JOHANNES KEPLER
UNIVERSITY LINZ

Author
Manuel Pöll, BSc
11707301

Submission
**Institute of
Networks and Security**

Thesis Supervisor
Univ.-Prof. Priv.-Doz. Dr.
René Mayrhofer

Assistant Thesis
Supervisor
Dr. **Michael Roland**

September 2022

Towards a Privacy-focused Biometric Identity System Through a Personal Identity Agent 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

A Personal Identity Agent (PIA) is a digital representative of an individual and enables their authentication in the physical world with biometrics. Crucially, this authentication process maximizes privacy of the individual via data minimization. The PIA is an essential component in a larger research project, namely the Christian Doppler Laboratory for Private Digital Authentication in the Physical World (Digidow). While the project is concerned with the overall decentralized identity system, spanning several entities (e.g. PIA, sensor, verifier, issuing authority) and their interactions meant to establish trust between them, this work specifically aims to design and implement a PIA for Android. The latter entails three focus areas: First, an extensive analysis of secret storage on Android for securely persisting digital identities and/or their sensitive key material. Specifically, we are looking at the compatibility with modern cryptographic primitives and algorithms (group signatures and zero knowledge proofs) to facilitate data minimization. Second, we reuse existing Rust code from a different PIA variant. Thereby we analyze and adopt a solution for language interoperability between the safer systems programming language Rust and the JVM. And third, we strengthen the trust in our Android PIA implementation by evaluating the reproducibility of the build process. As part of the last focus area we uncovered and fixed a non-determinism in a large Rust library and subsequently achieved the desired reproducibility of the Android PIA variant.

Kurzfassung

Ein Personal Identity Agent (PIA) ist ein digitaler Repräsentant einer Person und ermöglicht dessen Authentifizierung in der physischen Welt durch Biometrie. Entscheidend ist hierbei die Maximierung der Privatspähre der Person mittels Datensparsamkeit. Der PIA ist eine essentielle Komponente in einem größeren Forschungsprojekt, nämlich dem Christian Doppler Labor für Private Digitale Authentifizierung in der Physischen Welt (Digidow). Während das Projekt sich mit dem gesamten dezentralen Authentifizierungssystem befasst, bestehend aus mehreren Entitäten (z.B. PIA, Sensor, Verifier und Ausgabestelle) und deren Interaktionen zum Schaffen von Vertrauen, widmet sich diese Arbeit dem Design und der Umsetzung eines PIA für Android. Letzteres beinhaltet drei Schwerpunkte: Erstens, eine ausführliche Analyse zur Speicherung geheimer Daten auf Android, um digitale Identitäten und/oder deren vetrauliches Schlüsselmaterial sicher zu persistieren. Konkret betrachten wir dabei die Kompatiblität mit modernen kryptographischen Primitiven und Algorithmen (Gruppensignaturen und Zero-Knowledge-Proofs) zur Förderung der Datensparsamkeit. Zweitens, verwenden wir bestehenden Rust-Quellcode von einer anderen PIA-Variante. Dadurch Analysieren und Verwenden wir Programmierspracheninteroperabilität zwischen der sichereren Systemprogrammiersprache Rust und der JVM. Und drittens, stärken wir durch die Evaluierung der Reproduzierbarkeit des Build-Prozesses das Vertrauen in unsere Android PIA Implementierung. Als Teil des letzten Schwerpunktes haben wir in einer großen Rust Bibliothek einen Nichtdeterminismus gefunden und behoben. Dadurch wird in weiterer Konsequenz die Reproduzierbarkeit der Android PIA-Variante erreicht.

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 for Digital and Economic Affairs (BMDW), 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 Figures

Chapter 1

Introduction

1.1 Motivation

An authentication process, entailing both identification – the action of claiming a certain identity – and authentication – the action of verifying a claimed identity – is a crucial part in many daily actions. Such a process takes place in the physical world and/or in the digital one. We observe that a purely physical verification process requires carrying several documents with specific purposes. E.g.

- operating a car requires a driver's license,
- crossing a border requires a passport, and
- enrolling in university (in Austria) requires a photo ID and a high school graduation certificate (or similar).

On the surface level this trend of specific credentials for specific purposes is even more pronounced in the digital realm. Consider that most websites or apps require the creation (i.e. registration) of specific credentials per user. However, many registration and login procedures permit the use of single sign-on (SSO) or federated identity management (FIM) processes. Common examples include:

- Many large platforms (e.g. Google, Facebook, or Apple) act as identity provider in FIM solutions for numerous third party applications.
- Additionally, government schemes also provide SSO functionality (e.g. Austrian "Handy-Signatur" allows signing in to "Digitales Amt", "FinanzOnline" and the public health portal) or even FIM (e.g. the **e**lectronic **ID**entification, **A**uthentication and trust **S**ervices (eIDAS) EU regulation enables login with national government identities at other EU member countries; each country is both identity provider and relying party).

Such identity systems vastly increase usability since a user only needs to remember a few core credentials.

Nowadays, most verification processes involve both the physical and digital realm to some degree. Note that e.g.

- a driver's license is usually checked against a digital database,
- in addition to checking a passport, a border agent commonly verifies that a traveler is not on a blocklist and there is no active search warrant, etc.

This aspect of duality between the physical and digital realm for verification is on the rise. One current prominent example for this are the certificate schemes around COVID-19 used all over the world, most of which involve scannable barcodes[1] that are digitally signed [75]. This increasing duality is the starting point

[1] E.g. the EU Digital COVID Certificate (EUDCC) uses a QR code, one specific type of barcode.

for the vision of decentralized digital identities that can be used in the physical world.

Authentication is generally classified into three different factors:

1. knowledge factor: something the individual knows, e.g. password or PIN, graphical pattern;
2. ownership factor: something the individual has, e.g. smartcard, cell phone, hardware security token (e.g. FIDO2);
3. inference factor: something the individual is or does, e.g. fingerprint, face, retina/iris, voice pattern, gait.

The inference factor, especially the well-known fingerprint and face detection biometrics, have high adoption rates and continue to rise. A study by Cho et al. [29] showed that fingerprint detection is the most popular authentication method. 88%-93% of study participants were in favor (multiple positive and negative classifications were possible). This was because "*fingerprint is fast and convenient to use with one hand*" [29].

This master thesis is part of a larger research project, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World (Digidow). Therefore we present an overview of the larger project vision along with specific motivation for the focus of this thesis.

1.1.1 Architecture

In the Digidow project we envision a system of decentralized digital identities for the physical world. Such a system involves multiple entities, a detailed overview of the architecture is visualized in Figure 1.1. The main ones are:

- A *Personal Identity Agent (PIA)* is a digital representative of an individual and thus acts in their best interest during interactions with the other entities.
- A *Sensor* is the interface from the physical world to the digital one. It mainly reports proofs of sensor measurements for individuals to their respective PIAs.
- A *Verifier* controls a (physical) service and thus grants access upon receipt of a credential from a PIA.
- An *Issuing Authority (IA)* issues digital identities (comprised of relevant attributes) to PIAs.

Initially there are several boostrap processes meant to establish static trust between some parties. This includes

- the user initially enrolling themselves with their PIA,
- the verifier establishing trust in IA and sensor by checking their respective certificate or signature, and
- both verifiers and sensors, optionally, registering themselves in public directories that facilitate accountability.

After an individual registered with an IA, their PIA receives a digital identity. Alternatively, existing identities of the individual may be imported into their PIA. Finally, the primary interaction between the PIA, verifier, and sensor can take place to permit an individual access to a desired service:

1. The location tracking model of the PIA is following (and predicting) the movement of the individual and thus aware of their physical whereabouts. It uses that information to register biometrics to nearby sensors.

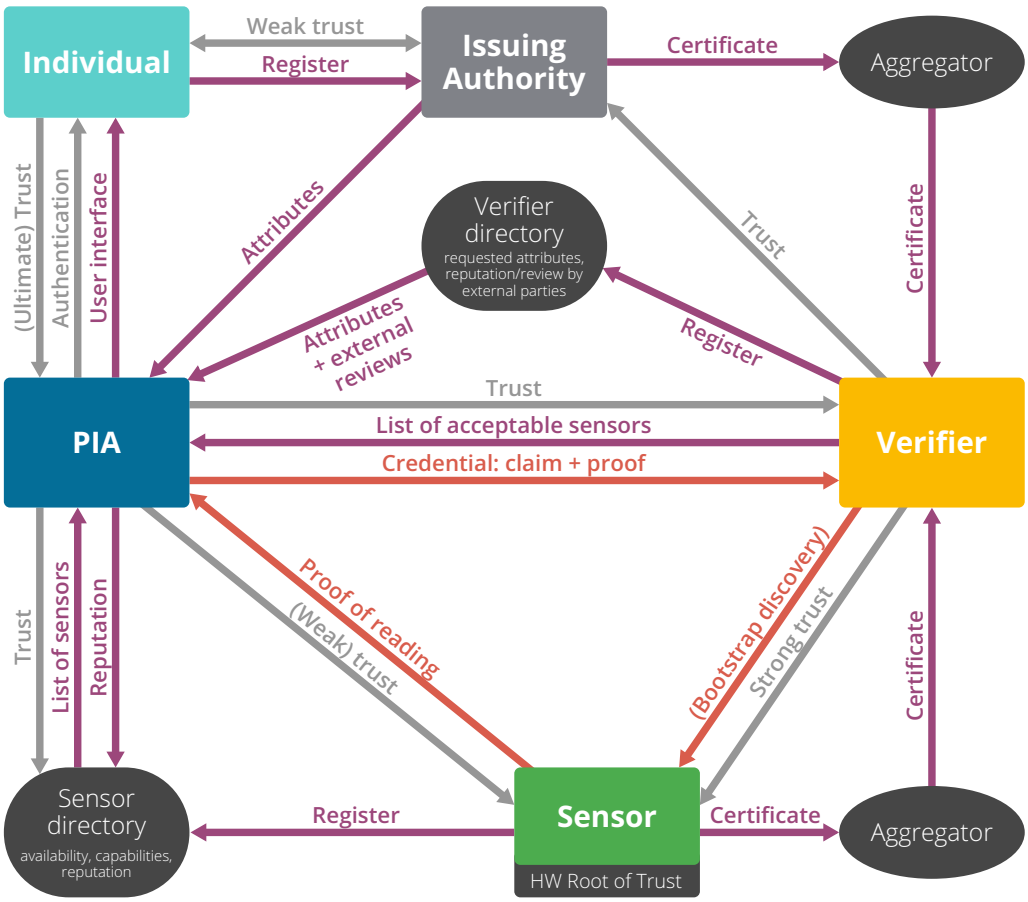


Figure 1.1: Detailed architecture of the Digidow system with all important entities and their major interactions visualized. Updated version of figure from previously published poster [82].

2. Once the user is recognized, a sensor transmits a proof of sensor measurement to the PIA.
3. The PIA combines this proof with a suitable credential, optionally employing a derived credential based on the original one (i.e. only revealing a minimum set of required properties), via a zero-knowledge proof (ZKP) and submits both to the verifier.

After the verifier has checked the combination of the submitted credential and the sensors' proof of measurement, access is either granted or denied.

1.1.2 Biometrics and Usability

The Digidow architecture is focused around biometrics, since all authentication attempts start with sensors in the physical world. As such we gain the aforementioned convenience that comes with the biometric authentication factor.

With regards to usability we also need to consider the need for interactions between the individual and PIA. As an end goal we envision a PIA that makes all decisions (in the best interest of the represented individual) on its own and thus makes carrying a smartphone optional. However, we recognize that initially many decisions are going to require feedback from the individual. This is a good fit for a PIA running on a mobile phone that can interact with its owner via a UI. Thus this work is focused on implementing a PIA for Android, the most widely adopted mobile OS for end users. With continued adoption and usage of such a smartphone based PIA we can also establish and increase trust in our system, easing the transition to a more autonomous implementation running on a remote system (e.g. in the cloud).

1.1.3 Decentralization and Privacy

Authentication in the physical world via biometrics can be trivially implemented via a centralized approach based on biometrics databases maintained by governments or big technology companies. Figure 1.2 sketches an outline how such a centralized architecture, with our terminology, might look like. However, such a centralized system has significant problems in the area of privacy and security in general, most notably among them:

- Mass surveillance of all individuals enrolled in such a system is trivial due to readily available biometrics.
- Individuals may be censored by blacklisting their credentials for one or all services offered by such a system, essentially violating the availability aspect from the perspective of an individual.
- A (possible) breach of such a large database filled with personal information is problematic on many fronts and would enable identity theft, stalking, etc. by malicious individuals.

These deficiencies are the main reasons why the Digidow project envisions a decentralized approach, granting individuals (via their PIAs) full autonomy over their data. A high level architectural view of this vision can be seen in figure 1.3.

Hansen [62] defines privacy as "*the right to informational self-determination*", effectively enabling individuals to "*control, edit, manage, and delete information about themselves and decide when, how, and to what extent that information*

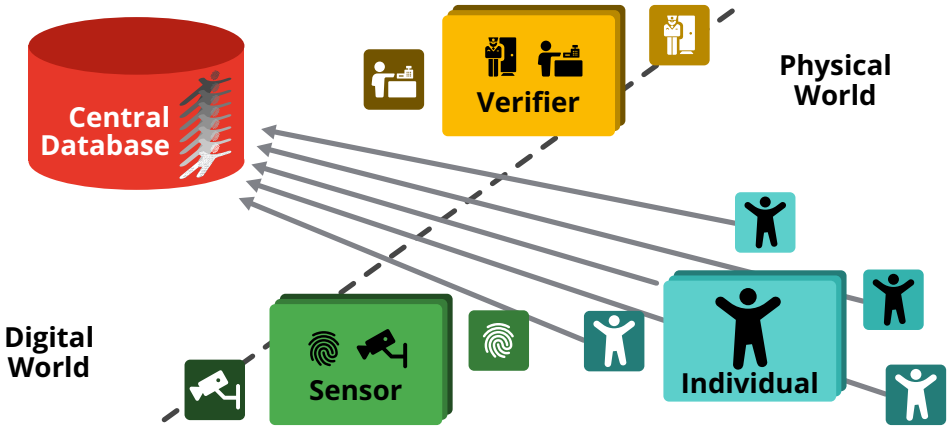


Figure 1.2: Conceptual architecture with the main entities of a centralized biometric authentication system [81]. This is a an explicit non-goal for Digidow.

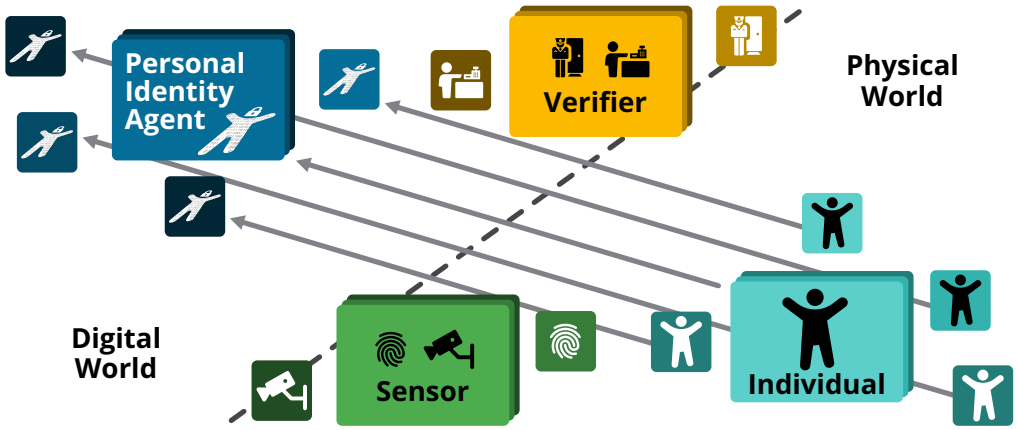


Figure 1.3: Conceptual architecture with the main entities of a decentralized biometric authentication system [81]. This is how we envision Digidow on a high level.

is communicated to others." Further work by Deng et al. [34] also worked with this definition and defined related terms and privacy properties. Their terminology distinguishes between

- hard privacy with the goal of *data minimization*, where data subjects provide as little data as possible and
- soft privacy, which assumes that data subjects have lost, or given up, control of personal data and rely on the honesty and competence of data controllers.

Any excessive data that is not required for the functionality of an action is a possible source of malicious or accidental data leakage. Thus, a key privacy goal of Digidow is data minimization.

Unlinkability, one specific part of privacy, is defined by Pfitzmann and Hansen [100] as "*unlinkability of two or more items of interest (IOIs, e.g., subjects, messages, actions, ...) from an attacker's perspective means that within the system (comprising these and possibly other items), the attacker cannot sufficiently distinguish whether these IOIs are related or not.*" A straight forward implementation of our architecture provides unique fingerprints, also called quasi-identifiers in the literature, on multiple levels. Among others, fingerprints can be found in biometrics, cryptography and network protocols. Wherever possible, we want to remove fingerprints and thus prohibit linkability per design.

As we previously introduced in section 1.1, verification in the digital realm commonly happens via a FIM system. It is worth pointing out that our envisioned decentralized identity system has similar convenience advantages as such SSO and FIM solutions. All of the digital identities and attributes are maintained by the PIA and provided wherever appropriate.

1.2 Objectives and Approach

The main objective of this work is the development of a PIA running on Android. Our general approach is adherence to secure development standards and best practices. This entails several focus areas that are briefly introduced in this section and preview the main contributions of this work.

1.2.1 Secret Storage on Android

A straight forward requirement for a PIA is the capability to store digital identities of the individual. Storing sensitive data should always be done with caution, preferably according to the security guidelines of the respective system or platform. We approach this focus area by analyzing both

- the official APIs provided by the platform vendor Google and
- taking a look at existing solutions by third parties.

We also present the basics of the Android security model, an important prerequisite for such an analysis.

1.2.2 Rust Integration for our Android app

In the Digidow project there is already an ongoing effort to develop a standalone PIA in Rust. We want to reuse existing work and, wherever possible,

share the business logic with this standalone implementation. Rust is not an officially supported programming language for Android apps[2]. Android app development supports only Java and Kotlin as first-class programming languages. It follows that this work is also concerned with programming language interoperability and, more specifically, presents our approach to use Rust code on Android apps.

1.2.3 Trust and Reproducibility

Several entities in our Digidow architecture have a trust relationship with other entities, meaning that they implicitly rely on the proper functioning of these. By design a PIA stores all digital identities and performs extensive tracking of the associated individual. Hence, the trust of the individual in their PIA is of utmost importance. Not every person is going to program their own PIA. In fact we envision that there will be a reasonable number of open source implementations that will be monitored closely by auditing efforts (e.g. driven by civil liberty unions). We assume that most individuals are going to chose among these. There are currently two planned variants of the PIA, namely:

- An *embedded* one running on the personal smartphone of the individual and
- a *standalone* (i.e. *remote*) one that is hosted on a remote server, either run by the individual themself or by a trusted (cloud) vendor (e.g. bank).

However, as famous work by Thompson [128] points out: Merely inspecting source code is definitely not enough to create trust. An important step in bridging the gap between an open source implementation and the executable artifact are reproducible builds. Thus we will investigate to what degree our Android implementation of the PIA is reproducible and work towards achieving this property.

1.3 Outline

This master thesis is structured in the following chapters. After presenting the overall vision of the Digidow project as motivation and specific objectives for the Android PIA developed as practical component of this work in this chapter, we show a selection of related scientific work, established technology, and their historic evolution in chapter 2. Afterwards, a more focused analysis of the Person Identity Agent (PIA) is performed in chapter 3, including requirements, threat model and technical measures that guide the implementation. Chapter 4 analyzes solutions for storing digital identities on Android and how they fit into the Android security model. In chapter 5 we look at the software architecture behind our app and show how we were able to leverage the modern systems programming language Rust for shared business logic that runs on Android. Chapter 6 looks at the issue of reproducibility for the embedded PIA. This covers both the theoretical basics around trust and the software supply chain, as well as an evaluation of the embedded PIA implementation for Android. Finally, in chapter 7 we conclude with a summary of our contributions and provide an outlook towards future work, both for the Digidow research project and the scientific community at large.

[2] Not to confused with the existing support for Rust in the Android Open Source Project (AOSP), which enables the creation of OS components in Rust. See https://source.android.com/docs/setup/build/rust/building-rust-modules/overview.

Chapter 2

History and Related Work

The first computers used by multiple individuals gave rise to the requirement of an authentication process. Simple user accounts for a computer evolved into digital identities that can be used for countless online services. Nowadays, many physical world authentication processes involve digital components or are in the process to move towards such approaches. This chapter presents basic terminology, the history and noteworthy related work around digital identities.

2.1 Basics of the Authentication Process

An *authentication process*, which is used to gate access to some resource, consists of two steps:

1. An individual performs *identification*, the action of presenting credentials with the intent to access a user account or claiming a certain identity.
2. These credentials are then checked by the system for validity, a process called *authentication*.

If valid, the individual has proven their control over the digital identity. While some terminology has evolved (e.g. user account vs. identity), the above definitions apply to all presented systems in this chapter.

Additionally, complex systems often feature many permissions. Mapping between identities and permissions is done via *authorizations*. These are encoded via a security model (e.g. discretionary access control (DAC), mandatory access control (MAC), role-based access control (RBAC)). While interesting in their own right, they are not a focus area for us and will only be mentioned tangentially.

2.2 User Accounts with Passwords

Passwords are the oldest authentication process used in computer science. By remembering a secret password string associated with a user account, an individual proves control over the latter. After their introduction by the Compatible Time-Sharing System (CTSS) in 1965 developed at the MIT [32], the security of password storage was augmented via the first *key derivation function (KDF)* `crypt`, entailing password hashing and salting, by Morris and Thompson [88] to enhance the security of the first UNIX systems. Even though they have been in use for over 50 years, their basic functionality remains unchanged.

The classical problem of password security is that “strong” passwords are inherently at odds with memorability by humans, thus leading to widespread use

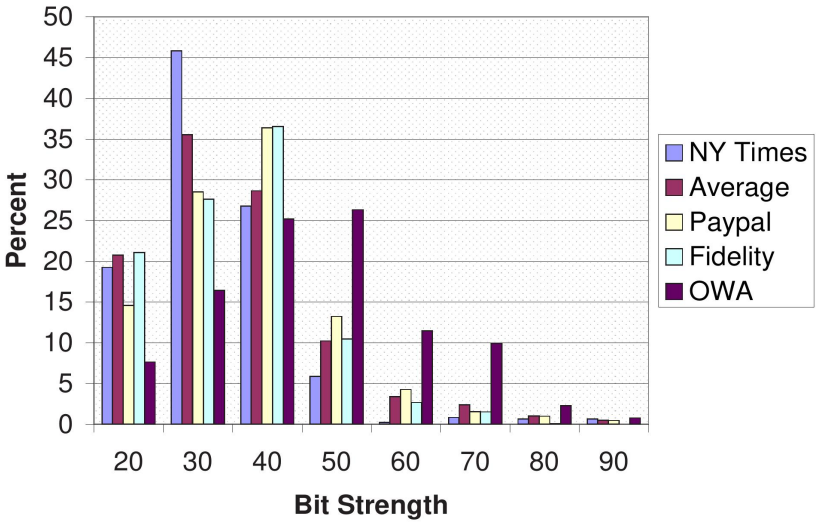


Figure 2.1: Distribution of password entropy as chosen by users across some services, grouped in buckets of 10 bits. From "A Large-Scale Study of Web Password Habits" by Florencio and Herley [40].

of weak passwords that are either short and/or guessable. One study by Voyiatzis et al. in 2011 [131] has found an average password length of 7 characters. According to another large study by Florencio and Herley [40], users tend to choose insufficient passwords even for important services. Figure 2.1 shows the distribution of password entropy for some prominent services in 2007. Even for an important service like PayPal users opted to use a password with an average entropy of mere 42 bits, which was considered weak even back then[1]. Improved KDFs have helped to a small degree. Their principle of *key stretching* requires increased computational cost for all verification attempts. At the same time adversaries have also evolved techniques to crack hashed passwords, specifically rainbow tables that trade extreme storage requirements of password dictionaries for manageable computational cost. Overall, increased computing power has shifted the requirements for a strong password to be even longer and more convoluted.

2.3 Handling Numerous User Accounts/Digital Identities

An ever increasing number of digital services require user accounts. It is no surprise that password reuse, i.e. one individual using the same password for multiple services, is widespread and a well understood problem [33, 70]. As Ives et al. [70] have found, password reuse results in a domino effect where any compromised service results in trivial access to other services where the user opted to reuse a password. We highlight two approaches to improve the usability of passwords when interacting with numerous digital services.

[1]These two references are slightly dated and research did not surface more recent empirical studies that are based on surveys of large real world applications. However, this is a good indication that technical best practices of password storage are followed and thus proper hashing makes a large scale survey of password complexity infeasible.

2.3.1 Password Managers

Password managers, also known as keychain or keyring (inspired by the eponymous physical keychain/keyring), are used to store credentials in a password database and thus permit access to multiple digital accounts. During registration for a new service the individual chooses a "strong" password, preferably utilizing an integrated password generator utility[2]. The entire credential, consisting of password, username and other optional metadata, is stored encrypted in a password database. Therefore the individual is no longer required to remember credentials for individual user accounts, only the master password that protects access to the password database itself. Subsequently each credential can (and should) use a unique password unrelated to all others, effectively solving the issue of password reuse.

A password manager can be used for arbitrary digital services. Storing credentials within it is transparent to the service and automatic insertion of credentials is available for the majority of applications, including web browsers[3]. Essentially, this approach does not require any additional effort by the service provider. Instead, each individual is required to select and configure their preferred password manager solution and manually maintain the password database.

One of the oldest password managers is the MacOS Keychain, after being originally developed and integrated into Apple's e-mail system PowerTalk in the early 1990s it was promoted as a system component, part of Mac OS 9 in 1999 [103]. Nowadays, there are a variety of password managers, each equipped with unique features (e.g. varying OS support, browser integration, multi-factor integration, etc.) The following list is categorized based on the delivery format of the application and password database:

- Local application with no integrated DB synchronization (e.g. KeePass[4], KeePassXC[5], GNOME Keyring, KWallet, etc.),
- Local application with integrated DB synchronization (e.g. 1Password[6], Bitwarden[7], LastPass[8], etc.)
- and fully cloud-based applications, including their DB (e.g. Firefox Lockwise[9], Meldium, Mitro, etc.)

