additional confirmation of the user's identity once the device is unlocked.

Common methods of user interface protection include passwords, PINs or biometric authentication, where during the app's startup a biometric prompt is shown to the user, either with or without the CryptoObject to protect the data. This protection is often implemented as a by-product, as the biometric authentication is meant to protect the data, and the user interface is protected as a side effect. It's important to note that the biometric authentication does not protect the data itself, but only the user interface. If the data is not protected by the Android KeyStore, the data is not secured.

An example for such UI protection is Zap Android (see section 7.1.4), which starts the protected home screen activity from a successful biometric authentication callback (see Listing 7.1). This means in order to circumvent the UI protection of the app, an attacker would only have to start the activity without the callback.

Similarly, AirGap Vault (section 7.1.4) uses the biometric authentication to protect the UI. It actually has the code shown in Listing 7.2 in its secure storage class[2].

7.2.4 Permissions

It is clear that the number of permissions an app requires often correlates with the size of the app and the breadth of its feature set. Larger applications with more features typically require more permissions. However, it's worth noting that certain apps, such as Bitpay Wallet, Coinbase Wallet: NFTs & Crypto, and Trust: Crypto & Bitcoin Wallet, request a significant number of permissions that are not essential for their core functionality.

7.2.5 Updates

A significant majority of applications, specifically 20 out of 22, receive regular updates, ensuring their functionality remains up-to-date and secure. The application WalletPasses | Passbook Wallet is especially noteworthy, since it had a

[2]https://github.com/airgap-it/airgap-vault/blob/60feba0bfd7cf3129587ec2b0e39a48eb70503ef/android/app/src/main/java/it/airgap/vault/plugin/securityutils/storage/Storage.kt

```
BiometricPrompt(this, executor, object : BiometricPrompt.AuthenticationCallback()
    {
    override fun onAuthenticationError(errorCode: Int, errString: CharSequence) {
        super.onAuthenticationError(errorCode, errString)
        _resultDeferred.complete(false)
    }

    override fun onAuthenticationSucceeded(result: BiometricPrompt.
    AuthenticationResult) {
        super.onAuthenticationSucceeded(result)
        _resultDeferred.complete(true)
    }

    override fun onAuthenticationFailed() {
        super.onAuthenticationFailed()
        _resultDeferred.complete(false)
    }
})
```

Listing 7.2: AirGap Vault Biometric Authentication Callbacks

period of stagnation where it did not receive updates for several years. Recently, this trend has reversed, and the application has started receiving frequent updates again.

7.2.6 Root Detection

Out of the 22 evaluated applications, 5 of them utilize root detection mechanisms to inform users that their device is rooted. Furthermore, 2 out of the 22 applications go a step further by using root detection to block the operation of the application on rooted devices. It's noteworthy that these applications rely on established libraries such as Play Integrity or other purpose-designed libraries like RootBeer for root detection, rather than implementing custom solutions.

7.2.7 Reproducible

10 out of the 22 evaluated applications provide the capability to be built from source. The apps were not tested for binary reproducibility, as that would require us to acquire the signing keys of the apps, as comparing a signed with an unsigned binary is an open research question.

7.2.8 Native Code

Applications developed using React Native, Apache Cordova, and Mono/C# inherently utilize native code. This is due to the nature of these frameworks, which compile down to native code. Furthermore, a significant number of applications, particularly those related to cryptocurrency, incorporate natively compiled cryptocurrency libraries. This results in a substantial amount of native code within these applications. Only a minority of the evaluated applications, specifically 5 out of 22, do not incorporate any business logic as native code.

Chapter 8

Results

8.1 Overview

With the current state of the art, it is possible to implement a digital wallet on a smartphone that is at least as secure as a physical wallet.

The actual security of a digital wallet currently depends on the security of the smartphone it is running on. If the smartphone is compromised, then in most cases the wallet is easily compromised as well.

On Android, data is vulnerable if it is somehow possible that it is unencrypted at some point, even if this is just due to reordering of program functions or by abusing weak encryption algorithms. In a digital identity wallet the data is usually meant to be stored offline, and transmitted to an also potentially offline verifier.

Storing data at rest on a relatively modern device is not an issue. The issue is the fact that the data needs to be accessed at some point, and this is where the problems start. There needs to be a way to import data into the wallet, to store it, and there needs to be a way to export data from the wallet, to use it.

Data at Rest

Properly protecting data at rest can be called a solved problem, as modern devices provide secure storage mechanisms, like TEEs, in combination with encryption algorithms. Android apps can use the KeyStore API, together with encryption libraries, such as SQLCipher or Google Tink, to store data in a secure way.

Data in Transit

Protecting data in transit is relatively easy, as there are well established protocols and algorithms to do so. Most apps already employ secure mechanisms, such as HTTPS, and in even more extreme cases VPN tunnels, so this area is not an issue anymore. Still, when implementing it needs to be implemented correctly, and implementers should not just rely on the security of the transport medium.

Data in Use

This area is the most unexplored and thus difficult to protect. In the explored applications, most Systems expect at least one party to be trustworthy [128],

which is not always the case in the Digidow scenario. If there is no trusted environment to run code in, the only way to protect data in use is to not decrypt it at all, if possible.

In our context, not all data needs to be available at the same time. The identity documents are independent from each other, and sometimes support selective disclosure, where even less of a document needs to be decrypted to be usable.

This is feasible for the importing of identity documents, for example, by transmitting a trust zone backed public key to the issuer, which is then used to encrypt the identity document, which in turn can only be decrypted by the trust zone. This way of protecting data does not work for the transmission of documents or elements thereof to a verifier, as a client would need to decrypt the data in order to re-encrypt it for the verifier. An approach here would be to encrypt the data twice, and to apply the decryption only after the data has been re-encrypted, but in a naive implementation this would imply the possibility of reordering the code on the client to get a state of complete decryption, as seen in Apples iTunes DRM [138].

