JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Bernhard Gründling, BSc

Submitted at
Institute of Networks and Security

Supervisor
Univ.-Prof. Dr. René Mayrhofer

October 2020

App-based (Im)plausible Deniability for Android

Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Acknowledgments

I would like to thank my supervisor, Prof. Dr. René Mayrhofer, for his highly motivating guidance and all the great opportunities he offered for discussions. I would also like to thank his colleagues at Google – Bram Bonné and Eran Messeri – who took the time to answer my questions and provided interesting insights.
Special thanks go to my friend and fellow student, Konstantin Hartl, for being the best study partner I could have asked for.
Finally, I would like to express my gratitude and appreciation for all the support I have received from my friends and family throughout the years.

Abstract

Confidentiality of data stored on mobile devices depends on one critical security boundary in case of physical access, the device's lockscreen. If an adversary is able to satisfy this lockscreen challenge, either through coercion (e.g. border control or customs check) or due to their close relationship to the victim (e.g. intimate partner abuse), private data is no longer protected.

Therefore, a solution is necessary that renders secrets not only inaccessible, but allows to plausibly deny their sole existence. This thesis proposes an app-based system that hides sensitive apps within Android's work profile, with a strong focus on usability. It introduces a lockdown mode that can be triggered inconspicuously from the device's lockscreen by entering a wrong PIN for example. Usability, security and current limitations of this approach are analyzed in detail.

Kurzfassung

Die Vertraulichkeit von Daten auf Mobilgeräten hängt im Falle eines physischen Zugriffs von einer kritischen Sicherheitsgrenze ab, dem Sperrbildschirm des Geräts. Wenn ein Angreifer in der Lage ist, das Gerät zu entsperren, entweder durch Zwang (z.B. Grenz- oder Zollkontrollen) oder aufgrund seiner engen Beziehung zum Opfer (z.B. Missbrauch durch Partner), sind private Daten nicht mehr geschützt.
Daher ist eine Lösung notwendig, die Geheimnisse nicht nur unzugänglich macht, sondern es auch ermöglicht sogar deren Existenz glaubhaft abzustreiten. In dieser Arbeit wird ein App-basiertes System vorgeschlagen, das vertrauliche Apps im Arbeitsprofil von Android verbirgt, wobei ein Schwerpunkt auf Benutzerfreundlichkeit gelegt wurde. Es wird ein Sperrmodus eingeführt, der über den Sperrbildschirm des Geräts unauffällig ausgelöst werden kann, indem beispielsweise eine falsche PIN eingegeben wird. Benutzerfreundlichkeit, Sicherheit und aktuelle Limitationen dieses Ansatzes werden detailliert analysiert.

Contents

List of Figures

1 Introduction

The term *Plausible Deniability* can be traced back to 1948, a year after the CIA was created and gained popularity in the era of the Cold War [1]. The concept itself is arguably older. Looser hierarchical structures were created to protect officials from legal persecution in the event illegal activities by the CIA became public. Information was withheld from higher-ranking officials, so one could plausibly deny knowledge of said activities. The US National Security Council once defined covert operations as *"so planned and executed that any US Government responsibility for them is not evident to unauthorized persons and that if uncovered the US Government can plausibly disclaim any responsibility for them"* [2].

Generally, plausible deniability enables the person or institution in question to claim to not have known or done something and there is no way to prove the contrary [3].

Many implementations of plausible deniability make use of special cryptographic systems, first proposed in 1997 by Canetti et al. [4]. Deniable encryption enables a person to hide the fact that they are using cryptography to keep secret data inaccessible. The typical situation arises when an adversary has access to the victim's ciphertext and forces the person to disclose their encryption key to decrypt the content. In an ideal deniable encryption system, it would be impossible for the adversary to even prove the existence of ciphertext in the first place [5]. There exist some applications that implement a less ideal, but still effective approximation of this idea which do not enable users to completely deny the existence of ciphertext. The most popular way of handling this situation is by handing out a key that decrypts a ciphertext to only harmless or decoy data. With another, really secret key, the actually private data is decrypted. In such systems, it is impossible to prove the existence of other keys that decrypt the ciphertext in different ways. The most commonly known application might be TrueCrypt, which has been widely used before its mysterious disappearance [6], and was endorsed by people like Edward Snowden [7].

The goal of this thesis is to bring plausible deniability to mobile devices in a usable way. The project aims to protect confidential data in scenarios where users are coerced into unlocking their devices or their device's lockscreen challenge can be satisfied by an adversary.

1.1 Motivation

Today's political developments make new solutions to protect individual privacy necessary. Already in 2008, the Electronic Frontier Foundation (EFF) published recommendations on how to securely cross borders with devices holding private data [8]. Besides more obvious advice like strong passwords, they also explicitly mention deniable encryption as a very secure solution [8]. In 2017, the EFF published another article on digital privacy at the border, which shows a five-fold increase in the number of device searches at the U.S. border in a single year, from 4764 searches in 2015 to 23877 in 2016 [9]. Many of these searches are potential violations of privacy, in which travelers find themselves in a no-win dilemma [9]. If a traveler does not comply with the imposed scrutiny, their devices may be seized, they may be subject to further interrogation or detention and their entry into the country might be refused [9].

We already have reliable solutions for deniable encryption for desktop computers, but for mobile devices, the options are limited and unattractive for non-technical users. At the same time, our mobile devices hold plenty of private data. For many people, mobile devices carry precise documentation of their life, including private conversations, photos, medical data, banking information, location and movement histories. Furthermore, many people have special interest in keeping their stored information private, for example journalists, activists and lawyers. Apart from sensitive professions, there might also be particular demand for privacy for people living in difficult relationships, where physical access to a device through another malevolent but close person cannot be avoided. In summary, this project tries to mitigate two particular threats to privacy:

- Border control arbitrariness: Innocent people are asked or forced to unlock their mobile devices when crossing borders.
- Intimate partner abuse: An abusive partner searches their victim's mobile device for certain private data.

The goal is to protect data stored locally on the device, but also apps that are logged into online accounts.

Plausible deniability is a nuanced topic. There are already some interesting technical approaches, especially for desktop environments. However, a classic, deniable encryption application like TrueCrypt is not sufficient, because it would not allow us to encrypt all the data of an Android application in a user-friendly way and a dedicated encryption app will always raise suspicion. Therefore, a more holistic approach is necessary. The idea is to protect apps and data stored locally on the device in a way that does not interfere with everyday usage.

Applying some form of plausible deniability system is especially interesting for a group of professions, but can also be a way for individuals to protect their human right to privacy.

1.2 Proposal

We propose a system that enables an Android device owner to hand their unlocked or unlockable device to an adversary without disclosing sensitive information. The obvious solution would be to use encryption, but with encryption alone, we always have the problem of possible coercion by the adversary. Because of this, the system should enable us to plausibly deny the existence of sensitive information.

The sensitive information can be inside one or multiple Android apps. These secret apps may be in a logged-in state (e.g. an email-client, instant-messaging app, or a browser with session cookies). The device owner chooses which apps should be protected in advance. For day-to-day usage, the apps are easily accessible via the app launcher and can be used like any other app. If the device owner runs into danger of coercion, they can choose an intuitive and inconspicuous method of hiding the secret apps, like deliberately failing the device's lockscreen challenge once, which triggers the system.

The working title of our application will be Due Process, referring to the constitution of the United States of America, where it states that *"No person shall [...] be deprived of life, liberty or property, without due process of law."* [10]. The ambiguity of the word *process* with its meanings in both law and computer science, depicts the fundamental motivation for this thesis.

1.3 Task and Scope

The project includes:

- Analysis of implementation options.
- Development of an Android application – "Due Process" – that uses Android's existing Device Policy Manager (DPC) interfaces to hide apps and data.
  - Paying special attention to usability.

  - Hardening the system by applying restrictive policies to interfere with forensic analysis.

- Evaluation of the system by performing a forensic analysis.
- Evaluation of the system by conducting a user study with two parts:
  - Non-technical and technically adept users are handed an Android device with the activated system, to find out if the existence of secret data can be plausibly denied in front of them.
  - A group of users with varying levels of technical skills is asked to use the system as a victim. It should be evaluated if the system is usable in a day-to-day scenario.
- Documentation of theoretical concepts and results.

1.4 Outline of Contents

In the next chapter, we will introduce theoretical concepts and our analysis of the threat model and implementation options. After that, we will be able to discuss the technical implementation details of DUE PROCESS in chapter 3. For users of the mobile application, chapter 4 might be the most interesting, where the usage of the software is described. The evaluation of the project, including the results of the user study will be covered in chapter 5. We will also take a look at related work concerning implementations of plausible deniability in chapter 6. Finally, in chapter 7, a summary is given and we explore what could be achieved in the future.

2
Theory

In this chapter, we focus on the theoretical concepts surrounding the topic. This includes the desirable features of plausible deniability, the threat model and assumptions, the requirements for our implementation and the considered options.

2.1 Definitions

In this section, we provide important definitions relevant for our problem space.

2.1.1 Steganography

Steganography is the art or science of concealing information [11]. It originates from the Greek word *steganos*, meaning *covered* or *concealed.* Information may be a message, a file, an image or any other type of data. It usually is hidden inside another "container" message.

2.1.2 Cryptography

Cryptography, meaning *secret writing*, enables secure transmission and storage of information. By applying cryptography, an adversary is unable to understand what the content of a transmitted or stored message is [12].

2.1.3 The Principle of Open Design

One of the core principles in information security is called open design: The security of a system should not depend on the secrecy of its design [13]. It has to be assumed that an attacker knows the inner mechanisms of a system. Therefore, the system has to depend on user provided information, like keys or passwords, which can be protected more easily [13]. Open design prohibits the principle of security through obscurity, where the security of a system depends on the secrecy of its implementation.

Steganography can be used to hide the sole existence of information, however it cannot protect its confidentiality once it becomes known where or how the information is hidden. In this sense, pure steganography is an example for security through obscurity.

Secure cryptosystems depend on a key for making encrypted information readable again. Even if an attacker knows everything about the algorithm used for encryption, he would not be able to decipher the message.

2.1.4 Plausible Deniability

Plausible deniability enables the person or institution in question to claim to not have known or done something and there is no way to proof the contrary [3]. In this section, we look at the different nuances of plausible deniability, which make this a hard problem.

Plausibility

Plausibility operates on a continuum and is a vague concept [1]. When we combine this with a technical system, which has definite features (based on mathematics), it gets interesting.

Most related work in this domain uses some kind of encryption scheme to create plausible deniability (see chapter 6). These cryptographic systems are based on mathematics which enables us to mutually and objectively agree about their workings.

Cryptography makes it possible that we can keep a secret. However, keeping the fact that we are keeping a secret a secret, goes beyond that. Steganography essentially hides the fact that one is hiding information, but it does not address facing adversaries [5].

Generally, it is out of one's control what somebody else finds plausible. In [5], Ragnarsson et al. state *"Plausibility is in the eye of the beholder."* – what another party would find plausible is not clear. An independent judge in a functional constitutional state might accept something as plausible while it may look entirely different in a repressive, rogue state [5].

We have to consider some important nuances of plausibility. Let's look at them in regard of deniable encryption and storage mediums which can be accessed directly by an adversary.

Imagine the situation of an interrogation, where our storage devices are searched for compromising data. We have to assume that an adversary is well aware of cryptography and the concept of plausible deniability. Cryptography produces ciphertexts with high entropy [14], therefore an adversary with access to the raw bits of our storage unit might search for random looking data. Therefore, our ability to plausibly deny the existence of a ciphertext, is associated with how likely the suspected ciphertext is actually just random data [5]. In most cases, this will be assessed as unlikely since the free, unused space of a hard drive is usually not filled with random data. If it was, the issue would be solved, because people who use cryptography could effectively blend in with people who do not [5].

Flawed Plausibility

In order to evaluate the plausibility of different approaches, we need to define the boundaries of the plausibility spectrum. First, we take a look at some imperfectly plausible ways of preventing the disclosure of secret data:

Full Disk Encryption Consider a storage medium where full disk encryption was applied. When interrogated and asked for the decryption key, the only way to handle the situation might be to explain that the key has been forgotten, was destroyed or lost. There is no way to prove this statement, so this is the weakest form of deniability [5].

Encrypted Partitions Instead of a single encrypted partition, we could introduce additional decoy partitions containing harmless data, prepared to be revealed. When asked for the key, we disclose a key for one of the decoy partitions. This is only marginally better than before, because we still need to convince the adversary that the rest of the partitions are just random data, which might seem very suspicious [5].

Hidden Volumes We can improve the situation by taking the idea of encrypted partitions one step further. Instead of storing the secret and decoy partition side by side, we could hide the really secret partition inside of the harmless partition. This provides deniability by convincing the adversary that the free space in our harmless partition is just random data. This is how TrueCrypt works [15].

In summary, unreferenced bits and high-entropy data may always raise suspicion [5].

Ideal Plausibility

A definition for ideal plausibility is given in [5], where a completely plausible plaintext is described by the following three points:

1. Plausible plaintext is data which will not raise suspicion. An interrogator believes this data is normally found on a device.
2. Unreferenced, high-entropy data is always suspected ciphertext.
3. A completely plausible plaintext may either contain only plausible plaintext or ciphertext, for which a key must be supplied which will decrypt it to plausible plaintext. It may not contain unreferenced random data.

Thus, for an ideal system, we would need to be able to explain every single bit on a storage unit. Most practical implementations do not fulfill the notion of ideal plausibility, but there exist some theoretical approaches and proof of concept implementations [5][16][17].

Effective Plausibility

Due to the holistic approach and usability requirements of our proposed system, the implementations go beyond an isolated cryptosystem and ideal plausibility cannot be fulfilled. Therefore we will orient our approach by the following statements regarding effective plausiblity [5]:

- The specific location of random data seems to contribute towards plausibility.
- The existence of random data should be explainable or can be linked to some event or process.
- While under suspicion, there should not be a distinct stop condition for an interrogator: The amount of withheld information should not be quantifiable by the adversary.

2.2 The Android Operating System

In this section, relevant aspects of the Android platform security model are explained.

Android is the most popular end-user operating system in the world [18]. Its open ecosystem offers possibilities for new experiments [19]. Moreover, the Android Open Source Project (AOSP) makes the code of the operating system publicly available. Therefore, it makes sense to develop our proposed system for this platform.

Many architectural security features in Android are based on the Linux kernel, which has been used, examined and improved for many years, so it is a trusted foundation for security critical operations [20].

Android's security model is based on meaningful consent. Essentially, the app developer, the user and the platform have to consent before any action can be executed [18].

2.2.1 Authentication

The primary mechanism of authentication is the device's lockscreen. It ensures that only the device owner can interact with the device. The user's PIN/pattern/password is used to derive the keys for encrypted storage [18].

Our threat model assumes that a third party, an adversary, is able to authenticate as the user. Once authenticated, the user is authorized to access any resource or action. This means, there are only binary states of security: the device is either locked or unlocked, but there is nothing in between.

2.2.2 Isolation Layers

Android consists of many layers of isolation. In this section we look at the most relevant ones.

Process isolation creates the most important boundary, where decisions are made and enforced. Figure 2.1 depicts the isolation layers on top of the main processor of a device. We are interested in the layers provided by the Linux kernel, which enforces the separation of users. Android supports multiple user profiles and additionally, so-called work profiles can be created within a user's personal profile. Installed applications are only available to the respective user/profile.

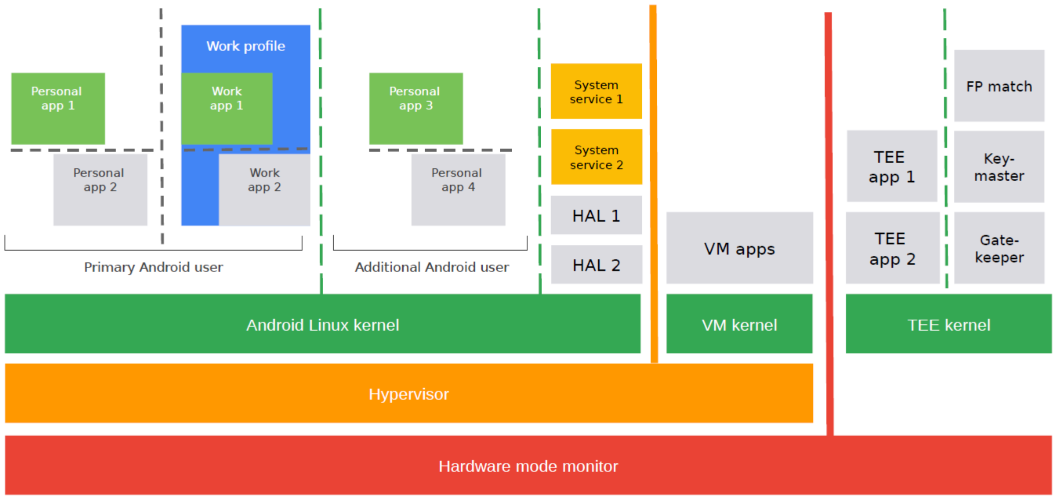


Figure 2.1: Android architecture: layers of isolation [19]

Applications

Applications running on Android are signed and run in an isolated sandbox, which defines the available privileges for the app. Developers sign apps with private keys, the public keys are shipped with the individual apps.

When an application is installed, the package manager creates a unique user ID (`uid`) for it. The app will run with this `uid`, which makes isolation possible. An app without additional permissions may read and write data only in `/data/data/<package-name>`, which is enforced by SELinux [19].

Permissions

The user can grant permissions to an app. There exist several types of permissions, all of them have to be defined in the app's manifest file by the developer [18]:

1. Normal permissions and are granted upon installation by default, e.g. network communication.
2. Runtime permissions require the explicit approval within a dialog box by the user at runtime, e.g. access to storage outside of the sandbox directory.
3. Special access permissions have to be granted by the user in the system settings, and cannot be requested by a simple dialog box, because they are associated with higher risk than runtime permissions. The privilege to display on top of other apps is one example for a special access permission.
4. Privileged permissions are for pre-installed privileged applications only.

5. Signature permissions can only be requested by components signed with the same key as the component which defines the permission. For example, the Android platform itself defines multiple such permissions that are only granted to apps signed with the platform key.

2.2.3 Device Policy Controller

The Device Policy Controller (DPC) offers interfaces for device management for enterprise deployment. A DPC is an application that enforces policies in its ”owned” environment, which may either be the whole device or the work profile. A device owner (DO) is installed in the main user account and manages device-wide policies. A profile owner (PO) is installed on a secondary user, the work profile.

The work profile is based on the same multi-user concept as normal primary users. The difference is that it shares the same common user interface with the primary user. Apps and notifications are identified with a little badge icon to distinguish them from personal apps and notifications [20]. All apps and data in the work profile are separated from the personal profile by the same isolation and containment concepts that protect apps from each other [18]. Additionally, a work profile can also have its own lockscreen, which means the data is encrypted with a separate encryption key [20].

A DPC adds a fourth party to the consent model described in the beginning. Only if the enforced policy allows an action it may be executed [18].

2.2.4 Android Debug Bridge

The Android debug bridge (adb) allows debugging, file transfer, package installation, Unix shell access, etc. from a computer [21]. The adb daemon runs on Android devices when ”USB debugging” is enabled in settings. A connection is initiated with the adb command-line tool running on a computer connected over USB or WiFi, or locally on the device itself [19].

It offers a powerful set of features, which has to be considered as a potential attack surface [19].