2.3.2 Single Sign-On and Federated Identity Management

Single sign-on systems (SSO) require each individual to perform the authentication process only once, subsequently they can use all services that participate in the SSO system. After a user is registered, all services that use the SSO accept the previously created credential. Even better, a user with an active session should not be prompted for his credentials again when accessing a different service integrated with the SSO. Instead, the SSO system should perform a silent login

[2] Obviously users are free to divine their own passwords, but from a security perspective we recommend to use integrated utilities. The ability to choose arbitrary passwords is valuable for various scenarios, e.g. where credential usage without password manager is desirable or existing "good" credentials should be recorded as a reminder.
[3] Some rare exceptions exist, such as the Windows UAC not permitting auto-type for security reasons
[4] https://keepass.info/
[5] https://keepassxc.org/
[6] https://1password.com/
[7] https://bitwarden.com/
[8] https://www.lastpass.com/
[9] https://lockwise.firefox.com/

for the active user. Web-based SSO systems often use dedicated subdomains, which store sessions cookies or similar, for this (e.g. `account.my-corp.tld`). On a technical level SSO systems are often implemented with the help of standardized protocols. Prominent ones are:

- **O**pen **Auth**orization (OAuth) [63] is an open standard related to access delegation, i.e. performing (eponymous) authorization between parties. It enables users to grant applications access to specific data or permissions that are part of the SSO account.
- Security Assertion Markup Language (SAML) [97] is an open standard addressing both authentication and authorization between the parties in an SSO.

An SSO solution needs to be supported by the digital service being accessed. The service provider is responsible for the integration with the SSO system, whereas the individual has no additional overhead and can simply use their dedicated credential for the SSO system. Therefore, we observe a symmetry w.r.t to usability and burden of effort compared to the previously presented password managers.

Federated identity management (FIM) systems are specific SSO systems that serve loose federations, typically spanning multiple legal entities [26]. Within such a system there is one identity provider (IdP) that manages all identities in the system and is responsible for authentication of individuals. Participating service providers (SP) enable login via the aforementioned IdP, effectively outsourcing the authentication process.

The first widely adopted FIM was Microsoft Passport back in 1999 [1], which evolved into today's "Microsoft Account" (aka. Windows Live ID). Beyond Microsoft owned services (MSN, Hotmail, etc.) it was also adopted by various other sites [87], including prominent e-commerce retailers (e.g. barnesandnoble.com, Buy.com and Costco.com) back then. Nowadays the most popular federated identity services in the private industry are Facebook Connect and Google Account [60]. During Q4 of 2015 they had a market share of 62% and 24% respectively in the "social authentication market", i.e. among FIM solutions. Modernization initiatives of governments also give rise to more and improved digital services for citizens. Thus, governments also tend to offer SSO or FIM solutions, e.g.

- in Austria "Handy-Signatur", Bürgerkarte [9], and ID Austria [10] (the last is the intended replacement for the two former ones) act as SSO for government related services,
- on a higher EU level there is the **e**lectronic **ID**entification, **A**uthentication and trust **S**ervices (eIDAS) EU regulation creating a FIM and permitting access to digital government services across EU member states.

Academia has their own dedicated federated identity solutions. Most notably Shibboleth[10] for accessing various academic publishers and journals with your university credentials. But even network access is often done via the prominent eduroam WLAN[11], allowing seamless network access at all participating university premises.

The OpenID Foundation is a non-profit organization working on the development of open standards around FIM. Their original OpenID Authentication process standards [37, 111], going back to 2005, used a custom standardized protocol and are now obsolete (all versions, i.e. from 1.0 through 2.0). These

[10] https://www.shibboleth.net/
[11] https://eduroam.org/

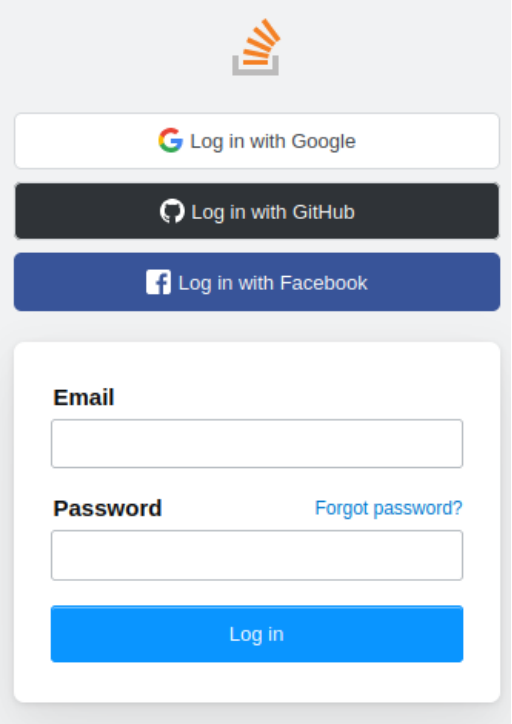


Figure 2.2: Example of a relying party (RP) login screen in a federated identity management (FIM) system (here: StackOverflow).

have been replaced with OpenID Connect (OIDC) [116]. Service providers, also known as relying parties (RP) in the OpenID terminology, can accept identities from third party identity providers (IdP). Similar to OAuth 2.0, which this standard is building on, OIDC is a technical standard that can be used to implement a FIM system. OIDC is concerned with authentication and defining a standardized set of claims around the user identity (e.g. `given_name`, `family_name`, `email`, etc.) that are exchanged between RP and IdP. In fact, starting with the introduction of OIDC in 2014, the vast majority of IdPs (e.g. Google Account, Facebook Connect, "Login in with Amazon/Twitter/Github", etc.) implemented OIDC as technical basis for their FIM systems [65]. However, instead of allowing logins with arbitrary OIDC IdPs, the relying parties typically limit the permitted IdPs to a group of well-known ones (see Figure 2.2 for an example). The OIDC standard even included Self-Issued OpenID Providers (SIOP), a personal, self-hosted IdP.

According to Allen [1] the next evolution after federated identity is *user-centric identity*, which is defined as "*individual or administrative control across multiple authorities without requiring a federation*". The vision of OIDC with SIOP would have been one such identity system. Unfortunately it was not realized due to restrictive RPs that limit the choice of supported IdPs [65].

2.4 User-Centric Identity Systems

All identity systems presented up to this point manage their digital identities (i.e. user accounts) either via a centralized instance or within a federation. First user-centric identity systems that shifted the creation and management of identities to the user are old. It is worth highlighting these systems

to show their fundamental differences to the previously presented approaches w.r.t. their trust model. These identity systems have their own unique strengths and weaknesses.

2.4.1 SSH Identities and the SSH Agent

Secure Shell (SSH) is a protocol to manage remote servers over an insecure network. The actual functionality of the SSH protocol, issuing commands to remote servers, is not relevant for our work. We are interested in the authentication process between SSH client and server. Although there is support for password-based authentication of users, it is commonly discouraged in favor of using SSH identities. These are public-private keypairs that are generated and managed by SSH clients. They are used to authenticate the client, who holds the private key, to the server, which has a list of authorized public keys. To protect the sensitive private key of an SSH identity, it is stored in an encrypted format and requires the individual to enter a passphrase to unlock it. The authors of SSH recognized that this scheme, requiring the user to enter the passphrase for all usages of an SSH identity, is not very convenient. Thus, there is the helper program `ssh-agent` that maintains unlocked SSH identities in memory. After startup of the former, one can add and unlock SSH identities via `ssh-add`. Subsequently any usage of the SSH protocol checks the running SSH agent for unlocked SSH identities, avoiding repeated entry of the passphrases for the same SSH identities. The `ssh-agent` utility has been part of the prominent OpenSSH implementation at least since their inception in 1999[12], back when OpenSSH was born as fork of the original SSH version 1.2.12 [92]. However, note hat our definition of an agent requires an active role of the software without explicit user input (cf. chapter 3) and thus an "SSH agent" should not be confused with or considered equivalent to a personal identity agent (PIA) in this work.

A fundamental difference to the previously presented identity systems is the trust model. The SSH server trusts a client because an authorized public key was previously installed via an out-of-band channel[13]. On the flip side, the SSH client relies on a trust-on-first-use (TOFU) authentication approach [132]. In case of SSH this trust model entails that the application user manually checks the presented server fingerprint via an out-of-band channel and thus confirms that the client established a connection with the legitimate server (instead of a MitM attacker). The SSH server fingerprint is then remembered in a `known_hosts` file and ensures authentication of the server to the client for future connections. This user-centric, i.e. decentralized, approach to identity management is possible due to the asymmetric cryptography involved. This is in stark contrast to centralized or federated identity systems. The latter place ultimate trust in a central identity provider that can arbiter over account validity as it pleases, whereas the former is a peer-to-peer approach without central authority.

One way of highlighting the novel nature and power of a user-centric identity system, like SSH, is by comparing it with previously presented use cases:

- A very simple usage of SSH only employs a single SSH identity for a real-world individual. In this case, the individual unlocks their singular SSH identity within in the SSH agent and can subsequently access all remote

[12] We confirmed the presence of the tool in the `openssh-1.0pre2-linux.tar.gz` source code release.

[13] A practical way of doing this is the usage of the same insecure network channel, but using the password based SSH authentication. After initial key provisioning the password-based authentication should be disabled.

servers where they are registered. Effectively, this type of usage makes the SSH agent behave as an SSO system.

- However, an individual may add any number of SSH identities to an SSH agent. A usage where an individual has multiple unlocked SSH identities means that the SSH agent behaves similar to a password manager. While the SSH agent manages multiple SSH identities, there is, unlike with real password managers[14], no master password[15].

In summary, one can see that SSH identities are a powerful identity system that can be used as a SSO with some features of a password manager.

User-centric identity systems without any central trust come with new challenges. A specific one for SSH identities is the transitive trust between multiple SSH agents running on different nodes in a network, i.e. SSH agent forwarding: Connecting to a remote host H_1 that also has an SSH agent, the remote instance of the SSH agent is granted access to all the identities in the local one, allowing convenient access to other hosts $H_2, H_3, \ldots$ over SSH via jump host H_1. However, Kogan et al. [77] observe that SSH agent forwarding infers full trust to a remote host. If the aforementioned jump host H_1 was compromised, an attacker with system-level privileges can issue arbitrary signing requests to the local ssh-agent and thus impersonate the user. Kogan et al. developed the Guardian Agent, which is a delegation system with granular control over what remote hosts may execute which actions, drastically limiting the attack surface. Notably, the official OpenSSH implementation has now adopted a similar feature [93], allowing user-defined restrictions on the usage of SSH identities for different purposes. This is based on the hostname and is part of the OpenSSH 8.9 release.

2.4.2 Pretty Good Privacy and Web of Trust

OpenPGP is an open RFC-based standard[16] [38, 71, 122] that uses asymmetric cryptography to securely exchange private messages and files. The meat of PGP is how to establish trust from Alice to Bob (and vice versa) that enable the secure exchange of sensitive data, i.e. performing an authentication process (in both directions). In an initial setup step Alice creates a PGP identity, consisting of a public-private key pair, associated (user-)name and e-mail address. The authentication process and message exchange consists of the following three steps:

1. The desired communication partner Bob also has a PGP identity and Alice receives his public key (via an out-of-band channel).
2. A message or file sent from Alice to Bob is both digitally signed (using the private key of Alice) and then encrypted (using the public key of Bob).
3. After receiving the file or message over an insecure channel, Bob is performing decryption (using his private key) and checking the digital signature (using Alice's public key).

[14] The fact that the SSH agent by itself cannot automatically select the correct SSH identity for a given host is not really a distinction to a password manager. One needs to specify the intended SSH identity (either during CLI usage or in a host configuration file), which is the equivalent to configuring a password manager entry with metadata that allows association from credential to application or website (auto-type window name or URL).

[15] This detail means that it can be useful to store SSH identity passphrases in a regular password manager, especially if a user has many SSH identities and wishes to rely solely on a single master password. E.g. KeePassXC features integration with the SSH agent, unlocking and locking SSH identities along with the password database, see https://keepassxc.org/docs/KeePassXC_UserGuide.html#_ssh_agent.

[16] Based on the original Pretty Good Privacy (PGP) program by Phil Zimmerman.

Performing all these steps ensures the CIA properties of a secure channel, i.e. confidentiality, integrity and authentication, as well as non-repudiation for this exchange[17].

As usual with asymmetric cryptography, the hard problem is key management and the trust model born from a decentralized identity system. The lack of a central governing authority, or at least a list of implicitly trusted root identities (as is the case with the public key infrastructure (PKI), consisting of root certificate authorities (CA) used by TLS), is both a curse and blessing at the same time. Initial distribution of the public portion of identities (public key + username + e-mail) is commonly done via websites or dedicated keyservers[18]. Anyone may create and publish any PGP identity, giving rise to the possibility to impersonate others.

A trust signature is an assertion by one individual (via their PGP identity) that attests that another PGP identity does indeed belong to the individual named in the metadata. As a prerequisite for secure communication one needs trust signatures for each communication partner, either signed by oneself or via a trusted proxy (that acts similar to a CA in a PKI). This ensures that the initial communication between two parties can be properly authenticated, i.e. it does not rely on the TOFU trust model. The graph of trust signatures among all participating PGP identities is the *web of trust*. This complicated system is a key reason for the bad usability of PGP [112, 133], especially among average users. At the same time, the web of trust cannot be undermined by any single entity, since users establish trust to the PGP identities themselves in a peer-to-peer fashion.

A noteworthy aspect of asymmetric cryptography, as introduced with user-centric identity, is the unavoidable need to persistently store the private key in such a way that it remains accessible to the actual signing/decryption process. This is noteworthy, because credentials for classical logins (including a SSO or FIM system) can be protected via password hashing schemes (i.e. password key derivation functions) at rest and thus, provided proper usage and work factors, are well protected against extraction. To address this shortcoming, one can employ dedicated hardware, like smart cards (cf. section 3.1.3), that either generate or import the private keys into security hardware. From then on, cryptographic operations are performed inside the secure hardware and keys can no longer be exported (i.e. they are "sealed" into hardware). Facilitating this usage for PGP keys, there is a *Functional Specification of the OpenPGP application on ISO Smart Card Operating Systems* [102]. One noteworthy implementation of this OpenPGP specification, aside from classical smart cards, is GoKey[19], with support for sealing and usage of PGP and SSH keys. It is based on the TamaGo framework[20], which enables the execution of bare metal Go binaries on ARM SoCs. Said software was originally developed for the USB Armory[21], a tiny single board computer with focus on security features that is connected to a regular computer via USB. The authors of GoKey envision that a software stack that solely relies on the managed language Go is slimmer and less susceptible to memory-related attacks found in C/C++.

[17] This intentionally skips over PGP details like subkeys, different key types (**C**ertification, **S**igning, **E**ncryption, etc.) and other auxiliary features, instead focusing on the core use case.

[18] One popular website is https://keybase.io. Dedicated keyservers are exclusively concerned with the distribution of PGP public keys, e.g. Symantec (current owners of the original PGP software) runs https://keyserver.pgp.com/, OpenPGP runs https://keys.openpgp.org/.

[19] https://github.com/usbarmory/GoKey

[20] https://github.com/usbarmory/tamago

[21] https://github.com/usbarmory/usbarmory/wiki

2.4.3 Partial Solutions with (Some) Centralized Trust

The two presented identity systems are extreme examples of their user-centric nature. In addition to enabling management and creation of identities by anyone, they abstain from using centralized or federated trust. Instead, their peer-to-peer nature makes them highly resistant to centralized control and censorship. Furthermore, the two specifically highlighted systems are relevant to a large user base and continue to evolve.

The need for an initial out-of-band exchange in a TOFU trust model and the complexities around key management hinder mainstream adoption of "pure" user-centric identity systems without centralized trust (cf. PGP usability). Many modern FIM systems adopted some weak aspects of user-centric ones. E.g. Facebook Connect and Google Account have permission systems that give users control over what account data/control is shared with which service providers. The OpenID standards were heavily inspired by the movement towards user-centric identity. This manifests itself in the previously noted support for multiple independent identity providers.

2.5 Self-Sovereign Identity

Self-sovereign identity (SSI), as introduced by Allen [1], is the next evolution of digital identities beyond user-centric identity approaches and emphasizes true user control and autonomy, portability of identities, and capability to make complex claims. More precisely, his ten proposed principles of SSI were categorized by Tobin and Reed [129] into three groups and are a starting point for further academic work by Mühle et al. [89]. The aforementioned are:

- Security: The identity information must be kept secure, entailing protection, persistence, minimisation.
- Controllability: The user must be in control of who can see and access their data, entailing existence, persistence, control, consent.
- Portability: The user must be able to use their identity data wherever they want and not be tied to a single provider, entailing interoperability, transparency, access.

Achieving these goals is not straight forward. Many projects, (non-profit) organizations, and interest groups across the industry and academia have emerged and aim to create modern identity systems that realize the vision of SSI. We highlight a selection of (supra)national initiatives and briefly explain their main objectives to give a clearer overview of the subject.

2.5.1 International and Supranational Initiatives

The Decentralized Identity Foundation (DIF) [72] is intended as a place to collaborate around decentralized identity. Based on discussions and experiments it aims to cultivate ideas and create specifications. The vision is focused on enabling "*a world where decentralized identity solutions allow entities to gain control over their identities and allow trusted interactions*" [72]. To this end, they want to realize this goal via an open-source implementation-driven approach and develop an interoperable identity stack. Subsequently, mature concepts and specifications are meant to be included or guide standardization work in this

area. According to them, SSI systems are a subset of decentralized identity systems[22]. Specifically, SSI systems are used to address human identity use-cases and require sovereignty and privacy. Beyond that, decentralized systems also aim to support collective non-human identities. The exact boundary between SSI decentralized systems is complicated by the fact that sovereignty and privacy are social constructs. Since their focus is technological, they are supportive of any decentralized identity system, whether they feature SSI qualities or not.

The Sovrin Foundation [123] aims to create an identity layer for the internet and wants to offer a global public utility that enables creation and usage of digital identity for people and other entities (e.g. organizations and devices). Specifically, they develop the Sovrin Network (short "Sovrin"), "*a public service utility enabling self-sovereign identity on the Internet*" [123]. Their vision entails the possibility for individuals to hold identity credentials without a strict dependency on siloed databases that control access to them. Nodes of this network can be found in several countries spanning four continents, underlining their international global focus.

The European Self-Sovereign Identity Framework (ESSIF) project [99] by the European Commission aims to develop a generic and interoperable self-sovereign identity (SSI) system, entailing both specification and the supporting infrastructure that is needed for citizens to control their own identity. It is designed to interact with other systems and platforms across public and private organizations. GDPR-compliance and alignment with the existing eIDAS regulation are explicit goals. ESSIF is part of the European Blockchain Services Infrastructure (EBSI) [35], an initiative to "*leverage blockchain to the creation of cross-border services for public administrations and their ecosystems to verify information and make services trustworthy*".

2.5.2 "Schaufenster Sichere Digitale Identitäten"

The project "Schaufenster Sichere Digitale Identitäten" by the German federal ministry for economic affairs and climate action [41] aims to develop German eIDAS solutions. It is setup as a competition and after an initial step there are four showcase projects currently being implemented:

- ID-Ideal[23],
- IDunion[24],
- ONCE - Online einfach anmelden[25], and
- SDIKA[26].

All projects mention SSI either as goal or understand it as building block to enable their respective vision of digital identity.

[22] https://identity.foundation/faq/#is-decentralized-identity-different-from-self-sovereign-identity
[23] https://id-ideal.de/
[24] https://idunion.org/
[25] https://once-identity.de/
[26] https://www.sdika.de/ and https://www.digitale-technologien.de/DT/Redaktion/DE/Standardartikel/SchaufensterSichereDigIdentProjekte/sdi-projekt_sdika.html

2.6 Further Noteworthy Standards and Projects

There are many more past and current digital identity projects beyond the ones listed so far. Additional ones that are relevant and/or related to our work, but do not fit well into the previous sections are listed here. However, the digital identity space is vast and we cannot possibly offer an exhaustive list of projects.

2.6.1 Digital Wallets and Trusted Service Managers

A digital wallet, i.e. e-wallet, is an application or online service on a mobile phone that (originally) enabled monetary payment for goods and services [64], but has in some instances evolved to include functionality of digital identity systems. The origins of digital payment (on mobile phones) can be traced back to 1997: Coca-Cola installed two vending machines in Helsinki that allowed payment via text messages [78, 114].

Google Wallet[27] was introduced in 2011 as one of the first digital wallets, being one of two notable available options on the US market back at that time [119], and allows storage of a dedicated virtual EMV card in a secure element (SE). Subsequently, this virtual EMV card can be used like a classical one to perform payments with a mobile phone. This was implemented via Near Field Communication (NFC), a wireless technology for close proximity data exchange.

On the flip side, the introduction of Apple Passbook, later called (Apple) Wallet, in 2012 did originally not support any payment, but was used for the storage of identity documents (e.g. boarding passes, coupons and tickets). Two years later, in 2014, the Apple Pay mobile payment system was added. By storing existing EMV card details in the Apple Wallet, users are able to perform payment as if they were using the physical plastic cards. Similar to Google Wallet, NFC is the technology that enables usage of mobile phones as a drop-in replacement for traditional wireless EMV card payment.

The aforementioned digital wallet applications were the original reason for including SEs into mobile phones. While the procedure for loading applets onto the SE (cf. section 3.1.3) depends on the specific SE issuer [2], it commonly anchors trust in the SE issuer by cryptographic means. E.g. Loading and execution of a third party applet may require a MAC with a symmetric key only known to the SE issuer or a digital signature rooted in a key pair only known by the latter. An SE issuer may delegate organizational control to a Trusted Service Manager (TSM). The latter is a neutral broker that is intended to provide access to the SE for legitimate third party applications (e.g banking or a public transport agency). Unfortunately, open TSM infrastructure did not gain widespread adoption in the industry yet and some TSM initiatives were even discontinued [14]. The absence of TSMs with a wide market reach essentially force third parties to interact with multiple SE issuers directly. As a consequence, only third parties with significant political capital have a realistic chance to be approved. A single third party needs to collaborate with all SE issuers with noteworthy market shares. Otherwise, this gives rise to fragmentation where a share of technically capable phones are not supported[28]. Later,

[27] Later in 2018, the functionality of this wallet was simultaneously rebranded as a standalone "Google Pay Send" app and integrated into a unified app called "Google Pay" [11, 25]. While the standalone Google Pay Send was discontinued in 2020, the unified app was renamed back to "Google Wallet" in 2022 [76].

[28] For specific use cases that may be acceptable. Consider the recently announced collaboration between the German BSI and Samsung as mobile phone OEM at https://news.samsung.com/de/samsung-und-bsi-intensivieren-zusammenarbeit. As a federal agency, the BSI can simply enforce the deployment of eSE-equipped Samsung mobile phones to employees. Subsequently, the custom applet can be loaded into the SEs due to sanctioning of Samsung.

host card emulation (HCE) enabled the storage and usage of virtual cards without dedicated SE hardware in phones.

Related to this TSM subject is the project "OPTIMOS - Sichere Identitäten für mobile Dienstleistungen" [21] by the German Bundesdruckerei. Secure storage of digital identities, e.g. public transport tickets, boarding pass, etc. on a smartphone is tricky due to the heterogeneous landscape of mobile devices and operating systems. In the past, each provider has created their own complex infrastructure for this purposes. OPTIMOS set out to simplify this situation by developing a TSM applet running either on a eSE or eUICC and an associated management app. Providers can use the interface of the TSM to store personal identities and other sensitive data in a secure fashion, utilizing dedicated hardware wherever available.

A second, refined iteration of this project, called "OPTIMOS 2.0 - Entwicklung der offenen, praxistauglichen Infrastruktur für mobile Services" was focused on creating a generic ecosystem for the creation of TSMs. After completion, the project became the basis of BSI (currently draft) *TR-03165 Trusted Service Management System* [20]. Said guideline defines a Trusted Service Management System (TSMS), which aims to streamline the operation and adoption of TSMs.

2.6.2 Mobile Driving License and Beyond

Driving licenses are one type of identity document that today is predominantly issued and presented in an analogue format, but the recently finished ISO/IEC 18013-5 mobile driving licence (mDL) application standard [68] establishes an interface specification for *mobile driving license (mDL)* applications and thus paves the way for a broader adoption of digital driving licenses. The standard is focused on the verification of an mDL. This involves the connection of the license holder with the mDL, a machine representation and digital transfer of the mDL data via an mDL reader, and verification of the received mDL data by the verifier. Provisioning of mDLs, an essential component of a full implementation, is not in scope of this standard. However, the standard does mention that *the transaction and security mechanisms in this document have been designed to support other types of mobile documents, specifically including identification documents*, short *mdoc*, which hints at the possibility of usage beyond the scope of driving licenses. As technological ground work both Apple (iOS 15 via Apple Wallet [5]) and Google (Jetpack support library - Identity Credentials API [56]) have introduced support for the ISO standard. Several states in the USA [5] aim to introduce digital driving licenses on this basis during 2022.

The ISO/IEC 23220 [69] standard series on building blocks for identity management via mobile devices is currently being developed. It inherits and enhances the generic parts of the ISO/IEC 18013-5 standard. Specifically, the first part of the new ISO standard series "*specifies generic system architectures and generic life-cycle phases of mobile eID systems in terms of building blocks for mobile eID system infrastructures and normalizes interfaces and services for mobile eID-Apps and mobile verification applications*" [69]. The standard series is targeting a wide range of applications (e.g. health assurance cards, payment cards, government IDs, electronic passports, driver license, etc.), but does not preclude specialized standards that target specific application areas.

2.6.3 COVID-19 Certificate

A contemporary topic are certificate schemes around COVID-19, used all over the world to prove vaccinations status, a negative diagnostic test result or re-

covery from a past infection. There is a survey paper by Karopoulos et al. [75] that looks at the landscape of certification schemes both in real-world deployments by governments and proposals by academia. On a technical level, schemes presented in the survey can be divided into

- blockchain (e.g. AOKpass, Cerify.health initiative, and IBM Digital Health Pass)
- and public key approaches that root their trust into a PKI (e.g. WHO Smart Vaccination Certificate, EU Digital COVID Certificate (EUDCC), and CommonPass).