Another approach would be to use a proxy re-encryption (PRE) scheme. PRE is a cryptographic method that allows a proxy to transform a ciphertext from one public key to another, without learning anything about the underlying plaintext [46]. This way the issuer could encrypt the identity document, and the client could re-encrypt it for the verifier, without ever decrypting it. This approach sadly has some issues as well, since the proxy would still need to be trusted to a certain degree to guarantee that the re-encryption is done confidentially and correctly [109]. Another limitation would be the fact that the proxy cannot interact with the data themselves, as they would need to decrypt it first, which is not possible in this scenario.

Zero-knowledge proofs are another way to prove that a certain data is correct, without actually revealing the data itself. These approaches are usually complex, but they offer many more features to transmit data securely, as data should not have to be transmitted at all, but only the proof that the data is correct. Having hardware support for Zero-Knowledge Proof (ZKP) methods in Android devices could be useful. These methods are computationally heavy and offer interesting features, so having them built into the device could improve performance.

8.2 Best Practices for Wallet Apps

8.2.1 Security

There are a lot of different security aspects that need to be considered when creating a new wallet. The data in the wallet needs to be protected from unauthorized access, and may not provide any information outside of what the user absolutely needs to share. Additionally, security needs to be considered when planning for a longer term use of the wallet, as the user might switch to a different device, might reset this device or might even lose it. Revocation of documents (such as a driver's license) is also a security aspect that needs to be considered from a provider's perspective.

Secure Storage

The most important aspect of a secure wallet is the secure storage of the data. Luckily Android and iOS provide a secure storage for the user's data, where keys

to data can be set to be protected by biometrics, such as fingerprints or face recognition. Sadly at the point of writing only Android, not iOS, supports the creation of a certificate that validates that a key is actually protected by biometrics.

Even though the storage is secure when it comes to unauthorized access, it is still possible to read the data when it is inserted into or extracted from the storage, as the data is usually not encrypted in those cases.

The actual storage should work as follows:

- The identity document is stored in a self containing format, such as a PDF file or a JSON-LD document.
  - The self containing format could be split internally to support selective disclosure.
- Create a cipher that is protected by the KeyStore (maybe with biometrics).
- The identity document or its parts are encrypted with said cipher.
- The encrypted identity document is stored in a SQL database, such as SQLite or Room-db, together with the cipher identifier, a boolean value indicating the need for biometrics to unlock and the proof-of-possession data.
- The SQL database is encrypted with a cipher that is protected by the KeyStore, maybe utilizing biometrics (User Preferences).

These multiple layers of encryption ensure that the data is secure at all times, that the data is only accessible when the user is actually using the wallet, and if a single layer is compromised, the data is still secure.

Authenticity

For a valid identity it is important that the user is actually presenting an official, valid and authentic document. This can be achieved by using a trusted third party, such as a government agency or a company, to verify the authenticity of the initial document that is stored in the wallet. In a classical scenario this would be a CA that issues a certificate that is then used to sign the identity document.

To avoid the constant need to verify the document against the original and to ensure maximum privacy for the user the identity document and every subpart that is available through selective disclosure should be stored in a self containing format that is not dependent on the original document or the third party providing it.

Device Binding

The identity provider must also sign the protection key of the identity document to assure to the party requesting an authentication that the identity document is authentic and has not been extracted from another device or wallet. This signature can be verified by the requesting party to ensure that the identity document is stored on the original device, which in turn is the device that is owned by the user and verified by the issuing authority.

User Binding

The issue with most wallets is the fact that they store credentials without any proof of possession mechanisms, so there is no way to prove the user is the original holder this credential was issued to. User binding is a problem, because the user (or a 3rd party) could just copy the credential and send it to another person, who could then use it as well.

A similar problem exists with physical wallets, as there is usually no way to prove the user (person showing an ID) is the original holder (the person an ID was issued to) of the credentials (like a drivers license). If going by the most accurate form of verification, the (biometric) photo, then a person looking similar to the photo could use the ID as well.

Proof of possession and proof of intention mechanisms can and should be implemented in digital wallets to increase their security.

These mechanisms are usually only specific to a single device, as anybody with pre-enrolled biometric data would be a valid user of the wallet. These mechanisms also need un-breakable Trusted Execution Environments (TEE) to avoid authenticating with wrong biometrics, and they need to be invalidated when a new biometric is enrolled.

In an ideal world, the setup process includes the complete removal of all pre-enrolled biometrics, so that only the biometrics of the current, actual, interacting user are enrolled. This would need an attestation that verifies that no other biometrics are enrolled, which is not possible/implemented with current TEEs.

Secure Interface

This also belongs in part to the secure storage, as a key protected with biometrics can only be accessed when accessing the wallet with said biometrics. It is therefore necessary and a good practice to use biometrics to unlock the wallet.

Data Migration

With usability in mind, it is also important to consider the possibility of the user setting up the wallet on a different device. It is necessary to provide a way to transfer to a new device without losing any data and without risking the security of the stored documents. This might be achieved by using a different verification method, such as logging in to a service provided by the issuing authority, or in higher security cases by physically confirming the device ownership to the issuing authority. Android already provides a way to securely transfer app data between devices, but there is no way to transfer the contents of a KeyStore, so the backup and restore process would need to be implemented by the wallet itself.

Secure Communication

A basic requirement for a secure wallet is the secure communication between the wallet and the issuer of the identity. This can be achieved by using a secure channel, such as HTTPS, to communicate with the issuer or by using physically close range technologies, such as NFC or QR codes, to import the identity document into the wallet. Close range technologies are more secure, as they automatically imply a physical presence of the user, and the communication does

not have to use the internet. This is all dependent on the issuer's capabilities and the proposed use case and security level. In a high security case, the issuer might require the user to visit their office to verify the identity document, in a lower security case the issuer might only require the user to confirm the identity document by scanning a QR code sent by mail, and in a very low security case the issuer might only require the user to download the identity document from their website.