2.3 Threat Model and Assumptions

In this section, we discuss Due Process’s threat model and its operational assumptions.

2.3.1 Threat Model

We consider an adversary that is able to access an Android device physically. The device is configured and protected by the Android lockscreen, which authenticates the user. The adversary is able to satisfy the lockscreen challenge. It does not matter whether the device owner unlocks the device for the adversary or discloses the correct PIN/pattern/password. The adversary captures the device to search for secrets and/or to install additional applications/malware. The device may be connected to a computer, e.g. via USB.

According to the principle of open design (section 2.1.3), the adversary may be aware of Due Process's design and will use this knowledge to determine the existence of the application on the system.

This specific threat of physical access to Android devices is also described in [18], where intimate partner abuse and border control situations are explicitly mentioned.

We consider threat actors with various capabilities under the assumption that each of them can satisfy the lockscreen challenge:

[T1] Non-technical interrogators, which will perform a device search without any additional tooling.

[T2] Interrogators who will install additional tools to perform an on-device search.

[T3] Interrogators who will install additional tools on the device and connect it to a computer, using all available communication protocols, including adb to interact with the device.

[T4] Highly sophisticated attackers capable of violating device integrity assumptions listed in section 2.3.2, e.g. zero-day bootloader exploits.

2.3.2 Assumptions

The following assumptions characterize the environment and scope of Due Process's application.

Android Version

The implementation is based on Android 10, the current stable release of Android. When implementing an application that targets widespread adoption, it is usually recommended to consider the trade-off of including older versions and therefore more devices and users versus newer platform features and less devices and users [22]. For most apps, it is good practice to support 90% of active devices [22]. To simplify development, we disregard any limitations of older versions and only consider API level 29, which is Android 10.

Device Integrity

Android's specification are enumerated in the Android Compatibility Definition Document (CDD), which includes the Android security model [23]. If a device does not conform to CDD, it is not Android [18]. We assume that the device conforms since the proposed application's security depends on its environment. Specifically, the system imposes the following requirements on the device's integrity:

Rooting Rooting is the process of modifying the system in a way that allows running processes without isolation and sandboxing [18]. This modification is an example of a CDD violation. Rooting can happen both intentionally by the user or unintentionally via yet unknown vulnerabilities. An adversary could use a process with elevated privileges to access otherwise protected paths and modify sensitive settings, which allows thorough analysis of the stored data and installed applications. Therefore, we assume that the system is not rooted.

Bootloader Unlocking If a device supports unlocking its bootloader, it allows flashing modified firmware and consequently rooting as well [18]. We assume that the device's bootloader is locked and unlocking will cause a factory reset of the device, which wipes the writeable data partitions including all user data.

Device Policy Controller Persistence The Device Policy Controller (DPC) and the policies enforced by it are an important part of Due Process's security. Therefore, we assume that an adversary cannot disable the active DPC without wiping the data of its controlled domain.

Malware We assume that the device is malware-free during setup and normal use of Due Process. The OS, the kernel and the bootloader are trusted. The user does not install apps that could monitor the input and/or output while using secret apps or Due Process itself.

If an adversary installs malware later during a device interrogation, the user will stop trusting their device and restore it to factory defaults.

Encryption

We assume that all data is encrypted with file-based encryption, which is the case with all new devices running Android 10 [20]. As file-based encryption enables the use of different keys for separate storage areas, this ensures the protection of sensitive data even if the main user's key is disclosed.

Furthermore, file system metadata, including the directory structure, file names and sizes, permissions and modification dates are protected by an additional layer of encryption [20]. This is also supported since Android 9 [20].

Adversary

We assume the adversary is rational: it will stop forcing the device owner once it is convinced that all secrets have been revealed.

2.4 Requirements

Considering the threat model and the assumptions under which DUE PROCESS will operate, it should fulfill certain requirements in three main categories: Security, plausibility and usability.

2.4.1 Security

A system is secure, if the cost of successfully attacking it exceeds the potential profit [19]. The user's data and apps (content and meta-data), which they have chosen to protect with DUE PROCESS, have to be kept confidential. This should also be the case if an adversary ([T1]-[T4]) finds the storage location of the data.

Furthermore, the system should be maintainable, both for the developer and user without disproportionate effort, in special consideration of the continually evolving Android platform.

2.4.2 Plausibility

The system should enable effective plausibility as defined in section 2.1.4. Ideally, an adversary would quickly arrive at the conclusion that the user is not hiding any secrets after access to their unlocked Android device.

2.4.3 Usability

The system should be user-friendly. Installation, setup, usage and maintenance should be possible and attractive for non-technical users. Complex user manuals should be avoided.

Moreover, the system should allow for seamless switching between normal and sensitive data operations/application usage. This has two advantages: First, it allows for a convenient every-day usage. Second, if the user is apprehended with the device in sensitive operation, deniability loss can be avoided by quickly switching to the normal/deniable mode.

2.5 Implementation Options

In this section, we will examine and compare various implementation options that could be suitable for our purposes.

2.5.1 Autonomous Encryption App

Cryptographic schemes in the domain of deniable encryption offer the most promising level of plausibility. Their mathematical properties can even make the notion of ideal plausibility possible [5].

Autonomous cryptography software like TrueCrypt works well for systems where data and applications are mostly independent [15]. For most desktop software, it does not matter where the corresponding files are stored. Due to the sandbox isolation on mobile OS, the data is often coupled with an application itself. Therefore, an autonomous encryption app might not work as well for mobile environments.

Additionally, previous work has demonstrated flaws in hidden container systems like TrueCrypt, which stem from the mentioned separation of data and software [24]. The operating system and applications accessing files in a hidden/deniable place can reveal the existence of such a file system or encrypted container, e.g. sensitive files in a recently opened file list [24].

Due to the requirement of sensitive application protection, which cannot be fulfilled by autonomous encryption apps, we do not consider them in our further evaluation.

2.5.2 Virtual Container App

Virtualization allows apps to run inside apps. These container apps are highly popular on the Google Play Store. Plausible deniability is enabled by hiding and running the private apps inside a harmless looking decoy app – the container app. Some of these solutions pretend to be a calculator app which unlocks to the containerized apps when a secret calculation is entered.

However, this approach was not considered due to inherent security issues. The main concern is that for containerized apps, Android's security guarantees are rendered ineffective. The containerized apps all run with the `uid` of the container app and therefore – from the OS and kernel's perspective – potentially inherit the superset of granted permissions of all containerized apps. Previous work has documented multiple security flaws that allows exploitation of this issue [25]. Furthermore, if containerized apps put data in the user's media storage, this data is not protected at all.

2.5.3 Device Policy Controller App

A DPC app has an extensive set of capabilities, which can be used for our purposes. Besides setting restrictive policies, it can also hide apps. Setting an application hidden effectively uninstalls them and makes it unavailable for use, but the package file and application data remain [26]. Although not designed with plausible deniability in mind, this feature could be used to hide the existence of apps.

Policies can be used to restrict analysis of the system by an adversary, e.g. disallowing the installation of additional apps.

Additionally, a DPC can be informed when the device's lockscreen challenge is failed. This allows us to implement a trigger for a lockdown of the system, e.g. when the user deliberately enters the wrong PIN/pattern/password, we can inconspicuously switch the device into ”deniable mode”.

The DPC can run in either profile owner (PO) mode or device owner (DO) mode.

Profile Owner Mode

When a DPC initiates PO mode, the provisioning process of creating a work profile is started. From the user's perspective, only an app has to be installed like any other conventional application. The sensitive apps are then installed inside the work profile, where we can hide them at any time. In the standard launcher running on Google Pixel, the dedicated work section disappears if all apps from the work profile are set to hidden. A user can access apps in the personal and work profile side by side, which enables seamless switching between normal and sensitive apps usage.

As described in section 2.2.3, the work profile's data can be encrypted with a separate key, which is a useful property for our security requirements, preventing loss of confidentiality facing [T1]-[T4]. This is, as long as the user is not forced to disclose this key as well and the key is evicted from memory.

A PO can enforce useful policies like restricting app installation inside the work profile, disallowing app installation from unknown sources globally and disabling cross-profile copy and paste [26]. With these, [T1]-[T2] can be directly countered.

However, an active work profile shows up in various spots of Android's user interface, e.g. in the following system settings: Sound; Security; Accounts; Language and Input; etc. This is inherently counterproductive concerning plausibility, so according to our definition of effective plausibility (2.1.4), the user may have to prepare an explanation for the existence of the profile.

Device Owner Mode

Compared to PO mode, the DO can set additional policies to further restrict analysis by an adversary. Most notably, it is possible to disallow USB debugging features [26] globally, which would limit the attack surface especially for [T3].

Moreover, a DO can create and manage secondary users, which provides similar isolation like the work profile. We could use the secondary user for all sensitive apps and data. To enable deniability, the DPC can log itself out of the secondary profile and disallow switching users, which disables the user interface for switching to the secondary profile [26]. The slight advantage in contrast to PO mode is that the secondary user is not visible anymore in the UIs, which may help in avoiding key coercion for this user.

The installation process is clearly more inconvenient for the user: To enable a DO, a user is required to remove any secondary users and also all accounts managed by the Account Manager API, which includes e.g. the Google Account. Furthermore, the device has to be connected to a computer via adb and the following command has to be issued to set the DO [27]:

```
adb shell dpm set-device-owner <package-name>/.<AdminReceiver>
```

As soon as the DO is active, a message appears in Android's quick settings area: Device is managed by your organization. This message cannot be removed, as a user should be aware of an active DO DPC. The DO variant is less conspicuous than the PO in terms of quantity of UI elements, the quick settings message may make plausible explanation by the user necessary nevertheless.

2.5.4 Custom Firmware

When it comes to security and plausibility, customizing the firmware to suit our requirements allows for versatile possibilities.

Extending the DPC approach, we can remove all traces of the work profile in the user interfaces of the operating system. It would also allow the implementation of secondary (real and decoy) work profiles and we could introduce creative unlocking mechanisms, e.g. invisible key-pads.

However, installation of custom operating systems is an advanced process and highly inconvenient compared to the approaches before, as it requires a complete wipe of the device's data. In addition, the cost of maintenance is much higher, compared to app-based systems. Therefore, we will only describe necessary or useful changes to AOSP which would improve the DPC app-only approach, rather than implementing them in section 7.1.

We take a look at some previous work which is based on customized AOSP code in chapter 6.

2.6 Evaluation of Implementation Options

To determine the different advantages and disadvantages of the described implementation options, we evaluated them based on five questions in three categories, derived from the defined threat model (section 2.3.1) and requirements (section 2.4). The questions were answered based on technical experiments and research of official documentation and related work.

2.6.1 Questions

Security

[QS1] Would sensitive data stay confidential during a local device search ([T1]-[T2])?

[QS2] Would sensitive data stay confidential during an adb search ([T3])?

[QS3] Would sensitive data stay confidential if the underlying system is exploited ([T4])?

[QS4] Is the system maintainable without high effort (developer and user)?

[QS5] Is the system compatible with Android's platform security model?

Plausibility

[QP1] Is the usage of the system safe from inexplicable questions at first glance (e.g. launcher, top-level settings) ([T1])?

[QP2] Is the usage of the system safe from inexplicable questions when device is searched manually, on-device? ([T2])?

[QP3] Is the usage of the system safe from inexplicable questions when device is searched via adb ([T3])?

[QP4] Is the usage of the system safe from inexplicable questions when the underlying system is exploited ([T4])?

[QP5] Does the system fulfill the notion of ideal plausibility?

Usability

[QU1] Can the system be installed like a conventional app?

[QU2] Does the setup process work without a desktop computer?

[QU3] Does the setup process work without an unlocked bootloader?

[QU4] Are the sensitive apps easily accessible for day-to-day use for the user?

[QU5] Is it easy to recover the system from deniable mode?

2.6.2 Results

Category	Container	Profile Owner	Device Owner	Custom Firmware
[QS1] (local)	no (data in media storage)	yes*	yes*	yes
[QS2] (adb)	no (data in media storage)	yes, if profile key is evicted and not disclosed	yes, adb can be disallowed [26]	yes
[QS3] (exploit)	no, direct access	yes, if profile key is evicted and not disclosed	yes, if user key is not disclosed	yes
[QS4] (maintainability)	yes	yes	yes	no
[QS5] (platform)	no, security issues	yes	yes	yes
Security Sum	1	4	4.5	4
[QP1] (first glance)	yes	yes*	yes*	yes
[QP2] (local)	no (data in media storage)	yes*	yes*	yes
[QP3] (adb)	no (data in media storage)	no, inexplicable data can be found	yes, adb can be disallowed [26]	yes
[QP4] (exploit)	no, direct access	no, inexplicable data can be found	no, inexplicable data can be found	theoretically yes
[QP5] (ideal)	no	no	no	theoretically yes
Plausibility Sum	1	2	3	5
[QU1] (conventional install)	yes	yes	yes, but significant effort	no
[QU2] (w/o computer)	yes	yes	no	no
[QU3] (w/o bootloader)	yes	yes	yes	no
[QU4] (easy day-to-day)	yes	yes	no, requires user switch	theoretically yes
[QU5] (deniable recovery)	yes	yes, depending on implementation	yes	yes
Usability Sum	5	4.5	2.5	2
Total Sum	**7**	**10.5**	**10**	**11**

* Detailed evaluation after implementation identified some caveats, see section 5.1. Not considered here.

The table sums up the questions which could be answered with *yes*. If there are any caveats, we counted it as 0.5 *yes*. This analysis is an attempt of comparing the different options and determining the best trade-off between plausibility, usability and security including maintainability.

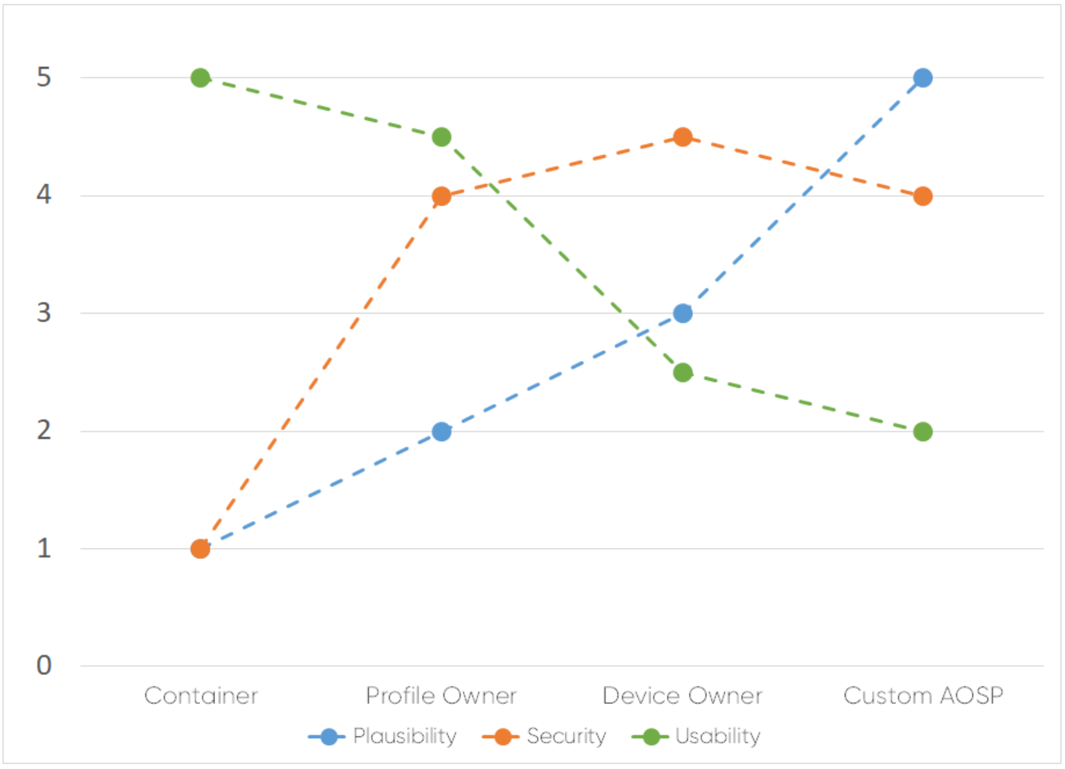


Figure 2.2: Visual representation of results

Figure 2.2 shows the results in a diagram, where the y-axis indicates the number of positively answered questions while the x-axis plots the different implementation options. Connecting the dots helps visualizing that security and plausibility improves with technical sophistication, but usability unfortunately decreases. Custom AOSP received the highest sum in this analysis, however, since usability was a major goal of this project, this approach was not taken. Security and usability almost intersect at the profile owner approach, which justifies our decision to implement this variant.

3 Implementation

This chapter presents the architecture and implementation details of Due Process.

Due Process is a native Android application written in Java with language level 1.8. The app targets Android SDK version 29, which corresponds to Android 10's API level [28]. The published builds are signed with the developers key and are not debuggable.

The application acts as a Device Policy Controller (DPC) in a work profile, provisioned as the profile owner.

The architectural design orients itself towards the recommended standard model for Android applications. It consists of several Activities and Fragments responsible for the user interface (UI), classes for data models and data persistence, and various utility classes, especially for communication with OS APIs. One of the core components is the Device Admin Receiver, which implements the device administration functionality and its definition in the manifest file publishes the app as a DPC to the operating system.
Most of the UI's layout is defined in XML files.

3.1 General Architecture

This section describes the general technical architecture and program flow of the application.
The app can be split in three parts from a user's perspective: First, the setup UI, which is initially presented to the user after installing the application in their personal profile. Second, the introduction UI, which gives important explanations to the user and is responsible for essential initial configuration before the user can enter the third, main part of the application, the management UI. This functions as the control and configuration center of the app.

The `MainActivity` is the UI entry point when the app is launched from the home screen and controls which one of the three parts will be shown to the user. The setup's `SetupFragment` will only be shown if the app is started from a profile where the app is not the owner (usually the personal profile of the primary user). The introduction's `IntroActivity` is launched after the setup is done and the user is transferred into the work profile for the first time. When the initial configuration is finished, the `MainActivity` will always show a `LockscreenFragment` when launched inside the work profile, protecting the management UI, which consists of the navigation `ManagementNavFragment`, the `LockdownFragment`, `AppManagementFragment`, and `ConfigFragment`. To configure the custom lockscreen, a `CodeConfigActivity` can be launched from the `IntroActivity` during initial configuration or the `MainActivity` when the corresponding preference is chosen in the `ConfigFragment`.