They observed a trend that most schemes permit the user to present their proof either paper-based or paperless, facilitating usability. Overall Karopoulos et al. found that the EU Digital COVID Certificate (EUDCC) provide better privacy than comparable schemes from Asia and America. This is a notable example on the combination of digital signatures for a physical certificate.

Chapter 3

Personal Identity Agent

On a high level, a personal identity agent (PIA) is a digital representative of an associated individual. It is a key component in the larger research project explained in section 1.1.1. More precisely, we define the term as follows [83] "*A hardware/software system actively representing individuals in their Digidow interactions with sensors, verifiers, and issuing authority (IA). A PIA is under control of the individual it represents and assumed to act in the best interests of that individual. The PIA collects and manages claims from different issuers (such as IA, sensors, or self-issued by the PIA).* "

3.1 Concepts and Technology for Digital Identity

There are several academic research areas and technological standards that cater to the sphere of digital identities. We present a small selection of these as preparation for the architectural work on PIAs in the next section. Effectively, these are building blocks of existing work that we want to build on and incorporate into our vision.

3.1.1 Zero-knowledge Proofs as Enabler for Privacy

A *zero-knowledge proof (ZKP)* is a cryptographic protocol that allows one entity, called prover, to prove to a different entity, the verifier, that a specific assertion is true, while minimizing the exposure of unrelated data [47, 48]. If the secret s is a digital identity, we call an assertion over it a *derived credential* or attribute based credential [59]. Derived credentials based on ZKP are essential tools to achieve data minimization, i.e. only exposing the minimal amount of data required for specific actions. Notably, this may include a "randomization" of the digital signature, meaning that the same proof originating from the same credential produces different, but nevertheless valid, signatures for each creation. Such a process maintains the cryptographic security guarantees of regular digital signature algorithms, but makes the signature unsuitable for fingerprinting the user. We introduce two different group signature schemes, building blocks for non-interactive ZKPs [108], distinguished by their powerfulness.

The CL signature scheme, by Camenisch and Lysyanskaya, is both a group signature scheme [23] and an anonymous credential system [24]. The prover has access to a secret s, which could be anything (e.g. simply a password string, collection of related digital identities, each consisting of attributes), and is signed by a trusted authority via a ZKP protocol with the signature sig_s. They want to convince the verifier of a specific assertion, i.e. predicate, $p(s)$ without simply disclosing the secret s and associated signature sig_s itself. The distinguishing

feature of a ZKP based on a CL signature over a standard asymmetric cryptosystem used for message signing is the possibility to derive a signature $sig_{p(s)}$ that asserts only the true statement $p(s)$. The exact capabilities of the predicate depend on the group signature scheme and the ZKP. A well-known capability that is available in CL are *range proofs*, i.e. proving an inequality (e.g. $x > threshold$) based on a secret value x without actually revealing the raw value. After transmitting the derived signature $sig_{p(s)}$ to the verifier, the latter can confirm the usual authenticity and integrity properties, which are rooted in the trust to the signing authority of s and hence assert the correctness of $p(s)$. E.g. if a digital identity contains the date of birth, a zero-knowledge range proof enables the prover to assert that the person in question is older than a specific age. The verifier never learns the full date of birth. Many real world use cases (e.g. buying cigarettes, entering a night club), are sufficiently served by asserting a minimum age (e.g. 18, 21).

Another possible building block for ZKPs are pairing-based signature schemes. These are by themselves less powerful and limited to implementing, what we informally call ZKPs with subset capability. The most prominent example is the BBS+ signature scheme by Boneh et al. [16]. Here, the secret s is made up of several individual messages $s = s_1, s_2, ...s_n$, all of which are signed by a single signature value $sig_s = sig_{s_1,s_2,...s_n}$. The prover can now select an arbitrary subset of these messages $s_i, s_j, ...s_m = s'$ and the ZKP allows the creation of a matching signature $sig_{s_i,s_j,...s_m} = sig_{s'}$. In this case the verifier receives s' with the associated signature $sig_{s'}$ and verifies it. E.g. a digital identity consists of attributes related to an individual and this variant of a ZKP allows the prover to disclose only attributes relevant to the specific transaction. Consider that the signup to a customer loyalty program may only require a small number of attributes (e.g. full name, gender, e-mail address). All other attributes (e.g. physical address, date of birth, social security number) are omitted in s' and hence hide not just their value, but even their existence altogether.

Related to this is the concept of an *anonymous credential*, a type of credential that has no identifier or quasi-identifiers with longtime correlation capabilities about the individual [27]. By that we mean the omission of attributes, that, either alone or in combination act as identifiers (e.g. social security number or the combination of full name and date of birth). This also prohibits pseudonyms (e.g. online forum username), since their long-lived nature allows correlating multiple transactions with each other. Using an anonymous credential is only possible if the transaction has no hard requirement for a (pseudonymous) identifier[1]. By using ZKPs one can derive anonymous credentials from any regular credential, which are often equipped with a unique persistent identifier from the IA. Ultimately, anonymous credentials achieve cryptographic unlinkability, a key measure to increase privacy.

3.1.2 W3C Verifiable Credentials

W3C published the *Verifiable Credentials (VC) Data model Recommendation* [124], "*a standard way to express credentials on the Web in a way that is cryptographically secure, privacy respecting, and machine-verifiable*". The aforementioned standard is the primary output of the Verifiable Credentials Working Group, but is augmented by a "working group note" on use cases [98].

[1]This even prohibits attributes like a serial number of a one-time use ticket. On first glance this is neither an identifier of the individual, nor pseudonym (since it's used only once). However, collusion between ticket vendor (IA in Digidow sense) and verifier does allow linkability that is not required for functionality.

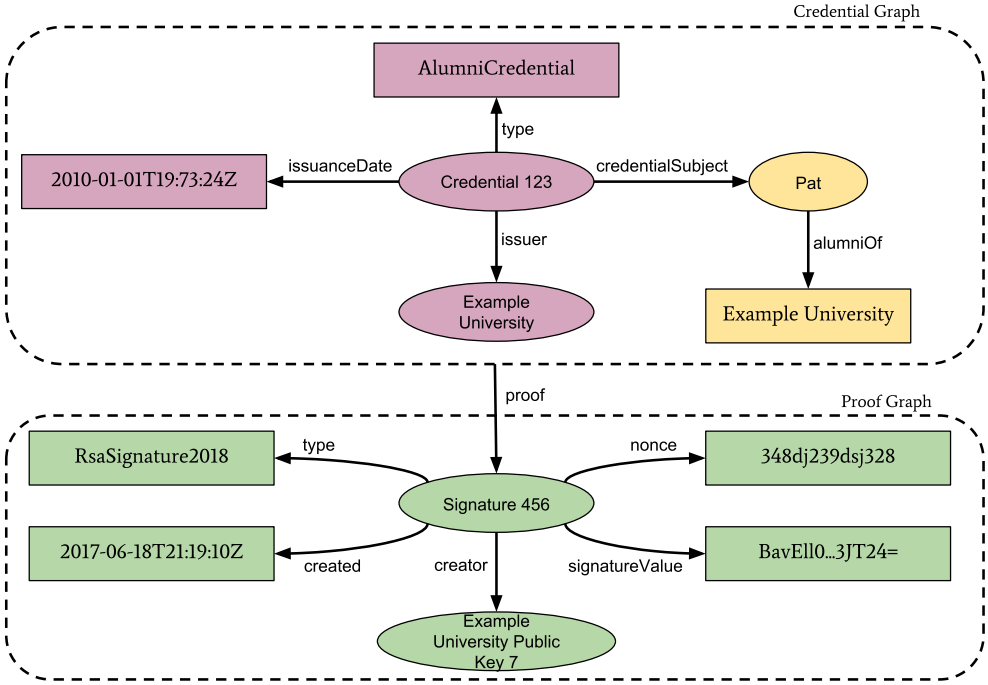


Figure 3.1: Example for a verifiable credential in the W3C VC data model. The credential is shown as an abstract graph and all information is encoded as subject-property-value triples, visualized as ovals, arrows and rectangles respectively. From "Verifiable credentials data model 1.1: Expressing verifiable information on the web" by Sporny et al. [124].

A *verifiable credential (VC)* is an identity document that is digitally signed . Figure 3.1 contains an example about a university alumni named Pat. Any single VC contains three components, namely

1. credential metadata (highlighted in purple in the figure): e.g. who issued the credential? when was the credential issued?,
2. claim(s) (highlighted in yellow in the figure): specific statement about the credentials subject; e.g. Jon holds a drivers license for vehicles of a certain type and
3. proof(s) (highlighted in green in the figure): cryptographic signatures by the issuer of the credential.

A *verifiable presentation (VP)* is used to

- bundle up one or more credentials to form rich and complex statements (e.g. enrollment into a university typically requires the assertion of many claims) and/or
- to formulate and digitally sign a derived credential via a ZKP.

Figure 3.2 contains a simple VP, which wraps the previous example from Figure 3.1. VPs are themselves composed of 3 components, namely

1. presentation metadata (highlighted in violet in the figure): e.g. are there specific terms of use,
2. verifiable credential(s) (sub-areas highlighted in previously mentioned colors in the figure): a combination of one or more credentials and/or a derived credential and
3. proof(s) (highlighted in blue in the figure): cryptographic signatures by the composer of the presentation.

3.1.3 Hardware Supported Security

Using dedicated hardware to strengthen or establish security is a popular approach. The reduced attack surface of dedicated security hardware (in relation to general purpose feature-rich hardware) and the additional layer of defense against threat actors are obvious advantages over classic software-only approaches. In this section we present a selection of hardware security technologies that are relevant for mobile devices and therefore interesting for storing sensitive data.

Trusted Execution Environments (TEE)

In general, trusted execution environments (TEE) [113] are special tamper resistant environments that run trusted applications on top of a separated kernel in order to ensure authenticity of the executed code.

ARM features a security extension, marketed as “ARM TrustZone”, that implements a TEE on the main application processor (AP) and is similar to virtualization technology or CPU rings. Each ARM core can switch between

- a non-secure world, typically running a feature rich OS like Linux, and
- a secure world execution mode, intended for security sensitive trusted applications on top of a dedicated kernel.

Figure 3.2: Example for a verifiable presentation in the W3C VC data model, embedding a verifiable credential. The credential is shown as an abstract graph and all information is encoded as subject-property-value triples, visualized as ovals, arrows and rectangles respectively. From “Verifiable credentials data model 1.1: Expressing verifiable information on the web” by Sporny et al. [124].

Smart Cards

Smart cards (integrated circuit card, ICC) are used to securely store and process data for many use cases [110]. Interoperability requirements have resulted in the widespread adoption of the ISO/IEC 7816 standards as a common basis. Specifically, the ISO/IEC 7816-4 standard defines "*Organization, security and commands for interchange*" and as part of this, introduces the *application protocol data unit (APDU)* as atomic communication unit between a smart card reader (issuing commands) and a smart card (replying with a response). Furthermore, it defines that multiple applications, each identified by an application identifier (AID), can exist and run in parallel on a smart card OS.

Typical smart card OSes, like the the Java Card OpenPlatform (JCOP) or MULTOS, have become quite powerful. In case of JCOP the ICC runs an implementation of the Java Card platform edition. The latter is a tiny Java virtual machine running Java applets on top of the native portions of the card OS stack. Since all smart cards relevant to our purposes use the JCOP platform, we us the term applet as synonym for applications running on any ISO compatible smart card for this thesis. This improves readability since possible confusion with applications running on other processors are avoided. Many use cases have their own dedicated standards on top of the ISO ones, including

- EMV for payment,
- universal integrated circuit card (UICC)[2] for identification of a user in the mobile network, and
- other special purposes usages (e.g. FIDO2 authenticators provides strong authentication for the web).

Secure Element (SE)

A secure element (SE) is an integrated circuit chip that adheres to the previously mentioned ISO/IEC 7816 standard and is embedded into electronics. The term originates from the NFC domain [30, 90], where it is often combined with an NFC controller into a single package. We focus on SEs that are embedded into mobile phones and thus provide fully autonomous computing and storage capability independent from the main AP. Just like with smart cards, an SE runs a card OS and can thus run various applications.

Secure Enclave

Secure enclaves are security-dedicated subsystems embedded into the main system on a chip [6]. Conceptually they have the same goals as SEs, but they use custom protocols to communicate with other system components (instead of the ones standardized in ISO/IEC 7816). On a technical level they usually consist of, at a minimum, a small CPU or micro-controller and a true random number generator. Within the mobile domain they can primarily be found in Apple devices [6], while in the Desktop space there are the Intel SGX[3] and AMD Memory Encryption [73] solutions.

[2] A standard that holds a subscriber identity modules (SIM) card as primary component.
[3] https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html

3.2 Requirements and Architectural Measures

3.2.1 Requirements and Threat Model

The functional requirements for a PIA are derived from the larger project vision [83] and are as follows:

R1 The PIA stores digital identities, each consisting of attributes related to the associated individual.

R2 The PIA interacts with other entities in the Digidow system to perform authentication for the individual.

R3 The individual has full legal control over the PIA.

These requirements were synthesized from project meetings [83].

Among the non-functional requirements we are primarily interested in security and privacy. We build on existing threat modeling by Mayrhofer et al. [84]. With regards to the individual themself, our scope contains the following threats:

TI1 *Privacy leak*: Private information about individuals should remain private. Leaks to unauthorized parties can be used for targeted phishing, blackmail, or even identity theft (if sensitive identifier or payment information is involved).

TI2 *Identity loss*: Individuals should always have access to their identities. If individuals become unable to prove their rightful association with an identity they will be unable to access related services.

Beyond that we recognize the following threats, roughly based on the CIA triad, for the PIA directly:

TP1 *Unauthorized attribute disclosure*: A PIA should only disclose attributes to authorized parties. A violation of this results in a privacy leak for the individual, as described in threat TI1. Note that any action by the individual is considered legitimate, even if it is performed under duress. Such a case is not considered a thread to the PIA, but obviously constitutes a privacy leak for the individual (cf. threat TI1).

TP2 *Denial of service*: A PIA should always be accessible by their individual. Disruption of such access leads to (temporary) loss of identities (cf. threat TI2).

TP3 *Unauthorized modification*: Any changes to the PIA (identities, configuration, etc.) should be possible only for the associated individual. The storage of wrong identities or the modification of existing ones can result both in privacy leaks (cf. threat TI1) and identity loss (cf. threat TI2).

3.2.2 Measures for Security and Privacy

Based on the functional requirements and the threat model we derived security and privacy related measures for the PIA. These are meant as a software design-level intermediate step between the theoretical threat model and the actual implementation. The measures act as guidelines that inform the actual implementation work to achieve better security and privacy guarantees w.r.t the modeled threats.

M1 *Transport security*: All data that the PIA sends or receives should be subject to secure transport encryption. This protects communication of the PIA from threats TP1 and TP3 by ensuring confidentiality, authenticity and integrity. On a technical level the PIA currently uses HTTP for communication with other Digidow entities and thus all traffic needs to be protected with TLS. Based on the current state of the art we propose to use at least TLS 1.2 and preferably TLS 1.3 (or future newer iterations). An alternative approach to satisfy the goal of transport security is the usage of Tor (The Onion Router). Among other guarantees, Tor establishes a secure connection with authentication and confidentiality.

M2 *Secret key protection*: The PIA needs to securely store secret keys[4] that are used to digitally sign its actions. An on-the-wire attacker with access to the secret key of a certain credential may alter authentication from the PIA to the verifier, constituting a threat TP3. In addition, if such an attacker exposes the secret key publicly, it needs to be revoked, constituting a threat TI2. Securely storing sensitive keys involves the usage of tamper resistant hardware on modern mobile devices. Note that the exact choice of keys and algorithms is constrained by the security and privacy requirements (cf. section 3.1.1). Section 4.2 analyzes the support available in contemporary hardware.

M3 *Encryption at rest*: Sensitive data that the PIA does not need regularly can be protected by additional application encryption at rest. Modern Android uses key material that involves a user secret[5] and protects against some basic threats (e.g. attacker reading the storage of a powered off device), but it does leave the user vulnerable to other realistic threats. E.g. an attacker coercing an individual to operate their PIA or attackers with system level privileges observing the execution of the PIA on the Android device. All data that needs to be processed or transmitted by the PIA as part of regular use cases, e.g. digital identities and biometric embeddings, do not gain additional security by application encryption at rest. Consider the following two attack scenarios:

- The user can be coerced to use the PIA, including the authentication of private key usage.
- Similarly, an attacker with system level privileges can observe the plaintext data during usage.

However, sensitive data that is not involved in regular use cases could be encrypted at rest on the application level. This defense-in-depth measure provides an additional security perimeter for sensitive data against the threat TI1. E.g. consider the following for transaction history data:

a) Transactions are being fed once into location tracking model of the PIA. This is not reversible, i.e. one cannot extract exact transactions from the model.
b) They are subsequently encrypted by the PIA via special key material.
c) In exceptional cases, the transaction history can be decrypted again. E.g. to migrate a full backup with all history data to a new PIA, where it can be used to retrain a clean location tracking model.

The essential part of this usage is that the special key material is not available during regular usage. One such possibility are printed backup codes stored in a secure physical location (e.g. safe).

[4] Note that in ZKP protocols we use the term *secret key*, whereas classical asymmetric cryptography systems use the term private key.
[5] Called Android file based encryption (FBE) with credential encryption (CE).

M4 *Network privacy*: Straight forward communication of the PIA over the network, i.e. Digidow entities establish direct connections between each other, leaves metadata traces in many places[6] [67]. Such metadata can be used to correlate two or more pieces of network communication with each other [67], enabling third parties to create fingerprints of PIAs, and in turn their associated individuals. This is problematic for unlinkability and possibly also anonymity, depending on the specific circumstances and the knowledge of the attacker[7]. Ultimately, this is at least a violation of the threat TI1 and, depending on the observed metadata, possibly even an issue with the threat TP1. In Digidow we use Tor (The Onion Router) to cloak all sensitive network operations. We also use Tor Hidden Services, a special type of network address that provides strong privacy guarantees for the server side of the network exchange, to increase the privacy of both communicating entities. While this is not a silver bullet against all network attackers, especially not against state level actors, it does prohibit or drastically harden network metadata against fingerprinting for a large percentage of threat actors.

The implementation of this measure is not a contribution by the author of this thesis, but rather by other work streams of the Digidow project. Regardless, it is listed for completeness sake since it is essential to satisfy the requirements and threat model of the PIA.

M5 *Biometric unlinkability*: Biometric data on individuals is inherently personal and allows (mostly) unique identification of users in large data sets. The following text is focused on facial recognition, but we believe similar procedures are possible for other biometrics. Facial recognition is commonly done by a neural network that processes a raw image of a face into a high-dimensional feature vector[8], an embedding. Subsequently, a similarity score between two embeddings can be computed and based on threshold values a (non-)match is determined. The unmodified embeddings used in such a scheme are personal data that allow linkability between two transactions and possibly even de-anonymization, provided an attacker has a face image and knows the neural network involved. This constitutes a threat TI1 or possibly even a threat TP1. We combated this issue by employing a novel privacy preserving hash, which we refer to as *fuzzy hash*, on the embedding data [120]. The fuzzy hash uses a trap door function to generate a hash of one, or preferably multiple, embeddings. This generated hash cannot be reversed to the original biometric data, prohibiting correlation to individuals, i.e. de-anonymization, while maintaining sufficient information to allow comparisons. Due to the addition of a salt value to the fuzzy hash, behaving similar to salt values used in conventional hashing, we can ensure that similar, or even identical, embeddings result in different hashes. As a consequence, unlinkability between two related transactions is achieved.

The implementation of this measure is not a contribution by the author of this thesis, but rather by other work streams of the Digidow project. Regardless, it is listed for completeness sake since it is essential to satisfy the requirements and threat model of the PIA.

M6 *Cryptographic data minimization*: Digital credentials that are signed by classic asymmetric cryptography systems (e.g. DSA, ECDSA) require that the

[6] Here, we assume proper transport security as precondition, cf. measure M1.

[7] Consider an individual x with PIA y using the same IP address z over a significant time frame, where y contacts another Digidow entity, e.g. verifier or sensor, regularly. Any on-path-attacker breaks unlinkability by correlating packets with IP address z to PIA y. If z is a fixed IP address associated with an individual, e.g. residential internet connection, and the attacker has state level capabilities, they can even infer the individual x, thus breaking anonymity.

[8] Common dimensionality ranges from 128 to 512.

prover discloses the entire credential with all attributes. In many use cases only a small subset of attributes or a predicate of an attribute from the issued credential are sufficient to satisfy the verifier. Hence, any exposed information beyond the bare minimum required for an interaction is considered a threat TP1. We address this topic by signing digital credentials via a ZKP protocol. The emergent derived credentials only expose the absolute minimum data required by the verifier, while maintaining the usual guarantees of digital signatures. In many cases it is even viable to create and use an anonymous credential, i.e. a credential without a unique fingerprint. Data minimization on the level of the cryptographic system is an important aspect to enhance privacy. Concretely, an analysis as part of the Digidow project has resulted in a short list with the following group signature schemes:

- Camenisch-Lysyanskaya (CL, [23]) signature scheme
- Boneh-Boyen-Shacham (BBS+, [16]) signature scheme and
- Pointcheval-Sanders (PS, [104]) signature scheme.

A noteworthy downside of the CL signature scheme is the size scaling of the signature, which is $O(n)$, where n is the number of attributes in the signed credential. In contrast, the other two schemes are $O(1)$.

Note that there are no measures that address the threat TP2 specifically. Unfortunately, we believe there is little that can be done to address this one in full. The measure M4, which involves the usage of Tor, may help against on-path attackers that control some intermediate portion of a connection that is not a choke point. A textbook example of this is the usage of Tor to bypass the Great Firewall of China [127]. However, any on-path attacker that controls even a singular choke point, or a set of points that together constitute a choke point, can arbitrarily suppress legitimate traffic and thus fully violate the threat TP2.

Chapter 4

Secret Storage on Android

As noted in the requirement R1 we need to store digital identities in the PIA. Subsequently, we enumerate and analyze approaches for secret storage on Android in this chapter. To this end we start with an overview of Android security in general and introduce important related terms.

4.1 Android Security

Android is an OS for mobile phones and therefore has distinct security requirements from OSes used in primarily in other domains (e.g. desktop, embedded). A foundational understanding of the Android platform model is important understand how hardware supported security can provide enhancements.

4.1.1 Android Platform Security Model

The Android platform security model, as defined by the equally named publication by Mayrhofer et al. [85], is a theoretical informal model that has informed the design and implementation of Android. Even tough it was only published in 2021 and is based on Android 11, the described model has been used informally in the past and applies similarly to past releases of Android. We highlight some aspects, those relevant to our goals, of this model.

Rule 5 of the security model states that *applications are security principals*. This is in contrast to many other OSes that assume actions of an application are performed on behalf of a user. E.g. consider how UNIX related OSes inherit the user/group of the logged in individual to executed applications. Since apps are not considered fully authorized agents on behalf of the user, the former do not run with the context of the logged in user. Instead, they are treated as security principals, meaning each app runs under a dedicated user.

This ties into rule 1 of the security model, the *Multi-party consent*. The three main parties in the Android ecosystem are

- the user,
- the platform vendor, and
- the app developer.

All of them need to be in agreement to perform an action. In the world of Android we distinguish between

- active subjects (users and application processes) and
- passive objects (files, network sockets and IPC interfaces, memory regions, virtual data providers, etc.)

4.1.2 Hardware Supported Security for Android

TEE for Android

The Android Open Source Project (AOSP) includes Trusty [4], a TEE that works in close collaboration with the Android OS itself. Trusty is made up of several components that work together, namely

- a secure world kernel,
- the Trusty Driver, a Linux kernel driver that belongs the Android OS and interfaces with the secure world kernel, and,
- the Trusty Lib, a userspace library that facilitates communication with trusted applications over the aforementioned kernel driver.

The Trusty API[1], which is the interface between these three components, is subject to change. Implementations of the secure world kernel are custom on a per-SoC basis. This means that OEMs and SoC vendors in the Android domain each have their own secure world kernel implementation, e.g.

- Google created the Trusty Kernel [4] (based on the Little Kernel embedded OS),
- Qualcomm Trusted Execution Environment (QTEE) runs on Qualcomm SoCs [107],
- Samsung TEEgris runs on their own Exynos chips [43, 117] and
- Huawei iTrustee runs on (at least one) Kirin SoC [42].

All of the aforementioned kernels are based on ARM TrustZone.

Embedded Secure Element (eSE) and Alternatives

Several current Android devices, mainly high-end ones and a few select mid-range ones, contain an embedded SE (eSE), i.e. an SE that is resides on the motherboard of the mobile phone. The main AP, driven by its requirements for a big feature set (latest ISA with numerous extensions) and performance, has significant complexity. In contrast to this, an eSE only needs to support a specific tailored OS and very few select applications. Therefore, it has a substantially smaller attack surface than a full-fledged main AP. As Mayrhofer et al. [85] note, this difference can be seen in security issues related to the main AP (e.g. Spectre/Meltdown) and dependent hardware (e.g. Rowhammer affecting the main memory), all of which can pose a threat to applications running on a main AP TEE, but not on an eSE.

As section 2.6.1 elaborates SEs were originally included into smartphones to provide a highly secure basis for digital wallets. However, over time digital wallet applications were switched to the host card emulations schemes and thereby no longer depend on SEs. More recently, Google pushes for the usage of SEs to increase the security of the Keystore system (cf. section 4.2.2).

The first implementation of an eSE to strengthen the Keystore system was the Google Titan M chip, a security chip introduced alongside the Google Pixel 3[2]. As Melotti et al. [86] found, previous to their publication, not much information was available about the internals of the Titan M chip and the firmware was

[1] https://source.android.com/docs/security/trusty/trusty-ref
[2] https://blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/

not released as open source (despite such promises in the announcement blog post). Nugget OS, abbreviated `nos`, is the name of the OS running on the Titan M chip. The hardware security module itself is referred to as "citadel". Furthermore, with the help of the custom structure-aware black box fuzzer developed by Melotti et al. [86], they proved that the security of the Titan M chip is not insurmountable by uncovering multiple vulnerabilities in the chip.