Device Integrity

The integrity of the device is also an important aspect of a secure wallet. It should be checked regularly if the device is still in a secure state, and if not, the user should be notified about potential risks. Using Device Attestation, as described in section 4.11.5, it is possible to check the integrity of a device. Blocking the wallet when the device is not in a secure state is a good practice, as it prevents the user from accidentally using the wallet in an insecure environment, yet there should be a way to manually allow the use of the wallet in an insecure environment if users are truly aware of the risks. Fully banning insecure or modified devices from using the wallet could lead to users rooting their devices to use the wallet, which is a security risk in itself.

Update Policy

A secure wallet should also have a good update policy, so that the user is always using the latest version of the wallet. This is important, as new security issues might be discovered regularly, and it is important to fix them as soon as possible. This means that the building and deployment process should be as automated as possible, whilst still going through a thorough an advanced automated testing process to avoid regressions.

8.2.2 Visual Design

The design of a wallet also plays a role in the usability and thus the security of the wallet.

The need for an easy and user-friendly way of navigating through different options and screens is an important aspect of a secure wallet, as the user should not be confused or overwhelmed by the interface and accidentally provide more information than necessary [53].

It should also be easy to integrate the wallet into the user's daily life, so that the user does not have to think about the wallet and can use it without much effort. Similarly, this also applies to the issuing authorities and parties requesting authentication, as they should be able to easily integrate the wallet into their systems.

From a purely optical point of view an appealing design is also important, as the user should be motivated to use the wallet. This could even go as far as to provide a replaceable frontend for the wallet, so that the user can choose a design that fits their personal taste. This would, of course, bring new challenges, such as the need to verify the frontend obstructs no information and does not affect usability in any way. This could help with getting rid of users installing modded versions of the wallet, or using root permissions to change the design of the wallet.

A good, simple way to integrate modern design into a wallet is to use the Material You[1] design system, which provides variables with the color scheme of a user, and can be used to easily create a design that fits the user's personal taste.

[1]https://material.io/blog/start-building-with-material-you

Chapter 9

Conclusion

In this thesis, we explored the security features available in the Android operating system and checked how well different digital wallet apps use these features. We examined several types of wallets, such as cryptocurrency wallets, payment wallets, ticket wallets, and identity wallets. By looking at each category, we assessed the security features each app used, based on our understanding of Android's architecture and its security tools like sandboxing, the KeyStore system, biometrics, and encryption.

Our findings showed that the use of Android's security features varies widely among different wallets. Some wallets use best practice approaches to security, while others lack important protections, making them more vulnerable. For example, the use of biometric authentication differed significantly, affecting the overall security of the affected wallets.

We used specific criteria to evaluate the wallets, such as storage protection, network security, user interface protection, permissions management, update policies, root detection, and the use of native code. These criteria helped us systematically analyze each wallet's security. The results highlight the need for a comprehensive approach to security, integrating multiple layers of protection to ensure user data is safe.

This research points out weaknesses in current wallet security practices and offers recommendations for developers. Future research could build on this work by exploring new features as Android updates or by implementing the missing security features like user binding.

Bibliography

[1] Abi Raja. 2024. screenshot-to-code Github Repository. (January 2024). Retrieved 02/08/2024 from https://github.com/abi/screenshot-to-code.

[2] Abraham Augustine. 2023. The Limits of Accelerating Digital-Only Financial Inclusion. (July 2023). Retrieved 04/22/2024 from https://carnegieendowment.org/2023/07/13/limits-of-accelerating-digital-only-financial-inclusion-pub-90175.

[3] AirGap. 2024. airgap-vault Github Repository. (April 2024). Retrieved 04/25/2024 from https://github.com/airgap-it/airgap-vault.

[4] Sumaya Almanee, Mathias Payer, and Joshua Garcia. 2021. Too Quiet in the Library: A Study of Native Third-Party Libraries in Android. *CoRR*. DOI: 10.48550/ARXIV.1911.09716. arXiv: 1911.09716.

[5] Iman M. Almomani and Aala Al Khayer. 2020. A Comprehensive Analysis of the Android Permissions System. *IEEE Access*, 8, 216671–216688. DOI: 10.1109/ACCESS.2020.3041432.

[6] Amber Steel. 2023. The Compromised Credentials Crisis: A Challenge Plaguing the Cybersecurity Industry. (June 2023). Retrieved 04/22/2024 from https://blog.lastpass.com/posts/2023/06/the-compromised-credentials-crisis-a-challenge-plaguing-the-cybersecurity-industry.

[7] Andre Lipke. 2020. Reverse Engineering Flutter Apps. (March 2020). Retrieved 09/05/2023 from https://blog.tst.sh/reverse-engineering-flutter-apps-part-1/.

[8] Andrew Kennedy. 2022. Employee Identity and Access Management: A BITS Primer. (March 2022). Retrieved 04/22/2024 from https://bpi.com/employee-identity-and-access-management-a-bits-primer/.

[9] Android Open Source Project. 2023. Android 14 Compatibility Definition: Biometric Sensors. (November 2023). Retrieved 01/24/2024 from https://source.android.com/docs/compatibility/14/android-14-cdd#7310_biometric_sensors.

[10] Android Open Source Project. 2024. Android Keystore system. (January 2024). Retrieved 01/24/2024 from https://developer.android.com/training/articles/keystore.

[11] Android Open Source Project. 2022. APK Signature Scheme v2. (September 2022). Retrieved 05/09/2023 from https://source.android.com/docs/security/features/apksigning/v2.