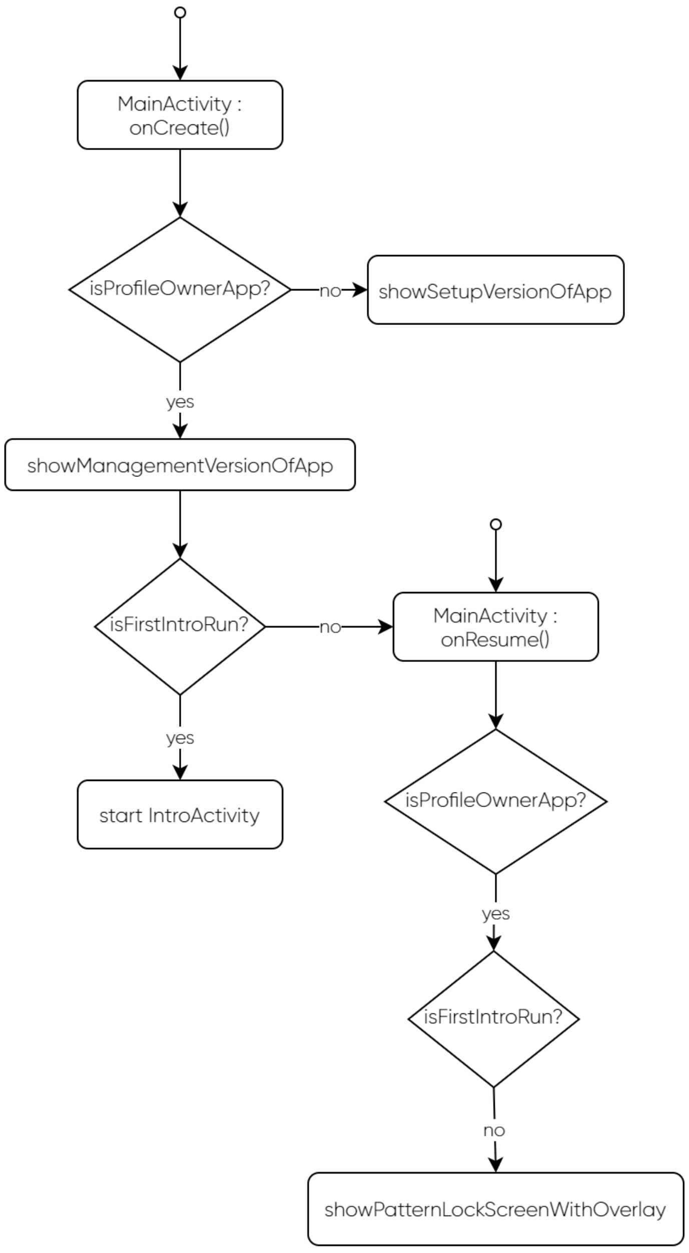


Figure 3.1: `MainActivity` launch flow

All of the mentioned UI classes are inside of an `ui` package, which has three corresponding sub-packages, namely `setup`, `intro` and `management`. Additionally, it holds two classes for the custom lockscreen, `PatternUnlock` and a `PatternUnlockListener`.

```
dueprocess
├─ ui
│  ├─ MainActivity
│  ├─ PatternUnlock
│  ├─ PatternUnlockListener
│  ├─ setup
│  │  ├─ SetupFragment
│  │  └─ PostSetupFragment
│  ├─ intro
│  │  ├─ IntroActivity
│  │  └─ SampleSlideFragment
│  └─ management
│     ├─ LockscreenFragment
│     ├─ ManagementNavFragment
│     ├─ AppManagementFragment
│     ├─ AppListAdapter
│     ├─ LockdownFragment
│     ├─ ConfigFragment
│     └─ CodeConfigActivity
├─ model
│  ├─ AppInfo
│  └─ AppInfoMapper
├─ util
│  ├─ PostProvisioningTask
│  ├─ HidingUtil
│  ├─ NotificationUtil
│  ├─ Util
│  └─ CornerHelper
├─ DueProcessDeviceAdminReceiver
└─ AppSettings
```

Figure 3.2: DUE PROCESS's packages and classes

In the following sections, we will describe the details of each part of the application. In the beginning of each section, the related and relevant classes are listed.

3.2 Setup

```
dueprocess
├─ ui
│  └─ setup
│     ├─ SetupFragment
│     └─ PostSetupFragment
├─ util
│  ├─ PostProvisioningTask
│  ├─ HidingUtil
│  └─ NotificationUtil
└─ DueProcessDeviceAdminReceiver
```

When the app is installed and started in the personal profile, the user can initiate the setup process. A button in the `SetupFragment` will create and start a new intent with action `DevicePolicyManager.ACTION_PROVISION_MANAGED_PROFILE`. The intent also specifies the admin component name, the `DueProcessDeviceAdminReceiver`. With this intent, the provisioning of the work profile is executed. Android will create a new separate profile, assigning an identifier – the `userId` – to the new user. The primary user has ID 00, when a new user is created it starts with `userId` 10 and increments this with every additional user [29]. This is then used for all created directories to separate the profiles, e.g. `/data/user/<userId>/<package>` for all the application data and `/data/media/<userId>/` for the profile's media storage. The sub-directories in these locations are owned by the `userId` of the respective user and application [29]. Applications are only installed once in `/data/app/<package>` and then enabled or disabled for each user. An application's `uid` (mentioned in chapter 2.2.2) consists of the two-digit `userId` (e.g. 00 or 10, 11, 12, ...) followed by a five-digit `appId` (e.g. 10052). When this app is run by the primary user, it runs with `uid` 0010052, when started from the work profile it runs with 1010052.

Before a device admin can enforce policies, it has to declare them in the manifest file. To prevent fingerprinting Due Process based on its metadata about requested policies, it simply declares all of them. The actually relevant ones are: `WATCH-LOGIN` to watch device login attempts by the user, `FORCE-LOCK` to lock the profiles, `WIPE-DATA` to wipe the profile data,

When the system's provisioning operations are finished, it informs the `DeviceAdminReceiver` via a callback method. In order to receive this intent, the intent filter `android.app.action.PROFILE_PROVISIONING_COMPLETE` has to be declared in the manifest. Our `DueProcessDeviceAdminReceiver` will create a new `PostProvisioningTask`, which will execute the following operations: First, it enables the work profile, after that, we add a new policy that disallows unified passwords. This makes sure that the managed profile cannot share the lockscreen challenge with the primary user [30].
Then, some default apps are enabled for the profile, e.g. Google Chrome, Gmail and the Google Play Store, so the user can install additional apps. It also marks

the Play Store app as sensitive as a later recommendation to the user. Apps marked sensitive will be hidden when the user activates the deniable lockdown mode.

The `MainActivity`'s standard label for the launcher icon is Setup. Once the app is enabled in the work profile, this label does not make sense anymore. To solve this, the application's manifest declares an alias for the `MainActivity`, so that the app inside of the work profile can have a different label in the launcher, namely Management. The `HidingUtil` is used to hide the Setup and show the Management launcher icon, by disabling and enabling the respective activity(-alias) with Android's `PackageManager` class.

When the system's provisioning activity returns with a successful result code, the `PostSetupFragment` is presented to the user. In this fragment, a button is created that transfers the user to the app inside the managed profile. This is possible with Android's `CrossProfileApps` class, which allows interaction of instances of the same app across the profile boundary [31].
Finally, a timer is set for a few minutes that prepares a notification sent from the personal profile's version of the app, which reminds the user to perform an uninstall for this instance. The `ACTION_DELETE` intent to trigger an uninstall prompt can only be sent for an app's own instance. The removal of the app makes sure we leave the personal profile in a cleaned up state, mitigating analysis possibilities of an attacker ([T1]-[T3]).

3.3 User Introduction

```
dueprocess
├─ ui
│  ├─ intro
│  │  ├─ IntroActivity
│  │  │
│  │  └─ SampleSlideFragment
│  │
│  └─ management
│     └─ CodeConfigActivity
│
└─ AppSettings
```

After the user has been transferred to the app's instance in the work profile for the first time, the `IntroActivity` is started. Quite some development effort went into this part of the application, as usability was a key requirement for the system. This activity is a typical app introduction and uses a public library to create a slide-based UI [32], where each slide explains one key feature. Besides critical explanations on default slides, the `SampleSlideFragments` are employed to create custom slides with buttons. These require the user to perform important basic configurations and cannot be skipped. There are three of these initial required settings:

1. Disallowing unified passwords as described in section 3.2 is not enough, we

also need to set a separate lockscreen challenge. An intro slide helps the user to configure this and only then allows proceeding.

2. The current implementation of the custom lockscreen (see section 3.6.2) requires the special access permission to draw on top of other apps. The user is sent to the corresponding settings screen, where this has to be enabled before proceeding.

3. Finally, the custom lockscreen itself has to be set up. The user is sent to the `CodeConfigActivity`. The `AppSettings` store the unlock pattern of the user, so only if there is a valid value we let the user continue.

3.4 Management

After initial setup and configuration is done, the user may enter the management version of the app. This is the main user facing component for everyday usage. The actual UI is protected by the custom lockscreen and only if the correct code is entered, the `ManagementNavFragment` is shown. This holds the bottom navigation bar and a view for further fragments, which provides access to the `AppManagementFragment`, `LockdownFragment` and the `ConfigFragment`. The following sections describe each of these components.

3.4.1 Managing Apps

```
dueprocess
├─ ui
│  └─ management
│     ├─ AppManagementFragment
│     └─ AppListAdapter
├─ model
│  ├─ AppInfo
│  └─ AppInfoMapper
├─ util
│  └─ HidingUtil
└─ AppSettings
```

The work profile contains privacy sensitive apps and the `AppManagementFragment` enables administration of the visibility status and behavior of these apps. It holds a `RecyclerView` which is populated with all work profile apps, but by default, filters out Android system packages without a launch intent and Due Process itself. Optionally, this filter may be disabled in the application's settings, so all (system) apps can be managed.

Android's `PackageManager` class allows us to retrieve the list of all installed applications. With the flag `MATCH_UNINSTALLED_PACKAGES` it also returns application information from the list of uninstalled apps, where the `DONT_DELETE_DATA` flag has been set. This is necessary, because we also want to show the applications which were hidden by the device policy manager.

The returned list holds `ApplicationInfo` objects, which are mapped by the `AppInfoMapper` to our own `AppInfo` wrapper object, which holds additional information about an app, such as its visibility and sensitivity status.

For each application, two buttons are available: One – a simple toggle – shows and changes the visibility status, the other one – a switch – shows and changes the behavior for lockdown mode. When the user toggles the first one for a visible app, the device policy manager will immediately hide this app and vice versa. These operations are conveniently wrapped by the `HidingUtil`. The switch marks an application as sensitive. All apps marked this way will be hidden when the user activates lockdown mode. The list of sensitive apps is stored in the preferences with the `AppSettings` class.

3.4.2 In-App Trigger Buttons

```
dueprocess
└─ ui
   └─ management
      └─ LockdownFragment
```

The `LockdownFragment` is a simple interface with two large buttons, a lockdown and a wipe button, which provide additional possibilities to trigger these features in the app. Both of them are attached to a long click listener. A long button click on the lockdown button will immediately activate lockdown. The wipe button will show a confirmation prompt before executing a wipe.

3.4.3 Settings

```
dueprocess
├─ ui
│  └─ management
│     ├─ ConfigFragment
│     │
│     └─ CodeConfigActivity
│
└─ AppSettings
```

All settings concerning the general behavior of Due Process can be configured inside the `ConfigFragment`. This is mostly handled by Android's `Preference` library, so the list of available preferences is actually defined in an XML file [33]. Every read and write operation not handled by the library is accomplished with the `AppSettings` class. The custom lockscreen is configured with an extra activity, the `CodeConfigActivity`.

3.5 Lockdown & Wipe

```
dueprocess
├─ ui
│  └─ management
│     └─ LockdownFragment
├─ util
│  └─ HidingUtil
├─ DueProcessDeviceAdminReceiver
└─ AppSettings
```

The lockdown and wipe feature enable the user to switch the device into a deniable mode, where the existence of apps can effectively be hidden and hardens it against adversarial analysis. To make this process as convenient and inconspicuous as possible, there are multiple ways to trigger it.

3.5.1 Triggers

There are currently two main ways to trigger lockdown and wipe:

1. Convenience in-app buttons as described in section 3.4.2.
   If lockdown is triggered by the in-app buttons, the lockdown actions are executed in a separate thread and the UI is closed with `finishAndRemoveTask()`, which will finish the activity and additionally remove it from the recent apps overview. Handling the lockdown procedures in a separate thread will prevent the profile lockscreen from immediately popping up, since the management app is already closed at the time of locking the profile.
2. Inconspicuous triggers on the device lockscreen.
   This option allows a user to trigger lockdown and/or wipe by deliberately failing the device's lockscreen challenge. This is possible because the `DueProcess-DeviceAdminReceiver` is notified about failed PIN/pattern/password attempts. When the user configured threshold of fails is reached, we either activate lockdown or wipe.

3.5.2 Actions

Lockdown

When lockdown mode is activated, multiple actions are executed by a call to the corresponding method of the `HidingUtil`.

First, all apps marked sensitive are retrieved from the `AppSettings` and then hidden. Optionally, Google Play Services (`com.google.android.gms`) are hidden as well. This makes sure that a potential Google account will be removed from the managed profile, which would show up in the accounts listing in system settings even with all Google launcher apps hidden. Usually it is sufficient to hide all apps with an account managed by the `AccountManager` API to remove the association and settings entry, but with Google apps, the account is administered by the Play Services application. This will log the user out of all Google apps inside the work profile even if not all are hidden. At the time of development however, hiding the Play Services had unexpected side effects which affected the Google Account login state and apps in the personal profile as well, where suddenly all Google apps were rendered unusable (manifesting as complete hangup when launching). This reproducible behavior was reported to Google as a bug.

If sensitive data is stored in the profile's media/data directories, it is especially important that all means of directly accessing these locations are inhibited. The following packages are hidden to realize this:

- Media Storage (`com.android.providers.media`) which is a Content Provider for common media types, such as audio, video, images, etc. [34].
- External Storage (`com.android.externalstorage`) which provides access to the SD card contents (which can either be removable storage or non-removable storage).
- Downloads (`com.android.providers.downloads`) which is another Content Provider for all downloaded items [34].
- System Tracing (`com.android.traceur`) which is used to record system traces to analyze performance-related bugs and may contain sensitive data as well [35].

The result of hiding these packages can be observed in the Files application, depicted by a before and after comparison in figure 3.3. Any application using these to access files, e.g. an upload file chooser dialog in a web browser – which can also be used to browse sensitive directories – will not work anymore.

Furthermore, all apps with runtime permissions to access external storage (`READ-_EXTERNAL_STORAGE`, `WRITE_EXTERNAL_STORAGE`) are retrieved from the `Package-Manager`. Then, these are stored in a list and the permission grant state will be set to denied. We have to keep track of these apps for later recovery of the original grant state, when lockdown is ended. Finally, a policy is set which will auto deny newly requested storage permissions.

Next, some policies are set to harden the system against further analysis. Android's share dialog supports sharing into the managed profile. To remove this UI

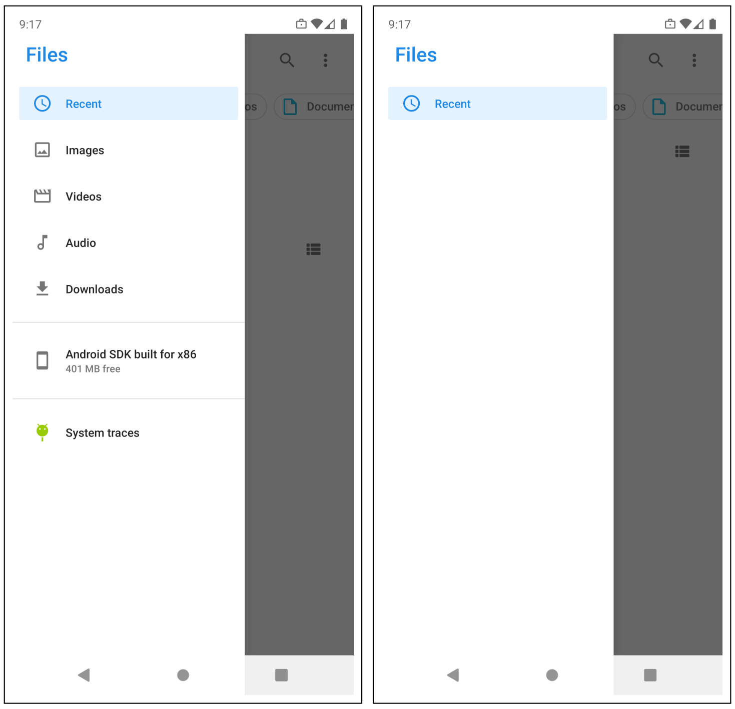


Figure 3.3: Effect of hiding specific content provider packages in the Files app

occurrence, this feature is disabled. The cross profile clipboard function is disallowed as well, to prevent pasting of sensitive data in the personal profile. Moreover, the installation of any app inside the profile is blocked and the installation of apps from unknown sources is blocked globally. This greatly limits the possibilities for installation of additional analysis tools or malware. Debugging features are also disabled for the work profile, which will block starting activities, making service calls, accessing content providers, sending broadcasts, installation and uninstallation of apps, clearing user data, and so on [30]. This only applies to the managed profile unfortunately.
Additionally, all biometric authentication is disabled for both profiles, since these authentication methods may be more vulnerable to coercion.

Optionally but recommended, DUE PROCESS's launcher icon will be hidden by disabling the component of the activity-alias. This disables the intent-filters `android.intent.category.LAUNCHER` and `android.intent.action.MAIN` which advertises it to the system as launcher activity and main entry point to the application. Unfortunately, this will not hide it in the same way as the sensitive apps (which is similar to uninstalling), because that would mean we remove the profile owner and without it, the work profile could not exist anymore.

Finally, by default, the managed profile is locked silently without a notification, by calling the device policy manager's `lockNow()` method. This does not remove the encryption key from the key-ring [26]. Only if the flag `FLAG_EVICT_CREDENTIAL_ENCRYPTION_KEY` is set for the `lockNow()` call, the user's credential will be removed from memory [26]. This however will also show a persistent notification informing the user that the work profile is locked, and it will only go away once the user enters the profile's credentials again. This is a trade-off between security and plausibility, which is analyzed in section 5.1.1. To provide users options to align the respective behavior with their specific threat model, this can be configured in settings.

Wipe

When a wipe is requested by the user, a call to the device policy manager's method `wipeData()` including the flag `WIPE_SILENTLY` is made. This will remove the work profile and all policies set by the profile owner from the system immediately [26]. After a wipe, no traces are left on the system, which obviously provides maximum security and plausible deniability for the sensitive data and apps. A practical use-case would be for example if there are only apps which store their data in the cloud anyway and one wants to keep the fact that one is using these cloud-based services confidential. Users could also prefer a wipe over lockdown depending on the actual threat actor they are facing. If they are usually confronted with [T1], they might use lockdown most of the time, but if suddenly a higher skilled adversary is encountered, they could opt for a wipe. This decision can even be made from the device's lockscreen, if configured.

3.6 Lockdown Recovery & Custom Lockscreen

```
dueprocess
├─ ui
│  ├─ MainActivity
│  ├─ PatternUnlock
│  ├─ PatternUnlockListener
│  ├─ intro
│  │  └─ IntroActivity
│  └─ management
│     ├─ LockscreenFragment
│     ├─ ConfigFragment
│     └─ CodeConfigActivity
├─ util
│  ├─ HidingUtil
│  └─ CornerHelper
├─ DueProcessDeviceAdminReceiver
└─ AppSettings
```

Since lockdown mode sets several restrictive policies, makes apps inaccessible and the management app's launcher icon can be hidden as well, a way of recovering from this state is necessary. The implementation of this recovery has to be well thought out. For this unlocking mechanism two requirements were identified:

1. It needs to be inconspicuous: Either it makes use of some benign feature native to Android or it is invisible.
2. It has to depend on a user provided secret (e.g. a PIN/pattern/password) [13].