In order to expand the adoption of (e)SEs to more OEMs, Google and Secure Element (SE) vendors founded a collaboration effort called Android Ready SE Alliance [51]. As part of this alliance, SE vendors and Google collaborate to design open-source hardware backed security applets that can run on SEs. These applets are meant to address use cases around Strongbox (cf. section 4.2.2), digital keys for cars and homes, identity credentials (e.g. mobile driving license and national IDs) and e-money solutions involving digital wallets. Ultimately, a wider adoption of SE and the accompanying applets is going to strengthen the security of the Android platform as a whole and provide desirable functionality for mobile devices.

While the vast majority of modern SE deployments in mobile phones are eSEs, there are other options. Specifically, Android supports SEs located inside UICC chips. With the continued trend for increased security we expect that the majority of new UICC models are going to feature SE capabilities. Another rising trend is the usage of embedded UICC (eUICC) chips that are fixed components of the mobile phone. Using an UICC-based SE is only really sensible if the UICC in question is an eUICC. Otherwise, each switch of the mobile network operator (MNO), meaning the physical UICC card is swapped, mean the loss of all sensitive data stored into SE. Data migration between the old UICC and the new one is not possible, since data stored into a specific SE should remain sealed there and not be extractable. In order to maintain generality, we are going to use the general SE term from here on.

SE Access Control

An OS running applications from untrusted third parties, as is the case with Android, needs to restrict access to connected SEs in order to avoid malicious abuse [46]. One very common threat are denial of service attacks (e.g. excessive allocation of keys or selection of non multi-selectable applets) prohibiting legitimate use. This is addressed by the *Secure Element Access Control* standard [46], which is tightly connected to the Open Mobile API (OMAPI) and implemented by Android (cf. section 4.2.4). The general architecture consists of access rules that reside inside the SE. These are retrieved by the access control enforcer, in our case Android, and applied to rich execution environment (REE) applications. An access rule consists of

- a `DeviceAppID`, uniquely identifying an REE application, and
- an AID for the applet in the SE.

Specifically for Android, the `DeviceAppID` is the "*SHA-1 hash of the certificate used to sign the APK*" [44].

According to the *Secure Element Access Control* standard [46], the SE issuer[3] deploys the access rules over the air (OTA). At the very least this is required as an initial setup step and can be updated later on. Application providers for third

[3]More precisely, whatever legal entity has organizational control over a deployed SE. For an eSE or eUICC that is most likely an original equipment manufacturer (OEM) or original device manufacturer (ODM), whereas a UICC is controlled by a mobile network operator (MNO).

party applets deployed into a SE can update their own access rules, called Access Rule Application Client (ARA-C), on their own terms via an OTA mechanism. The aggregation of all access rules is exposed by the SE to the access control enforcer.

4.2 Analysis of APIs

The PIA stores digital identities and related auxiliary data. For our purposes of this analysis we logically partition data related to each identity into the following 3 components:

IC1 *Main credential*: The entire credential issued by the IA, spanning all attributes and metadata. Notably, this includes biometric data, like embeddings, essential to the functionality of our system. One exception to this, i.e. not included in this component, is the secret key associated with this identity.

IC2 *Secret key*: The secret signing key of the ZKP signature scheme associated with the specific identity. It is used by the PIA to create derived credentials from the original one. Because of the specific requirements we treat it as a separate component. While both measures M2 and M6 are guiding the implementation, the latter trumps the former due to it's critical nature for the overall project.

IC3 *Auxiliary data*: Aside from the main credential, as issued by the IA, the PIA maintains some auxiliary data for each digital identity. E.g. a flag whether the identity is active and should be used by the PIA, or cosmetic color information used by a smartphone PIA for visualization.

For each of the enumerated storage techniques we provide an overview of functionality, an analysis concerning the compatibility across the broader user basis and finally an analysis of suitability w.r.t. to our threat model and the related measures.

4.2.1 Data and File Storage

Android offers several APIs for storing regular data and files [54]. Each come with their own advantages and disadvantages.

- *App-specific storage*: This API is similar to the ones offered by other OSes and allows arbitrary text or binary data to be placed in one or more files. All created files and directories reside in dedicated directories only accessible by the app. There are two variants: Internal storage is always available, whereas external storage tends to provide more space at the cost of availability. On some devices the latter type of storage is physically located on a removable media. The former can be located via `getFilesDir()` or `getCacheDir()`, while the latter can be found via `getExternalFilesDir()` or `getExternalCacheDir()`.
- *Shared storage*: As the name implies, this API is used to store files that are shared across apps and can be managed by the user directly (via file manager apps). Technically, there is a distinction between media files, accessed via the `MediaStore` API, and all other files, managed via the Storage Access Framework (SAF). In SAF an entity offering files implements a subclass of `DocumentsProvider`, whereas a consumer of files triggers an intent action of type `ACTION_CREATE_DOCUMENT`, `ACTION_OPEN_DOCUMENT` or `ACTION_OPEN_DOCUMENT_TREE`.

- *App preferences*: This API is used to store simple key-value pairs for the app and supports some basic data types for values (boolean, float, integer, long, string and string set). Instances of a `SharedPreference` object can be retrieved via `getSharedPreferences()` or `getPreferences()`.
- *App database*: Using this API, a developer can store structured data in an Android Jetpack Room database [57]. While conceptually similar to the app preferences, this is a more powerful and mature approach for storing larger structured datasets. Developers include the Jetpack Room dependencies, located under the `androidx.room` coordinates, in their build scripts and can subsequently use the library in their app.

The best choice depends on the requirements of the use case in question. All of these APIs are relatively easy to use, convenient for storage of simple properties (app preferences), structured data (app database), or arbitrary data (app-specific or shared storage), and nowadays allow usage of ample space[4]. Furthermore, since they are a core part of the Android functionality, these methods of data storage are available on all Android devices.

The usage of shared storage via the `MediaStore` API requires permissions for most use cases [54]. Apps targeting Android version 9 or older were required to have the `READ_EXTERNAL_STORAGE/WRITE_EXTERNAL_STORAGE` permission to read/write any file outside of the app data directory, Starting with Android 10, an app only requires the `READ_EXTERNAL_STORAGE` permission if it needs to access other apps' files and the `WRITE_EXTERNAL_STORAGE` permission no longer has any effect. In practice this means that limited functionality is available on recent Android versions (starting with 10) even without permissions. The SAF on the other hand does not require permissions at all.

Security Caveats

However, none of these APIs use dedicated security mechanisms like a TEE or SE to harden access to data. Instead, data is simply stored in the file system, physically residing either on the internal flash memory or an SD card. While the Linux discretionary access control (DAC) does protect app-private data from other apps and (mostly) the user, the Android system itself has full access.

Another problem is the possibility of exporting app-specific data via the `adb backup` command. Such a backup contains, for each app, the APK file(s) and the following app-specific data:

- Nearly the entire internal app data directory, comprised of
  - internal app-specific storage (`files` sub-directory),
  - app preferences (`shared_prefs` sub-directory) and
  - app databases (`databases` sub-directory), as well as
  - all other sub-directories (e.g. WebView data with a browser profile).

  The only exceptions to this are
  - caches (`code_cache` and `cache` sub-directory) and
  - a specific `no_backup` sub-directory[5].

[4] While there are no artificial hard limits concerning disk usage by individual apps in Android itself, some OEMs opt to warn users if an app takes up excessive amounts of storage. Thus, a single app may use the entirety of the `userdata` partition in theory.

[5] Which can be found via the `getNoBackupFilesDir()` on the app context.

- The external app-specific storage is also included.

Effectively, this means that users do have access to supposedly app-private data.

Note that this behavior has changed with Android 12. Before that release, apps were able to opt-out of this manual backup mechanism by declaring the `android:allowBackup` in their manifest file to be `false`. All apps that target API level 31 or higher are no longer going to be included in manual backups via `adb backup`[6] [53, 101]. In line with our observation, Google states that this is "*to help protect private app data*". However, apps targeting API level 31 or above may opt-in to the manual backup by declaring the `android:debuggable` attribute[7] in their manifest to be `true`.

Usage in Digidow

Even though all stored identity data in a PIA is about the associated individual, and they exert full legal control of it, it would be ill-advised to place such data in shared storage. Doing so would enable arbitrary read or write access by many other apps, all of them untrusted independent security principals in the Android ecosystem. I.e. we do not give the user easily accessible full technical control over their data. The legal control, together with other trust measures, has to be sufficient for an individual to trust their PIA. The manual backup mechanism via `adb backup` is a double edged sword. As a power user, it is convenient that one can inspect the app data directory. However, a regular user should never need this access and it is a possible extraction mechanism for sensitive data that can be used under duress (e.g. by border agents). At this point in time we don't have a strong opinion on this subject and trust the default behavior of the Android platform with the latest API level. We cannot identify any benefit to the external app-specific storage for our use case either. The remaining options, namely internal app-specific storage, app preferences and app databases, all ultimately place their data in the internal private data directory of the app. These APIs are meant for different types of data (unstructured, key-value and structured), but fundamentally provide the same security guarantees. Hence, for our purposes we will collectively refer to these as *internal app-private data*.

All three components of a digital identity can be stored in the internal app-private data. In fact, this will be our working assumptions for now and acts as baseline. Any other APIs or approaches to storing one or more component(s) of a digital identity will be measured against this naïve model. However, that is not an endorsement of internal app-private data APIs from a security and privacy point of view. Malicious actors with system level privileges can read and modify internal app-private data as they please. There are many possible approaches how a malicious actor may achieve such privileges, e.g. an unpatched vulnerability (chain), zero-day vulnerability or possibly even a user granting system-permissions for apps running on a rooted device. Clearly, such issues occur often enough that it motivated the platform vendor Google to create dedicated APIs for secure storage of key material or identities.

[6] At the same time, Android 12 also renames the *Android Auto Backup* to *Cloud backups*. These continue to respect the `android:allowBackup` attribute of apps targeting API level 31 (default is `true`). In parallel to this, it also introduces the new *Device to Device (D2D) transfer*, which allows a direct transfer of app-private data to a new device and is always possible, regardless of the aforementioned attribute.

[7] Note that this flag has severe security implications because of the ability of a debugger to inject and execute arbitrary code inside the debuggable app.

4.2.2 Keystore System

The Android keystore system is an API to securely store cryptographic keys [50]. Once a key is loaded into the keystore it can no longer be exported. From then on, the usage is constrained by key use authorizations, a list of conditions that need to be met in order for the key to be usable in a specific setting. Key use authorizations can be grouped into the following categories:

- *cryptographic algorithms and parameters*, e.g. operations or purposes (encrypt, decrypt, sign, verify), padding schemes, block modes and digests with which the key can be used;
- *temporal validity interval* defining a time range wherein the key is considered valid;
- *user authentication* prompting the user to perform a lock screen authentication and/or a strong biometric authentication, i.e. requiring explicit consent by the user for each usage of the key.

The Android keystore supports key attestation, essentially a challenge response process between a trusted host (the verifier) and the keystore system (the prover), where the latter cryptographically asserts metadata of a stored key.

There are different implementations of the keystore system in Android and the one in use depends on the API level and hardware capabilities [50]. *Keymaster* and *Keymint* together are the original implementation of the Android keystore system that can optionally be backed by a TEE. In any case, key storage on the host system is also available. The newer implementation of the keystore system, available on devices with an SE running Android 9 (API level 28) or later, is referred to as *Strongbox Keymaster*. Device running Android 9 without an SE continue to use the regular Keymaster and Keymint implementations.

Security Levels

Each key stored in the keystore system has an associated security level (called `attestationSecurityLevel` in the ASN.1 schema of the key attestation). The following values can occur:

- `Software (0)`: This key was created and is managed by software running on the main Android OS. Even though such a key does not use any hardware security features and is thus vulnerable to malicious actors with system level privileges, it is better than storing the key in the application directly. Since the key material never resides in app process memory, any attacker that gained privileges of the executing app cannot access it.
- `TrustedEnvironment (1)`: Whenever this value is observed, it means that the key resides in a TEE (cf. section 4.1.2). The extraction of such a key is protected from attackers with system level privileges (e.g. an exploitable security vulnerability in the customized Linux kernel, the "main" kernel of Android). However, without a user-bound key use authorization an attacker with system level privileges can use the TEE keystore as an oracle to perform malicious key use.
- `StrongBox (2)`: This key is being managed in an SE (cf. section 4.1.2). Going beyond the guarantees of the previous level, the key material stored with this security level is considered especially well protected. Based on the nature of an SE, this includes resistance against issues related to the main AP and dependent hardware, e.g. Spectre, Meltdown and Rowhammer.

Note that the security level of the key does not have to match the highest security level offered by the device. If the specified combination of key type, key size and cryptographic parameters is not supported by the highest security level of the device, the keystore system is going to use a best-effort approach. Whenever the usage of an SE is mandatory, a developer can specify the `setIsStrongBoxBacked()` flag in the `KeyGenParameterSpec.Builder` class and subsequently a key may only be stored in a dedicated hardware security module.

API Feature Overview

From a developer point of view the keystore system consists of APIs that originate from the Java Cryptography Architecture (JCA) by Oracle [94] and, thus, adopts terminology and API interfaces from there, but was also extended by Google in several places for their own purposes in Android. On a high level view, depending on the required usage scenario, an application designer can choose between the following two API "families":

- The `KeyChain` API is used to manage system-wide credentials and has been available since Android 4.0 (API level 14). Application can use this API to request some credential. Upon such a request, the user chooses a suitable credential in a system-provided UI and the selected credential is then returned to the application. While the classes concerned with the actual algorithms and keys come from JCA, the `KeyChain` class itself is an addition by Google.
- In contrast, the Android keystore provider feature allows secure storage of credentials in an app-private way since Android 4.3 (API level 18). Since these credentials are per definition scoped to the app, there is no need for a selection UI. While the `KeyStore` API originates from JCA and, therefore, has been around since the inception of Android[8], the Android keystore provider feature specifically refers to an instance with the type value `AndroidKeyStore` [19]. Only such an instance can, in case of available hardware and software support, store keys with a higher security level than `Software`.

Note that Android developers should not provide an explicit value for the `provider` parameter of the JCA instance retrieval methods, but rather use the overloads without these and let the platform figure out the optimal provider on its own [130]. Additionally, the Android Jetpack offers a wrapper API that simplifies the usage of the Android keystore system [58].

More concretely, the current list of supported cryptographic algorithms, entailing key type and/or cryptographic operations, is limited to a selection of well-known primitives and sensible combinations of these. Namely, for the Android keystore provider feature these are:

- `Cipher` algorithms are a triple of a symmetric or asymmetric cipher, block mode and padding scheme, together used to encrypt/decrypt data (e.g. AES/CBC/PKCS7Padding, RSA/ECB/OAEPWithSHA-256AndMGF1Padding),
- `KeyGenerator` and `SecretKeyFactory` algorithms handle symmetric keys (e.g. AES, HmacSHA256), the former creates new ones altogether while the

[8] Android offers many cryptographic facilities [49], spanning a wide range of keys and algorithms. Since Android API level 1 developers could access software-only, but feature rich `KeyStore` implementations that allow writing cryptography code based on the JCA. E.g. an instance of the `KeyStore` with the `PKCS12` type value. Some of the algorithm implementations come from the Bouncy Castle library, which provides numerous cryptography algorithms and exists for several platforms[9]. This is the source of two identifiers for `KeyStore` instance types [19], namely the eponymous `BouncyCastle` and the `BKS` (short for Bouncy Castle Keystore).

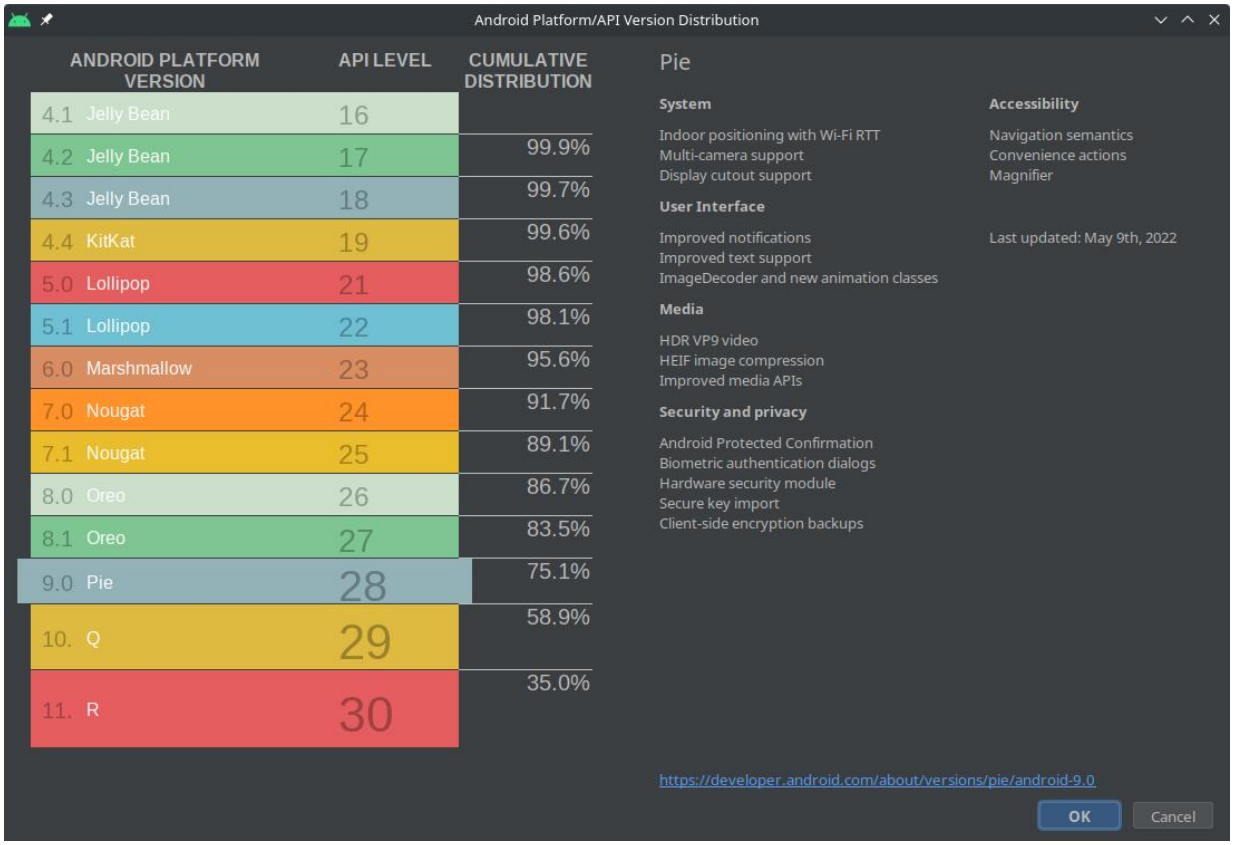


Figure 4.1: Distribution of different Android versions according to Google as of 2022-05-09. Screenshot obtained from Android Studio Chipmunk 2021.2.1 Patch 1 running on Linux [52].

latter converts an existing key specification (containing given key material) to an opaque `Key` object and vice versa,

- `KeyPairGenerator` and `KeyFactory` algorithms handle asymmetric keys (e.g. EC, RSA), the former creates new ones altogether while the latter converts an existing key specification (containing given key material) to an opaque `Key` object and vice versa,
- the `KeyStore` API is at the heart of the eponymous keystore system and manages cryptographic keys and certificates stored at least with system level privileges, whereas each entry can be any of the symmetric or asymmetric keys supported by the previously named API abstractions,
- `Mac` algorithms perform message authentication code signing and verify operations (e.g. HmacSHA1, HmacSHA256)[10] and
- `Signature` algorithms combine a cryptographic hashing algorithm (e.g. SHA256, SHA1) with an asymmetric cipher (e.g. ECDSA, RSA) to perform signing and verification operations, where the public key is sufficient for verification.

Compatibility Across User Base

We analyze the compatibility of the keystore system across the Android user basis by cross checking the introduction of the associated APIs against the current usage statistics for Android. Google itself provides data on active Android devices, offering aggregated statistics on the platform version, screen size/density and supported graphics API (versions)[11]. The latest update to the platform usage statistics considered in this thesis happened in May 2022 and provides data for major and minor releases of Android from 4.1 through 11, cf. Figure 4.1. According to this data

- the `KeyChain` API (available starting with Android 4.0, API level 14) is available for 100% of users and
- the Android keystore provider (available starting with Android 4.3, API level 18) can be used by 99.7% of users.
- Many of the algorithms supported by the Android keystore provider feature were added in later versions. The latest additions were made in Android 6.0, i.e. API level 23 and correspond to 95.6% of active users.
- Another interesting data point is the percentage of active Android users that employ the StrongBox Keymaster with an eSE. Unfortunately we do not have any good numbers on this. Via the platform statistics we can infer an upper bound (Android 9, API level 28) at roughly 75.1% of active Android users. Furthermore, industry experts at Eurosmart estimate the number of overall shipped eSE to device manufacturers, including *mobile phones, tablets, navigation devices, wearables and other connected devices without SIM application*, at 490 million units in 2021 (vs. 450 million in 2020) [36]. However, these data points do not permit clear conclusions for this metric. Many open questions remain: How many of the shipped eSE units end up in Android devices? How to combine statistics on active users with others on shipped units?.

Suitability for Usage in Digidow

As the name suggests, we would like to use the keystore to store the secret keys associated with the digital identities, i.e. component IC2. While the PIA needs to sign messages sent to other Digidow entities via the secret key, there is no need to access or distribute the raw value of the secret key. Thus, we want to use the key pair generation inside the keystore system. In case of a StrongBox Keymaster, such a usage would mean that the secret key is sealed into an SE. In our case we want to restrict usage

- to a cryptographic signing operation,
- prohibit usage of the secret key before or after the validity of the accompanying credential, and
- (possibly) bind the usage to user authentication.

The last key use authorization is a trade-off between security (each usage requires explicit user consent) and convenience (no interaction required, i.e. PIA acts independent). We propose the following three policies as a starting point: Explicit consent is required for

[10] In contrast to the subsequent Signature algorithms a MAC needs the secret symmetric key for both signing and verification.

[11] While most of this data can be found at the "Distribution Dashboard" located at https://developer.android.com/about/dashboards, the statistics on platform versions can only be accessed via Android Studio in the *Android Platform/API Version Distribution* window [52].

1. all usages,
2. only the first usage within a certain timespan or
3. under no circumstances.

There is no "correct" answer to this and thus the policy choice varies based on several factors. An IA may enforce a certain policy to have reasonable confidence in the secure usage of issued credentials. On the other end, an individual may configure this aspect on each identity to account for their own threat model. However, per the design of the keystore we can only set this policy once at the time of the key generation or import. Overall, from a conceptual level, the keystore is a good fit to handle the secret key.

Analyzing the technical point of view, we review the list of supported algorithms/keys of the keystore system against our shortlist of ZKP protocols presented in measure M6. Within the former (cf. the list of supported algorithms by Google [50]) we are interested in

- asymmetric keys, where the `KeyPairGenerator` and `KeyFactory` APIs support
  - RSA with key sizes 512, 768, 1024, 2048, 3072 and 4096 and
  - EC with the named curves P-224 (i.e. secp224r1), P-256 (i.e. secp256r1 and prime256v1), P-384 (i.e. secp384r1) and P-521 (i.e. secp521r1) and
- signature algorithms, where the `Signature` API supports nearly all pairs made up of the following
  - cryptographic hashing algorithms, including NONE, MD5, SHA1, SHA-224, SHA-256, SHA-384, SHA-512 and
  - the asymmetric cryptosystems ECDSA with the named NIST curves, RSA and RSA/PSS.

Note that this list is exhaustive for Android 6.0 (API level 23) and above, but can only be guaranteed for the software security level. Support for higher security levels is subject to further restrictions. The preferred `StrongBox` level with an SE only mandates support for ECDSA with P-256 as signature algorithm and key[12], as mandated by the Android Compatibility Definition Document (CDD). Beyond that, a hardware Keymaster device (either TEE or StrongBox) can optionally support P-224 (i.e. secp224r1), P-384 (i.e. secp384r1) or P-521 (i.e. secp521r1). In contrast, the latter is comprised of

- the Boneh-Boyen-Shacham (BBS+) signature scheme with pairing friendly elliptic curves as keys (cf. [115]), common examples include
  - the Barreto-Naehrig (BN) curves (e.g. BN256I, BN254N, BN512I) and
  - the Barreto-Lynn-Scott (BLS) curves (e.g. BLS12_381, BLS48_581), and
- the Pointcheval-Sanders (PS) signature scheme with, again, pairing friendly elliptic curves as keys. We found a Rust implementation named `ps-sig`[13] for the pairing friendly curve BLS12_381.

[12] See the current version of the Keymaster (currently version 4) Hardware Abstraction Layer (HAL): https://android.googlesource.com/platform/hardware/interfaces/+/refs/tags/android-12.0.0_r1/keymaster/4.0/IKeymasterDevice.hal#444.

[13] The Rust implementation `ps-sig` at https://github.com/evernym/ps-sig is based on the `amcl_wrapper` Rust library at https://github.com/lovesh/amcl_rust_wrapper. While the latter supports four different elliptic curves (secp256k1, ED25519, BN254 and BLS12_381), cf. https://github.com/lovesh/amcl_rust_wrapper/blob/9d4d4e2d009e3c8c2de17e97e9a4d36721896496/src/lib.rs, the former is only implemented and validated for the pairing friendly curve BLS12_381 (cf. https://github.com/evernym/ps-sig/blob/b1a1d530a8fa65a80212c8f8f5f8bd8c80a75365/Cargo.toml).

Clearly, there is no overlap, neither for the keys nor for the signing algorithms. As a consequence, there is no straight forward technical approach to satisfy our requirements. We considered an indirect approach envisioning the usage of the available high level primitive operations of the JCA Signature API (`sign` and `verify`) to emulate a ZKP protocol inside the keystore system. Unfortunately, our assessment resulted in negative findings. Overall, this inhibits the usage of the keystore system to handle the secret keys associated with the digital identities, i.e. component IC2.