[12] Android Open Source Project. 2022. APK Signature Scheme v3. (October 2022). Retrieved 05/09/2023 from https://source.android.com/docs/security/features/apksigning/v3.

[13] Android Open Source Project. 2022. APK Signature Scheme v4. (September 2022). Retrieved 05/09/2023 from https://source.android.com/docs/security/features/apksigning/v4.

[14] Android Open Source Project. 2023. Application Sandbox. (October 2023). Retrieved 01/24/2024 from https://source.android.com/docs/security/app-sandbox.

[15] Android Open Source Project. 2022. Cryptography. (October 2022). Retrieved 04/26/2023 from https://developer.android.com/guide/topics/security/cryptography.

[16] Android Open Source Project. 2023. Explain access to more sensitive information. (May 2023). Retrieved 05/22/2023 from https://developer.android.com/training/permissions/explaining-access.

[17] Android Open Source Project. 2023. Face Authentication HIDL. (August 2023). Retrieved 09/26/2023 from https://source.android.com/docs/security/features/biometric/face-authentication.

[18] Android Open Source Project. 2024. File-Based Encryption. (January 2024). Retrieved 01/24/2024 from https://source.android.com/docs/security/features/encryption/file-based.

[19] Android Open Source Project. 2023. Implementing dm-verity. (April 2023). Retrieved 05/24/2023 from https://source.android.com/docs/security/features/verifiedboot/dm-verity.

[20] Android Open Source Project. 2023. Jetpack: Security. (April 2023). Retrieved 05/31/2023 from https://developer.android.com/jetpack/androidx/releases/security#security-identity-credential-1.0.0-alpha01.

[21] Android Open Source Project. 2022. Measuring biometric unlock security. (October 2022). Retrieved 04/26/2023 from https://source.android.com/docs/security/features/biometric/measure.

[22] Android Open Source Project. 2023. Permissions on Android. (April 2023). Retrieved 04/26/2023 from https://developer.android.com/guide/topics/permissions/overview.

[23] Android Open Source Project. 2023. Permissions updates in Android 11. (May 2023). Retrieved 05/22/2023 from https://developer.android.com/about/versions/11/privacy/permissions.

[24] Android Open Source Project. 2023. Platform architecture. (May 2023). Retrieved 04/10/2024 from https://developer.android.com/guide/platform.

[25] Android Open Source Project. 2023. Save data in a local database using Room. (February 2023). Retrieved 04/26/2023 from https://developer.android.com/topic/libraries/architecture/room.

[26] Android Open Source Project. 2023. Save data using SQLite. (March 2023). Retrieved 04/26/2023 from https://developer.android.com/training/data-storage/sqlite.

[27] Android Open Source Project. 2022. Security with network protocols. (December 2022). Retrieved 04/26/2023 from https://developer.android.com/training/articles/security-ssl.

[28] Android Open Source Project. 2024. Show a biometric authentication dialog. (January 2024). Retrieved 01/24/2024 from https://developer.android.com/training/sign-in/biometric-auth.

[29] Android Open Source Project. 2023. Shrink, obfuscate, and optimize your app. (June 2023). Retrieved 09/05/2023 from https://developer.android.com/studio/build/shrink-code.

[30] Android Open Source Project. 2023. Sign your app. (April 2023). Retrieved 04/26/2023 from https://developer.android.com/studio/publish/app-signing.

[31] Android Open Source Project. 2023. SQLCipher for Android. (February 2023). Retrieved 04/26/2023 from https://github.com/sqlcipher/android-database-sqlcipher.

[32] Android Open Source Project. 2023. Tapjacking. (February 2023). Retrieved 04/26/2023 from https://developer.android.com/topic/security/risks/tapjacking.

[33] Anton Kumaigorodski. 2023. akumaigorodski/wallet Github Repository. (November 2023). Retrieved 04/25/2024 from https://github.com/akumaigorodski/wallet.

[34] Attido Mobile. 2023. PassWallet - Passbook Wallet. (July 2023). Retrieved 04/25/2024 from https://play.google.com/store/apps/details?id=com.attidomobile.passwallet.

[35] ax. 2024. apk.sh Github Repository. (February 2024). Retrieved 03/21/2024 from https://github.com/ax/apk.sh.

[36] Beem Development. 2024. Aegis Github Repository. (April 2024). Retrieved 04/25/2024 from https://github.com/beemdevelopment/Aegis.

[37] Beem Development. 2022. Aegis Github Repository: README.md. (December 2022). Retrieved 05/31/2023 from https://github.com/beemdevelopment/Aegis/blob/efd8e2d9ff25f59e3d4bcaab664641b0e45761e4/README.md.

[38] Bitcoin Wallet Contributors. 2023. Bitcoin Wallet Github Repository. (January 2023). Retrieved 01/24/2024 from https://github.com/bitcoin-wallet/bitcoin-wallet.

[39] BitPay. 2024. BitPay Wallet Github Repository. (April 2024). Retrieved 04/25/2024 from https://github.com/bitpay/wallet.

[40] Blockstream Corporation Inc. 2023. Blockstream Green. (July 2023). Retrieved 09/13/2023 from https://github.com/Blockstream/green_android/blob/3c861f5752a8db586ab81142d0e559bb84c6d326/green/src/main/java/com/blockstream/green/database/AppDatabase.kt.

[41] Blockstream Corporation Inc. 2023. Blockstream Green. (June 2023). Retrieved 09/13/2023 from https://github.com/Blockstream/green_android/blob/3c861f5752a8db586ab81142d0e559bb84c6d326/green/src/main/java/com/blockstream/green/database/WalletDao.kt.

[42] Blockstream Corporation Inc. 2024. Blockstream Green Github Repository. (April 2024). Retrieved 04/25/2024 from https://github.com/Blockstream/green_android.