3.6.1 Implementation Experiments

Multiple ways of implementing such an unlocking mechanism were explored and tested. Essentially, there are two kinds of events the application could listen to: First, there are implicit system intents/broadcasts. As of API level 26 and its power consumption/background execution restrictions, only a limited number of broadcast receivers can be registered however, such as `ACTION_NEXT_ALARM_CLOCK_CHANGED` and `ACTION_BOOT_COMPLETED` [36].
The other kind of event would be explicit activity intents, e.g. `ACTION_SEND`.

This section describes the findings of our unsuccessful but interesting experiments, skip ahead to section 3.6.2 for the approach that was implemented eventually.

System Alarm Clock

The first attempt was using the system alarm clock, where a user has to set an alarm with the correct secret time and repeating day. We can register a broadcast receiver for `ACTION_NEXT_ALARM_CLOCK_CHANGED`, which will inform the application when the next upcoming alarm has changed. Usually this will be triggered by Android's system clock app, because it uses the `AlarmManager` class (but any app can use these interfaces). When DUE PROCESS receives such an intent, it asks for the time set. If this equals the secret, user configured time, the system unlocks. Regarding the number of possibilities, we would have 60 minutes per day, times seven for the repeating days. This would equate to $60 * 24 * 7 = 10080$ combinations, which is an acceptable number.

However, the mentioned broadcast is never delivered into the work profile. Since the setup instance should be removed from the personal profile for security reasons, there is no suitable way to receive the user's unlock secret.

Share Intent

Direct share intents can be used to exchange data between two applications. This does not work across the profile boundary but the profile owner can explicitly allow certain cross-profile intents [26]. The user could then send the password via a share intent (`ACTION_SEND`) to the management app.

However, cross profile intents only work for activity intents [26], which cannot be received by Due Process if its activity is disabled in lockdown mode. Consequently, even if we kept an instance in the personal profile in the previous alarm clock solution, it would not work. Cross profile intents are therefore not an appropriate tool.

Clipboard

Another attempt was to use the system clipboard. The idea is that the user writes down the secret unlock code somewhere and copies it to the clipboard. Then, the user unlocks the work profile triggering a `ACTION_BOOT_COMPLETED` broadcast, which is delivered into the work profile. When Due Process receives this, we retrieve the clipboard content. However, an application has to be in focus in order to access the clipboard, apparent in the following error message:

```
[...] /system_process E/ClipboardService: Denying clipboard access to
    com.bernhardgruendling.dueprocess, application is not in focus
   neither is a system service for user 39
```

Android allows creating invisible overlay views with the `WindowManager` class. We could draw such an overlay and listen for `onFocusChange()` and access the clipboard when the view gains focus. However, even with the view's `hasFocus()` method returning true, clipboard access is denied. This means, an overlay does not count as application with regards to the clipboard. This makes a clipboard based approach infeasible.

Invisible Keypad

Sticking to the idea of invisible overlay views created upon `ACTION_BOOT_COMPLETED`, we could use these to create an invisible keypad. One requirement is that we have to let all input pass through to the underlying UI, because otherwise this would raise suspicion. An approach would be to create just a little, e.g. 1px view in an overlay with the following parameters: `FLAG_WATCH_OUTSIDE_TOUCH` and `FLAG_NOT_TOUCH-_MODAL`. The first one will let us capture `MotionEvent`s for touches that are outside of our view [37]. The second one lets pointer events outside of the window to be sent to the UI behind it [37].
Conceptually, the coordinates of these touch events could be used for an invisible keypad. However, since Android 4.2, the `InputDispatcher` will only return zeros for the coordinates if the underlying view does not share its `uid` with the listening view [38][39]. This is an undocumented security feature [37][39], but a reasonable change as previous work has shown that keyloggers could be implemented this way [40].

3.6.2 Current Implementation: Invisible Pattern Unlock

Our experiments eventually lead to the current implementation, a pattern unlock mechanism based on the `ACTION_BOOT_COMPLETED` event and an invisible `WindowManager` overlay. The user experience is similar to how Android's native pattern lockscreen works. Three base components use this pattern unlock system: `DueProcessDeviceAdminReceiver` for lockdown recovery, `MainActivity` when showing the `LockscreenFragment` and the `CodeConfigActivity` when the unlocking secret is configured by the user.

Lockdown Recovery Procedure

This section describes the process of lockdown recovery if the management's app icon is hidden. The user initiates it by unlocking the work profile and then drawing a pattern with an invisible overlay.

The `DueProcessDeviceAdminReceiver` registers an intent-filter for `ACTION_BOOT_COMPLETED`, which makes the system independent from the disabled activity. When the intent is received, the user configured unlock credential is retrieved from the `AppSettings` and with this information, a new `PatternUnlock` object is created. This class handles and holds the invisible overlay and initializes the system with the stored secret. The `PatternUnlockListener`, which is an `OnTouchListener` will be attached to the created view. This listener implements the actual logic for the pattern drawing. Furthermore, it holds a callback listener which informs the calling component (one of the three mentioned above, in this case the admin receiver) about the user's input progress and result.

The device's screen and its four corners make up the drawing board. The user starts in one corner and moves the little overlay across the screen, from corner to corner. The unique combination and sequence of corners results in the user's unlocking secret. Upon initialization, a little invisible square is created and positioned in the starting corner, waiting to be moved by the user.

The touch listener discerns three events: `ACTION_DOWN` when the touch gesture starts, `ACTION_MOVE` for every change during the gesture (e.g. moving the view) and `ACTION_UP` when gesture is finished.

`ACTION_DOWN` The initial position is saved and the callback listener is invoked, informing the base component that the input has started.

`ACTION_MOVE` When the user starts moving the view, the current corner index is resolved by the `CornerHelper` class. This means, the screen coordinates are mapped to an index, ranging from 0-3, for each of the four corners. Then, every newly visited corner's index is saved in a list. Additionally, the base component is informed about the code input progress.

ACTION_UP Upon input termination, the original position of the overlay is restored. Additionally, the list of visited corners is sent to the base component, before clearing it for new attempts.

After DueProcessDeviceAdminReceiver has initialized the pattern unlock system, the user currently has 30 seconds to finish the pattern input before the overlay is removed. The class implements the callback listener's method to be informed when the user has finished drawing the pattern. Here, a kind of tarpit comes into play: If code input is failed for three or more consecutive times, the system stops to listen for new attempts for 10 seconds times the failed attempts. This means, after three fails, for 30 seconds any further attempt will be ignored. After four fails 40 seconds, five fails, 50 seconds - and so on. The next possible unlock attempt time is persisted by AppSettings, so this cannot be circumvented by simply restarting the process. Furthermore, an adversarial change of Android's system time to bypass the tarpit is currently handled by the admin receiver listening for the system broadcast ACTION_TIME_CHANGED.
If the correct pattern is drawn, the attempt counter and time is reset, the overlay is cleared and the recovery actions are executed.

Custom Lockscreen

If the management app is launched or resumed, it will always show the custom lockscreen, the LockscreenFragment. This is just a generic white surface, where a user configurable decoy message can be displayed. Additionally, it initializes the same PatternUnlock overlay on top of it. To enter the application, the user has to draw the correct pattern. By using the same method here for the app lockscreen as well as for the lockdown recovery, helps to remind and train the user.

Implementing an application lockscreen requires careful considerations. As the MainActivity controls the flow of the app, it ensures that the lockscreen is always shown as soon as the activity is created, paused or resumed. Moreover, interaction with other activities like the CodeConfigActivity has to be taken into account.

When the app is put into the background, MainActivity's onPause() method is called. To protect all sensitive user interfaces from prying eyes in Android's recent app overview, the WindowManager's FLAG_SECURE is set. This prevents the window to be shown on non-secure displays, like the recent app overview [37].
If the user is switching to a trusted activity, like the CodeConfigActivity, onPause() is called as well, but no further steps are taken. Otherwise, the current fragment is replaced by the LockscreenFragment, but no pattern unlock overlay is initiated.

When the app is resumed and `onResume())` is called, the `FLAG_SECURE` is cleared to enable screenshots inside the app. If the user is coming from a trusted activity, like the `CodeConfigActivity`, no further steps are taken – this means, the original fragment remains, e.g. the `ConfigFragment`. Otherwise, the `Lockscreen-Fragment` including the pattern unlock overlay is displayed. To prevent a bypass of the lockscreen with Android's back navigation, the back stack is cleared.

If the user backgrounds the `CodeConfigActivity`, its `onPause()` method is called. Here, the activity is finished and its result code is set to an unsuccessful value. The result code is processed by the `MainActivity` and resets the request code. This makes sure that a later resumption is not coming from a trusted activity and the lockscreen is shown.

Configuration

The third component employing the pattern unlock system is the `CodeConfig-Activity`, which can be started from the `IntroActivity` or the `ConfigFragment`. It allows the user to configure their secret pattern.
The initialization is a little bit different here. Normally, only one overlay view in the starting corner is created, but for configuration, four overlays have to be created, each in one corner. To help the user find the possible starting points for the pattern, the four views are temporarily made visible with a pulsing animation. The activity implements the callback listener's methods as well. When the user starts to draw the pattern, helpful tips are shown based on the input progress. After two consecutive and identically drawn patterns, the new unlock secret is stored. Currently, a pattern with a minimum of four corners is enforced. This results in at least $4 * 3^{n-1} = 108$ unique combinations, where n is the number of points included in the pattern (in this case 4).

3.6.3 Recovery Actions

Ending lockdown may be initiated in two ways: First, if the management app's icon was hidden, it will be carried out after the user performs the recovery procedure as described in section 3.6.2. Second, if the management app is left accessible and the user unlocks the custom lockscreen, the recovery actions are executed as well.

Recovering from lockdown essentially means reverting the actions of lockdown mode. The `HidingUtil` offers a convenience function for this. First, the Google Play Services app is unhidden again, so the Google Account works again. The content provider packages are unhidden as well. Next, all restrictive policies are cleared by the device policy manager and the original grant states of storage permissions are restored. Biometric authentication is enabled again. If the user has configured it before, the marked sensitive and now hidden apps are shown again in the launcher.

4 Usage

A key goal when implementing Due Process was to create the most user-friendly system possible. Due to the delicate aspects and critical requirements of the system, this was not an easy task. This chapter describes the functionality of Due Process out of the user's perspective and goes through every single use case and scenario. Moreover, particular adjustments to the user interface (UI) based on results of early usability experiments are detailed.

4.1 Installation

To make the system easily accessible from an Android device, it can be downloaded and installed like any conventional app, e.g. from a public repository like the Google Play Store. Unfortunately, this makes it possible to link the app's usage to the user, if it is downloaded with the user's personal Google account for example.

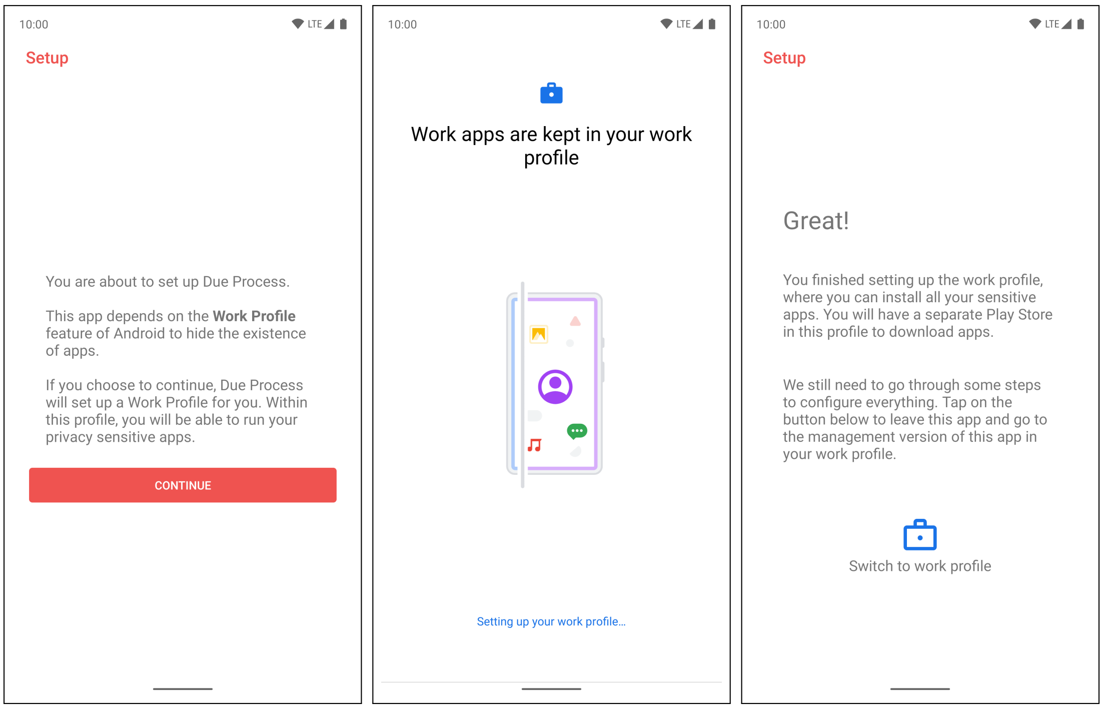
Therefore, to provide anonymity and additional plausibility, the application and its source code should also be hosted publicly on the Internet. On the one hand, this enables users to perform downloads without attribution to their person. On the other hand, it allows users to review the source code and make some changes to it that can further increase plausible deniability. After that, the application package has to be built from the code. The recommended changes are:

- Change package name of application. This requires software that allows automatic refactoring of source code to be efficient.
- Change label of application and labels of activity. This can be achieved by changing the strings inside the app's manifest.
- Change admin receiver's label. This can also be changed in the manifest.
- Change app icon. The drawable defined in `ic_app.xml` needs to be exchanged.

Altering these and obfuscating the code before building a unique APK can significantly improve plausibility when facing [T1]-[T3]. However, this process is not user friendly as it requires a computer and technical expertise. Since all of the listed steps could be automated, requiring the user only to provide some decoy strings and an app icon, the process could be greatly simplified. This could be achieved with a web application, where the user inputs all of the decoy information. The server then creates a unique application package, tailored to the user and offers it to download. All of this could be accomplished from a mobile web browser, making a computer unnecessary.

4.2 Setup

Once installed, the setup can be started. The initial screen shows a short explanation with a Continue button, which starts the provisioning process. If a work profile already exists, the user will be prompted to delete it or cancel provisioning. During the profile setup, Android shows multiple educational screens regarding the basic usage of work profiles. After successful provisioning, it returns to the second and last screen of the setup UI. Here, some more text explains how Due Process incorporates the work profile functionality. Figure 4.1 shows the sequence of screens.


10:00

Setup

You are about to set up Due Process.

This app depends on the **Work Profile** feature of Android to hide the existence of apps.

If you choose to continue, Due Process will set up a Work Profile for you. Within this profile, you will be able to run your privacy sensitive apps.

CONTINUE

10:00

Work apps are kept in your work profile

Setting up your work profile...

10:00

Setup

Great!

You finished setting up the work profile, where you can install all your sensitive apps. You will have a separate Play Store in this profile to download apps.

We still need to go through some steps to configure everything. Tap on the button below to leave this app and go to the management version of this app in your work profile.

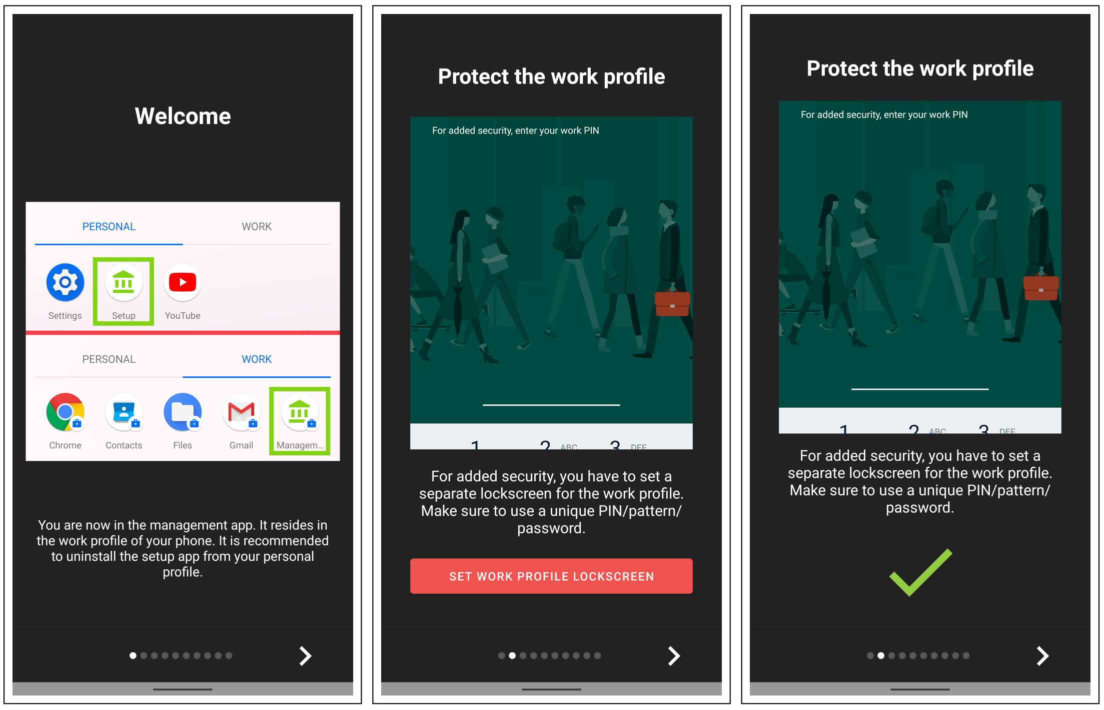
Switch to work profile

Figure 4.1: Setup UI flow

The Switch to work profile button conveniently transfers the user to the app inside the work profile, where all further introduction and configuration will take place.

4.3 Introduction

The introduction UI is based on multiple sliding screens and acts both as guide and configuration sequence. It was designed to educate the user with small pieces of information, moving forward step by step at the user's own pace. It also includes some fundamental configuration slides, which can only be progressed from when the user performed the requested actions. The figure below depicts the first two slides. Here, the visual differences between work and personal apps are introduced. Right after that, the separate lockscreen challenge has to be configured. The figure includes the second slide before and after this required configuration step.



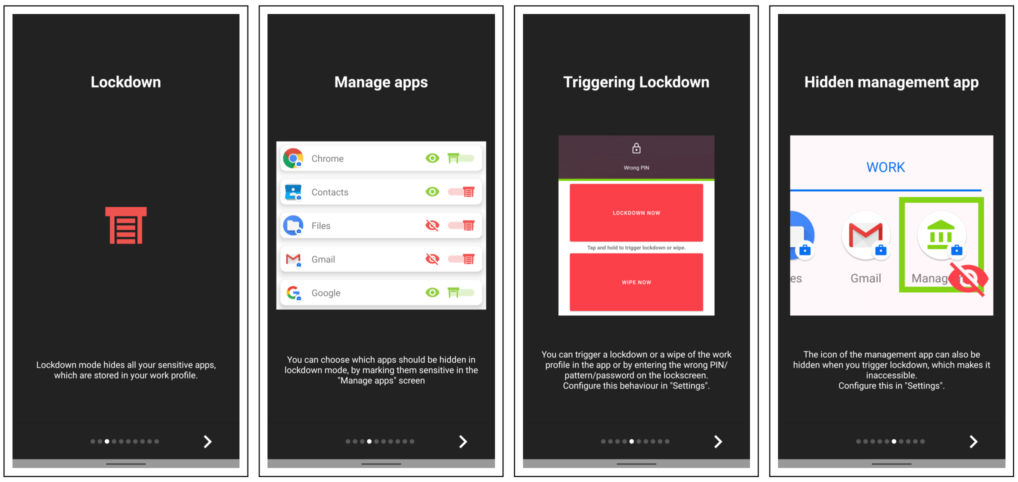
You are now in the management app. It resides in the work profile of your phone. It is recommended to uninstall the setup app from your personal profile.