The functionality of the keystore does not lend itself to storing arbitrary data, as would be required for the components IC1 and IC3. Even if we could seal arbitrary data into keystore, this would not be sensible for our purposes. Per design the PIA needs to share these components with other Digidow entities, like sensors and verifies, or present them to the individual themself in a UI. In all of these instances the identity data needs to be processed and thus has to be available in plain text on the various Digidow entities.

4.2.3 Identity Credentials API

The Identity Credentials API (also known as `security-identity-credential` Maven artifact ID) is purpose-built for the the secure storage of user identity documents [3, 56]. Google intentionally designed the API in a generic and abstract fashion. I.e. the semantics of data being exchanged with the credential verification devices and Issuing Authorities (IAs) are explicit non-goals of this feature. However, the data structures that are part of this API are based on the ISO/IEC 18013-5 standard [68]. The ISO/IEC 18013-5 mobile driving licence (mDL) application standard [68] defines an interface specification for *mobile driving license (mDL)* applications and includes a generic specification for the exchange of *mobile documents (mdoc)* as underlying basis (cf. section 2.6.2 for details). Specifically, said standard builds on previous ones and

- specifies that mdoc requests and responses are encoded with *Concise Binary Object Representation (CBOR)* [17], a binary serialization format,
- use the related *Concise Data Definition Language (CDDL)* as notational convention to express CBOR data structures as text [12], and
- *CBOR Object Signing and Encryption (COSE)* as cryptographic operations on that data format [118].

Due to its generic nature the Identity Credentials API is designed to work with mdocs in general, rather than be restricted to mDLs. The latter is an instance of the former, but uses the specified mDL data elements from the standard, all located under the namespace `org.iso.18013.5.1`.

Briefly speaking, the security requirements of the ISO/IEC 18013-5 standard mandate that

1. credentials are authenticated by their issuing authority (IA) and
2. that communication between an mdoc holder and mdoc reader satisfies the requirements of a secure channel, namely including proper (mutual) authentication, confidentiality, and integrity.

This is satisfied by the following approaches in the standard:

1. Every credential issued by an IA is authenticated by a digital signature anchored in a PKI. By extension of trusting the root certificates of the PKI, a verifier also trusts the signed credentials issued by any IA.

2. Addressing the security requirements of the secure channel between mdoc holder (i.e. device) and mdoc reader is multifaceted.
   a) The mdoc authentication of the device to the reader is done via the `SDeviceKey` asymmetric key pair. After being initially generated on the device, the public part is signed by the IA and thus embedded into a certificate during credential provisioning, while the private one remains on the device and is used by it to sign data elements in the mdoc responses. Due to privacy considerations the device wants to have many IA-signed `SDeviceKey` key pairs at their disposal, allowing rotation or possibly even single usage (akin to a TAN). This hardens against or even prohibits fingerprinting based on the key pair presented to the mdoc readers, even if multiple ones collude.
   b) Conversely, but technically an optional component, mdoc reader authentication is used to authenticate the mdoc reader to the device. Issued credentials can lock entries behind access profiles, which may include a reader certificate. Subsequently, only properly signed mdoc requests of the reader are fulfilled by the device.
   c) Encryption and integrity is satisfied by the usual combination of a hybrid cryptosystem. During session establishment both parties generate elliptic curve key pairs (`EDeviceKey` and `EDReaderKey`) and perform in-band key exchange. Subsequently, the initial key material from that exchange is derived via a KDF into two secret session keys (`SKDevice` and `SKReader`). These are then used for encryption with a symmetric block cipher and an authenticated block mode. An authenticated block mode ensures both confidentiality and integrity of transmitted messages.

The ISO/IEC 18013-5 standard defines a cipher suite to be a combination of cryptographic primitives that are used to secure the communication between mdoc holder and mdoc reader. At the time of writing, there is only one cipher suite (with the ID 1) containing the following primitives:

- ECKA-DH (with support for 11 different curves),
- HKDF-SHA-256,
- AES-256-GCM and
- HMAC-SHA-256.

These primitives are currently state of the art and (assuming correct usage) considered secure.

Extended Compatibility via Jetpack

The full-fledged version of the Identity Credentials API, entailing a dedicated `credstore` system service and the Identity Credential HAL as part of the Android OS, was introduced in Android 11 (API level 30) and requires secure hardware. As of May 2022 only 35% of active Android users run Android 11 (API level 30) [52] and an even smaller (unknown) percentage of these users own a mobile phone with the required secure hardware.

The Identity Credentials API is also included in the Android Jetpack support library. This is Google's usual approach to such a feature adoption dilemma and expands (in this case) compatibility to Android 7.0 (API level 24), the equivalent to approximately 91.7% of users.

From a developers perspective both variants of the API work exactly the same. If available, the Jetpack variant (located under `androidx.security.`

`identity`) simply calls the native platform variant of the API (located at `android.security.identity`). Otherwise, the Jetpack variant uses a Keystore-backed implementation. Google notes the following concerning the security of these approaches [56]: "*While the Android Keystore-backed implementation does not provide the same level of security and privacy it is perfectly adequate for both holders and issuers in cases where all data is issuer-signed*".

Data Exchange Formats and API

The basic (unextended) generic data model of CBOR is conceptually inspired by JSON and the standard explicitly specifies conversion procedures in both directions. More explicitly, we list the major primitive data types with their CDDL keyword and CBOR type number:

- `bool` (major type 7, additional information 20 or 21): Boolean value,
- `uint` (major type 0): unsigned integer,
- `nint` (major type 1): negative integer,
- `bstr` or `bytes` (major type 2): byte string,
- `tstr` or `text` (major type 3): text string.

Similar to JSON, CBOR supports

- arrays for repetition of uniform data elements and
- maps as a key-value dictionary, akin to JSON objects.

As alluded to, an mdoc is generic and can thus carry any CBOR element with arbitrary complexity.

Developers start interacting with the Identity Credential API by acquiring an instance of the `IdentityCredentialStore` class. A credential store can have different capabilities, the most notable properties are:

- A credential store is either hardware- or software-backed. Developers can explicitly request an instance of the credential store with either capability. However, hardware-backed instances are only supported with the native Android OS implementation, which requires Android 11 and secure hardware.
- Optionally, a credential store with direct-access support allows accessing credentials via specialized NFC hardware even while the Android OS is fully powered off. Note, that according to the Jetpack API documentation [56] any credentials stored in a direct-access store should always use reader authentication, since a device with no running instance of Android OS cannot verify the consent of the user.

Once the desired credential store instance is available, one can either retrieve existing `IdentityCredential` objects or store new ones. Every single credential is comprised of:

- A document type string. Usually, credentials can be of any document type, but direct-access credential stores can restrict support to a fixed number of document types.
- A list of one or more access control profiles, each with a locally unique ID. Each profile can specify whether user authentication is required or not and specify a timeout that defines how long a user authentication is valid. Additionally, the specification of a reader certificate enables reader authentication and thus limits access to entities that can present signed mdoc requests.

- A list of entries that store the actual data of the credential. A straight forward approach would be the storage of each identity credential attribute in a separate entry. Every entry has
  - a namespace, which groups entries into logical groups (a credential can contain entries for multiple namespaces),
  - a name, which is a string that, together with the namespace, uniquely identifies the entry,
  - a value, which is a CBOR data element and can thus be any arbitrarily complex semi-structured data, and
  - a set of access control profile IDs that are allowed to read this specific entry.
- A `CredentialKey` is an asymmetric key pair that is used to authenticate the mdoc holder to the IA and persists across the lifetime of the credential. After initial generation on the mdoc holder device, the certificate with the public key is sent and remembered by the IA. Some mdoc holder operations (e.g. provisioning and deletion of credential) provide a signed cryptographic proof for the IA.
- One or more tuples of *mobile security objects (MSO)* and `AuthKeys`. An instance of the former is a sufficient proof of validity of the mdoc to an mdoc reader, whereas an instance of the latter is an asymmetric key pair that authenticates the mdoc holder to the mdoc reader. They are closely related and part of the following provisioning workflow:
  1. Initially, the Android identity application requests the generation of one or more `AuthKeys`.
  2. The credential store generates them and signs certificates for each key with the persistent `CredentialKey`.
  3. Next, each `AuthKey` certificate is submitted to the IA and the latter creates corresponding MSOs, each signed by the IA and returned to the mdoc holder.

  This concludes provisioning of the mdoc and the mdoc holder can now use any MSO to authenticate the mdoc to an mdoc reader. The `AuthKey` of the Identity Credentials API is equivalent to the `SDeviceKey` of the ISO standard.

Note that the latest available version of the Jetpack implementation of the Identity Credentials API is currently `1.0.0-alpha03` and thus still an alpha release.

Possible Usage in Digidow

The Identity Credentials API is, as the name implies, designed from the ground up for the storage of identity documents. Thus, it is conceptually well aligned with the general goal of Digidow and is the obvious approach to implement the requirement R1 for the Android PIA. As per API design, we would like to store components IC1 and IC2 inside the credential store, while keeping the auxiliary data from component IC3 in the regular data and file storage. An entry in an mdoc is well suited to store an identity attribute and thus the composition of all entries/attributes logically forms a credential. Entries can be arbitrary binary values and can thus store biometric data (e.g. embeddings derived from face detection) directly without a bloated binary-to-text encoding. Additionally, the W3C Verifiable Credentials standard uses JSON as a de facto standard

encoding format and therefore the interoperability between CBOR and JSON is convenient for our purposes.

Currently, the ISO/IEC 18013-5 standard permits the following digital signature algorithms and curves (i.e. key pairs):

- "ES256": ECDSA with SHA-256, using either curve P-256 (aka. secp256r1) or brainpoolP256r1,
- "ES384": ECDSA with SHA-384, using either curve P-384 (aka. secp384r1), brainpoolP320r1 or brainpoolP384r1,
- "ES512": ECDSA with SHA-512, using either curve P-521 (aka. secp521r1) or brainpoolP512r1 or
- "EdDSA": EdDSA, using either curve Ed25519 or Ed448.

Support for the different signature algorithms and key sizes depends on the implementation (cf. section 4.2.3). Specifically,

- the Jetpack implementation has support for all mentioned algorithms above (due to delegation to BouncyCastle), while
- the native credential store implementation, backed by security hardware, supports only ECDSA with P-256 for all keys[14].

Section 4.2.2 lists the signature schemes (and associated key types) that we currently deem suitable for our purposes in Digidow to fulfill the measure M6. Unfortunately, there is no overlap between these and thus we are not aware of a (simple) solution to reconcile the technical capabilities of the Identity Credentials API with the requirements of Digidow.

Another hardship is the immaturity of the Jetpack Identity Credentials API. At the time of writing this work, said API is still an alpha release and thus subject to possibly major changes.

4.2.4 Direct Secure Element Access

A secure element (SE) has a dedicated smart card OS and can run multiple applets (cf. section 3.1.3). Direct access via APDUs to these applets may enable additional functionality that is not exposed via the Android OS (e.g. via the StrongBox Keymaster).

In addition to applets that are pre-installed by the SE vendor, new ones can also be provisioned. Depending on hardware capabilities, one could develop and deploy a custom applet for an SE that addresses the requirements of the PIA secret storage.

APIs Driven by Standards

Android defines hardware compatibility for SEs via their Compatibility Test Suite (CTS). Any SE that should be accessible by Android has to pass the Open Mobile API (OMAPI) test cases and satisfy test vectors that are provided by Google in the CTS [2]. Android implements the Open Mobile API (OMAPI) and

[14] See AIDL documentation for `CredentialKey` at https://android.googlesource.com/platform/hardware/interfaces/+/refs/heads/master/identity/aidl/android/hardware/identity/IIdentityCredentialStore.aidl#201 and AIDL documentation for AuthKey (i.e. signing key) at https://android.googlesource.com/platform/hardware/interfaces/+/refs/heads/master/identity/aidl/android/hardware/identity/IIdentityCredential.aidl#345

exposes all compliant SEs over this interface [44, 45]. Ultimately, after selecting a specific SE and opening a channel, an application developer exchanges APDUs with the SE via the `transmit` method of the `android.se.omapi.Channel` class.

ZKP via Elliptic Curve based Direct Anonymous Attestation (ECDAA)

The Crpyto library applets in some SEs[15] support a group signature protocol that is standardized under the name Elliptic Curve based Direct Anonymous Attestation (ECDAA) [22][16]. The standardized ECDAA protocol uses the CL signature scheme, one of the candidates listed in measure M6, and was originally developed for the Trusted Platform Module v2.0. As Chen and Li [28] demonstrate, the ECDAA protocol, here notably with trust anchored in security hardware, can be used to implement both

- a signature proof of knowledge (SPK), proving the possession of a secret without revealing any piece of the secret (hence the name "anonymous attestation"), and
- derived credentials that selectively disclose attributes, as shown via an ECDAA-based design of the U-Prove system[17].

The latter is a proof of concept for the usage of hardware supported ZKPs to realize derived credentials.

Potential Usage in Digidow

Direct access to an SE provides a wide range of possible usages in Digidow. In the simplest case, we would store the secret keys associated with the digital identities, i.e. component IC2. The support of ECDAA in pre-deployed applets of some contemporary SE hardware makes this approach possible from a technical perspective. ECDAA uses the CL signature scheme and thus, crucially, satisfies the requirements from measure M6.

Depending on the storage capabilities inside such a secure hardware, we might want to store the main credential, i.e. the component IC1, inside the SE. Such a scheme should involve asymmetric cryptography (or a similar cryptographic operation) between the SE of the PIA and the Digidow entity that needs to operate on the plain text attributes inside a credential. Making credential attributes inaccessible for the REE portion of the PIA provides stronger security against attackers with system level privileges. However, a simple solution comes with notable usability restrictions that prohibit even the Digidow manager app from showing attributes to the user. A possible trade-off is a more advanced scheme, where a subset of sensitive attributes are hidden inside the SE, while less sensitive ones are available to the REE.

Unfortunately, the complexities around SE access control and the current state of TSMs makes it unlikely that a research project of our scale will receive the necessary support from SE issuers. Without inclusion in the SE access rules, we can neither use custom APDUs to call ECDAA related features from pre-installed applets, nor deploy a custom applet into the SE.

[15] As far as the author is aware, that includes several NXP SEs: Namely at least SN100, SN200, SN220, SE050 and SE051. This was inferred based on the certifications for these products at https://www.commoncriteriaportal.org/products/#IC.

[16] The cited version of the proposed standard (dated 2017-04-11) is the exact version referenced by the certification documents from NXP. Newer version are available and the latest one is a "Draft 02" (dated 2018-07-02) at https://fidoalliance.org/specs/fido-v2.0-rd-20180702/fido-ecdaa-algorithm-v2.0-rd-20180702.pdf.

[17] https://www.microsoft.com/u-prove

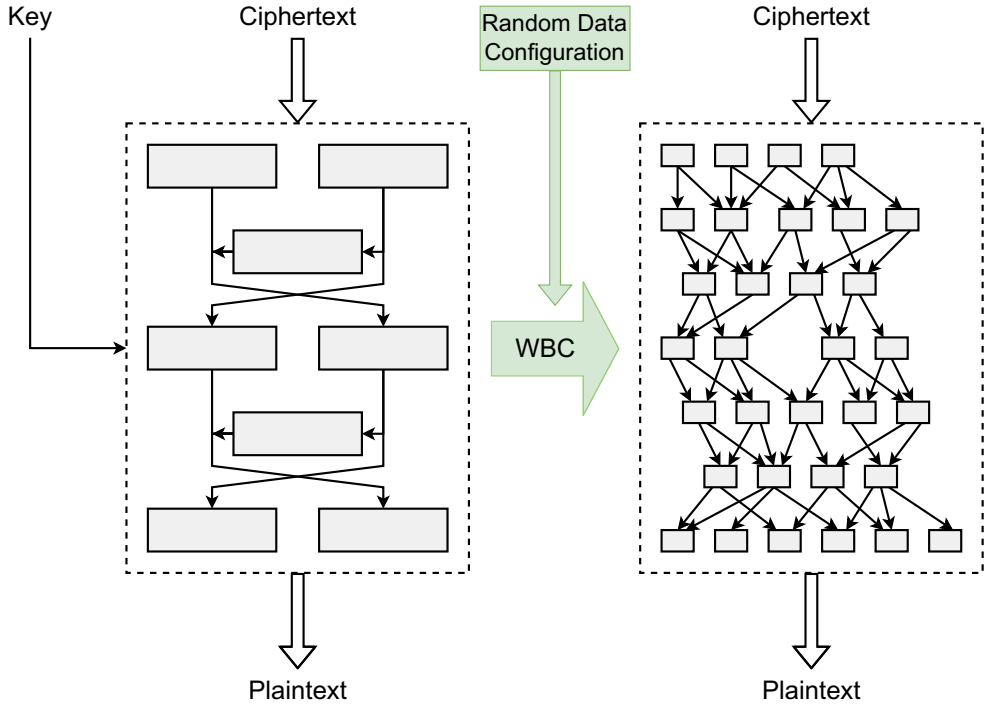


Figure 4.2: General architecture of white-box cryptography (WBC) with a fixed key [18].

4.2.5 White Box Cryptography

Cryptography aims to ensure security properties on a secure channel between two or more end points. Conventionally, any end point performing cryptographic operations with a private/secret key is assumed to be trustworthy and thus a mere oracle, i.e. an input-output black-box, from an attackers point of view. Conversely, a white-box context assumes that such a sensitive cryptographic operation is performed on a device with full access by an attacker [18]. Most notably, the latter has access to the binary of cryptographic operation and can observe its execution. Thus, the goal of white-box cryptography (WBC) is to protect sensitive key material from attackers in a white-box context.

The Kerckhoffs' principle, a standard assumption of modern cryptography, demands that cryptographic algorithms themselves are considered public knowledge. Crucially, the parametrization with sensitive key material needs to be protected. There are two main approaches to WBC:

- Either the hard coded sensitive key material is compiled into the algorithm in obfuscated format (cf. Figure 4.2) or
- a dynamic white-box implementation receives a protected version of the key at runtime and is processed by the obfuscated cryptographic operation in a way that never exposes the "raw" key directly.

Usefulness for Digidow

The main advantage of WBC for Digidow is the possibility to implement arbitrary group signatures and ZKP in the REE, while offering some protection against attackers with system level privileges. In case of Digidow, the secret

key, described as component IC2, would be unique for each provisioned credential. Subsequently the WBC code for each credential will be unique and would have to be provided by the IA. While we do have a weak trust assumption from the PIA to the IA, executing arbitrary black-box code provided by the IA is clearly problematic. Together, these circumstances mean that only a dynamic WBC system is reasonable for Digidow.

Beyond the mere technical possibility, it should be emphasized that WBC is inherently an obfuscation process. As such it is subject to an endless cat and mouse game between researchers creating and breaking WBC schemes [15]. With a sufficiently large incentive, as would undoubtedly be the case for a wide spread adoption of the Digidow system, any WBC scheme will be broken. Ultimately, while possible on a technical level, we are under the impression that WBC does not provide sufficient security guarantees.

Chapter 5

Implementation of a PIA for Android

Even when adhering to modern software engineering practices, which in our case heavily revolve around security and privacy aspects (threat modeling, designing conceptual security, and privacy measures), implementation work offers a significant degree of freedom. Therefore, we believe it is important to present the challenges and possible solutions of the implementation. This is essential to communicate the full picture of our work around the Android PIA.

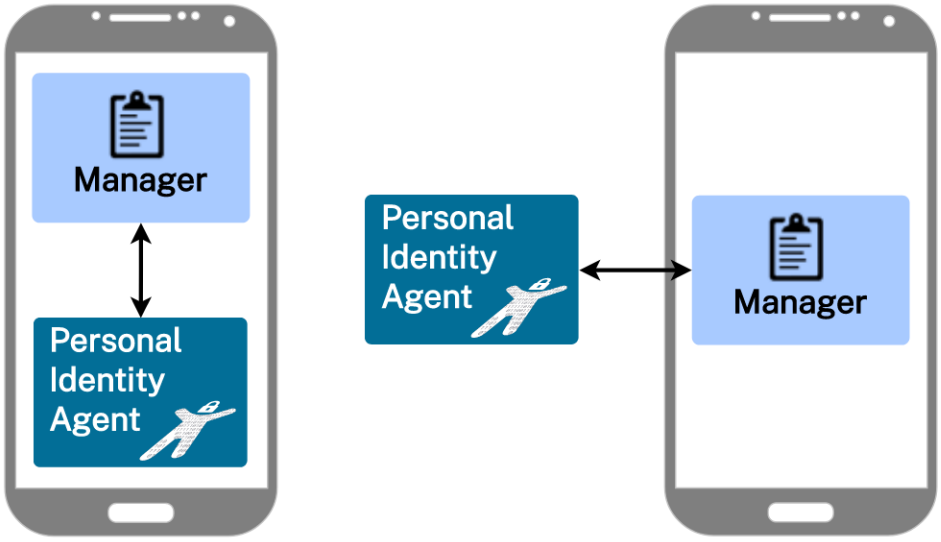
5.1 Auxiliary Requirements from Digidow Project Context

The larger Digidow project context gives rise to auxiliary requirements. Previously, other work streams started the development of a remote (i.e. standalone) PIA, written in Rust, that is expected to run on a trusted host. This is in contrast to the embedded PIA, running on the same mobile phone as the Digidow manager app, that is being developed as practical part of this work. Refer to Figure 5.1 for a comparison. We envision code sharing between these two variants has the following advantages and disadvantages:

+ Code reuse saves time that would be spent on implementing the same features twice.
+ Single source of truth for business logic avoids potential inconsistent behavior.
+ Having a single implementation of the core data structures and business logic reduces complexity and thus helps with maintainability. Furthermore, this promises to help with semi-formal verification.
+ Rust, as modern systems programming, language offers a more secure foundation and eliminates some classes of memory errors entirely via language design.
- Initial upfront development cost for unified core code base.
- Language interoperability between the official Android development languages Kotlin/Java[1] and Rust is immature and thus requires extra effort.

Overall, we believe that the advantages outweigh the downsides and thus opted for a shared PIA core code base. This means the following auxiliary technical requirements, in addition to requirements R1 through R3 introduced in section 3.2.1, arise for the embedded PIA and the (closely related) Android Digidow manager app implementation:

[1] Alternative approaches for the Android Digidow manager app (e.g. C++ via NDK, C# via Xamarin or Unity, Lua via Corona, JavaScript via PhoneGap) were considered, but none have significant unique aspects that would warrant their usage for our purposes. Therefore, the default choice for Android app development are the two first-class languages supported by Google.



(a) Embedded PIA variant (b) Remote (i.e. Standalone) PIA variant

Figure 5.1: The two PIA variants that are currently in scope for the Digidow research project, whereas this master thesis is mainly concerned with the embedded PIA. Both are shown with the Digidow manager app that allows user interactions by the individual.

R4 The embedded PIA is implemented in Rust.

R5 The Android Digidow manager app facilitates interactions with the individual and uses the embedded PIA on the same mobile phone.

5.2 Architecture

As architectural ground work, the code base of the PIA was restructured into the following 3 subprojects:

- The *core* subproject is a common basis for essential data structures and unified business logic.
- The *standalone* subproject contains all parts exclusive to the remote variant and thus mainly exposes features via REST API over Tor.
- The *embedded* subproject hosts portions exclusive to the eponymous PIA variant and is therefore mainly populated with glue code that bridges the gap to the Android Digidow manager app.

Both, the standalone and embedded subproject, depend on the core to provide them with an interface to the PIA features. Together these 3 subprojects form a Rust workspace.

5.2.1 Language Interoperability Through Java Native Interface

Interoperability between programming languages can be done in several ways. The most notable approaches are:

- Usage of inter-process communication (IPC) mechanisms from the OS (e.g. files, TCP/IP or Unix sockets, pipes, shared memory, signals) to exchange data between two or more processes implemented in different languages [125].
- Many languages offer one or more foreign function interfaces (FFI) that allow one programming language to call another within the same process. The majority of FFIs are between a high level language and the systems programming language C[2].
- A runtime, also called process virtual machine (VM), executes code in a hardware independent execution environment. It supports one or more high level programming languages that are all based on the same abstractions (e.g. data types, memory management) and therefore commonly feature interoperability between each other. Such applications spanning multiple languages are called polyglot.

In our specific case we need interoperability between

- the official Android development languages Kotlin/Java (cf. requirement R5) and
- Rust, due to requirement R4.

Rust, as an unmanaged systems programming language, produces platform dependent native code. Our implementation platform of choice, Android, provides the Android Native Development Kit (NDK) as FFI for integrating native code with the official Android development languages [55]. The provided tools enable the usage of the Java Native Interface (JNI) [96]. JNI is a well specified standard interface between high level JVM languages and native code. This standard enables portability for both high level and low level code across different conforming JVM implementations. Subsequently, we will present the basics of JNI, including the different strategies for handling objects and data structures over the FFI.

JNI Basics

We introduce the basics of JNI via the official documentation by Oracle [96]. This establishes the general features and rules of the standard FFI offered by the JVM.

As a starting point, the *interface pointer* of type `JNIEnv` points to an array of JNI function pointers, each offering some features of the JNI interface. It can be used by the low level code to perform a call to the JVM and is provided in all calls from a high level language to the low level code. Due to the multithreaded nature of the JVM and the high level languages running on it, compilers for low level code need to be multithread aware. Developers need to be mindful of the thread dependent nature of the interface pointer. The well defined specification has clear rules on mapping method names of the high level language to qualified procedure identifiers for the low level language.

All Java primitive types (e.g. `jboolean`, `jbyte`, `jint`) are simply copied between high and low level. On the flip side, objects (`jobject`) are passed by reference. By default, all objects are JNI *local references* and only valid for the duration of the specific JNI call. If an object needs to be retained for longer, a programmer can opt to promote it to a *global reference*. Both kinds of references can (in case of a global one must) be freed at any time to allow garbage collection (GC) of the

[2]Which is a de facto standard for the low level portions of user space.