[43] Boris Batteux. 2022. The Current State & Future of Reverse Engineering Flutter™ Apps. (June 2022). Retrieved 09/06/2023 from https://www.guardsquare.com/blog/current-state-and-future-of-reversing-flutter-apps.

[44] Build38. 2021. 5 Benefits of using Mobile ID Wallets and use-cases. (November 2021). Retrieved 06/02/2024 from https://build38.com/5-benefits-to-use-mobile-id-wallet-and-use-cases/.

[45] Bundesministerium für Wirtschaft und Klimaschutz. 2020. ONCE - Online einfach anmelden! (January 2020). Retrieved 01/24/2024 from https://www.digitale-technologien.de/DT/Redaktion/DE/Standardartikel/SchaufensterSichereDigIdentProjekte/sdi-projekt_once.html.

[46] Ran Canetti and Susan Hohenberger. 2007. Chosen-Ciphertext Secure Proxy Re-Encryption. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (*CCS '07*). ACM, Alexandria, Virginia, USA, pp. 185–194. DOI: 10.1145/1315245.1315269.

[47] Luca Casati and Andrea Visconti. 2018. The Dangers of Rooting: Data Leakage Detection in Android Applications. *Mobile Information Systems*, 2018, (February 2018). DOI: 10.1155/2018/6020461.

[48] Charles Guillemet. 2024. Scalability War, Data Availability, Account Abstraction: Exploring Crypto's "Broadband Moment". (January 2024). Retrieved 04/22/2024 from https://www.ledger.com/blog-scalability-war-account-abstraction-data-availability-exploring-cryptos-broadband-moment.

[49] Yoonjung Choi, Woonghee Lee, and Junbeom Hur. 2024. PhishinWebView: Analysis of Anti-Phishing Entities in Mobile Apps with WebView Targeted Phishing. In *Proceedings of the ACM on Web Conference 2024* (*WWW '24*). ACM, Singapore, Singapore, pp. 1923–1932. DOI: 10.1145/3589334.3645708.

[50] ClassEditor. 2004. Java Class File Editor. Retrieved 09/26/2023 from https://classeditor.sourceforge.net/.

[51] Coinbase. 2024. Toshi: Ethereum Wallet. (April 2024). Retrieved 04/25/2024 from https://play.google.com/store/apps/details?id=org.toshi.

[52] Connor Tumbleson. 2023. Apktool. (September 2023). Retrieved 09/26/2023 from https://apktool.org/.

[53] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *NDSS*. Volume 17. San Diego, CA, USA, pp. 10–14722. DOI: 10.14722/ndss.2017.23465.

[54] Cybersecurity Education Guides. [n. d.] Identity and Access Control in Information and Network Security. Retrieved 04/22/2024 from https://www.cybersecurityeducationguides.org/access-control/.

[55] Isai Damier, Kevin Chyn, and Curtis Belmonte. 2019. One Biometric API Over all Android. (October 2019). Retrieved 04/26/2023 from https://android-developers.googleblog.com/2019/10/one-biometric-api-over-all-android.html.

[56] Daniel Sogl. 2023. Awesome Cordova Plugins. (April 2023). Retrieved 06/23/2023 from https://github.com/danielsogl/awesome-cordova-plugins.

[57] Digital Credentials Consortium. 2024. Learner Credential Wallet Github Repository. (January 2024). Retrieved 04/22/2024 from https://github.com/digitalcredentials/learner-credential-wallet.

[58] Verena Distler, Matthias Fassl, Hana Habib, Katharina Krombholz, Gabriele Lenzini, Carine Lallemand, Lorrie Faith Cranor, and Vincent Koenig. 2021. A Systematic Literature Review of Empirical Methods and Risk Representation in Usable Privacy and Security Research. *ACM Trans. Comput.-Hum. Interact.*, 28, 6, Article 43, (December 2021), 50 pages. DOI: 10.1145/3469845.

[59] Dock. 2024. Digital Identity: The Key to Unlocking the Future of Work. (June 2024). Retrieved 07/05/2024 from https://www.dock.io/post/digital-identity.

[60] DocuSign. 2023. The Top Trends in Identity Verification Technology. (June 2023). Retrieved 04/22/2024 from https://www.docusign.com/blog/top-trends-identity-verification-technology.

[61] Axel Domeyer, Mike McCarthy, Simon Pfeiffer, and Gundbert Scherf. 2020. How governments can deliver on the promise of digital ID. (August 2020). Retrieved 04/25/2024 from https://www.mckinsey.com/industries/public-sector/our-insights/how-governments-can-deliver-on-the-promise-of-digital-id.

[62] Electrum Technologies GmbH. 2024. Electrum Github Repository. (April 2024). Retrieved 04/25/2024 from https://github.com/spesmilo/electrum.

[63] European Commission. [n. d.] European Digital Identity. Retrieved 04/25/2024 from https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/europe-fit-digital-age/european-digital-identity_en.

[64] Nathan S. Evans, Azzedine Benameur, and Yun Shen. 2015. All your Root Checks are Belong to Us: The Sad State of Root Detection. In *Proceedings of the 13th ACM International Symposium on Mobility Management and Wireless Access* (*MobiWac '15*). ACM, Cancun, Mexico, pp. 81–88. DOI: 10.1145/2810362.2810364.

[65] Evernym. 2023. Evernym GitLab Repository. (March 2023). Retrieved 04/25/2024 from https://gitlab.com/evernym.

[66] Federal Trade Commission. 2013. Fighting Identity Theft with the Red Flags Rule: A How-To Guide for Business. (May 2013). Retrieved 04/22/2024 from https://www.ftc.gov/business-guidance/resources/fighting-identity-theft-red-flags-rule-how-guide-business.