For added security, you have to set a separate lockscreen for the work profile. Make sure to use a unique PIN/pattern/ password.

For added security, you have to set a separate lockscreen for the work profile. Make sure to use a unique PIN/pattern/ password.

Figure 4.2: The first two intro slides

The next four slides introduce lockdown. The first of them displays its identifying icon. Then, management of apps, triggering lockdown and the optional deactivation of the management app are explained as well.



Lockdown mode hides all your sensitive apps, which are stored in your work profile.

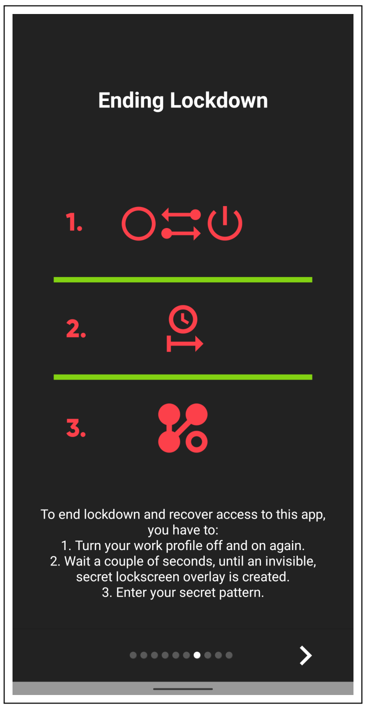
You can choose which apps should be hidden in lockdown mode, by marking them sensitive in the "Manage apps" screen

You can trigger a lockdown or a wipe of the work profile in the app or by entering the wrong PIN/ pattern/password on the lockscreen. Configure this behaviour in "Settings".

The icon of the management app can also be hidden when you trigger lockdown, which makes it inaccessible. Configure this in "Settings".

Figure 4.3: Slides explaining lockdown

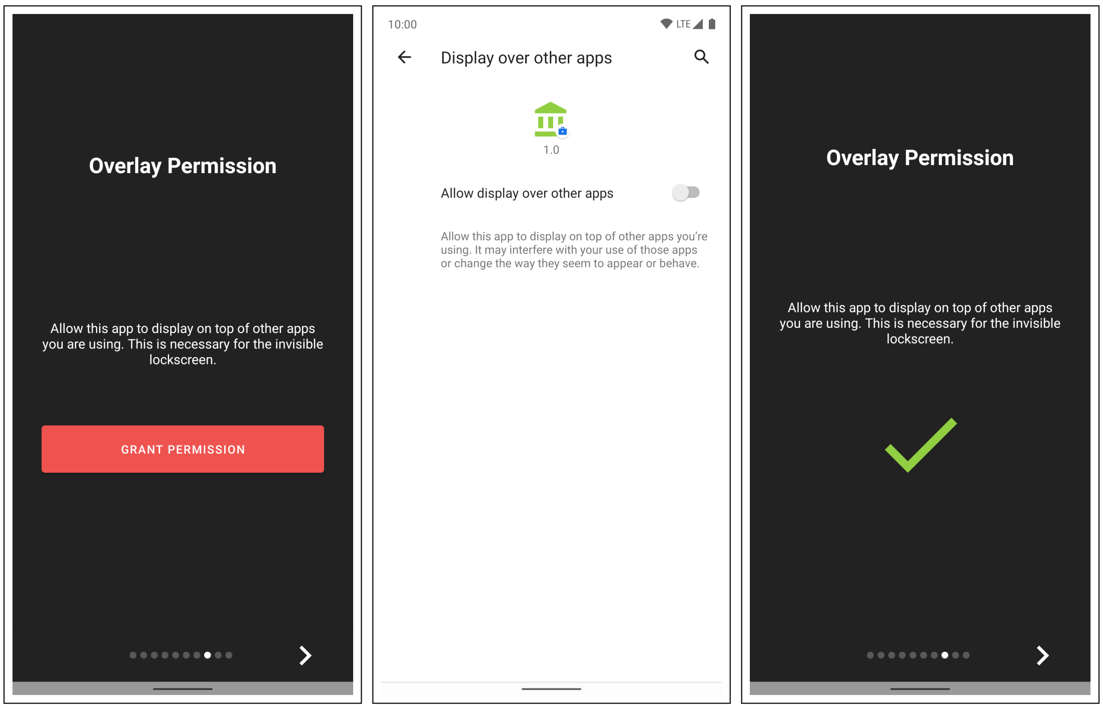
After the user is educated about lockdown, we can demonstrate how to end lockdown mode. Here, the invisible, secret lockscreen is mentioned the first time. This slide contains critical information, if users skip or are unable to understand, it could lead to usability issues.



To end lockdown and recover access to this app, you have to:
1. Turn your work profile off and on again.
2. Wait a couple of seconds, until an invisible, secret lockscreen overlay is created.
3. Enter your secret pattern.

Figure 4.4: Slide explaining recovery from lockdown

In preparation of setting up the secret lockscreen, we can now ask the user to grant the permission to display on top of other apps. As this is a special access permission, we have to transfer the user into the corresponding settings screen, where the switch has to be activated. Only if this is done, the user may proceed.



Overlay Permission

Allow this app to display on top of other apps you are using. This is necessary for the invisible lockscreen.

GRANT PERMISSION

10:00

Display over other apps

1.0

Allow display over other apps

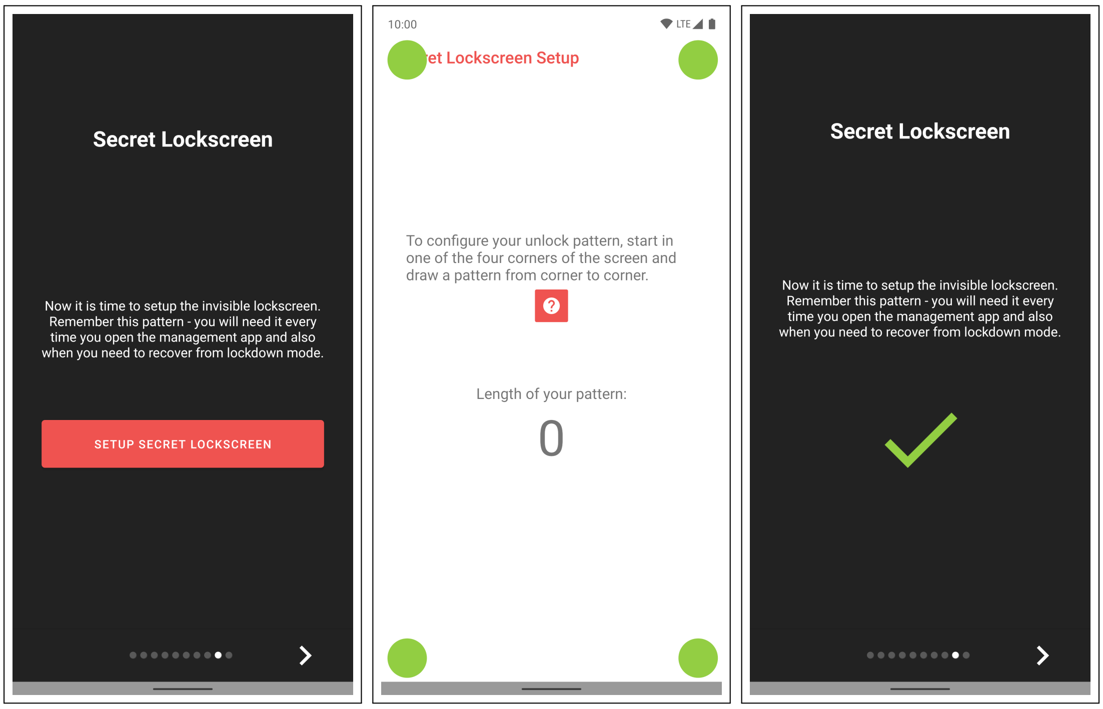
Allow this app to display on top of other apps you're using. It may interfere with your use of those apps or change the way they seem to appear or behave.

Overlay Permission

Allow this app to display on top of other apps you are using. This is necessary for the invisible lockscreen.

Figure 4.5: Overlay permission granting flow

Finally, the secret lockscreen can be configured. For this, the designated configuration activity is started. Upon opening it, the available starting points for the pattern are temporarily indicated with a pulsing animation. The user then has to draw the pattern including one repetition and consisting of at least four points. The pulsing animation can be re-enabled with a button.



Secret Lockscreen

Now it is time to setup the invisible lockscreen. Remember this pattern - you will need it every time you open the management app and also when you need to recover from lockdown mode.

SETUP SECRET LOCKSCREEN

10:00

et Lockscreen Setup

To configure your unlock pattern, start in one of the four corners of the screen and draw a pattern from corner to corner.

Length of your pattern:

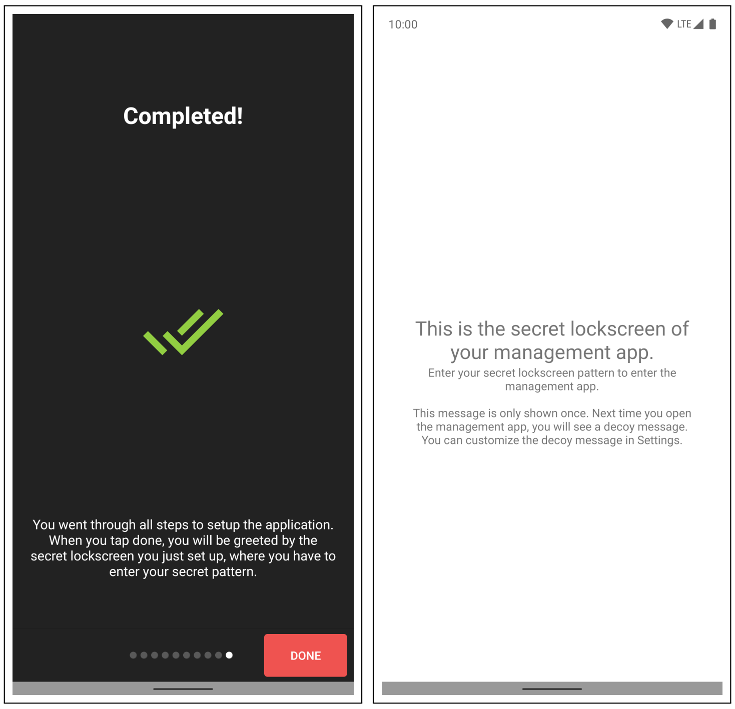
0

Secret Lockscreen

Now it is time to setup the invisible lockscreen. Remember this pattern - you will need it every time you open the management app and also when you need to recover from lockdown mode.

Figure 4.6: Secret lockscreen setup flow

This concludes the introduction process and after the user taps done, the custom lockscreen is shown for the first time. Usually, this lockscreen only displays a configurable decoy message. Early experiments with users have revealed that this can be confusing, therefore it now presents a one time clarification message.


Completed!

You went through all steps to setup the application. When you tap done, you will be greeted by the secret lockscreen you just set up, where you have to enter your secret pattern.

This is the secret lockscreen of your management app.
Enter your secret lockscreen pattern to enter the management app.

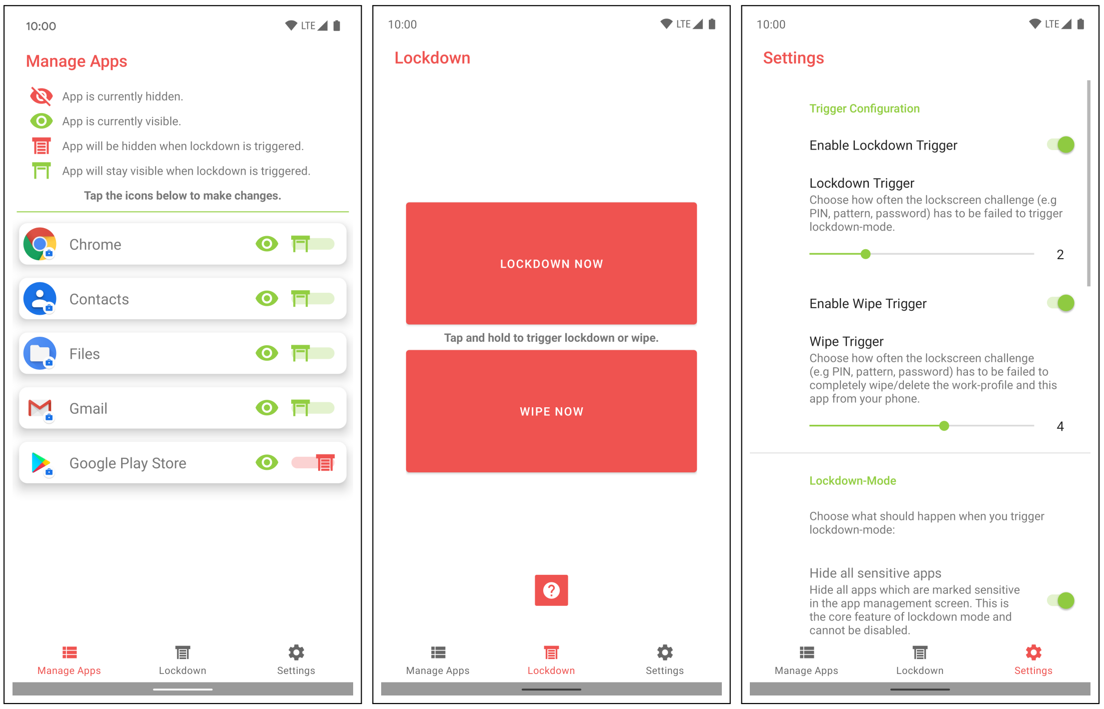
This message is only shown once. Next time you open the management app, you will see a decoy message. You can customize the decoy message in Settings.

Figure 4.7: Final slide and first time secret lockscreen

At some point during the introduction, a notification will be sent to the user from the app's instance in the personal profile. It reminds and prompts the user to uninstall the setup application, as it is no longer needed and deleting it avoids adversarial analysis.

4.4 Management

Whenever the management UI is started or resumed from the background, the custom lockscreen is shown and the secret pattern has to be solved. Inside the actual app, a bottom navigation bar lets the user open the three main UIs: Manage Apps, Lockdown and Settings (see figure 4.8, from left to right). For each app installed in the work profile, there are two controls: First, a toggle button which indicates and changes whether an app is currently hidden or not. Second, a switch, which indicates and changes whether an app is marked sensitive and will be hidden when lockdown is activated. As visible in the screenshot, the Google Play Store has been marked sensitive, which it is by default. The main colors chosen for the app were red and green. Early versions of the application had green color indicating hidden (*enabled the system's hiding mechanism*) and sensitive states (*will enable the system's hiding mechanism*). From the user's perspective however, this was perceived differently and created confusion in experiments: Green was associated with *app is accessible/usable* and *everything is okay*. For the user, lockdown and "hidden" was not the positive/green state, for the developer it was, since it enabled the system. This was adjusted in favor of usability.



App is currently hidden.
App is currently visible.
App will be hidden when lockdown is triggered.
App will stay visible when lockdown is triggered.
Tap the icons below to make changes.

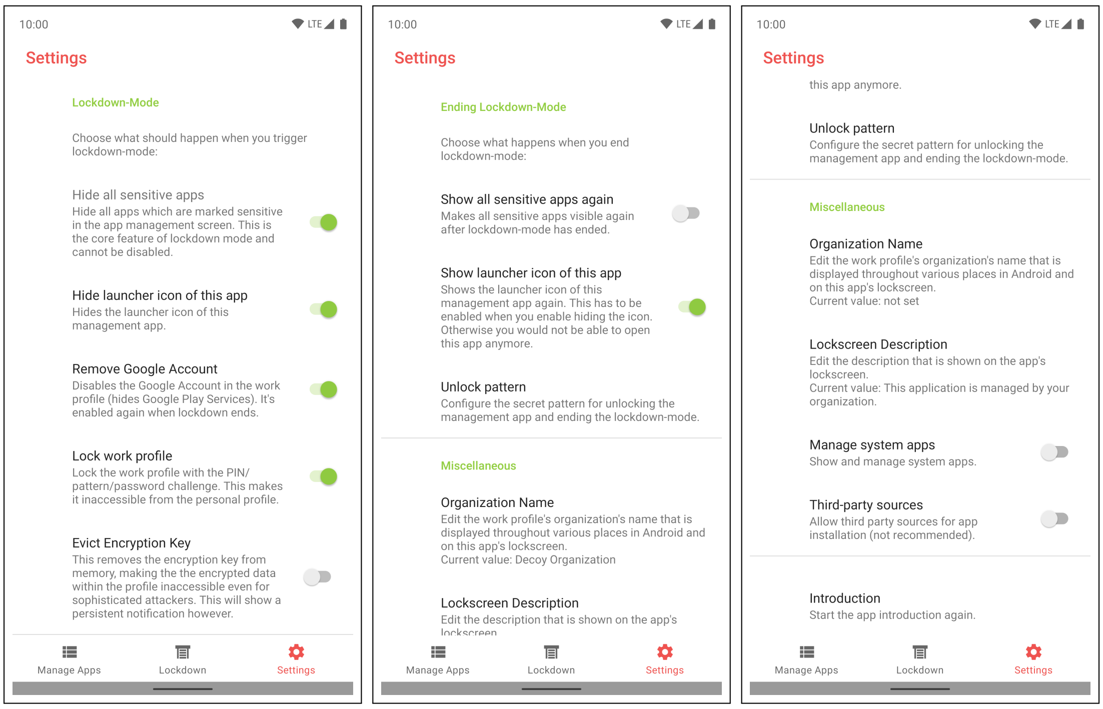
Tap and hold to trigger lockdown or wipe.

Enable Lockdown Trigger
Lockdown Trigger
Choose how often the lockscreen challenge (e.g PIN, pattern, password) has to be failed to trigger lockdown-mode.
Enable Wipe Trigger
Wipe Trigger
Choose how often the lockscreen challenge (e.g PIN, pattern, password) has to be failed to completely wipe/delete the work-profile and this app from your phone.
Choose what should happen when you trigger lockdown-mode:
Hide all sensitive apps
Hide all apps which are marked sensitive in the app management screen. This is the core feature of lockdown mode and cannot be disabled.

Figure 4.8: *The three main screens*

The lockdown and wipe buttons were designed to allow for quick activation of the respective feature. To avoid accidental actuation, tapping and holding is necessary, supporting the user with small animation cues. Furthermore, a button was added to open the Ending Lockdown slide from the introduction sequence as a reminder. This was also a result from early user tests.

The Settings allow the user to configure the lockdown and wipe triggers, the actions performed by lockdown mode, the actions performed when lockdown has ended and some miscellaneous options. Some of these options have logical dependencies on other options: The user can choose the threshold for device lockscreen fails with two seek bars, which logically depend on each other because a wipe cannot be triggered by fewer lockscreen fails than a lockdown. When the user activates Hide launcher icon of this app in lockdown mode actions, Show launcher icon of this app has to be activated for the ending lockscreen actions as well.