VM to proceed. A crucial design aspect of JNI is that direct C-structure like access to `jobjects` is generally not possible and developers need to use accessor functions. Only a small subset of APIs do support direct memory access (primitive arrays and strings depending on circumstances, new IO (NIO) related entry points), these will be addressed briefly later on. Exceptions can be in raised native code. In case of an exception during a JNI call into the high level language, the low level code may handle and clear these. Any outstanding exceptions are propagated back to the code that initiated the JNI call.

Strategies for Handling of Objects

Objects, regardless if they are simple struct-like aggregations of fields, arrays of data elements, or any arbitrary abstract data type, require some consideration for usage as part of JNI. This is mainly due to the inherent differences in memory management between managed JVM languages and unmanaged native languages. Based on experience and reflections on this topic the author is aware of the following four strategies to handle objects in JNI:

1. Any object that passes through low level code, but is only processed in high level code, can simply be passed as opaque `jobject` value in the low level code.
2. Conversely, any data structure that passes through high level code, but is only processed in low level code, can be represented by a pointer value and thus copied as primitive (commonly `jlong` or `jint`, depending on platform) in the high level code.
3. Whenever it is acceptable that an object traverses the FFI-boundary with a copy semantic (i.e. field changes in one domain do not reflect into the other), the object is serialized into a data type that can natively cross the JNI boundary and deserialized on the other side back into a rich object again. One such design is a string as data type[3] and JSON as format with highly efficient (de)serialization.
4. In case the exact object needs to be preserved across FFI domains, one has to settle on a storage location for said object. Effectively this results in three sub-strategies:
   a) If the object resides in the managed JVM language, all low level code will need to call high level accessor functions from low level code via JNI. E.g. by iterating fields of a class and calling the appropriate `Get<Type>Field` JNI function.
   b) Conversely, any object residing in the low level language is represented by an opaque pointer value and one needs to call custom low level accessor functions to access fields or perform operations on it.
   c) Java new IO (NIO) API can allocate direct buffers and JNI can, optionally, provide access to these for low level code. This provides arbitrary random read/write access to both domains and as such is the most generic approach to object/data structure management across FFI boundaries. However, it requires a mutual understanding of data layout.

[3] JNI offers APIs for efficient access to the low level representation of these.

Development Workflow for Standard JNI

The standard development workflow of JNI with Java and C/C++ roughly involves three steps [96]. Developers

1. write Java code, where the JNI methods are designated with the `native` keyword,
2. generate native header files by executing the `javac` compiler with the `-h` flag [95][4] and
3. compile the C/C++ code, including the implemented functionality corresponding to the generated headers, into a shared library.

From now on, the JVM portion of the application has a dependency on the native shared library, meaning the latter needs to be available to users executing the application. During runtime, the Java code needs to call `System.loadLibrary("my_library")`, which loads the appropriate shared library for the platform, and subsequently calls to the implemented native methods are possible. In summary, this standard JNI workflow entails a developer-defined interface in the high level JVM language and a tool-generated interface (in case of C/C++: preprocessor header files) for the low level language.

JNI in Rust Through flapigen

The official documentation, both for the NDK [55] and JNI [96], only mention support for C and C++ as low level languages. However, in practice many low level languages provide JNI interoperability due to the popularity of the JVM ecosystem.

Rust itself provides an FFI interface to other unmanaged languages based on several application binary interfaces (ABI)[5], most prominently C, as a first-class citizen of the language.

Thus we find, that neither Java nor Rust provide first-class support for an FFI between each other. Regardless, there are several Rust community projects, called "crates" in Rust terminology, that aim to enable JNI between these languages as follows:

1. They provide type information for native JNI data structures and methods, with varying degrees of abstraction across different crates.
2. Developers write Rust code that uses the aforementioned and implement required functionality of the specific interface.
3. Said Rust code is then compiled as a shared library with a C ABI (specify `crate_type = ["cdylib"]` in `Cargo.toml`).

At runtime, the JVM can load and use this shared library, which behaves indistinguishable from a C/C++ based implementation of the low level language portion. Prominent examples include the crates `jni`[6], `jni-sys`[7] and, specifically for the Android NDK, `ndk`[8].

[4] This feature was formerly available as standalone `javah` command.
[5] See https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#using-extern-functions-to-call-external-code for a basic introduction and https://doc.rust-lang.org/nomicon/ffi.html for a detailed documentation on all supported ABIs and an example.
[6] https://crates.io/crates/jni
[7] https://crates.io/crates/jni-sys
[8] https://crates.io/crates/ndk

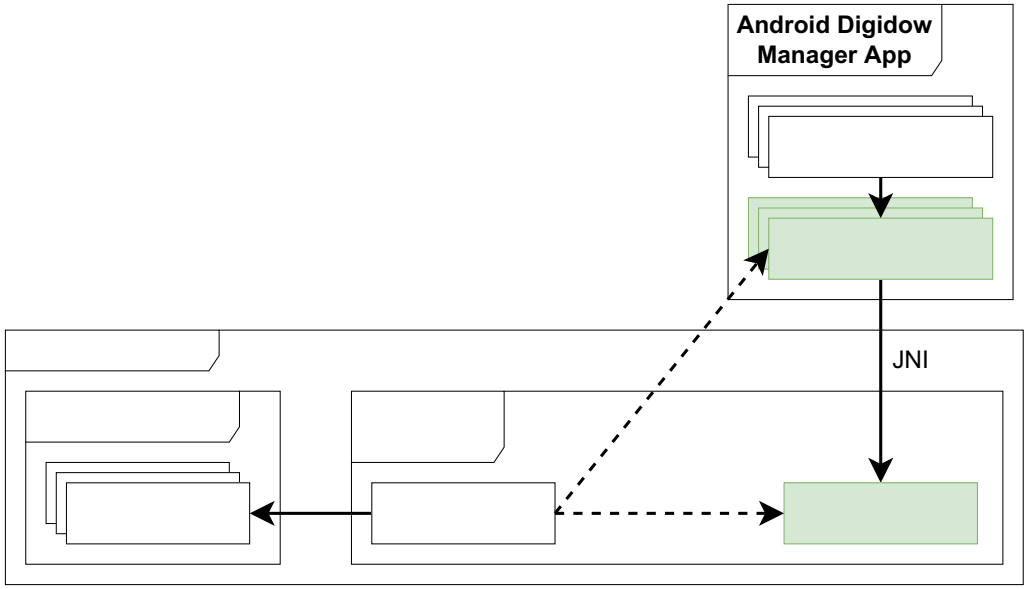


Figure 5.2: The combined architecture of the embedded PIA and the Android Digidow manager app with focus on parts relevant for the language interoperability via the Rust crate `flapigen`. Regular arrows represent dependencies. Green boxes indicate generated code with information flowing in the direction indicated by the dashed arrows.

We settled on using the crate `flapigen`[9], which is an abstraction layer over the crate `jni-sys`. Using the crates `jni-sys` or `jni` directly has some notable downsides:

- There is no automated way to generate the interface for one side of JNI via the developer-specified interface from the other side. This makes the connection between JVM language and Rust very brittle and requires manually syncing the interface specification between both sides.
- JNI code, especially one involving complex objects and data structures (cf. section 5.2.1), requires some boilerplate code on both sides of the FFI boundary.

`flapigen` provides a so called *foreign language API* and improves on these issues noticeably. A developer specifies an FFI interface for their application via the `flapigen` foreign language API. Subsequently, the developer-specified FFI interface in Rust is used to automatically generate both

- the lower portion of JNI code in Rust (based on the crate `jni-sys`) and
- the higher portion of JNI code in Java.

This architecture, involving both the embedded PIA and the Android Digidow manager app is visualized in Figure 5.2.

E.g. consider the functionality around identities that is visualized as an example in the aforementioned figure. The `trait IdentityApi` and its implementation reside in the `identity_api.rs` Rust source file in the core subproject. Building upon this, the embedded subproject contains a single `java_glue.rs.in` file with all developer-written rules in the `flapigen` foreign language API. This includes:

[9] https://crates.io/crates/flapigen

- `foreign_class!(/* class specification */)` macro rules that define a certain set of Rust functions, commonly taken from a trait, to be exposed as a generated Java class. In case of our running example, this involves the methods of the `trait IdentityApi` shown in Listing 5.1.
- `foreign_typemap!(/* mapping code */)` macro rules that define how types that already exist on both sides should be translated to each other. E.g. An object of Java type `Optional<String>` needs to be translated to a `String/jstring` that can cross the FFI boundary and then converted back into a Rust type `Option<String>`, as seen in Listing 5.2.

As previously mentioned, this FFI interface is used to generate the actual JNI code for Rust (in `java_glue.rs`) and Java (specifically in `IdentityApi.java` for the running example). Thus the Digidow manager app can access the identity related functionality of the PIA.

5.2.2 Implementation of Secret Storage

In chapter 4 we performed an extensive analysis on different approaches for secret storage. Subsequently we document the implementation of our currently selected approach and propose a concept for a more sophisticated future implementation.

Simple File-Based Storage

As the conclusion lays out (cf. section 7.1), we implement file-based storage in the app-private data directory as a fallback solution since more sophisticated approaches are currently not possible due to technical or organizational issues. The existing standalone PIA persists data in a semi-structured JSON file. This means that our current solution can reuse existing work and requires only minor extensions. Whereas the standalone PIA relies on the current working directory to be a good location for storing the PIA state, we cannot make this assumption for the embedded PIA. Instead we devised the following approach to handle the file storage on Android:

1. The Digidow manager app determines the location of the app-private file storage directory.
2. It passes this information to the embedded PIA via a context initialization method.
3. The embedded PIA stores the persisted PIA state under this directory.

The implementation of this design works as expected and allows keeping PIA persistence handling part of the core subject shared across variants.

```
foreign_class!(class IdentityApi {
    self_type dyn IdentityApi;
    constructor get_identity_api_trait() -> Box<Box<dyn IdentityApi>>;

    fn IdentityApi::enrollment(&self, enroll: Enroll) -> Result<(), std::io::Error>;
    fn IdentityApi::get_identities(&self) -> Vec<Identity>;
    fn IdentityApi::get_identity(&self, id: String) -> Result<Identity, std::io::Error>;
    fn IdentityApi::update_identity(&self, id: String, value: Identity) -> Result<Identity, std::io::Error>;
    fn IdentityApi::delete_identity(&self, id: String) -> Result<Identity, std::io::Error>;
});
```

Listing 5.1: flapigen foreign language API rule that exposes the Rust `trait IdentityApi` with all its methods as a Java class.

```
fn jstring_to_option_string(env: *mut JNIEnv, js: jstring) -> Option<String> {
    let chars = if !js.is_null() {
        unsafe { (**env).GetStringUTFChars.unwrap()(env, js, ::std::ptr::null_mut()) }
    } else {
        ::std::ptr::null_mut()
    };

    if !chars.is_null() {
        let s = unsafe { ::std::ffi::CStr::from_ptr(chars) };
        let r_str = s.to_str().unwrap();

        match r_str {
            s if r_str.starts_with("Some:") => Some(s[5..].to_string()),
            "None" => None,
            _ => panic!("unrecognized Option<String> value!"),
        }
    } else {
        None
    }
}

foreign_typemap!(
    ($p:r_type) Option<String> <= jstring {
        let r_option_string = jstring_to_option_string(env, $p);
        $out = r_option_string;
    };
    ($p:f_type, option = "NoNullAnnotations") <= "java.util.Optional<String>"
    r#"/*Option<String> <= jstring <= @NonNull java.util.Optional<String>*/
    $out = $p.map(s -> ("Some:" + s)).orElse("None");
    "#;
    ($p:f_type, option = "NullAnnotations") <= "@Nullable java.util.Optional<String>"
    r#"/*Option<String> <= jstring <= @Nullable java.util.Optional<String>*/
    $out = $p.map(s -> ("Some:" + s)).orElse("None");
    "#;
);
```

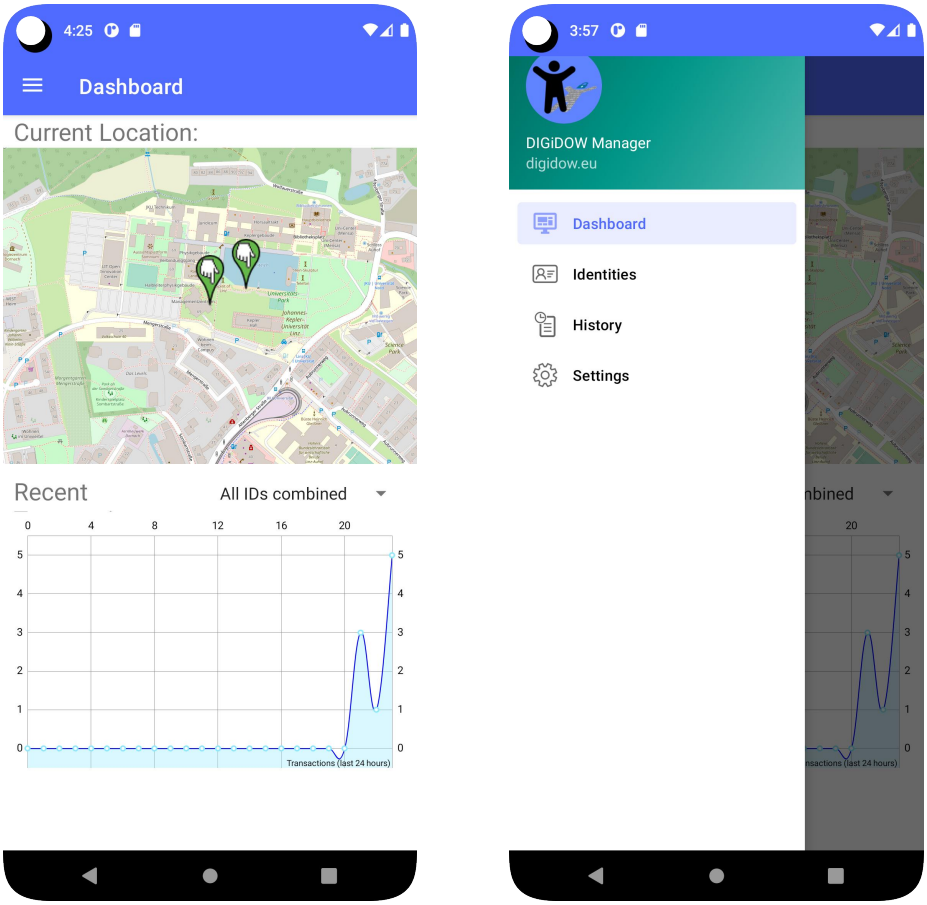
Listing 5.2: flapigen foreign language API rule (and a helper function) that defines a mapping of the Java type `Optional<String>` to a Rust type `Option<String>`

Concept for Usage of Dedicated Android APIs

The preferred solution for secret storage on Android, once a newer Android offers algorithms and keys suitable for ZKP, is the usage of dedicated Android APIs (e.g. Keystore system or the Identity Credentials API). This is not compatible with the current approach of a unified persistence handling in the PIA core subproject written in Rust. To overcome this issue, we envision the following architecture for a future implementation:

1. The PIA core subproject defines an interface for persisting identities and other sensitive data.
2. Each variant must provide an implementation for the persistence API. The embedded PIA (in close collaboration with the Digidow manager) should provide an implementation that delegates this task to the dedicated Android API. On the other hand, the standalone variant can continue using the current file-based approach (or whatever strategy is optimal for their platform).

Such a design would entail a more complicated control flow across the JNI boundary. Whereas the currently implemented file-based storage only performs calls from high level JVM languages into Rust, an implementation of the envisioned persistence API would require calls from Rust to dedicated Android APIs (which are implemented in JVM languages). While possible on a technical level, multiple crossings of the JNI boundary requires careful engineering work. Beyond the usual complexities (e.g. consistent interface with a single source of truth, handling of objects), one has to be mindful of the exception state of the



(a) Dashboard screen (b) Navigation drawer

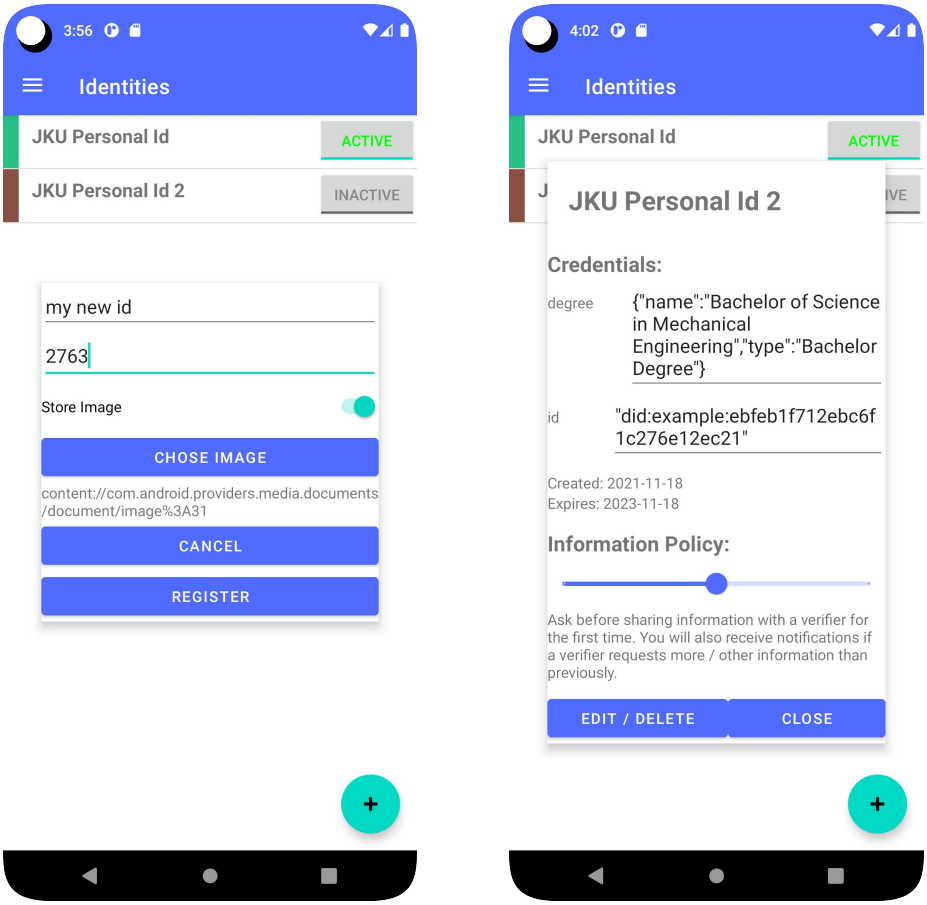
Figure 5.3: Dashboard and navigation drawer of the Android Digidow manager app.

JVM. As outlined in the previous subsection, any potential JVM exception need to be either cleared or propagated by the low level code. The correct behavior might vary depending on the specific exception that occurred on the JVM side.

5.3 Functionality Overview via the Current App UI

In order to convey the high level functionality of the embedded PIA we will present the current state of the Android Digidow manager app user interface (UI). It should be noted that the practical portion of this thesis was heavily focused on the embedded PIA and not the Android Digidow manager app. While some technical improvements were performed, the UI remains unchanged in the state that was available for the remote PIA. Furthermore, on the other end of the spectrum, the core subproject of the PIA is itself a work in progress and thus features placeholder data in some locations.

Upon launching the app, an initial dashboard is shown to the user (cf. Figure 5.3a). The current version entails a map for tracking Digidow entity interactions in the physical world and a line chart showing transactions over the course of time. As primary means of navigation, there is a navigation drawer in the left side of the app (cf. Figure 5.3b).

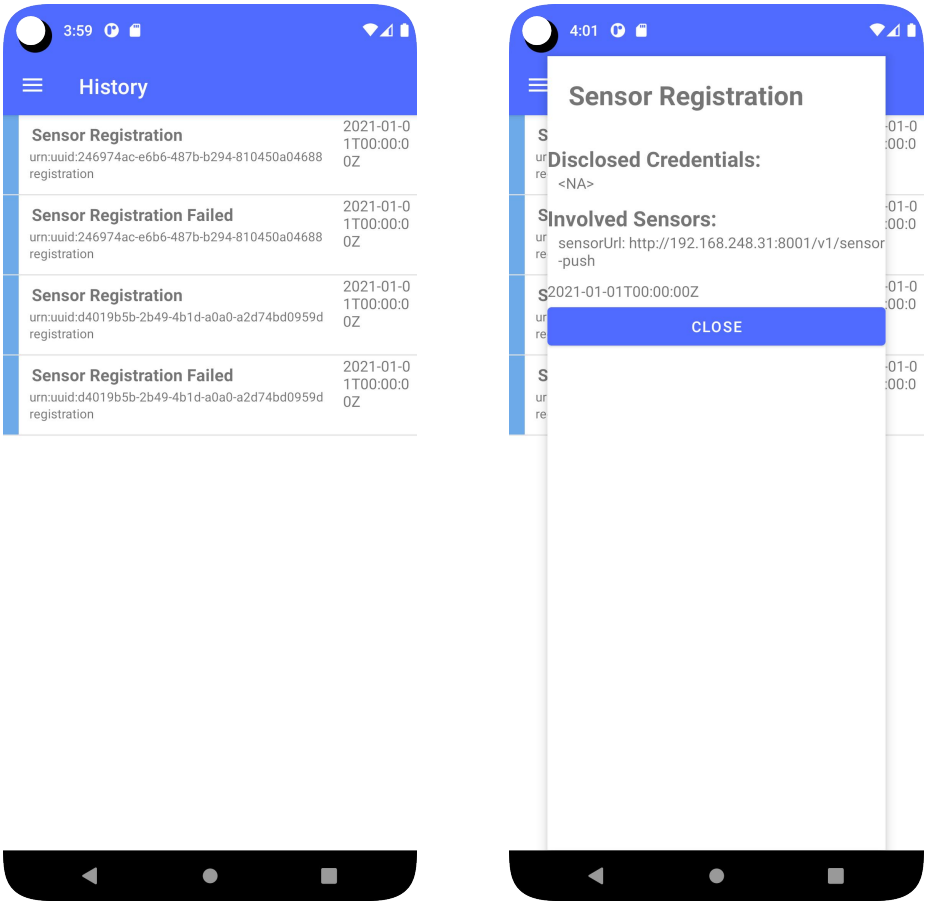


(a) Identity Enrollment (b) Identity Details

Figure 5.4: Identity related popups of the Android Digidow manager app. List of identities can be seen in the background of both screenshots.

At the heart of the application is the management of the identities that are stored in the PIA (cf. requirement R1). The user can see a list of identities and can enroll additional ones via a popup menu (both shown in Figure 5.4a). While technically functional, the current implementation of identity enrollment is geared towards quick development and does not feature all envisioned security aspects of the interaction between PIA and issuing authority. Specifically, the current version of the enrollment process allows any individual to create an identity by providing a photo and a valid password. Said photo is sent to the sensor, processed there into a into an embedding (i.e. feature vector), and subsequently returned to the PIA. This concludes the simple provisioning process and enables usage of the identity in the Digidow identity system. The current provisioning process is merely a functional prototype for development and does not reflect a proper process with security and privacy considerations. Among the numerous and severe issues are: There is no validation of the binding between the individual and the provided photo, the sensor receives the full biometric for embedding calculation, and identities are not trust-anchored into an IA.

Individual identities can be managed via a detail popup, where attributes are shown and the information privacy policy can be adjusted (cf. Figure 5.5b). The latter defines how often the PIA inquires with the individual (via the Digidow manager app UI) concerning the usage of an identity.



(a) History List (b) History Transaction Details

Figure 5.5: History of transactions in the Android Digidow manager app.

Furthermore, users can inspect the history of interactions between the PIA and other Digidow entities (cf. requirement R2). Once again we use a list to visualize the possibly numerous interactions (cf. Figure 5.5a). Each transaction can be inspected via a detail popup, as shown in Figure 5.5b.

Chapter 6

Investigating Reproducibility of the Embedded PIA

Trust in software is deeply linked to the software supply chain. Part of a good one is reproducibility, which bridges the gap between source code and executable artifact. We investigate the reproducibility of the embedded PIA and present the necessary adjustments to achieve this property.

6.1 Trust and the Software Supply Chain

A trust relationship between a user and their software (here: PIA) is important for acceptance and broader adoption of the latter. For our purposes we need to distinguish between soft and hard trust [66].

The Digidow project is complex and aims to use state-of-the-art science to achieve its designated goals. As such, we need to highlight that a precondition is the soft trust of an individual in their PIA and the larger Digidow system with all its underlying pieces. I.e. one needs to trust the abstract concept of a digital representative, that holds crucial credential data on an individual and performs physical location tracking, in order to enable a decentralized authentication system. This is inherently a complex question that involves e.g. personal knowledge and general attitude towards science. For these reasons the soft trust is out of scope for this work and simply assumed.

Relevant for this work is the hard trust relationship between an individual and their concrete software running on their device. I.e. that portion of trust that can be objectively measured by e.g. verifying a digital signature. Already in 1974 Karger and Schell [74] recognized that a compiler needs to be trustworthy. This notion was picked up by Thompson in his now infamous work "Reflections on Trusting Trust" [128], which observes that once a binary has been equipped with a back door, it matters little that all source code looks benign. This is especially troublesome if a malicious compiler is used to produce another compiler

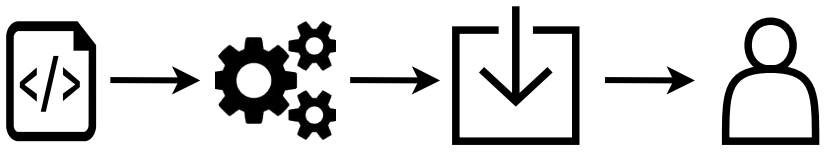


Figure 6.1: A simple model of a software supply chain with the following steps (from left to right) and transitions: Source code (1) is compiled into an executable artifact (2), which is packaged into one or several distribution formats (3) and finally delivered to the end user running the software (4).

and thus propagates the back door. Furthermore, he notes that this problem is not limited to a compiler. All "program-handling" programs (e.g. assembler, loader or even hardware microcode) can have a back door. This has led to the modern term software supply chain, which refers to the collection of all components and processes involved in the creation of an executable artifact run by an end user [80, 91]. The better the accountability of the software supply chain is, the higher the trust of end user. A simple model of a software supply chain is depicted in Figure 6.1. Strong trust assertions about any step in the software supply chain depend on all previous steps and underlying components.