[67] Hyperledger Foundation. 2024. Hyperledger Projects. Retrieved 07/08/2024 from https://www.hyperledger.org/projects.

[68] Fraud.com. 2021. Digital Wallet Fraud: How to Protect Your Business. (November 2021). Retrieved 04/22/2024 from https://www.fraud.com/post/digital-wallet-fraud.

[69] GeeksforGeeks. 2024. Android Architecture. (May 2024). Retrieved 04/10/2024 from https://www.geeksforgeeks.org/android-architecture/.

[70] Sérgio Manuel Nóbrega Gonçalves, Alessandro Tomasi, Andrea Bisegna, Giulio Pellizzari, and Silvio Ranise. 2020. Verifiable Contracting - A Use Case for Onboarding and Contract Offering in Financial Services with eIDAS and Verifiable Credentials. In *Computer Security - ESORICS 2020 International Workshops, DETIPS, DeSECSys, MPS, and SPOSE, Guildford, UK, September 17-18, 2020, Revised Selected Papers* (*LNCS*, volume 12580). Ioana Boureanu, Constantin Catalin Dragan, Mark Manulis, Thanassis Giannetsos, Christoforos Dadoyan, Panagiotis Gouvas, Roger A. Hallman, Shujun Li, Victor Chang, Frank Pallas, Jörg Pohle, and M. Angela Sasse, (Eds.) Springer, pp. 133–144. DOI: 10.1007/978-3-030-66504-3_8.

[71] Google. [n. d.] Application fundamentals. Retrieved 07/08/2024 from https://developer.android.com/guide/components/fundamentals.

[72] Google. 2023. D8 dexer and R8 shrinker. Retrieved 09/05/2023 from https://r8.googlesource.com/r8/.

[73] Google. 2024. Dart. Retrieved 07/08/2024 from https://github.com/dart-lang.

[74] Google. 2023. Flutter architectural overview. (June 2023). Retrieved 09/05/2023 from https://docs.flutter.dev/resources/architectural-overview.

[75] Google. 2023. Flutter Homepage. (June 2023). Retrieved 06/23/2023 from https://flutter.dev.

[76] Google LLC. 2024. Google Wallet. (April 2024). Retrieved 04/25/2024 from https://play.google.com/store/apps/details?id=com.google.android.apps.walletnfcrel.

[77] Guardsquare. 2017. New Android vulnerability allows attackers to modify apps without affecting their signatures. (November 2017). Retrieved 05/24/2023 from https://www.guardsquare.com/blog/new-android-vulnerability-allows-attackers-to-modify-apps-without-affecting-their-signatures-guardsquare.

[78] JD-GUI contributors. 2019. Java Decompiler. (December 2019). Retrieved 09/26/2023 from http://java-decompiler.github.io/.

[79] Ruth Halperin and James Backhouse. 2012. Risk, trust and eID: Exploring public perceptions of digital identity systems. *First Monday*, 17, 4, (April 2012). DOI: 10.5210/fm.v17i4.3867.

[80] Felix Hoops and Florian Matthes. 2024. A Universal System for OpenID Connect Sign-ins with Verifiable Credentials and Cross-Device Flow. *CoRR*. DOI: 10.48550/ARXIV.2401.09488. arXiv: 2401.09488.

[81] HorizontalSystems. 2024. unstoppable-wallet-android Github Repository. (April 2024). Retrieved 04/25/2024 from https://github.com/horizontalsystems/unstoppable-wallet-android.

[82] HorizontalSystems. 2023. unstoppable-wallet-android Github Repository: AppDatabase.kt. (January 2023). Retrieved 05/26/2023 from https://github.com/horizontalsystems/unstoppable-wallet-android/blob/ae2fcf3f8110fe4a8468c6d4948e26efe03488b9/app/src/main/java/io/horizontalsystems/bankwallet/core/storage/AppDatabase.kt.

[83] HorizontalSystems. 2022. unstoppable-wallet-android Github Repository: CipherWrapper.kt. (May 2022). Retrieved 05/26/2023 from https://github.com/horizontalsystems/unstoppable-wallet-android/blob/ae2fcf3f8110fe4a8468c6d4948e26efe03488b9/core/src/main/java/io/horizontalsystems/core/security/CipherWrapper.kt.

[84] iComply Investor Services Inc. 2024. Overcoming Challenges in Digital Identity Verification for Enhanced Security. (January 2024). Retrieved 04/22/2024 from https://icomplyis.com/icomply-blog/overcoming-challenges-in-digital-identity-verification-for-enhanced-security/.

[85] Idemia. 2023. What is Mobile ID? (November 2023). Retrieved 04/22/2024 from https://www.idemia.com/insights/what-mobile-id.

[86] Identity Defined Security Alliance. 2019. THE STATE OF IDENTITY: HOW SECURITY TEAMS ARE ADDRESSING RISK. White Paper. (December 2019). Retrieved 04/22/2024 from https://www.idsalliance.org/white-paper/identity-and-access-management-the-stakeholder-perspective/.

[87] Institute of Networks and Security. 2023. Christian Doppler Laboratory for Private Digital Authentication in the Physical World. Retrieved 05/19/2023 from https://www.digidow.eu/.

[88] JesusFreke. 2021. Home. (March 2021). Retrieved 07/08/2024 from https://github.com/JesusFreke/smali/wiki.

[89] JetBrains. 2024. Kotlin: Getting Started. (June 2024). Retrieved 07/08/2024 from https://github.com/JetBrains/kotlin-web-site/blob/c173c110d60b7684364d47ab9f38ac96e1e2b688/docs/topics/getting-started.md.

[90] John Wu (topjohnwu). 2023. Magisk. Retrieved 05/24/2023 from https://topjohnwu.github.io/Magisk/.