10:00

Settings

Lockdown-Mode

Choose what should happen when you trigger lockdown-mode:

Hide all sensitive apps
Hide all apps which are marked sensitive in the app management screen. This is the core feature of lockdown mode and cannot be disabled.

Hide launcher icon of this app
Hides the launcher icon of this management app.

Remove Google Account
Disables the Google Account in the work profile (hides Google Play Services). It's enabled again when lockdown ends.

Lock work profile
Lock the work profile with the PIN/ pattern/password challenge. This makes it inaccessible from the personal profile.

Evict Encryption Key
This removes the encryption key from memory, making the the encrypted data within the profile inaccessible even for sophisticated attackers. This will show a persistent notification however.

10:00

Settings

Ending Lockdown-Mode

Choose what happens when you end lockdown-mode:

Show all sensitive apps again
Makes all sensitive apps visible again after lockdown-mode has ended.

Show launcher icon of this app
Shows the launcher icon of this management app again. This has to be enabled when you enable hiding the icon. Otherwise you would not be able to open this app anymore.

Unlock pattern
Configure the secret pattern for unlocking the management app and ending the lockdown-mode.

Miscellaneous

Organization Name
Edit the work profile's organization's name that is displayed throughout various places in Android and on this app's lockscreen.
Current value: Decoy Organization

Lockscreen Description
Edit the description that is shown on the app's lockscreen

10:00

Settings

this app anymore.

Unlock pattern
Configure the secret pattern for unlocking the management app and ending the lockdown-mode.

Miscellaneous

Organization Name
Edit the work profile's organization's name that is displayed throughout various places in Android and on this app's lockscreen.
Current value: not set

Lockscreen Description
Edit the description that is shown on the app's lockscreen.
Current value: This application is managed by your organization.

Manage system apps
Show and manage system apps.

Third-party sources
Allow third party sources for app installation (not recommended).

Introduction
Start the app introduction again.

Figure 4.9: The rest of available settings

The Evict Encryption key option is disabled by default, refer to section 5.1.1 for the reasoning behind this decision. The configurable Organization Name is shown not only on the app's lockscreen, but also on Android's work profile lockscreen for example. The option Manage system apps lets (technical) users disable filtering of system apps in the Manage apps screen, so apps without a launcher icon can also be hidden.

4.5 Lockdown Recovery

Ending lockdown and recovery of the management app can be tricky. The invisible unlocking mechanism is created when the work profile is enabled. The default app launcher on Pixel devices offers a switch to enable and disable the profile. However, if all work apps are hidden, the launcher section disappears and so does the switch. To disable and enable the work profile in this case, the user has two options: Either with a switch in Android's system settings, Accounts, Work profile settings or with a quick settings toggle.

5 Evaluation

In this chapter, we will examine the software's capabilities in terms of security, plausibility and usability. Some unforeseen issues which came up during implementation are highlighted in the following sections as well.

5.1 Security & Plausibility Analysis

5.1.1 Key eviction

When the system is switched into deniable mode, the work profile can be made inaccessible with a lockscreen. The `lockNow()` method either performs a "soft lock" or a "hard lock". The first one will keep the encryption key in memory, the second one will remove it. Once the key is removed, it can only be retrieved again with the user's knowledge factor [26]. Assuming the user does not disclose their lockscreen secret, we compare the two cases:

When we perform key eviction, we win security (confidentiality), especially facing [T4] and specifically [QS3], because even with direct storage access, the data and file meta data will stay encrypted. At the same time, we lose some plausibility or at least raise suspicion against [T1], [QP1] due to a persistent notification, which cannot be disabled, not even manually by the user.

On the other hand, if the key is not evicted, [T2] would be able to enumerate the file and directory names inside the work profile. [T3] would be able to do the same, [T4] could potentially circumvent the permissions enforced by the kernel and therefore also access the content of files.

The hard lock is more secure but might lead to earlier coercion to disclose the work profile key anyway. The soft lock is less secure, but more inconspicuous. As this difficult decision is depending on the actual threat model of the user, it has been left to the user as a configuration option.

If the user is forced to disclose their secret in either situation and assuming all work profile applications are hidden, the attack vectors do not really differ from those of a soft locked profile. Since the attacker cannot install any apps inside of the profile, the access can only be made from outside, from the personal profile. Here, the same restrictions apply as if the profile is soft locked. Furthermore, key eviction is only really guaranteed, if the device has been rebooted and therefore all memory has been cleared. As most [T4] attacks might involve a reboot, the key will be evicted in these cases anyways (except for cold boot attacks perhaps).

5.1.2 Forensic Analysis

This section presents the results of a forensic analysis of the system, with the capabilities up to [T3]. Theoretical attack vectors of [T4] are described as well.

The analysis assumes that all apps have been hidden in the work profile and all identifiers (package name, app labels, icon, etc.) of Due Process have been changed to inconspicuous elements. The user enabled Google Account removal, management app hiding and work profile locking.

What [T1] can find

When the device is searched locally by a non-technical user, the first inspection point will most likely be the app launcher. The default launcher application in Google Pixel devices has a separate section for work apps, visible in the first two images of figure 5.1. When all work apps are hidden, the section disappears completely, effectively hiding the apps and the work profile at first glance (see third image in figure 5.1). Android's system settings are more revealing. The existence of a work profile can be determined in multiple menus. Moreover, a list of all installed applications is provided, and here the first plausibility issue arises: The primary user/device owner can always see all apps installed on the system for every user. If an app is only installed for the work profile and hidden, the app will still be listed here with a notice saying not installed for this user. This might raise suspicion and confusion, because another settings screen listing apps specifically installed in the work profile will not show this entry.
If an app is installed for both the personal and the work profile, the issue is remedied: The conspicuous entry is overloaded by the app's instance in the personal profile. To maximize plausible deniability, the user could install the app for both profiles, keeping the personal profile's instance clean or filled with decoy data.

Another problem is used storage: The primary user sees the amount of storage used by the managed profile, which becomes an issue especially if there is lots of data stored inside the profile.

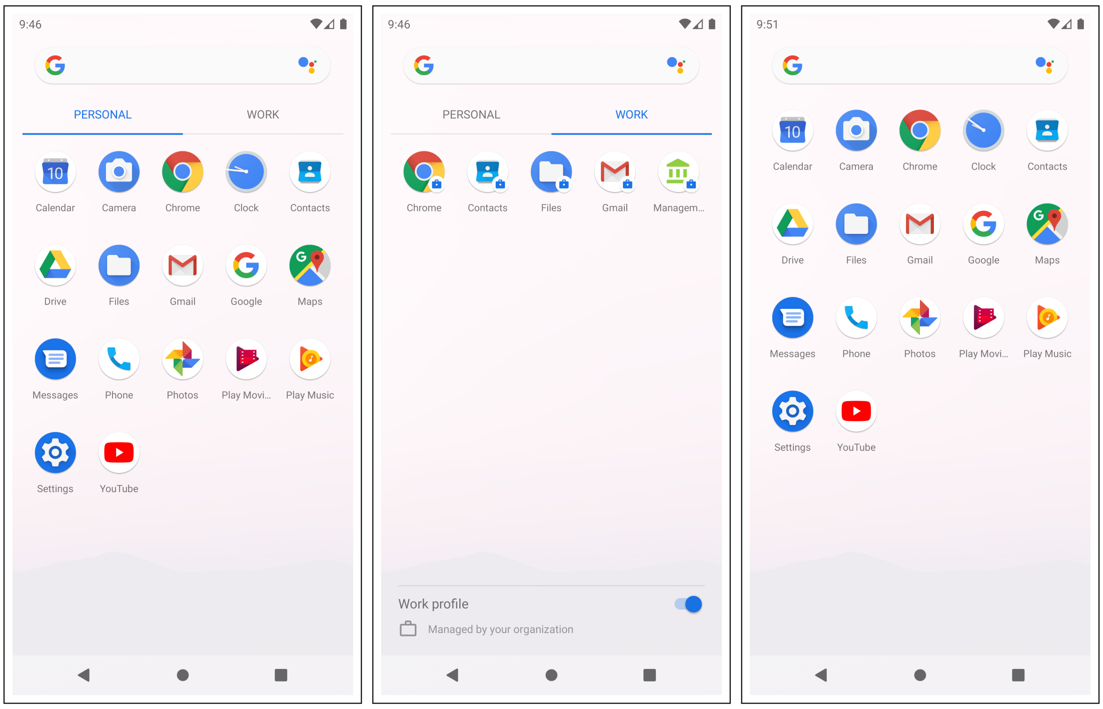


Figure 5.1: Effect of hiding all work apps: Launcher section disappears.

Android will create a notification when an app displays over other apps. The invisible pattern unlock system will trigger such a notification, which – if disabled by the user beforehand – is not an issue. Attempts were made to create an intent during app introduction that opens this specific settings screen for Android system notifications, but were not successful. It seems that this is not possible anymore in Android 10, as tests with older versions of Android had shown that it worked at some point [41].

If the mechanics of Due Process's invisible pattern unlock system are known to the adversary – which we have to assume based on the principle of open design – it would be possible to determine when it becomes active. An adversary would need to check all four corners after starting the work profile, and if the underlying UI is not responding to touch input, the initial corner for the user's pattern is found.

What [T2] can find

An attacker may also install additional apps inside the personal profile. Since installation of apps from unknown sources is blocked globally in lockdown, the only way to install apps is the Google Play Store. This reduces the risk of malware installed by an adversary. Interesting tools for forensic examination could be a file explorer which allows navigation to the file system's root directory or a terminal emulator to run shell commands. Both kinds of applications can be acquired from the Play Store.

Using a file explorer, an adversary will not be able to cross the profile boundary and therefore no interesting things will be found.
If the work profile key is disclosed or not evicted, a terminal emulator can be used to enumerate `userIds` of users and file and package names inside the work profile. Enumeration is possible because the operating system returns distinguishable messages for existing and non-existent directories.
First, the `userId` of the work profile is enumerated:

```
generic_x86:/ $ cd /data/user/
generic_x86:/data/user $ ls 0
ls: 0: Permission denied
1|generic_x86:/data/user $ ls 10
ls: 10: No such file or directory
1|generic_x86:/data/user $ ls 11
ls: 11: Permission denied
1|generic_x86:/data/user $ cd 11
generic_x86:/data/user/11 $
```

For every user/`userId`, a directory is created in `/data/user/` and as `userId` 0 is the primary user, this directory must exist. Regarding the directory permissions, only the execute bit is set for everyone other than root, so listing the contents is denied. To find valid `userIds`, the `ls <userId>` command can be used, which will return with `Permission denied` for existing directories and `No such file or directory` for non-existent directories. Since only the execute bit is set for existing sub-folders as well, we can only enter them, but not list the content. Inside of these directories, app-private data is stored for every user the app is installed for (`/data/user/0/` is linked to `/data/data/`, `/data/user/11/` contains app data for apps inside the work profile in this case). So, enumerating installed packages for the work profile works the same way:

```
generic_x86:/data/user/11 $ ls
ls: .: Permission denied
1|generic_x86:/data/user/11 $ cd com.nonexistent.app
cd: /data/user/11/com.nonexistent.app: No such file or directory
2|generic_x86:/data/user/11 $ cd com.twitter.android
cd: /data/user/11/com.twitter.android: Permission denied
```

This way, [T2] can determine which apps in the work profile store data, so a more definitive assertion than [T1]'s can be made.

Besides determining installed packages, file names can also be enumerated in `/storage/emulated/<userId>/<file-name>`.

If the profile key is not disclosed and evicted, enumerating attacks as described are not possible, because the directory and file names are encrypted. In this case, Android bug reports – which can be generated on demand – could potentially be analyzed and may contain indications of Due Process.

What [T3] can find

Adding a computer to the adversary's arsenal, more details can be gathered, especially with adb. The restriction on debugging features for the work profile however, still prevents the installation of further software. All enumeration attacks described above are possible via adb shell as well if the work profile is unlocked or soft locked. If the profile key is evicted and not disclosed, system logs could be examined when `logcat` is used, which may contain traces of e.g. DUE PROCESS's behavior, if a dump of the logs is retrieved early after lockdown before they are overwritten by more recent logs. Analyzing log output makes it possible to fingerprint any application's behavior, even with altered package names.

ADB also allows listing all users present:

```
$ adb shell pm list users
Users:
        UserInfo{0:Owner:13} running
        UserInfo{10:Managed Profile:30} running
```

In this case, the work profile has `uid` 10. Even more info on users can be retrieved:

```
$ adb shell dumpsys user
Current user: 0
Users:
  UserInfo{0:null:13} serialNo=0 isPrimary=true
    Flags: 19 (ADMIN|INITIALIZED|PRIMARY)
    State: RUNNING_UNLOCKED
[...]
    Effective restrictions:
      no_install_unknown_sources_globally
  UserInfo{10:Managed Profile:30} serialNo=65 isPrimary=false
    Flags: 48 (INITIALIZED|MANAGED_PROFILE)
    State: RUNNING_LOCKED
    Created: +2d2h23m50s496ms ago
    Last logged in: +1h18m16s716ms ago
    Last logged in fingerprint: google/walleye/walleye:10/QQ2A
    .200501.001.B3/6396602:user/release-keys
    Start time: +1h17m39s598ms ago
    Unlock time: <unknown>
    Has profile owner: true
    Restrictions:
      no_wallpaper
    Device policy global restrictions:
      no_install_unknown_sources_globally
    Device policy local restrictions:
      no_install_apps
      no_unified_password
      no_control_apps
      no_bluetooth_sharing
      no_sharing_into_profile
      no_install_unknown_sources
      no_cross_profile_copy_paste
      no_debugging_features
    Effective restrictions:
      no_install_apps
      no_unified_password
      no_control_apps
      no_install_unknown_sources_globally
      no_bluetooth_sharing
      no_wallpaper
      no_sharing_into_profile
      no_install_unknown_sources
      no_cross_profile_copy_paste
      no_debugging_features
[...]
  Recently removed userIds: [12]
  Started users state: {0=3, 10=1}
[...]
```

This reveals all kinds of details about the work profile, including the hardening restrictions. Based on this unique set of restrictions, an adversary could conclude that DUE PROCESS is indeed the profile owner. Recently removed `userIds` are also listed, which could raise questions if a wipe was performed, but as there is no timestamp, this is rather unlikely.

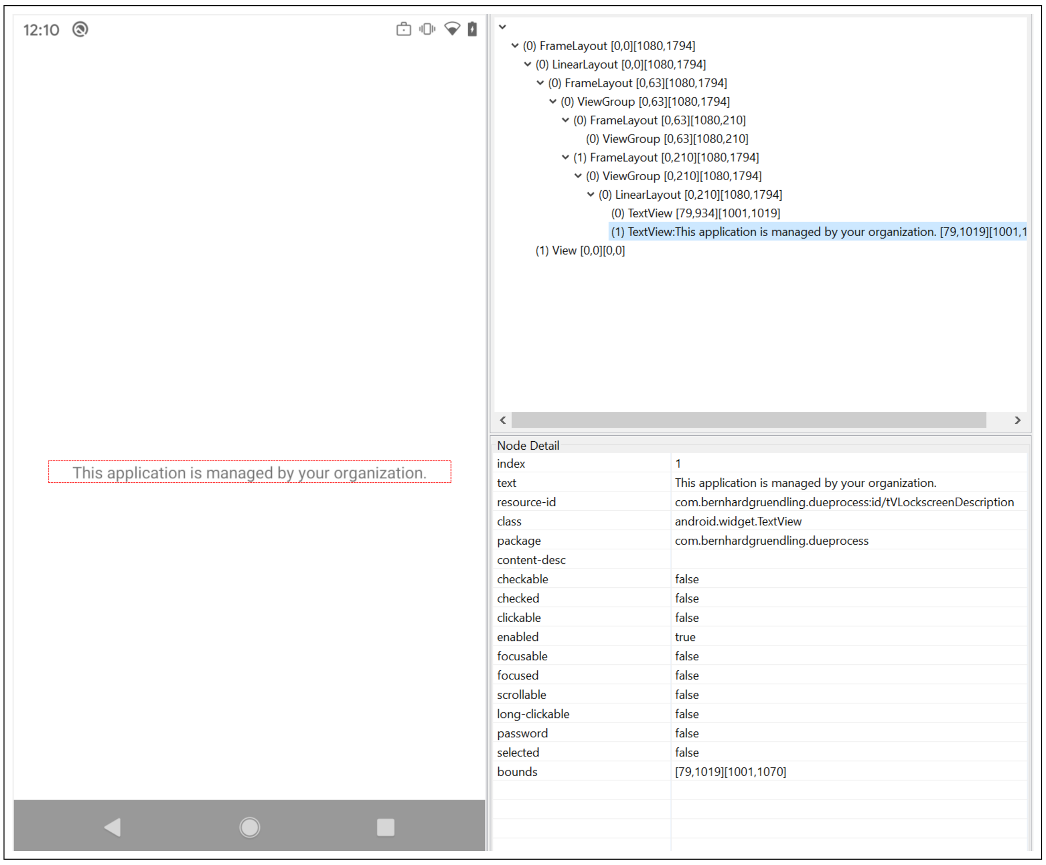
Information on all installed applications can be gathered with the following command:

```
$ adb shell dumpsys package packages
Packages:
[...]
  Package [org.thoughtcrime.securesms] (6a0c279):
    userId=10216
    dataDir=/data/user/0/org.thoughtcrime.securesms
[...]
    timeStamp=2020-05-23 20:21:22
    firstInstallTime=2020-05-23 20:21:26
    lastUpdateTime=2020-05-23 20:21:26
    installerPackageName=com.google.android.packageinstaller
    signatures=PackageSignatures{a93836c version:3, signatures:[46
   ed1dfa], past signatures:[]}
[...]
    User 0: ceDataInode=0 installed=false hidden=false suspended=
   false stopped=true notLaunched=true enabled=0 instant=false
```

```
    virtual=false
      gids=[3002, 3003]
[...]
     User 10: ceDataInode=1573040 installed=true hidden=true suspended
    =false stopped=false notLaunched=false enabled=0 instant=false
    virtual=false
[...]
```

Here, it becomes obvious that the Signal messsenger app (package `org.thoughtcrime-.securesms`) is installed only for the work profile and is currently hidden.

Assuming the management app was not hidden, it would be possible to use Android Device Monitor or Android Studio's Layout Inspector to analyze the app's view hierarchy [42]. This reveals the `resource-id` of the decoy label shown on the app's lockscreen, which affirms that thorough obfuscation is definitely necessary. Another important factor is that Due Process is properly signed and not a debug



```
(0) FrameLayout [0,0][1080,1794]
  (0) LinearLayout [0,0][1080,1794]
    (0) FrameLayout [0,63][1080,1794]
      (0) ViewGroup [0,63][1080,1794]
        (0) FrameLayout [0,63][1080,210]
          (0) ViewGroup [0,63][1080,210]
        (1) FrameLayout [0,210][1080,1794]
          (0) ViewGroup [0,210][1080,1794]
            (0) LinearLayout [0,210][1080,1794]
              (0) TextView [79,934][1001,1019]
              (1) TextView:This application is managed by your organization. [79,1019][1001,1
  (1) View [0,0][0,0]
```

Node Detail	
index	1
text	This application is managed by your organization.
resource-id	com.bernhardgruendling.dueprocess:id/tVLockscreenDescription
class	android.widget.TextView
package	com.bernhardgruendling.dueprocess
content-desc	
checkable	false
checked	false
clickable	false
enabled	true
focusable	false
focused	false
scrollable	false
long-clickable	false
password	false
selected	false
bounds	[79,1019][1001,1070]

Figure 5.2: UI dump with Android Device Monitor

build, and backup is disallowed in the manifest, so app-private data like shared preferences can not be extracted.