6.2 Reproducibility Challenges and Solutions

One key area of a software supply chain is the step from source code to executable artifact, which involves build tooling like previously mentioned compilers. This specific step is addressed by reproducible builds, which entail that identical source needs to result in identical executable artifacts. The build process is only one of many steps in a software supply chain. Therefore, trust in the previous steps (e.g. source code fetching, dependency auditing) and underlying components (e.g. running OS, hardware and their microcode) is a precondition. Only then it becomes possible to translate reproducibility to strong trust claims on the executable artifacts.

In contrast to the previously listed component, the build process (e.g. compiler, linker, packaging tools) is not assumed to be trusted. If reproducibility of a given software with a given build system does exist, it proves trustworthiness for the specific software being compiled. While this does increase the confidence in the build system in question, it is no formal proof that the specific build system behaves correctly, especially in a non-malicious way, for all possible input source code or configurations.

6.2.1 Deterministic Build System and Normalization

A general requirement for build reproducibility is a deterministic build system, which means that stable inputs for source code and configuration result in stable outputs for the executable artifacts [8, 13, 106]. Even if the deterministic property of the build system holds for a given source code and configuration, problems can arise from build context. Most build systems treat some of the latter as input as well and thus propagate any variations of these as varying outputs into the executable artifact. Common examples for such build context are

- various timestamps (e.g. compilation or file system timestamps),
- system properties (e.g. hostname, kernel version, specific hardware) and user information (e.g. username, locale),
- paths (e.g. absolute path to source or build output, additional paths for loader library lookup).

Ideally, output formats should not include any of these and source code should never query the build tooling for them. However, some file formats mandate the presence of certain build context properties.

The goal of a good deterministic build system is to normalize build context as much as possible. Depending on the necessity of a build context property, the build system should adopt either of the following *normalization strategies*:

- Omission/Removal: Provided a property is not essential for the functionality, it should be omitted or removed entirely from an artifact. The inclusion of many values is for convenience only (e.g. debugging purposes) and therefore, they belong in this category.
- Deterministic deduction: Otherwise, the build system should infer a deterministic value. E.g. if a package format mandates a build time stamp, one can take the timestamp of the latest commit and thus receives a stable, but still useful, value for unchanged source code.

Orthogonal to this is the *time of normalization*:

- At emission: If the tool that captures and injects the property in question can itself be configured or modified, one can apply the normalization ahead of property emission. This is preferable since it ensures that intermediary artifacts are not tainted via the build context property in question.
- Build wrapping: If the former approach is not possible, one can apply a pre- or post-process step that performs the normalization for individual build steps or the overall build artifact. Such a strategy is useful to wrap existing build tooling and thereby reduce or eliminate the leaked build context from the final build time artifact. E.g. the Nix package manager has a bundled shell script called `set-source-date-epoch-to-latest.sh`[1] which can easily be included in custom Nix build scripts. It sets the `SOURCE_DATE_EPOCH` environment variable to the most recent file modification timestamp in a pre-build hook and thereby instructs invoked build tools or entire systems to use the normalized value. On the other end of the spectrum, the `strip-nondeterminism` program[2] from the Reproducible Builds project strips several build context parameters from build artifacts.
- Semantic equivalence: Finally, a more theoretical approach is a semantic equivalence check that normalizes known leaked build context at runtime of the comparison [121]. This is closely related to *accountable builds* [106], which refers to builds that only differ by explainable differences. If these explainable differences can be normalized in an automated programmatic fashion, such a normalization can act as basis for a semantic equivalence check. Such a late time of normalization is the only possible approach to handle build parameters that do need to be different for the final build time executable artifact (e.g. digital signatures).

6.2.2 Verification of Reproducibility

A straight forward methodology for verification of reproducibility is based on performing multiple builds with a specific stable input. Ideally, independent builds are performed on different machines and hardware by different parties. This concept is visualized in Figure 6.2.

If the resulting outputs are stable across all builds, it follows that the specific build is reproducible. Part of the build input is (next to the actual source code) a configuration of the desired output (e.g. enable/disable build time features, target ISA/ABI). Therefore, it is also desirable to test different configurations for reproducibility to gain confidence that reproducibility holds for a certain source state[3].

[1] https://nixos.org/manual/nixpkgs/stable/#set-source-date-epoch-to-latest.sh
[2] https://salsa.debian.org/reproducible-builds/strip-nondeterminism
[3] However, as with any testing approach (in contrast to formal verification), this is only about increased confidence. Certainty about the reproducibility for all configurations is only possible by checking all configurations.

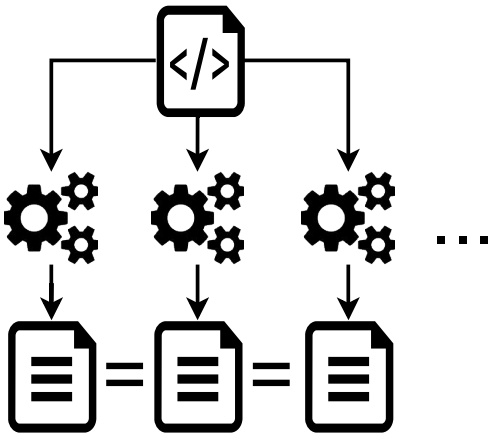


Figure 6.2: Multiple independent builds are performed and subsequently compared for equality.

6.3 Evaluation of Embedded PIA

As a preamble to the actual reproducibility evaluation we briefly explain the current build process of the embedded PIA and how it relates to the CI used by the Digidow project. The embedded PIA is part of a Rust codebase with subprojects (cf. section 5.2) and thus all subprojects use the cargo build tooling – the native build system for Rust. In addition to this, other Digidow work streams that are concerned with the standalone PIA have added a Nix build configuration for that subproject specifically. The Nix build configuration wraps the cargo build configuration and is used in the CI to perform precisely defined CI builds. Furthermore, this existing work uses Nix to perform a rebuild and thus check for reproducibility.

In order to enable measurement of reproducibility (and to maintain consistency with the larger project) we have decided to adopt a Nix build configuration for the embedded PIA as well. Such a configuration entails pinning of all dependencies and precise instructions of their assembly. It's worth noting that there are currently four prominent target tuples related to Android[4] and we build the embedded PIA for all of them.

Initial testing of the embedded PIA has shown non-reproducibility. Analysis of the binary artifact via the diffoscope tool[5] from the Reproducible Builds project indicates a reordering of global variable symbols. As a consequence, code accessing these symbols also exhibit differences. This difference can be traced back to the generated `java_glue.rs` file having variable definition order differences on the source level between compilations (cf. Listing 6.1). Conceptually, `java_glue.rs` is generated by flapigen (cf. section 5.2.1) via a Rust build script[6], a general purpose pre-build hook which is used in this context for code generation. Subsequently, we located the code responsible for the generation of these variable definitions inside flapigen. The issue was the usage of a `std::collections::HashMap` instance with a naïve invocation of the `values` method, which per API doc provides "*An iterator visiting all values in arbitrary order*" [109]. We swapped the data structure for a different map type with well defined iteration order and provided a pull request to the upstream project [105].

[4] Specifically these are `armv7-linux-androideabi`, `aarch64-linux-android`, `i686-linux-android` and `x86_64-linux-android`. The last two are useful for emulation.
[5] https://diffoscope.org/
[6] https://doc.rust-lang.org/cargo/reference/build-scripts.html

From a theoretical point of view, flapigen build scripts were making the build process non-deterministic. Fortunately, once the root cause was understood, this was a simple fix (in contrast to complex issues around build context properties that are mandated by various file formats or use cases). After adoption of this fix, our builds of the embedded PIA are now fully reproducible for all tested targets, including the four Android target tuples.

```
@@ -1526,13 +1526,29 @@
     let this: Box<Box<dyn TransactionApi>> = unsafe { Box::from_raw(this) };
     drop(this);
 }
-static mut JAVA_UTIL_OPTIONAL_LONG: jclass = ::std::ptr::null_mut();
-static mut JAVA_UTIL_OPTIONAL_LONG_OF: jmethodID = ::std::ptr::null_mut();
-static mut JAVA_UTIL_OPTIONAL_LONG_EMPTY: jmethodID = ::std::ptr::null_mut();
+static mut JAVA_LANG_DOUBLE: jclass = ::std::ptr::null_mut();
+static mut JAVA_LANG_DOUBLE_DOUBLE_VALUE_METHOD: jmethodID = ::std::ptr::null_mut();
+static mut JAVA_LANG_FLOAT: jclass = ::std::ptr::null_mut();
+static mut JAVA_LANG_FLOAT_FLOAT_VALUE: jmethodID = ::std::ptr::null_mut();
+static mut JAVA_LANG_STRING: jclass = ::std::ptr::null_mut();
+static mut FOREIGN_CLASS_TRANSACTIONAPI: jclass = ::std::ptr::null_mut();
+static mut FOREIGN_CLASS_TRANSACTIONAPI_MNATIVEOBJ_FIELD: jfieldID = ::std::ptr::null_mut();
+static mut JAVA_UTIL_OPTIONAL_INT: jclass = ::std::ptr::null_mut();
+static mut JAVA_UTIL_OPTIONAL_INT_OF: jmethodID = ::std::ptr::null_mut();
+static mut JAVA_UTIL_OPTIONAL_INT_EMPTY: jmethodID = ::std::ptr::null_mut();
 static mut JAVA_LANG_INTEGER: jclass = ::std::ptr::null_mut();
 static mut JAVA_LANG_INTEGER_INT_VALUE: jmethodID = ::std::ptr::null_mut();
 static mut JAVA_LANG_BYTE: jclass = ::std::ptr::null_mut();
 static mut JAVA_LANG_BYTE_BYTE_VALUE: jmethodID = ::std::ptr::null_mut();
+static mut FOREIGN_CLASS_CONFIGAPI: jclass = ::std::ptr::null_mut();
+static mut FOREIGN_CLASS_CONFIGAPI_MNATIVEOBJ_FIELD: jfieldID = ::std::ptr::null_mut();
+static mut JAVA_UTIL_OPTIONAL_LONG: jclass = ::std::ptr::null_mut();
+static mut JAVA_UTIL_OPTIONAL_LONG_OF: jmethodID = ::std::ptr::null_mut();
+static mut JAVA_UTIL_OPTIONAL_LONG_EMPTY: jmethodID = ::std::ptr::null_mut();
+static mut JAVA_LANG_SHORT: jclass = ::std::ptr::null_mut();
+static mut JAVA_LANG_SHORT_SHORT_VALUE: jmethodID = ::std::ptr::null_mut();
+static mut FOREIGN_CLASS_SENSORAPI: jclass = ::std::ptr::null_mut();
+static mut FOREIGN_CLASS_SENSORAPI_MNATIVEOBJ_FIELD: jfieldID = ::std::ptr::null_mut();
 static mut FOREIGN_CLASS_IDENTITYAPI: jclass = ::std::ptr::null_mut();
 static mut FOREIGN_CLASS_IDENTITYAPI_MNATIVEOBJ_FIELD: jfieldID = ::std::ptr::null_mut();
 static mut JAVA_UTIL_OPTIONAL_DOUBLE: jclass = ::std::ptr::null_mut();
@@ -1540,23 +1556,7 @@
 static mut JAVA_UTIL_OPTIONAL_DOUBLE_EMPTY: jmethodID = ::std::ptr::null_mut();
 static mut JAVA_LANG_LONG: jclass = ::std::ptr::null_mut();
 static mut JAVA_LANG_LONG_LONG_VALUE: jmethodID = ::std::ptr::null_mut();
-static mut JAVA_LANG_FLOAT: jclass = ::std::ptr::null_mut();
-static mut JAVA_LANG_FLOAT_FLOAT_VALUE: jmethodID = ::std::ptr::null_mut();
 static mut JAVA_LANG_EXCEPTION: jclass = ::std::ptr::null_mut();
-static mut JAVA_UTIL_OPTIONAL_INT: jclass = ::std::ptr::null_mut();
-static mut JAVA_UTIL_OPTIONAL_INT_OF: jmethodID = ::std::ptr::null_mut();
-static mut JAVA_UTIL_OPTIONAL_INT_EMPTY: jmethodID = ::std::ptr::null_mut();
-static mut JAVA_LANG_SHORT: jclass = ::std::ptr::null_mut();
-static mut JAVA_LANG_SHORT_SHORT_VALUE: jmethodID = ::std::ptr::null_mut();
-static mut FOREIGN_CLASS_CONFIGAPI: jclass = ::std::ptr::null_mut();
-static mut FOREIGN_CLASS_CONFIGAPI_MNATIVEOBJ_FIELD: jfieldID = ::std::ptr::null_mut();
-static mut JAVA_LANG_DOUBLE: jclass = ::std::ptr::null_mut();
-static mut JAVA_LANG_DOUBLE_DOUBLE_VALUE_METHOD: jmethodID = ::std::ptr::null_mut();
-static mut FOREIGN_CLASS_TRANSACTIONAPI: jclass = ::std::ptr::null_mut();
-static mut FOREIGN_CLASS_TRANSACTIONAPI_MNATIVEOBJ_FIELD: jfieldID = ::std::ptr::null_mut();
-static mut FOREIGN_CLASS_SENSORAPI: jclass = ::std::ptr::null_mut();
-static mut FOREIGN_CLASS_SENSORAPI_MNATIVEOBJ_FIELD: jfieldID = ::std::ptr::null_mut();
-static mut JAVA_LANG_STRING: jclass = ::std::ptr::null_mut();
 #[no_mangle]
 pub extern "system" fn JNI_OnLoad(
 java_vm: *mut JavaVM,
```

Listing 6.1: Different orderings for Rust static variables (comparable to global variables in C) in the generated `java_glue.rs` code that subsequently resulted in a non-reproducible binary artifact. Generated JNI code for registering and freeing the global JNI references in the `JNI_OnLoad` and `JNI_OnUnload` also have these ordering differences, but are omitted for brevity here.

Chapter 7

Conclusion and Outlook

We summarize the findings of this master thesis, which entail several focus areas. On the theoretical side we defined a threat model, derived resulting technical measures and analyzed approaches for secret storage on Android. On the practical side we performed the implementation of the embedded PIA (and its integration into the Android Digidow manager) and evaluated, as well as subsequently fixed, the reproducibility of this PIA variant. Concrete plans for future work streams of the Digidow project are presented right after the respective focus area summary. Finally, we conclude by proposing more general and abstract future work.

7.1 Summary of Contributions

We've iterated on previous threat modeling from other project work streams and outlined both requirements and the threat model for the PIA. Based on this, privacy and security related technical measures were derived. These are used to guide the software architecture and the choice of algorithms/technologies employed by our reference PIA implementation.

A brief summary of all the analyzed approaches for secret storage on Android is as follows:

- White-box cryptography (WBC) does not provide sufficient security guarantees and can thus be discarded right away.
- The usage of the Keystore system or the Identity Credentials API is currently not feasible for technical reasons. Specifically, the pairing friendly curves and group signatures required for ZKPs are not supported.
- Direct SE access does fulfill the technical requirements, but is not realistic on a large scale for the Digidow research project.
- The fallback solution, namely storage of all credential data inside app-private data, does have the highest flexibility (arbitrary algorithms possible) but lacks any kind of security against attackers with system level privileges.

Overall, no single API or implementation strategy provides an ideal solution for the Android PIA. We intend to initially use the data and file storage API and implement/use a software implementation of group signatures and ZKPs. Actual implementation of this ZKP software approach is an actionable item for the immediate future of the Digidow project. Ideally, the APIs dedicated to storage of key material (Keystore system) and/or identity (Identity Credentials API) should evolve to support the technical primitives required for ZKPs. In such a case, we would switch to these dedicated, platform sanctioned solutions. For now, we believe the most promising avenue for hardware supported security

is an agreement between the Digidow research project and a single SE vendor. This would allow us to demonstrate the feasibility of our vision in a proof of concept, even if limited to a single/few device(s).

Due to auxiliary requirements from the larger Digidow project, we have investigated the usage of the memory safe systems programming language Rust on Android. This focus area involves the general theoretical aspect of language interoperability, especially the foreign-function interface (FFI) solution. Based on an overview of the FFI capabilities of the languages involved, namely the Java Native Interface provided by the JVM running on Android and Rust's capability to compile a dynamic library, we have decided on and implemented a solution. The flapigen Rust library provides an abstraction layer above raw JNI bindings, allows declaring a single source of truth for the FFI interface in Rust and provides code generation for the convenience of the developer. Additionally, we have added compatibility between the existing Android Digidow manager and the embedded PIA. This has resulted in a unified core code base that implements the functionality of both PIA variants in the state-of-the art systems programming language Rust.

The importance of trust and the software supply chain is undisputed. Part of this larger topic are reproducible builds and deterministic build systems used to bridge the trust gap between source code and executable artifacts. Based on previous work by the author, we present our thoughts on build context normalization strategies and time of normalization. As a practical contribution, we evaluated the reproducibility of the embedded PIA. Our analysis uncovered a non-determinism in the flapigen Rust library. Through a subsequent fix we managed to make the embedded PIA reproducible and thereby increase the trust in our software supply chain. Our pull request with this fix was merged in the upstream flapigen project and thus benefits the wider open source community.

7.2 Future Work

Based on the current state of literature on digital identity and our contributions in this work, we see numerous future areas of interest. While they are inspired by the Digidow project, we believe that most of these topics are interesting to the scientific community at large.

Portability of digital identities, e.g. for backups or migration from one PIA to another, is a complex subject. If an identity has plain text key material available, the import into a security API (e.g. Android Keystore system) or hardware (e.g. secure element) is trivial. However, the inverse, the export of key material for an identity from an existing PIA is inherently at odds with the security principle of key sealing. How to facilitate legitimate portability demands by users, while limiting (or even prohibiting) possible damage from malicious attackers, is an open question for future research.

Another topic is the association between an individual and their PIA(s). The introduction of the PIA term in section 1.1.1 conceptually assumes that each individual has exactly one PIA. Furthermore, the project vision currently entails two distinct PIA implementation variants (remote and embedded; see Figure 5.1). Both are augmented by a Digidow manager application, running on the mobile phone of the individual, that enable user interactions. This leads to an obvious question: Can/Should one individual have multiple PIAs? For a meaningful answer, one needs to consider the motivations behind possible usage of multiple PIAs for a single individual.

One straight forward reason is redundancy and reliability. This approach entails multiple running PIA instances that are kept in sync, together constituting a single logical PIA and representing a single individual. We consider a combination of a remote and an embedded PIA to be especially useful. E.g. consider that limited internet connectivity of the mobile phone (e.g. due to travel abroad) makes the remote PIA instance the only reachable one. Beyond this configuration, the combination of multiple remote PIA instances may also be beneficial. E.g. provided they are hosted at different locations, such an arrangement promises to improve availability (cf. threat TP2), maintaining functionality of the logical PIA even in face of an on-path attacker that can fully choke traffic to one (but not the other) physical PIA instance.

Another obvious reason is the partitioning of identities related to different activities and roles in life (e.g. business, private, specific community). Here, a single device would run multiple PIA instances, each equipped with a subset of identities. E.g. per default a PIA instance with business related identities is only active during work hours. However, while this seems reasonable on the surface, we believe the same functionality can be achieved with a single PIA. For each identity a PIA implementation should at least offer toggleable enablement and can also provide time and/or location-based policy rules that automate such a switch. Unlinkability measures, like M4, M5 and M6, should ensure that the single PIA approach is indistinguishable from the multiple PIA one for involved parties.

Overall, the usage of multiple PIAs per individual is reasonable. Albeit, depending on the motivation, it may not be advantageous over a single PIA instance.

Bibliography

[1] Christopher Allen. 2016. Life With Alacrity - The Path to Self-Sovereign Identity. (April 25, 2016). Retrieved 01/06/2022 from https://www.lifewithalacrity.com/2016/04/the-path-to-self-soverereign-identity.html.

[2] Android Open Source Project. 2020. CTS Test for Secure Element. Android Open Source Project. (August 3, 2020). Retrieved 08/25/2022 from https://source.android.com/docs/compatibility/cts/secure-element.

[3] Android Open Source Project. 2022. Identity Credential. Android Open Source Project. (March 18, 2022). Retrieved 04/14/2022 from https://source.android.com/security/features/identity-credentials.

[4] Android Open Source Project. 2020. Trusty TEE. Android Open Source Project. Retrieved 01/23/2022 from https://source.android.com/security/trusty.

[5] Apple. 2021. Apple announces first states signed up to adopt driver's licenses and state IDs in Apple Wallet. Apple. Retrieved 01/09/2022 from https://www.apple.com/newsroom/2021/09/apple-announces-first-states-to-adopt-drivers-licenses-and-state-ids-in-wallet/.

[6] Apple. 2021. Apple Platform Security - Secure Enclave. Apple. (May 17, 2021). Retrieved 08/28/2022 from https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web.

[7] Donovan Artz and Yolanda Gil. 2007. A survey of trust in computer science and the Semantic Web. *Journal of Web Semantics*, 5, 2, 58–71. Software Engineering and the Semantic Web. DOI: 10.1016/j.websem.2007.03.002.

[8] Aspiration. *Open Technology Fund Community Lab: Reproducible Builds Summit*. Athens, Greece, (December 2015). Aspiration. Retrieved 07/14/2022 from https://reproducible-builds.org/files/AspirationOTFCommunityLabReproducibleBuildsSummitReport.pdf.

[9] Austria - Bundesministerium für Digitalisierung und Wirtschaftsstandort. 2022. Handy-Signatur und kartenbasierte Bürgerkarte. Austria - Bundesministerium für Digitalisierung und Wirtschaftsstandort. Retrieved 01/06/2022 from https://www.oesterreich.gv.at/themen/dokumente_und_recht/handy_signatur_und_kartenbasierte_buergerkarte.html.

[10] Austria - Bundesministerium für Digitalisierung und Wirtschaftsstandort. 2022. ID Austria - Mein Ich-organisiere-das-von-überall-Ausweis. Austria - Bundesministerium für Digitalisierung und Wirtschaftsstandort. Retrieved 01/06/2022 from https://www.oesterreich.gv.at/id-austria.html.

[11] Phat Bhat. 2018. Bringing it all together with Google Pay. Google. (January 8, 2018). Retrieved 09/22/2022 from https://blog.google/products/google-pay/announcing-google-pay/.

[12] Henk Birkholz, Christoph Vigano, and Carsten Bormann. 2019. Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures. RFC 8610. (June 2019). DOI: 10.17487/RFC8610.

[13] Jérémy Bobbio, Paul Gevers, Georg Koppen, Chris Lamb, and Peter Wu. 2020. Reproducible Builds Documentation: Deterministic build systems. reproducible-builds.org. (July 2020). Retrieved 07/14/2022 from https://reproducible-builds.org/docs/deterministic-build-systems/.

[14] Rian Boden. 2016. Semble discontinues SIM-based NFC mobile wallet in New Zealand. NFCW. (July 15, 2016). Retrieved 08/18/2022 from https://www.nfcw.com/2016/07/15/346220/semble-discontinues-sim-based-nfc-mobile-wallet-in-new-zealand/.

[15] Andrey Bogdanov and Takanori Isobe. 2015. White-Box Cryptography Revisited: Space-Hard Ciphers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (*CCS '15*). ACM, Denver, Colorado, USA, pp. 1058–1069. DOI: 10.1145/2810103.2813699.

[16] Dan Boneh, Xavier Boyen, and Hovav Shacham. 2004. Short Group Signatures. In *Advances in Cryptology – CRYPTO 2004*. Matt Franklin, (Ed.) Springer, Berlin Heidelberg, pp. 41–55. DOI: 10.1007/978-3-540-28628-8_3.

[17] Carsten Bormann and Paul E. Hoffman. 2020. Concise Binary Object Representation (CBOR). RFC 8949. (December 2020). DOI: 10.17487/RFC8949.

[18] Wyseur Brecht. 2012. White-box cryptography: hiding keys in software, (April 2012). Retrieved 08/11/2022 from http://whiteboxcrypto.com/files/2012_misc.pdf.

[19] Kamil Breński, Krzysztof Pranczk, and Mateusz Fruba. 2019. How Secure is your Android Keystore Authentication? F-Secure Corporation. (August 21, 2019). Retrieved 04/08/2022 from https://labs.f-secure.com/blog/how-secure-is-your-android-keystore-authentication/.

[20] BSI TR-03165. 2022. TR-03165 Trusted Service Management System. Draft Technical Guideline. Bundesamt für Sicherheit in der Informationstechnik, Bonn, DE, (May 2022).

[21] Bundesdruckerei GmbH. 2020. OPTIMOS - praxistaugliches Ökosystem sicherer Identitäten für mobile Dienste. Bundesdruckerei GmbH. Retrieved 01/08/2022 from https://www.bundesdruckerei.de/de/Innovation/Optimos.

[22] Jan Camenisch, Manu Drijvers, Alec Edgington, Anja Lehmann, and Rainer Urian. 2017. FIDO ECDAA Algorithm. Proposed Standard. (April 11, 2017). https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-ecdaa-algorithm-v1.2-ps-20170411.html.

[23] Jan Camenisch and Anna Lysyanskaya. 2003. A Signature Scheme with Efficient Protocols. In *Security in Communication Networks*. Stelvio Cimato, Giuseppe Persiano, and Clemente Galdi, (Eds.) Springer, Berlin Heidelberg, pp. 268–289. DOI: 10.1007/3-540-36413-7_20.

[24] Jan Camenisch and Anna Lysyanskaya. 2001. An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation. In *Advances in Cryptology — EUROCRYPT 2001*. Birgit Pfitzmann, (Ed.) Springer, Berlin Heidelberg, pp. 93–118. DOI: 10.1007/3-540-44987-6_7.

[25] Gerardo Capiel and Varouj Chitilian. 2018. Say hello to a better way to pay, by Google. Google. (February 20, 2018). Retrieved 09/22/2022 from https://blog.google/products/google-pay/say-hello-to-google-pay/.

[26] David W. Chadwick. 2009. Federated Identity Management. In *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*. Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, (Eds.) Springer, Berlin Heidelberg, pp. 96–120. DOI: 10.1007/978-3-642-03829-7_3.