[91] Matt Johnston, Manoj Ahuje, Stephen VanVaerenbergh, and Chad Yoder. 2023. Compromising Identity Provider Federation. (November 2023). Retrieved 04/22/2024 from https://www.crowdstrike.com/blog/compromising-identity-provider-federation/.

[92] Kivy Community. 2023. Kivy: The Open Source Python App Development Framework. (June 2023). Retrieved 06/23/2023 from https://kivy.org/.

[93] Anna-Magdalena Krauß, Sandra Kostic, and Rachelle A. Sellung. 2023. A more User-Friendly Digital Wallet? User Scenarios of a Future Wallet. In *Open Identity Summit 2023, Heilbronn, Germany, June 15-16, 2023* (*LNI*). Heiko Roßnagel, Christian H. Schunck, and Jochen Günther, (Eds.) Gesellschaft für Informatik e.V. DOI: 10.18420/OID2023_06.

[94] Miroslaw Kutylowski, Lucjan Hanzlik, and Kamil Kluczniak. 2016. Pseudonymous Signature on eIDAS Token - Implementation Based Privacy Threats. In *Information Security and Privacy - 21st Australasian Conference, ACISP 2016, Melbourne, VIC, Australia, July 4-6, 2016, Proceedings, Part II* (*LNCS*, volume 9723). Joseph K. Liu and Ron Steinfeld, (Eds.) Springer, pp. 467–477. DOI: 10.1007/978-3-319-40367-0_31.

[95] Ethan P Larsen, Arjun H Rao, and Farzan Sasangohar. 2020. Understanding the scope of downtime threats: A scoping review of downtime-focused literature and news media. *Health Informatics Journal*, 26, 4, 2660–2672. DOI: 10.1177/1460458220918539.

[96] Lauren Hendrickson. 2023. The Importance of Interoperability in Digital Identity. (September 2023). Retrieved 04/22/2024 from https://www.identity.com/the-importance-of-interoperability-in-digital-identity/.

[97] LSPosed. 2023. LSPosed Github Repository. (September 2023). Retrieved 09/26/2023 from https://github.com/LSPosed/LSPosed.

[98] Massachusetts Institute of Technology. 2023. Learner Credential Wallet. Retrieved 05/19/2023 from https://lcw.app.

[99] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. 2021. The Android Platform Security Model. *ACM Trans. Priv. Secur.*, 24, 3, Article 19, (April 2021), 35 pages. DOI: 10.1145/3448609.

[100] Carlo Mazzocca, Abbas Acar, Selcuk Uluagac, Rebecca Montanari, Paolo Bellavista, and Mauro Conti. 2024. A Survey on Decentralized Identifiers and Verifiable Credentials. *CoRR*. DOI: 10.48550/ARXIV.2402.02455. arXiv: 2402.02455.

[101] Michael Troger. 2024. GreenPass Android Github Repository. (April 2024). Retrieved 04/25/2024 from https://github.com/michaeltroger/greenpass-android.

[102] Microsoft Corporation. 2024. Microsoft Authenticator. (April 2024). Retrieved 04/25/2024 from https://play.google.com/store/apps/details?id=com.azure.authenticator.

[103] Microsoft Corporation. 2024. Microsoft Entra Verified ID. (April 2024). Retrieved 04/25/2024 from https://www.microsoft.com/en-us/security/business/identity-access/microsoft-entra-verified-id.

[104] Mitmproxy Project. 2023. Mitmproxy. (September 2023). Retrieved 09/26/2023 from https://mitmproxy.org/.

[105] Mobile Security Framework. 2023. Mobile Security Framework. (September 2023). Retrieved 09/26/2023 from https://mobsf.github.io/Mobile-Security-Framework-MobSF/.

[106] Mozilla. 2024. JavaScript. Retrieved 07/08/2024 from https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[107] Niklas Merz. 2022. Cordova Plugin Fingerprint All-In-One. (November 2022). Retrieved 06/23/2023 from https://github.com/NiklasMerz/cordova-plugin-fingerprint-aio.

[108] Noetic Cyber. 2024. The Hidden Threat: Understanding the Identity Attack Surface. (March 2024). Retrieved 04/22/2024 from https://noeticcyber.com/understanding-the-identity-attack-surface/.

[109] David Nuñez, Isaac Agudo, and Javier Lopez. 2017. Proxy Re-Encryption: Analysis of constructions and its application to secure access delegation. *Journal of Network and Computer Applications*, 87, (June 2017), 193–209. DOI: 10.1016/j.jnca.2017.03.005.

[110] Oracle. 2024. JAR File Overview. Retrieved 07/08/2024 from https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html.

[111] Oracle. 2023. jarsigner. Retrieved 05/22/2023 from https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jarsigner.html.

[112] Oracle. 2023. Signed JAR File. Retrieved 05/09/2023 from https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#Signed_JAR_File.

[113] Oracle. 2006. Summary of Tools for Java Platform Security. (July 2006). Retrieved 05/22/2023 from https://docs.oracle.com/javase/8/docs/technotes/guides/security/SecurityToolsSummary.html.

[114] Oracle. 2024. The Java Language Environment. Retrieved 07/08/2024 from https://www.oracle.com/java/technologies/introduction-to-java.html.

[115] Oracle. 2013. The Java® Virtual Machine Specification - Chapter 4. The class File Format. (February 2013). Retrieved 07/08/2024 from https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html.

[116] David Ortinau, David Britch, Julius Zint, Craig Dunn, Nick Schonning, and Alex Soto. 2017. Creating Bindings with Objective Sharpie. (October 2017). Retrieved 03/05/2024 from https://learn.microsoft.com/en-us/xamarin/cross-platform/macios/binding/objective-sharpie/.