What [T4] can find

If an adversary manages to root the device, our device integrity assumptions do not hold anymore. In this case, the attacker can escape Android's sandbox and is not limited by directory permissions anymore.
If the profile key is not evicted or disclosed by the victim, everything stored in the work profile can be accessed.
If the profile key is evicted and not disclosed, confidentiality of data is given, but the existence of data is not plausibly deniable.

5.1.3 Security States

As described in 2.2.1, mobile operating systems currently provide only binary security states – locked or unlocked. The state diagram in figure 6.2 depicts these two states, where the state designator symbol indicates the current lock state $\{L; U\}$ of the device lockscreen. Locked is symbolized by L, while U represents an unlocked state. The edges are labeled with actions which trigger state transitions.

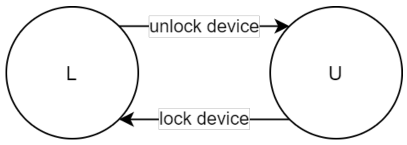


Figure 5.3: Diagram of standard security states

The diagram in figure 5.4 illustrates how Due Process introduces further security states. Here, a state is designated by four symbols, $S_1S_2S_3S_4$ where position 1 to 4 denote the lock state $\{L; U\}$ for the following security boundaries:

1. Device (boundary enforced by lockscreen)
2. Profile (boundary enforced by lockscreen)
3. Lockdown (boundary enforced by invisible management app recovery lockscreen)
4. Management app (boundary enforced by lockscreen)

Note that the following diagram only considers the base case, with the default lockdown behaviour. The user may change certain options concerning lockdown and lockdown recovery, which would result in a different state diagram.

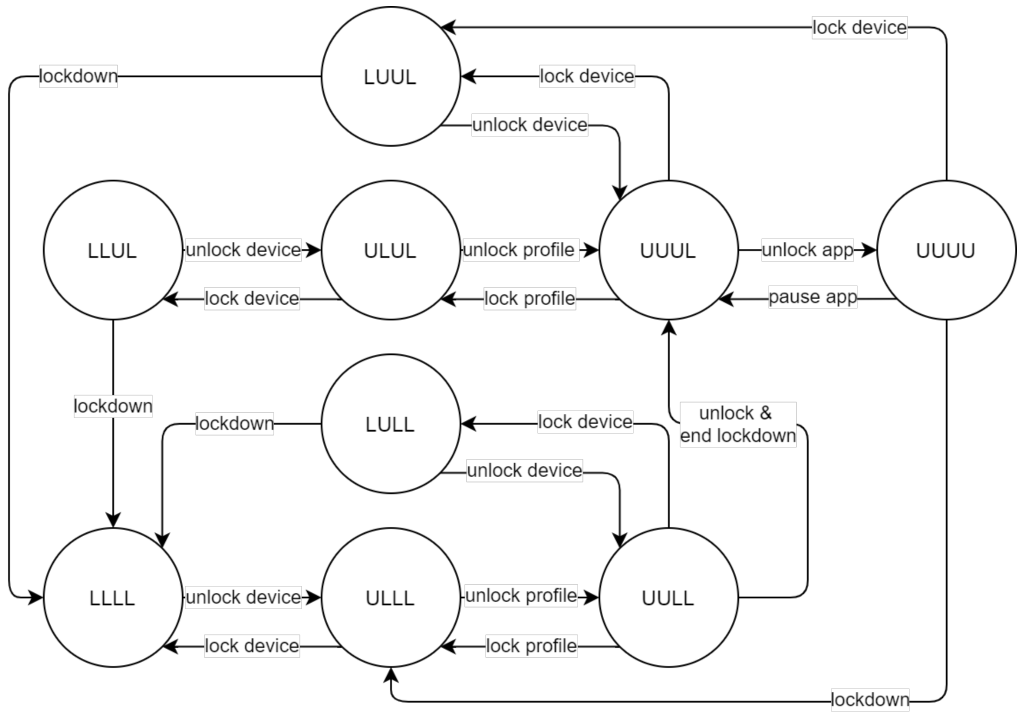


Figure 5.4: Diagram of security states with Due Process

When lockdown was activated, the following states have to be traversed in order to access hidden sensitive applications:

1. LLLL (Device locked in lockdown): Everything is inaccessible.
2. ULLL (Unlocked device lockscreen/personal profile): Work profile is inaccessible.
3. UULL (Unlocked work profile): The management app and hidden apps are inaccessible.
4. UUUL (Unlocked lockdown mode): The management app icon is now visible, but the app is locked, so hidden apps are still inaccessible.
5. UUUU (Unlocked management app): Everything is accessible and sensitive apps can be unhidden.

Regarding the security boundaries in terms of separate knowledge factors, three user provided secrets are involved, since the secrets for recovery of the management app and the app itself are the same:

1. Device lockscreen secret (PIN/pattern/password)
2. Work profile lockscreen (PIN/pattern/password)
3. Due Process app recovery and app lockscreen (pattern)

5.1.4 Plausible Deniability Effectiveness

Clearly, the system has limitations that decrease plausibility when a user wants do deny the existence of a specific application on the device. So the system is not ideal, considering the definition for ideal plausibility (section 2.1.4). Regarding the requirements for effective plausibility (section 2.1.4), the location of random data is bound to the work profile's context. Therefore, the existence of random data can be explained by claiming something along the lines of *"I use this work profile for work."*. When a DPC is deployed for the purposes of actual mobile device management, companies could leverage the mechanics of our system to protect company secrets under the given threat model.

5.1.5 Adversary Simulation

In addition to our theoretical and experimental security analysis, a simulation of real attacks on the system was performed. As genuine threat scenarios as defined by our threat model are not common enough for a real world study, these situations had to be simulated. To achieve this, some participants of the usability study were handed a device with the system in lockdown mode. This was done before the usability part of the study. Based on self-reporting concerning their technical knowledge about Android, they were then classified according to the threat model.

The device had a specific app installed inside the work profile, and this app was hidden by lockdown mode. The subject was asked to search the device for this specific application with all possible means without breaking the hardware. Two participants classified as [T1] performed the search, where one of them managed to find the mentioned not installed for this user message for the target app in Android settings. This raised doubts, but no conclusive statement about the existence of the app could be made. Two other participants with [T3]-level skills, who also searched the device, were able to use the described adb commands after some research, which increased their confidence that the app was installed. The inaccessibility of the app raised both suspicion and doubts at the same time. None of the participants were able to launch the application or retrieve any confidential data (besides meta data). As expected, these results are in line with the outcome of our forensic analysis.

The typical inspection points by all participants were as follows:

- Home screen and list of apps in launcher
- Google Play Store, installed applications
- System settings, listing of installed apps
- File explorer applications
- System settings, listing of accounts
- Work profile settings, work profile lockscreen

5.2 Usability Study

The study was conducted to recognize problems in usability. In case of DUE PROCESS, usability problems could lead to serious security and plausibility problems. The following problems could arise:

1. Users do not follow the setup and introduction process until the end, leading to frustration.
2. Users do not understand the need for the separate lockscreen for the work profile.
3. Users do not understand the need for the secret lockscreen.
4. Users do not understand how to setup the secret lockscreen.
5. Users finish the setup, but are unable to use the management app, e.g. because of the secret lockscreen.
6. Users do not understand how to use the switches and toggles in the Manage Apps screen.
7. Users do not understand how to install an app inside the work profile.
8. Users have difficulties configuring particular settings.
9. Users fail to trigger lockdown when it is necessary.
10. Users fail to trigger a wipe when it is necessary.
11. Users lock themselves out after a lockdown because they cannot recover the management app.

The secret lockscreen is one of the most critical parts of the user experience. If users fail to understand how the lockscreen works, they cannot use the management app. This issue could arise directly after finishing the introduction activity, when they need to use it for the first time. Later, when a lockdown has been triggered and the management app was hidden, they could either have forgotten how it works or still fail because of bad timing.

5.2.1 Pre-Test Questionnaire

Before the tests were started, the following information was documented for each subject with this questionnaire:

1. Experience with Android in general: I have been using Android for m months.

2. Experience with Android work profiles: I know what a work profile is. ($true/false$)

   If true: I have used a work profile in the past. ($true/false$)

3. Expertise in computer science in general: Rate your expertise in computer science on a scale of 1-10. 1 = I have no technical knowledge at all; 10 = I am a computer scientist with extensive experience; ($1 - 10$)

4. Expertise with computer security: Rate your expertise in computer security on a scale of 1-10. 1 = I have no security knowledge at all; 10 = I am a computer scientist with specialization in security; ($1 - 10$)

5. I know how to install applications from "unknown sources" (other than the Google Play Store) on an Android device. ($true/false$)

6. I know how to use the adb command line tool. ($true/false$)

7. I know the concept of plausible deniability. ($true/false$)

8. I have used a system enabling plausible deniability (e.g. True Crypt's hidden container) in the past. ($true/false$)

5.2.2 Scenarios and Tasks

The user study put the users in a scenario with three consecutive situations described in the following sections.

The Scenario

Imagine you are an investigative journalist, dealing with highly sensitive information. You are going to fly to another country and meet a whistleblower, who will hand over lots of sensitive data. The fact that you communicate with this person and all the provided data have to be kept a secret under all circumstances. You use the following apps on your Android smartphone to communicate and store sensitive data: Gmail, Contacts and Files. The device is protected by a PIN which is only known to you. However, you fear that you will be asked to unlock your phone at the border, either when entering the foreign country or on your return home.

Situation One: Preparation

When looking for a solution, you came across an app called DUE PROCESS, *which should enable you to hide the sensitive apps on your phone. You downloaded and installed the app on your phone and are now going to set it up according to the threat described. Make sure that Gmail, Contacts and Files will be protected in case of coercion. For now, these apps should stay accessible, but be prepared for sudden* *interrogations in the future. Take your time and make yourself familiar with the application and its features.*

Situation Two: Coercion

You have landed at the airport and are in line for the passport checkpoint to formally enter the country. You already feel watched by the security personnel and now fully expect to be taken aside, interrogated and searched. You might be asked to unlock your device or disclose your PIN to the border agent. You have to avoid looking suspicious now, so you cannot risk opening any sensitive apps (DUE PROCESS is one of them!). The sensitive data and apps on your phone have to kept confidential under all circumstances.

Situation Three: Recovery

You successfully entered the country and have arrived in a safe environment. Send an email with Gmail to your contact, the whistleblower.

Tasks

The following tasks result from the scenarios, which should be derived by the users implicitly.

1. Peform the setup.
2. Unlock the management app.
3. Delete the setup app in the personal profile.
4. Mark Gmail, Contacts & Files app as sensitive.
5. Configure a lockdown trigger in Settings.
6. Trigger lockdown, which should hide all sensitive apps and the management app.
7. Recover the management app.
8. Unhide Gmail.

5.2.3 Results

The study had 6 participants with varying levels of skill and knowledge about plausible deniability and computer science in general. The participants have been using Android for 73 months on average. Half of the participants knew what a work profile is, and only one has used it in the past. The self reported average expertise in computer science was 6 (on a scale of one to ten). The average security expertise was $4.1\overline{6}$ (on a scale of one to ten). Half of them knew how to install apps from unknown sources, only 2 of 6 were familiar with adb. The concept of plausible deniability was known to all but one participant, but only 2 have used software in this context. The general lack of knowledge about adb makes it clear that an approach involving adb – like a device owner DPC or custom rom installation – might not be user friendly.

On average, $6.\overline{6}$ of 8 tasks could be completed without any support or intervention. Only one participant was able to perform all tasks without any difficulties. Tasks 1,2,4,6 and 8 posed no problems for anyone. One third forgot about deleting the setup app in the preparation situation (task 3), because they did not spot the notification that prompts the uninstall. One participant was slightly confused with task 5, how to set a trigger for lockdown in the settings. The most serious problem manifested in task 7, which all but one participant struggled with: Recovering the management app after lockdown was triggered. One participant completely forgot how to initiate the recovery process (turning the work profile on and off again), but was able complete it after she was reminded of the first step. The rest of the participants remembered the three steps, but had trouble with timing. Most of them drew the pattern too early, some then completed them either by luck or after a small hint at the correct timing.
With small hints, all participants were able to eventually complete all tasks.

Besides completion of tasks, other little problems were observed as well: Two participants expressed their lack of understanding for the wipe function, which might stem from insufficient explanation during introduction. Most participants needed help finding the work profile apps in the launcher, two participants were surprised that now two instances of the apps existed. During Android's PIN/pattern/password setup sequence for the secondary lockscreen challenge, most found it a little bit confusing that they had to enter their device lockscreen PIN first, before actually configuring the profile lockscreen.

The general feedback was positive and no one reported frustration after completion of the procedure. The application's visual design was received positively. One participant recommended an additional video during the introduction that goes through lockdown recovery. Another participant suggested the use of a screen pop-up guide during lockdown recovery when lockdown is triggered for the first time.

6
Related Work

Past work and literature describe a variety of approaches for plausible deniability. In this chapter, notable examples are highlighted, some of which are more theoretical than others.

6.1 Cryptographic Schemes

A thesis on plausible deniability cannot go without mentioning Canetti et al., who describe the concept of deniable encryption in great detail for the first time in 1997 [4]. Their paper is quoted by most work in this field. For example, in [16], the authors state that the encryption scheme of Canetti et al. is insufficient, because its only purpose is deniable encryption, which would immediately indicate that there is a hidden message when using it. This is a core problem of all systems enabling plausible deniability. Therefore, Klonowski et al. show fairly practical approaches of deniable encryption which modify existing and widely known conventional encryption schemes. These small modifications then allow hiding a secret message. One example is the asymmetric ElGamal cryptosystem, where the modified variant even provides ideal plausibility [16].

6.1.1 TrueCrypt

TrueCrypt is a popular and classic open source desktop software that offers on-the-fly-encryption and has the ability to create deniable hidden volumes as described in section 2.1.4. The project's development has ceased in 2014, but since then, new derivatives (e.g. VeraCrypt) have been initiated. These succeeding projects have similar features, so we will describe them based on the original TrueCrypt software. These hidden volumes are not mandatory but an optional feature – therefore they are deniable [24]. A volume is an encrypted file on a storage medium that can be decrypted and mounted as a virtual drive.

TrueCrypt's regular encrypted volumes are containers stored as a file on top of a regular file system (e.g. `/home/user/containerFile`). It is also possible to encrypt a whole dedicated disk partition. Both are referred to as a TrueCrypt container [15]. These containers allocate a predefined amount of space and the free space is filled with random data. To decrypt containers, a passphrase (and optionally a key file) is provided by the user. Standard volumes do not support plausible deniability.

Hidden volumes are stored inside a standard TruCrypt volume. From outside, it is not discernible if a standard volume without or a volume with a hidden container inside is present. The hidden volume is stored in the free space of the regular volume, space that is usually filled with random data [15]. The container can be opened by entering either the key for the outter, regular volume or the key for the inner, hidden volume.

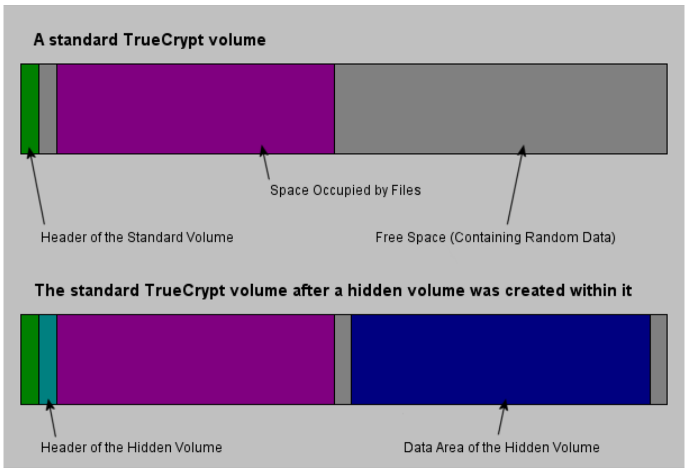

A standard TrueCrypt volume
Space Occupied by Files
Header of the Standard Volume
Free Space (Containing Random Data)
The standard TrueCrypt volume after a hidden volume was created within it
Header of the Hidden Volume
Data Area of the Hidden Volume

Figure 6.1: Standard volumes with and without a hidden container [15]

TrueCrypt does not fulfill the notion of ideal plausibility due to unreferenced random data, but it can be regarded as quite effective.

Deniable encryption solutions like TrueCrypt only target desktop operating systems.

6.2 Approaches on Mobile OS

The special architecture of mobile operating system poses particular requirements for systems enabling plausible deniability. We will look at two categories of related work: customized AOSP and app-based approaches.

6.2.1 Customized AOSP

Regarding modified versions of Android, there exist multiple excellent concepts and proof-of-concept implementations [43][44][45], however they all have complicated setup processes, so they do not fulfill our usability requirements. Furthermore, since they are based on a modified version of the Android Open Source Project (AOSP), continuous maintenance is hard. The authors therefore state that their implementation would have to be merged into the offical AOSP code or some other popular custom Android project [43].

MobiFlage [43], MobiHydra [44] and MobiPluto [45] are all modified versions of AOSP and were developed in this order, so each system improves on some aspects of the predecessor(s). They allow users to hide data in several hidden volumes, providing plausible deniability. They support multiple, user-controllable deniability levels, which is achieved by choosing the number of hidden volumes during setup [45]. When the system boots, the user chooses the volume by entering either a public decoy password or a private, hidden password. Then, an offset is derived from the password and the respective volume is mounted onto the file system mount point. The hidden volumes are located in the empty space of the device's storage. When empty space is encrypted, high entropy data is produced, so the hidden volumes are filled with random data to make them indistinguishable [44]. The basic concept of hidden volumes is comparable to the one of TrueCrypt.

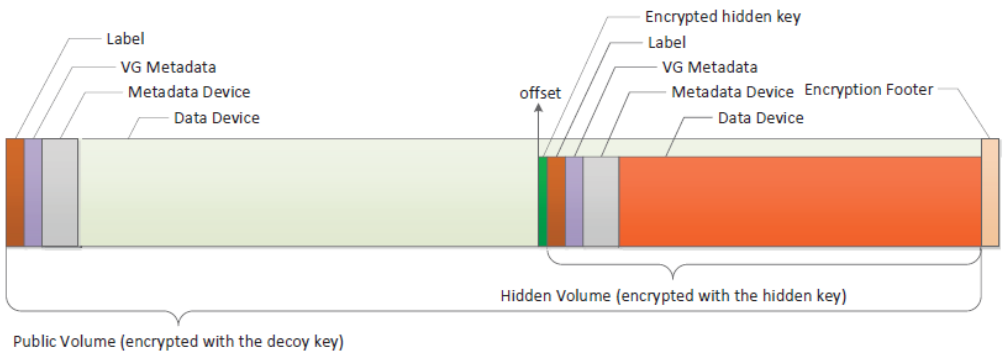


Figure 6.2: Storage layout of MobiPluto [45]

To switch between sensitive mode and normal operation, a reboot of the device is always necessary, which does not contribute towards usability. Moreover, if a user is apprehended with the device in sensitive mode, plausible deniability is not given. Furthermore, the normal, public storage spaces have to be used regularly, otherwise they might not convince an adversary when the data is old and makes the obvious impression of 'dummy data'.