[27] David Chaum. 1985. Security without Identification: Transaction Systems to Make Big Brother Obsolete. *Commun. ACM*, 28, 10, (October 1985), 1030–1044. DOI: 10.1145/4372.4373.

[28] Liqun Chen and Jiangtao Li. 2013. Flexible and Scalable Digital Signatures in TPM 2.0. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (*CCS '13*). ACM, Berlin, Germany, pp. 37–48. DOI: 10.1145/2508859.2516729.

[29] Geumhwan Cho, Jun Ho Huh, Soolin Kim, Junsung Cho, Heesung Park, Yenah Lee, Konstantin Beznosov, and Hyoungshick Kim. 2020. On the Security and Usability Implications of Providing Multiple Authentication Choices on Smartphones: The More, the Better? *ACM Trans. Priv. Secur.*, 23, 4, Article 22, (August 2020), 32 pages. ISSN: 2471-2566. DOI: 10.1145/3410155.

[30] Bishwajit Choudhary and Juha Risikko. 2005. Mobile device security element, Key Findings from Technical Analysis. Retrieved 09/09/2022 from https://www.academia.edu/1579251/Mobile_Device_Security_Element.

[31] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. 2014. Analysis of Secure Key Storage Solutions on Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (*SPSM '14*). ACM, Scottsdale, Arizona, USA, pp. 11–20. DOI: 10.1145/2666620.2666627.

[32] P.A. Crisman. 1965. CTSS Programmer's Guide. (1965). Retrieved 09/09/2022 from https://people.csail.mit.edu/saltzer/Multics/CTSS-Documents/CTSS_ProgrammersGuide_1966.pdf.

[33] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. 2014. The tangled web of password reuse. In *Network and Distributed System Security Symposium (NDSS Symposium 2014)* number 2014. Volume 14, pp. 23–26. DOI: 10.14722/ndss.2014.23357.

[34] Mina Deng, Kim Wuyts, Riccardo Scandariato, Bart Preneel, and Wouter Joosen. 2011. A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. *Requirements Engineering*, 16, 1, (March 1, 2011), 3–32. DOI: 10.1007/s00766-010-0115-7.

[35] European Commission. 2022. European Blockchain Services Infrastructure (EBSI). European Commission. Retrieved 01/08/2022 from https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/EBSI.

[36] Eurosmart. 2022. Worldwide Market of Secure Elements Confirms its Resiliency in 2021. Eurosmart. (February 4, 2022). Retrieved 03/27/2022 from https://www.eurosmart.com/worldwide-market-of-secure-elements-confirms-its-resiliency-in-2021/.

[37] Barry Ferg, Brad Fitzpatrick, Carl Howells, David Recordon, Dick Hardt, Drummond Reed, Hans Granqvist, Johannes Ernst, Johnny Bufu, Josh Hoyt, Kevin Turner, Marius Scurtescu, Martin Atkins, and Mike Glover. 2007. OpenID Authentication 2.0. Technical report. (December 5, 2007). Retrieved 01/06/2022 from https://openid.net/specs/openid-authentication-2_0.html.

[38] Hal Finney, Lutz Donnerhacke, Jon Callas, Rodney L. Thayer, and David Shaw. 2007. OpenPGP Message Format. RFC 4880. (November 2007). DOI: 10.17487/RFC4880.

[39] Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. 2020. Memory Corruption Attacks within Android TEEs: A Case Study Based on OP-TEE. In *Proceedings of the 15th International Conference on Availability, Reliability and Security* (*ARES '20*), Article 53. ACM, Virtual Event, Ireland, 9 pages. DOI: 10.1145/3407023.3407072.

[40] Dinei Florencio and Cormac Herley. 2007. A Large-Scale Study of Web Password Habits. In *Proceedings of the 16th International Conference on World Wide Web* (*WWW '07*). ACM, Banff, Alberta, Canada, pp. 657–666. DOI: 10.1145/1242572.1242661.

[41] Germany - Bundesministerium für Wirtschaft und Klimaschutz. 2022. Schaufenster Sichere Digitale Identitäten. Germany - Bundesministerium für Wirtschaft und Klimaschutz. Retrieved 01/08/2022 from https://www.digitale-technologien.de/DT/Navigation/DE/ProgrammeProjekte/AktuelleTechnologieprogramme/Sichere_Digitale_Identitaeten/sichere_digitale_ident.html.

[42] GlobalPlatform. 2020. Certificate of Security Evaluation Huawei iTrustee on Kirin 980 Version 3.0. GlobalPlatform. (November 6, 2020). Retrieved 02/20/2022 from https://globalplatform.org/certified-products/huawei-itrustee-v3-0-on-kirin-980/.

[43] GlobalPlatform. 2020. Certificate of Security Evaluation Samsung TEEgris v4.1 on Exynos 990/980 devices. GlobalPlatform. (September 22, 2020). Retrieved 02/20/2022 from https://globalplatform.org/certified-products/samsung-teegris-v4-1/.

[44] GlobalPlatform. 2018. Open Mobile API - Android Binding. Technical report. Version 1.0 for OMAPI v3.3. (October 2018). Retrieved 09/09/2022 from https://globalplatform.org/specs-library/open-mobile-api-omapi-android-binding-v1-0-for-omapi-v3-3/.

[45] GlobalPlatform. 2018. Open Mobile API Specification. Technical report. Version 3.3. (July 2018). Retrieved 09/09/2022 from https://globalplatform.org/specs-library/open-mobile-api-specification-v3-3/.

[46] GlobalPlatform. 2014. Secure Element Access Control. Technical report. Version 1.1. (September 2014). Retrieved 09/09/2022 from https://globalplatform.org/specs-library/secure-element-access-control-v1-1/.

[47] Oded Goldreich and Yair Oren. 1994. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7, 1, (December 1, 1994), 1–32. DOI: 10.1007/BF00195207.

[48] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1989. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing*, 18, 1, 186–208. DOI: 10.1137/0218012.

[49] Google. 2021. Android Cryptography. Google. (October 27, 2021). Retrieved 04/08/2022 from https://developer.android.com/guide/topics/security/cryptography.

[50] Google. 2021. Android keystore system. Google. (October 27, 2021). Retrieved 02/20/2022 from https://developer.android.com/training/articles/keystore.

[51] Google. 2022. Android Ready SE Alliance. Google. Retrieved 03/27/2022 from https://developers.google.com//android/security/android-ready-se.

[52] Google. 2022. Android Studio - Android Platform/API Version Distribution. Retrieved 07/15/2022 from Android Studio Chipmunk 2021.2.1 Patch 1 running on Linux. Google, (May 9, 2022).

[53] Google. 2022. Behavior changes: Apps targeting Android 12. Google. (June 8, 2022). Retrieved 06/19/2022 from https://developer.android.com/about/versions/12/behavior-changes-12.

[54] Google. 2022. Data and file storage overview. Google. (March 9, 2022). Retrieved 03/19/2022 from https://developer.android.com/training/data-storage.

[55] Google. 2020. Get started with the NDK. Google. (September 30, 2020). Retrieved 07/06/2022 from https://developer.android.com/ndk/guides.

[56] Google. 2022. Jetpack support library - Security - Security-Identity-Credential Version 1.0.0. Google. (February 23, 2022). Retrieved 04/14/2022 from https://developer.android.com/jetpack/androidx/releases/security#security-identity-credential_version_100_2.

[57] Google. 2022. Save data in a local database using Room. Google. (February 23, 2022). Retrieved 03/19/2022 from https://developer.android.com/training/data-storage/room.

[58] Google. 2022. Work with data more securely. Google. (February 23, 2022). Retrieved 03/20/2022 from https://developer.android.com/topic/security/data.

[59] Mohit Gupta. 2017. Attribute Based Credentials. University of California - Berkeley. (November 28, 2017). Retrieved 01/08/2022 from https://privacypatterns.org/patterns/Attribute-based-credentials.

[60] Reeyaz Hamirani. 2016. The Landscape of Customer Identity: Facebook Slides Again. Gigya, Inc. (January 27, 2016). Retrieved 01/06/2022 from https://web.archive.org/web/20180105215556/https://www.gigya.com/blog/the-landscape-of-customer-identity-facebook-slides-again/.

[61] Mohamed Tahar Hammi, Badis Hammi, Patrick Bellot, and Ahmed Serhrouchni. 2018. Bubbles of Trust: A decentralized blockchain-based authentication system for IoT. *Computers & Security*, 78, 126–142. DOI: 10.1016/j.cose.2018.06.004.

[62] Marit Hansen. 2008. Linkage control-integrating the essence of privacy protection into identity management. In *Proceedings of eChallenges*, pp. 1585–1592.

[63] Dick Hardt. 2012. The OAuth 2.0 Authorization Framework. RFC 6749. (October 2012). DOI: 10.17487/RFC6749.

[64] Md Arif Hassan and Zarina Shukur. 2019. Review of Digital Wallet Requirements. In *2019 International Conference on Cybersecurity (ICoCSec)*, pp. 43–48. DOI: 10.1109/ICoCSec47621.2019.8970996.

[65] Nader Helmy. 2021. The State of Identity on the Web. MATTR Limited. (March 15, 2021). Retrieved 06/05/2022 from https://medium.com/mattr-global/the-state-of-identity-on-the-web-cffc392bc7ea.

[66] Serena Hillman, Carman Neustaedter, John Bowes, and Alissa Antle. 2012. Soft Trust and MCommerce Shopping Behaviours. In *Proceedings of the 14th International Conference on Human-Computer Interaction with Mobile Devices and Services* (*MobileHCI '12*). ACM, San Francisco, California, USA, pp. 113–122. DOI: 10.1145/2371574.2371593.

[67] Tobias Höller. 2019. Towards establishing the link between a person's real-world interactions and their decentralized, self-managed digital identity in the Digidow architecture. In *IDIMT-2019: Innovation and Transformation in a Digital World*. Trauner Verlag, Kutná Hora, Czech Republic, (September 2019), pp. 327–332. ISBN: 978-3-99062-590-3.

[68] ISO/IEC 18013-5:2021. 2021. Personal identification — ISO-compliant driving licence — Part 5: Mobile driving licence (mDL) application. International Standard. International Organization for Standardization, Geneva, CH, (September 2021).

[69] ISO/IEC 23220-1. 2021. Cards and security devices for personal identification — Building blocks for identity management via mobile devices — Part 1: Generic system architectures of mobile eID systems. Draft International Standard. International Organization for Standardization, Geneva, CH, (November 2021).

[70] Blake Ives, Kenneth R. Walsh, and Helmut Schneider. 2004. The Domino Effect of Password Reuse. *Commun. ACM*, 47, 4, (April 2004), 75–78. DOI: 10.1145/975817.975820.

[71] Andrey Jivsov. 2012. Elliptic Curve Cryptography (ECC) in OpenPGP. RFC 6637. (June 2012). DOI: 10.17487/RFC6637.

[72] Joint Development Foundation Projects, LLC, Decentralized Identity Foundation Series. 2022. Decentralized Identity Foundation. Joint Development Foundation Projects, LLC, Decentralized Identity Foundation Series. Retrieved 06/04/2022 from https://identity.foundation/.

[73] Kaplan, David and Powell, Jeremy and Woller, Tom. 2021. AMD Memory Encryption. Technical report. Version 9. Advanced Micro Devices, Inc., 2485 Augustine Drive, Santa Clara, CA 95054, (October 18, 2021), 12 pages. Retrieved 08/28/2022 from https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v9-Public.pdf.

[74] Paul A. Karger and Roger R. Schell. 1974. Multics Security Evaluation: Vulnerability Analysis. Technical report ESD-TR-74-193, Vol. II. Electronics Systems Division (AFSC), L. G. Hanscom AFB, MA, USA, (June 1974), 156 pages. Retrieved 02/04/2021 from https://csrc.nist.gov/publications/history/karg74.pdf.

[75] Georgios Karopoulos, Jose L. Hernandez-Ramos, Vasileios Kouliaridis, and Georgios Kambourakis. 2021. A Survey on Digital Certificates Approaches for the COVID-19 Pandemic. *IEEE Access*, 9, 138003–138025. DOI: 10.1109/ACCESS.2021.3117781.

[76] Dong Min Kim. 2022. Aus der Google Pay-App wird Google Wallet. Google. (July 27, 2022). Retrieved 09/22/2022 from https://blog.google/intl/de-de/produkte/android-chrome-mehr/google-pay-app-wird-google-wallet/.

[77] Dmitry Kogan, Henri Stern, Ashley Tolbert, David Mazières, and Keith Winstein. 2017. The Case For Secure Delegation. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (*HotNets-XVI*). ACM, Palo Alto, CA, USA, pp. 15–21. DOI: 10.1145/3152434.3152444.

[78] Raenu Kolandaisamy and Kasthuri Subaramaniam. 2020. The Impact of E-Wallets for Current Generation. *Journal of Advanced Research in Dynamical and Control Systems*, 12, (February 2020). DOI: 10.5373/JARDCS/V12SP1/20201126.

[79] Hsu-Hui Lee and Mark Stamp. 2006. P3P Privacy Enhancing Agent. In *Proceedings of the 3rd ACM Workshop on Secure Web Services* (*SWS '06*). ACM, Alexandria, Virginia, USA, pp. 109–110. DOI: 10.1145/1180367.1180389.

[80] Elias Levy. 2003. Poisoning the software supply chain. *IEEE Security & Privacy*, 1, 3, (June 2003), 70–73. DOI: 10.1109/MSECP.2003.1203227.

[81] René Mayrhofer. 2020. Virtuelle Eröffnungsfeier des CD-Labors DIGIDOW - Vorstellung des CDL: Kontext, Motivation, Ziele, Überblick. presentation. (May 26, 2020). https://www.digidow.eu/opening/.

[82] René Mayrhofer, Michael Roland, and Tobias Höller. 2020. Poster: Towards an Architecture for Private Digital Authentication in the Physical World. In *Network and Distributed System Security Symposium (NDSS Symposium 2020), Posters*. San Diego, CA, USA, (February 2020).

[83] René Mayrhofer, Michael Roland, Tobias Höller, and Philipp Hofer. 2020. Digidow Architecture: Personal Identity Agent (PIA). Christian Doppler Laboratory Private Digital Authentication in the Physical World (Digidow). (November 11, 2020). Retrieved 06/19/2022 from https://pad.ins.jku.at/digidow-documentation-pia.

[84] René Mayrhofer, Michael Roland, Tobias Höller, and Martin Schwaighofer. 2021. Towards Threat Modeling for Private Digital Authentication in the Physical World. Technical report. Johannes Kepler University Linz, Institute of Networks and Security, Christian Doppler Laboratory for Private Digital Authentication in the Physical World, (April 2021). https://www.digidow.eu/publications/2021-mayrhofer-tr-digidowthreatmodeling/Mayrhofer_2021_DigidowThreatModeling.pdf.

[85] 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.

[86] Damiano Melotti, Maxime Rossi-Bellom, and Andrea Continella. 2021. Reversing and Fuzzing the Google Titan M Chip. In *Reversing and Offensive-Oriented Trends Symposium* (*ROOTS'21*). ACM, Vienna, Austria, pp. 1–10. DOI: 10.1145/3503921.3503922.

[87] Microsoft Corporation. 1999. Microsoft Passport: Streamlining Commerce and Communication on the Web. Microsoft Corporation. (October 11, 1999). Retrieved 01/06/2022 from https://news.microsoft.com/1999/10/11/microsoft-passport-streamlining-commerce-and-communication-on-the-web/.

[88] Robert Morris and Ken Thompson. 1979. Password Security: A Case History. *Commun. ACM*, 22, 11, (November 1979), 594–597. DOI: 10.1145/359168.359172.

[89] Alexander Mühle, Andreas Grüner, Tatiana Gayvoronskaya, and Christoph Meinel. 2018. A survey on essential components of a self-sovereign identity. *Computer Science Review*, 30, 80–86. DOI: 10.1016/j.cosrev.2018.10.002.

[90] NFC Forum. 2007. Mobile NFC Technical Guide. Technical report. Version 1.0. (April 2007).

[91] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (*LNCS*, volume 12223). Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, (Eds.) Springer, 17th International Conference, DIMVA 2020, Lisbon, Portugal, pp. 23–43. DOI: 10.1007/978-3-030-52683-2_2.

[92] OpenBSD Foundation. 2022. OpenSSH History. OpenBSD Foundation. Retrieved 01/03/2022 from https://www.openssh.com/history.html.

[93] OpenBSD Foundation. 2021. SSH agent restriction. OpenBSD Foundation. (December 16, 2021). Retrieved 01/06/2022 from https://www.openssh.com/agent-restrict.html.

[94] Oracle. 2020. Java Cryptography Architecture (JCA) Reference Guide. Oracle. Retrieved 04/06/2022 from https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html.

[95] Oracle. 2022. Java Development Kit Version 17 Tool Specifications - The javac Command. Oracle. Retrieved 07/09/2022 from https://docs.oracle.com/en/java/javase/17/docs/specs/man/javac.html.

[96] Oracle. 2022. Java Native Interface Specification Contents. Oracle. Retrieved 07/06/2022 from https://docs.oracle.com/en/java/javase/17/docs/specs/jni/index.html.

[97] Organization for the Advancement of Structured Information Standards (OASIS). 2020. OASIS SAML Wiki - SAML V2.0 Standard. Organization for the Advancement of Structured Information Standards (OASIS). (June 26, 2020). Retrieved 01/06/2022 from https://wiki.oasis-open.org/security/FrontPage#SAML_V2.0_Standard.

[98] Nate Otto, Sunny Lee, Brian Sletten, Daniel Burnett, Manu Sporny, and Ken Ebert. 2019. Verifiable Credentials Use Cases, (September 24, 2019). Shane McCarron, Joe Andrieu, Matt Stone, Tzviya Siegman, Gregg Kellogg, and Ted Thibodeau, Jr., (Eds.) https://www.w3.org/TR/vc-use-cases/.

[99] Marta Pastor and Gregory Steenbeek. 2021. European Self-Sovereign Identity Framework (ESSIF) - High-level scope. European Commission. (August 27, 2021). Retrieved 01/08/2022 from https://ec.europa.eu/cefdigital/wiki/pages/viewpage.action?pageId=379913698.

[100] Andreas Pfitzmann and Marit Hansen. 2010. A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management. v0.34. (August 2010). https://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf.

[101] Lam Pham. 2021. Another Story of Backup and Restore In Android (12). (November 8, 2021). Retrieved 06/19/2022 from https://medium.com/mobile-app-development-publication/another-story-of-backup-and-restore-in-android-12-2500731fbad2.

[102] Achim Pietig. 2020. Functional Specification of the OpenPGP application on ISO Smart Card Operating Systems. Technical report. Version 3.4.1. Lippstädter Weg 14, 32756 Detmold, Germany, DE, (March 18, 2020).

[103] David Pogue. 2000. *Mac OS 9: The Missing Manual*. O'Reilly Media, Inc. https://archive.org/details/mac_Mac_OS_9_The_Missing_Manual_2000/page/n305/mode/2up.

[104] David Pointcheval and Olivier Sanders. 2016. Short Randomizable Signatures. In *Topics in Cryptology - CT-RSA 2016*. Kazue Sako, (Ed.) Springer International Publishing, Cham, pp. 111–126. DOI: 10.1007/978-3-319-29485-8_7.

[105] Manuel Pöll. 2022. flapigen - pull request #438: Make build script runs reproducible. (July 14, 2022). Retrieved 09/07/2022 from https://github.com/Dushistov/flapigen-rs/pull/438.

[106] Manuel Pöll and Michael Roland. 2022. Automating the Quantitative Analysis of Reproducibility for Build Artifacts derived from the Android Open Source Project. In *WiSec '22: Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, San Antonio, TX, USA, (May 2022), pp. 6–19. DOI: 10.1145/3507657.3528537.

[107] Qualcomm Technologies. 2019. Guard Your Data with the Qualcomm Snapdragon Mobile Platform - Hardware-Backend Mobile Secure Storage. Qualcomm Technologies. (April 27, 2019). Retrieved 02/20/2022 from https://www.qualcomm.com/media/documents/files/guard-your-data-with-the-qualcomm-snapdragon-mobile-platform.pdf.

[108] Charles Rackoff and Daniel R. Simon. 1992. Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack. In *Advances in Cryptology — CRYPTO '91*. Joan Feigenbaum, (Ed.) Springer, Berlin Heidelberg, pp. 433–444. DOI: 10.1007/3-540-46766-1_35.

[109] Anthony Ramine. 2019. Rust - Struct std::collections::HashMap - Method values. Rust Library team. (February 13, 2019). Retrieved 09/07/2022 from https://doc.rust-lang.org/stable/std/collections/struct.HashMap.html#method.values.

[110] Wolfgang Rankl and Wolfgang Effing. 2008. *Handbuch der Chipkarten*. (5th ed.). Hanser Verlag, München. ISBN: 978-3-446-40402-1.

[111] David Recordon and Brad Fitzpatrick. 2006. OpenID Authentication 1.1. Technical report. (May 2006). Retrieved 01/06/2022 from https://openid.net/specs/openid-authentication-1_1.html.

[112] Scott Ruoti, Jeff Andersen, Daniel Zappala, and Kent E. Seamons. 2015. Why Johnny Still, Still Can't Encrypt: Evaluating the Usability of a Modern PGP Client. arXiv: 1510.08555.

[113] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted Execution Environment: What It is, and What It is Not. In *2015 IEEE Trustcom/BigDataSE/ISPA*. Volume 1, pp. 57–64. DOI: 10.1109/Trustcom.2015.357.

[114] Navanwita Sachdev. 2019. The evolution of ewallets: history, benefits and withdrawals. Espacio Media Incubator. (February 28, 2019). Retrieved 06/04/2022 from https://sociable.co/mobile/evolution-ewallets-history-benefits-withdrawals/.

[115] Yumi Sakemi, Tetsutaro Kobayashi, Tsunekazu Saito, and Riad S. Wahby. 2021. Pairing-Friendly Curves. Internet-Draft draft-irtf-cfrg-pairing-friendly-curves-10. Work in Progress. Internet Engineering Task Force, (July 2021). 54 pages. https://datatracker.ietf.org/doc/draft-irtf-cfrg-pairing-friendly-curves/10/.

[116] Nat Sakimura, John Bradley, Michael B. Jones, Breno de Medeiros, and Chuck Mortimore. 2014. OpenID Connect Core 1.0 incorporating errata set 1. OpenID Foundation. (November 8, 2014). Retrieved 01/06/2022 from https://openid.net/specs/openid-connect-core-1_0.html.

[117] Samsung Electronics. 2020. Samsung TEEGRIS. Samsung Electronics. Retrieved 02/20/2022 from https://developer.samsung.com/teegris/overview.html.

[118] Jim Schaad. 2017. CBOR Object Signing and Encryption (COSE). RFC 8152. (July 2017). DOI: 10.17487/RFC8152.

[119] Rainer Schamberger, Gerald Madlmayr, and Thomas Grechenig. 2013. Components for an interoperable NFC mobile payment ecosystem. In *2013 5th International Workshop on Near Field Communication (NFC)*, pp. 1–5. DOI: 10.1109/NFC.2013.6482440.

[120] Gerald Schoiber. 2021. Privacy Preserving Hash for Biometrics. Technical report. Johannes Kepler University Linz, Institute of Networks and Security, Christian Doppler Laboratory for Private Digital Authentication in the Physical World, (October 2021). https://www.digidow.eu/publications/2021-schoiber-tr-hashforbiometrics/Schoiber_2021_HashForBiometrics.pdf.

[121] Martin Schwaighofer and Manuel Pöll. Personal discussion on reproducibility and build context normalization. (August 2022).

[122] David Shaw. 2009. The Camellia Cipher in OpenPGP. RFC 5581. (June 2009). DOI: 10.17487/RFC5581.

[123] Sovrin Foundation. 2022. Sovrin. Sovrin Foundation. Retrieved 06/04/2022 from https://sovrin.org/.

[124] Manu Sporny, Dave Longley, and David Chadwick. 2021. Verifiable credentials data model 1.1: Expressing verifiable information on the web, (November 9, 2021). Manu Sporny, Grant Noble, Dave Longley, Daniel C. Burnett, Brent Zundel, and Kyle Den Hartog, (Eds.) https://www.w3.org/TR/vc-data-model/.

[125] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems*. (4th ed.). Pearson Prentice Hall Press, Boston, MA, USA. ISBN: 978-0-13-359162-0.

[126] Thales Group. 2022. What is an eSE? Thales Group. Retrieved 03/27/2022 from https://www.thalesgroup.com/en/markets/digital-identity-and-security/mobile/secure-elements/embedded-secure-element.

[127] The Tor Project, Inc. 2022. How to circumvent the Great Firewall and connect to Tor from China? The Tor Project, Inc. Retrieved 03/15/2022 from https://support.torproject.org/censorship/connecting-from-china/.

[128] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM*, 27, 8, (August 1984), 761–763. DOI: 10.1145/358198.358210.

[129] Andrew Tobin and Drummond Reed. 2017. The Inevitable Rise of Self-Sovereign Identity. Technical report. Sovrin Foundation, (March 2017). https://sovrin.org/wp-content/uploads/2018/03/The-Inevitable-Rise-of-Self-Sovereign-Identity.pdf.

[130] Adam Vartanian. 2018. Cryptography Changes in Android P. Google. (March 8, 2018). Retrieved 04/11/2022 from https://android-developers.googleblog.com/2018/03/cryptography-changes-in-android-p.html.

[131] Artemios G. Voyiatzis, Christos A. Fidas, Dimitrios N. Serpanos, and Nikolaos M. Avouris. 2011. An Empirical Study on the Web Password Strength in Greece. In *2011 15th Panhellenic Conference on Informatics*, pp. 212–216. DOI: 10.1109/PCI.2011.6.

[132] Dan Wendlandt and Adrian Perrig. 2008. Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*.

[133] Alma Whitten and J. Doug Tygar. 1999. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*. G. Winfield Treese, (Ed.) USENIX Association, Washington, DC, USA, (August 1999). https://www.usenix.org/conference/8th-usenix-security-symposium/why-johnny-cant-encrypt-usability-evaluation-pgp-50.