[117] Stack Overflow. 2020. Difference between AAR, JAR, DEX, APK in Android. (March 2020). Retrieved 07/08/2024 from https://stackoverflow.com/questions/33533370/difference-between-aar-jar-dex-apk-in-android.

[118] Passes Alliance. 2024. Wallet Alliance - Passbook Wallet. (April 2024). Retrieved 04/25/2024 from https://play.google.com/store/apps/details?id=com.passesalliance.wallet.

[119] Patrick Ahlbrecht. 2023. Raccoon Website. (June 2023). Retrieved 08/17/2023 from https://raccoon.onyxbits.de/.

[120] Philipp Crocoll. 2024. KeePass2Android Github Repository. (April 2024). Retrieved 04/25/2024 from https://github.com/PhilippC/keepass2android.

[121] PwC Luxembourg. 2022. EU Digital Identity: Why it makes sense and the challenges ahead. (February 2022). Retrieved 04/25/2024 from https://blog.pwc.lu/eu-digital-identity-why-it-makes-sense-and-the-challenges-ahead/.

[122] pxb1988. 2023. dex2jar. (September 2023). Retrieved 09/26/2023 from https://github.com/pxb1988/dex2jar.

[123] Raiffeisen Österreich. 2024. Mein ELBA-App. (April 2024). Retrieved 04/25/2024 from https://play.google.com/store/apps/details?id=at.rsg.pfp.

[124] Rohan Vaidya. [n. d.] Android Architecture: Layers and Important Components. Retrieved 04/10/2024 from https://www.elluminatiinc.com/android-architecture/.

[125] rovo89. 2023. XposedBridge Wiki. (September 2023). Retrieved 09/26/2023 from https://github.com/rovo89/XposedBridge/wiki.

[126] Scott Alexander-Bown. 2021. RootBeer Github Repository. (November 2021). Retrieved 01/24/2024 from https://github.com/scottyab/rootbeer.

[127] shoey63 on xda-developers.com. 2022. Comment 105 in Thread: [INFO] Play Integrity API - replacement for SafetyNet. (November 2022). Retrieved 05/26/2023 from https://forum.xda-developers.com/t/info-play-integrity-api-replacement-for-safetynet.4479337/page-6#post-87663135.

[128] Vasily Sidorov and Wee Keong Ng. 2015. Transparent Data Encryption for Data-in-Use and Data-at-Rest in a Cloud-Based Database-as-a-Service Solution. In *2015 IEEE World Congress on Services*. IEEE Computer Society, New York, NY, USA, pp. 221–228. DOI: 10.1109/SERVICES.2015.40.

[129] skylot. 2023. jadx. (September 2023). Retrieved 09/26/2023 from https://github.com/skylot/jadx.

[130] Manu Sporny, Dave Longley, and David Chadwick. 2021. Verifiable Credentials Data Model v1.1. W3C Recommendation. (June 2021). Retrieved 10/24/2023 from https://www.w3.org/TR/vc-data-model/.

[131] SQLite. 2020. SQLite Release 3.32.2 On 2020-06-04. (June 2020). Retrieved 05/05/2023 from https://www.sqlite.org/releaselog/3_32_2.html.

[132] SQLite. 2007. SQLite Release 3.4.0 On 2007-06-18. (June 2007). Retrieved 05/04/2023 from https://www.sqlite.org/releaselog/3_4_0.html.

[133] Tap2Pay. 2024. WalletPasses - Passbook Wallet. (February 2024). Retrieved 04/25/2024 from https://play.google.com/store/apps/details?id=io.walletpasses.android.

[134] The Apache Software Foundation. 2023. Apache Cordova Overview. Retrieved 06/23/2023 from https://cordova.apache.org/docs/en/10.x/guide/overview/.

[135] The kernel development community. 2023. fs-verity: read-only file-based authenticity protection. Retrieved 05/23/2023 from https://www.kernel.org/doc/html/latest/filesystems/fsverity.html.

[136] The kernel development community. 2023. Seccomp BPF (SECure COMPuting with filters). Retrieved 09/26/2023 from https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[137] Mary Theofanos. 2020. Is Usable Security an Oxymoron? *Computer*, 53, 2, 71–74. DOI: 10.1109/MC.2019.2954075.

[138] Tony Smith. 2005. DVD Jon: buy DRM-less tracks from Apple iTunes. (March 2005). Retrieved 10/24/2023 from https://www.theregister.com/2005/03/18/itunes_pymusique.

[139] Transmit Security. [n. d.] Simplify and Secure Account Recovery. Retrieved 04/22/2024 from https://transmitsecurity.com/solutions/account-recovery.

[140] Trulioo. 2024. What Is Digital Identity Verification? Retrieved 04/22/2024 from https://www.trulioo.com/identity-verification-use-cases/digital-identity-verification.

[141] Trust Wallet. 2024. Trust: Crypto & Bitcoin Wallet. (April 2024). Retrieved 04/25/2024 from https://play.google.com/store/apps/details?id=com.wallet.crypto.trustapp.

[142] UC Berkeley. [n. d.] Backing Up Your Data. Retrieved 04/22/2024 from https://security.berkeley.edu/education-awareness/backing-your-data.

[143] US Geological Survey. [n. d.] Backup & Secure. Retrieved 04/22/2024 from https://www.usgs.gov/data-management/backup-secure.

[144] Zap Solutions. 2024. Zap Android Github Repository. (April 2024). Retrieved 04/25/2024 from https://github.com/LN-Zap/zap-android.

[145] Zap Solutions, Inc. 2021. zap-android Github Repository: PrefsUtil.java. (December 2021). Retrieved 05/26/2023 from https://github.com/LN-Zap/zap-android/blob/acffc0ec81320d08c2bd4686991f0d2fa1a26a45/app/src/main/java/zapsolutions/zap/util/PrefsUtil.java.