6.2.2 Mobile Apps

There are already some apps in the Play Store that use the Device Policy Manager and work profiles to isolate, freeze and hide apps [46][47]. They are not designed with plausible deniability in mind, because everyone with access to the unlocked device could simply use the app themselves to unhide apps. These solutions are rather a way of separating apps from the rest of your personal apps, run a second instance of an app (e.g. for two different accounts), or freeze background-heavy apps [46].

One recent addition was released in fall 2019, called BatApps [48], which has a similar use case like Due Process. BatApps uses the work profile to store and hide private apps, and disguises itself as a calculator app to enable plausible deniability. When the user enters the correct PIN into the calculator, the hidden apps are revealed. Some features are only available after an in-app purchases, like the fake calculator feature or if more then three apps should be installed in the profile.
In our short analysis some flaws concerning security and plausible deniability have been determined: First, all of the functionality is controlled from the app's instance in the main profile. This means, an adversary can immediately observe the existence of the application on the system. This also enables an attacker to analyze the application package, since installed APKs of the personal profile are all accessible and can be copied to a computer. The second flaw is that the work profile is not secured with a separate lockscreen challenge, so the data is encrypted with the same user provided secret from the personal profile. In face of [T4], the data inside of the profile could potentially be decrypted. Another plausibility flaw arises if the calculator disguise feature is used, since at the same time, the app is shown as the work profile owner, which will potentially raise suspicion.
Furthermore, the lack of inconspicuous methods of triggering deniable mode is a small functional deficit compared to Due Process.

7
Future Work and Summary

This chapter describes potential future improvements and advancements of DUE PROCESS. Afterwards, a summary concludes this thesis.

7.1 Future Work

DUE PROCESS can be developed further in various spots. We will discuss two areas of development: Improvements and changes that concern DUE PROCESS on the application layer, and changes out of the app's scope, like AOSP modifications.

7.1.1 Improving the App

The user study revealed some issues in usability, which could be dealt with in multiple ways. First and foremost, the lockdown recovery procedure has to improved. Due its nature, being invisible and giving no feedback to the user during interaction, it proved to be the most problematic part of DUE PROCESS's user experience (UX). Of the three steps (turn profile off and on, wait a couple of seconds, draw the pattern), the second creates most confusion, since users do not know when to start drawing the pattern. After the user turns on the profile, Android takes some time to "boot" the profile, and then sends a broadcast which initiates the pattern unlock overlay. This unfortunate delay cannot be influenced by our application.
We currently see two approaches to correct this issue. As one study participant suggested, a short video could additionally visualize the recovery process during app introduction. Another user recommended a tutorial with small screen pop-ups that guides the user through the process the first time. This is a promising idea and compared to the video, it lets the user try it in a learning-by-doing style. Of course, this is only slightly mitigating the obvious, inherent UX issues of the current lockdown recovery implementation. In real world usage scenarios, lockdown might be triggered rarely on special occasions and only in long time intervals. This increases the risk that users have already forgotten the procedure again. Besides the usability

deficits of the current recovery implementation, plausible deniability may suffer if Android's system notification for displaying over other apps is not disabled by the user. Therefore, a completely different way of recovering from lockdown could be implemented. One option could be the introduction of a remote controller. The idea is that the app connects to a remote server over the Internet, which controls the DPC on the device. The user then logs into the controller software on the server (e.g. a web application) and sends a command to end lockdown to the device. This remote controller could also be used to remotely wipe the work profile or remotely trigger lockdown. This approach adds quite some technical complexity which contradicts the principle of keeping the design as simple and small as possible (economy of mechanism) [13]. The original goal for Due Process was to create an autonomous, device only solution. If the DPC starts communicating over the network, many more attack vectors have to be considered, e.g. we have to include a threat actor capable of having full control over the network infrastructure. Furthermore, a user would be unable to unlock the system without network access. A possible leak of information could also be the browser history of the user if it includes past requests to the controller web application.

In section 4.1, we mentioned a web application that generates a unique APK for the user before installing Due Process. The development of such a service could mitigate some plausibility issues that indicate the usage of our app.

Android R developer preview 3 brought some interesting additions to the device policy manager API, e.g. a feature called "secondary lockscreen". Unfortunately, this has been removed in developer preview 4 [49]. The suspected use case for this would be that the device policy controller can implement its own work profile lockscreen, replacing the system lockscreen. If this is indeed the case, it may be possible to replace the current invisible overlay lockscreen with this. It could act as the official lockscreen for the work profile, which unlocks the profile when a "public" key is entered, and additionally ends lockdown mode when a secondary, secret key is entered. This would greatly improve the usability of our system.

During initial provisioning, Android shows some educational screens about the work profile by default. These could be replaced by our own educational screens that explain our special usage of the work profile feature [26]. For this, the flag `EXTRA_PROVISIONING_SKIP_EDUCATION_SCREENS` needs to be set when provisioning is started.

The application stores the user's settings and authentication secret for the unlock pattern in shared preferences inside the app-private data directory. In the current implementation, this is stored in plaintext. If [T4] gains access to the configuration files, the list of apps marked sensitive and the pattern unlock sequence would be exposed. Following best practices for secure app development, this information should be encrypted [50]. Concerning security and plausible deniability otherwise, the possibilities seem exhausted with our app-based approach currently.

7.1.2 Modifying AOSP

In this section, we will go over some potential changes to AOSP that would improve our system.

During initialization, when provisioning is complete, the DPC makes a call to `setProfileEnabled(componentName)`, which enables the work profile. This call is not reversible and the only action that comes close is deleting the profile. In case of lockdown, this is obviously not desirable. If there was the option for a profile owner to temporarily disable a work profile, which would remove most of the related settings and UI, this would greatly help with plausible deniability.

As adb command output currently leaks most information countering plausible deniability, a good way of alleviating this would be allowing a profile owner to restrict USB debugging features globally and not only for the profile. To prevent the analysis of logcat output and bugreports, a strict separation of logs between profiles could be introduced.

Profile key eviction is currently accompanied with a persistent system notification, with no way of removing or disabling it. For normal use cases, notifying the user about this makes sense, since the (background) functionality of apps inside the profile is limited in this state. However, for our purposes it would help if the behavior of this notification was at least user controllable. When the user attempts to disable it, Android shows a message that it is impossible to disable notifications from ”your IT admin”. More granular control of notifications related to the profile admin would be appreciated.

To simplify the lockdown recovery process to some degree, it would be useful if a profile owner can make calls to the `UserManager`’s method `requestQuietModeEnabled()`. At this time, only apps which are a foreground default launcher or apps with permissions `MANAGE_USERS` or `MODIFY_QUIET_MODE` are allowed to execute this [30]. Both of these permissions are signature platform permissions which would require our app to be signed with the same key as the OS.

```
java.lang.SecurityException: Can't modify quiet mode, caller is
   neither foreground default launcher nor has MANAGE_USERS/
   MODIFY_QUIET_MODE permission
```

Requesting quiet mode on lockdown would simplify the first step of our recovery procedure: The user just needs to turn the work profile on (disable quiet mode), instead of off and on.

Android 9 introduced its own lockdown feature, which can be activated with a shortcut in a menu appearing when holding the power button. This lockdown feature disables biometric authentication for the device lockscreen and hides all notifications. A great improvement would be if Android sent a system wide broadcast when this lockdown is activated. Due Process (or any app) could register a receiver for this broadcast and perform its own lockdown procedures. This could be beneficial to a variety of applications dealing with sensitive information, assuming the user might be coerced into unlocking the device at a later point, e.g. logging the user out of an online session.

In terms of usability, it may be useful if the managed profile was renameable, so instead of ”work profile”, it could be presented to the user as ”secondary profile”, ”sensitive profile” or ”isolated profile” for use cases like Island or Shelter (see 6.2.2). This would somewhat rectify the confusion with the term ”work profile” in a non-work use case. Support for multiple secondary (work) profiles could be valuable for creating multiple layers of deniability.

Finally, if the suspected secondary lockscreen feature for work profiles mentioned in section 7.1.1 actually made it to a final release of Android, it would open up new interesting possibilities.

7.2 Summary

In this thesis, we explored the complex and nuanced topic of plausible deniability. We analyzed several implementation variants to bring plausible deniability to Android devices in a usable and yet secure way. Following the theoretical analysis, one approach was implemented, an application named Due Process. The implementation and usage of this mobile app were described in detail. Then, a thorough forensic examination of the created system was performed, which revealed some issues that were not initially anticipated during theoretical analysis. Besides the technical evaluation, we share the results of a small user study, which highlights the main usability problems with the current implementation. For comparison, we also looked at some related work in this problem space. Finally, we described several ways of improving and advancing the system.

Due to the nuanced nature of plausible deniability, it is difficult to find an approach that is both practical and (cryptographically) ideal. The feasibility of systems in this context often only comes to light after implementation. The approach we implemented does not fulfill the notion of ideal plausibility. However, we attempted to create a system that is as effective as possible, without lowering our demands for good usability too much. With some adjustments – which are mostly out of our sphere of influence – the concept could be leveraged to a powerful solution.
Nevertheless, one fact can never be eliminated, even with cryptographically ideal systems: Competent adversaries observing the mathematical possibility of hidden data could always assume that it has been employed.

The subject matter is interesting not only in respect of technical feasibility but also in terms of the societal and political reality we live in. Plausible deniability raises ethical questions, as it can be used for both good and evil purposes. Kaufman et al. state that whenever the term *plausible deniability* comes up, the person in question is almost certainly guilty [3]. Ragnarsson et al. conclude in [5]: ”[...] as a society making laws, we have to decide whether it is just and desirable to allow people to keep secrets in their heads, but not on their computers.”

This thesis emerged from a deep desire to protect the fundamental human right to privacy. Plausible deniability can be an effective way of preserving this right in certain adverse circumstances.

A

Appendix

Bibliography

[1] R. Cormac and R. J. Aldrich, “Grey is the new black: covert action and implausible deniability,” *International affairs*, vol. 94, no. 3, pp. 477–494, 2018.

[2] D. of State, “National security council directive on office of special projects (nsc 10/2),” 1948, accessed: 2020-03-29. [Online]. Available: https://history.state.gov/historicaldocuments/frus1945-50Intel/d292

[3] C. Kaufman, R. Perlman, M. Speciner, and M. Speciner, *Network Security: Private Communication in a Public World*, ser. Prentice Hall series in computer networking and distributed systems. Prentice Hall PTR, 2002.

[4] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky, “Deniable encryption,” in *Annual International Cryptology Conference*. Springer, 1997, pp. 90–104.

[5] B. Ragnarsson, G. Toth, H. Bagheri, and W. Minnaard, “Desirable features for plausibly deniable encryption,” University of Amsterdam, 2012, accessed: 2020-04-02. [Online]. Available: https://www.os3.nl/_media/2012-2013/courses/ssn/desirable_features_for_plausibly_deniable_encryption.pdf

[6] A. Hern, “Encryption software truecrypt closes doors in odd circumstances,” accessed: 2020-03-30. [Online]. Available: https://www.theguardian.com/technology/2014/may/30/encryption-software-truecrypt-closes-doors

[7] R. McMillan, “Snowden’s crypto software may be tainted forever | wired,” https://www.wired.com/2014/05/truecrypt/, (Accessed on 2020-06-04).

[8] J. Granick, “Eff answers your questions about border searches,” 2008, accessed: 2020-04-01. [Online]. Available: https://www.eff.org/de/deeplinks/2008/05/border-search-answers

[9] S. Cope, A. Kalia, S. Schoen, and A. Schwartz, “Digital privacy at the u.s. border: Protecting the data on your devices,” 2017, accessed: 2020-04-01. [Online]. Available: https://www.eff.org/wp/digital-privacy-us-border-2017

[10] “United States of America: Constitution,” United States of America, September 1787.

[11] Merriam-Webster, “Steganography,” 2020, accessed: 2020-04-01. [Online]. Available: https://merriam-webster.com

[12] J. Scharinger, “Class lecture: Cryptography,” Johannes Kepler University Linz, 2018.

[13] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[14] J. Aumasson, *Serious Cryptography: A Practical Introduction to Modern Encryption.* No Starch Press, 2017.

[15] “Truecrypt user’s guide, version 7.1a,” 2012, retrieved from software package TrueCrypt.

[16] M. Klonowski, P. Kubiak, and M. Kutyłowski, “Practical deniable encryption,” *SOFSEM 2008: Theory and Practice of Computer Science*, pp. 599–609, 2008.

[17] B. Gründling, “An implementation of ideal deniable encryption,” Bachelor’s Thesis, Johannes Kepler University Linz, 2018.

[18] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, “The android platform security model,” *CoRR*, vol. abs/1904.05572, 2019. [Online]. Available: http://arxiv.org/abs/1904.05572

[19] R. Mayrhofer, “Class lecture: Android security,” Johannes Kepler University Linz, 2019.

[20] “Android enterprise security white paper 2019,” 2019.

[21] “Android debug bridge (adb) | android developers,” https://developer.android.com/studio/command-line/adb, Google, (Accessed on 2020-04-17).

[22] “Support different platform versions | android developers,” https://developer.android.com/training/basics/supporting-devices/platforms.html, Google, (Accessed on 2020-04-07).

[23] “Android compatibility definition document,” https://source.android.com/compatibility/cdd, Google, (Accessed on 2020-04-07).

[24] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier, “Defeating encrypted and deniable file systems: Truecrypt v5. 1a and the case of the tattling os and applications.” in *HotSec*, 2008, pp. 7:1–7:7.

[25] G. Pandian and V. Gupta, “Perils of running apps in android virtual containers – android security symposium,” https://android.ins.jku.at/symposium/2020/program/gautam-arvind-pandian-and-vikas-gupta/, (Accessed on 2020-06-04).

[26] “Devicepolicymanager | android developers,” https://developer.android.com/reference/android/app/admin/DevicePolicyManager, Google, (Accessed on 2020-04-15).

[27] “android-testdpc/readme.md at master · googlesamples/android-testdpc,” https://github.com/googlesamples/android-testdpc/blob/master/README.md, Google, (Accessed on 2020-04-15).

[28] "Codenames, tags, and build numbers | android open source project," https://source.android.com/setup/start/build-numbers, Google, (Accessed on 2020-04-17).

[29] P. Ratazzi, Y. Aafer, A. Ahlawat, H. Hao, Y. Wang, and W. Du, "A systematic security evaluation of android's multi-user framework," *arXiv preprint arXiv:1410.7752*, 2014.

[30] "Usermanager | android developers," https://developer.android.com/reference/android/os/UserManager.html#DISALLOW_UNIFIED_PASSWORD, Google, (Accessed on 2020-04-22).

[31] "Crossprofileapps | android developers," https://developer.android.com/reference/android/content/pm/CrossProfileApps.html, Google, (Accessed on 2020-04-22).

[32] "Appintro/appintro: Make a cool intro for your android app." https://github.com/AppIntro/AppIntro, AppIntro, (Accessed on 2020-05-04).

[33] "androidx.preference | android developers," https://developer.android.com/reference/androidx/preference/package-summary, Google, (Accessed on 2020-04-23).

[34] "android.provider | android developers," https://developer.android.com/reference/android/provider/package-summary, Google, (Accessed on 2020-05-10).

[35] "Capture a system trace on a device | android developers," https://developer.android.com/topic/performance/tracing/on-device, Google, (Accessed on 2020-05-10).

[36] "Implicit broadcast exceptions | android developers," https://developer.android.com/guide/components/broadcast-exceptions, Google, (Accessed on 2020-04-25).

[37] "Windowmanager.layoutparams | android developers," https://developer.android.com/reference/android/view/WindowManager.LayoutParams, Google, (Accessed on 2020-04-25).

[38] "Flag_watch_outside_touch doesn't return location for action_outside events on 4.2+ [36998614] - visible to public - issue tracker," https://issuetracker.google.com/issues/36998614, (Accessed on 2020-04-25).

[39] "AOSP: services/input/inputdispatcher.cpp - platform/frameworks/base - git at google," https://android.googlesource.com/platform/frameworks/base/+/android-4.2.1_r1/services/input/InputDispatcher.cpp#1416, Google, (Accessed on 2020-04-25).

[40] T. Niedermayr, "Android keylogger," Bachelor's Thesis, Graz University of Technology, 2014.

[41] “android - ”app is displaying over other apps” notification - stack overflow,” https://stackoverflow.com/questions/48909610/app-is-displaying-over-other-apps-notification, (Accessed on 2020-05-26).

[42] “Android device monitor | android developers,” https://developer.android.com/studio/profile/monitor, Google, (Accessed on 2020-05-23).

[43] A. Skillen and M. Mannan, “Mobiflage: Deniable storage encryption for mobile devices,” *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 3, pp. 224–237, 2014.

[44] X. Yu, B. Chen, Z. Wang, B. Chang, W. T. Zhu, and J. Jing, “Mobihydra: Pragmatic and multi-level plausibly deniable encryption storage for mobile devices,” in *International Conference on Information Security.* Springer, 2014, pp. 555–567.

[45] B. Chang, Z. Wang, B. Chen, and F. Zhang, “Mobipluto: File system friendly deniable storage for mobile devices,” in *Proceedings of the 31st Annual Computer Security Applications Conference.* ACM, 2015, pp. 381–390.

[46] PeterCxy, “Shelter: Isolate your big brother apps,” https://github.com/PeterCxy/Shelter, (Accessed on 2020-05-30).

[47] oasisfeng, “Island for android,” https://github.com/oasisfeng/island, (Accessed on 2020-05-30).

[48] “Batapps – apps on google play,” https://play.google.com/store/apps/details?id=com.batapps.android&hl=de_AT, Be Anonymous Technologies, Inc, (Accessed on 2020-05-30).

[49] “Android R DP4, removed method in android.app.admin.DevicePolicyManager,” https://developer.android.com/sdk/api_diff/r-dp4-incr/changes/android.app.admin.DevicePolicyManager, Google, (Accessed on 2020-06-01).

[50] “Data storage on android - mobile security testing guide,” https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05d-testing-data-storage, OWASP, (Accessed on 2020-06-02).

BERNHARD GRÜNDLING

Computer Science Student & Associate Security Consultant

EDUCATION

Master's Program Computer Science
Johannes Kepler University Linz
June 2018 – ongoing (expected 2020)
Major: Networks and Security

Bachelor of Computer Science
Johannes Kepler University Linz
October 2013 – June 2018
Thesis: An Implementation of Ideal Deniable Encryption

Emergency Medical Technician
Red Cross Upper Austria
November 2013 – January 2014
Various additional qualifications

High School
BRG Enns
September 2005 – June 2013
Graduated with distinction

LANGUAGES

German	●●●●●
English	●●●●○
French	●●○○○
Latin	●●○○○

EXPERIENCE

Associate Security Consultant
SEC Consult Unternehmensberatung GmbH
August 2019 – ongoing   Linz/Vienna

Volunteer Emergency Medical Technician
Red Cross Upper Austria
July 2014 – ongoing   Bad Hall/Enns/Linz
Ambulance Service / Emergency Medical Service

Film Production
Freelance
2009 – ongoing   Austria

Alternative Civilian Service
Red Cross Upper Austria
November 2013 – July 2014   Bad Hall
Ambulance Service / Emergency Medical Service

Internship IT
EUROFIT Informationstechnologie GmbH
August 2011   Linz
August 2010   Linz
Internship in a systems house

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 7. Oktober 2020 Bernhard Gründling, BSc