BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2020 Bc. Lukáš Balaževič

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

SECURITY MECHANISMS OF OS ANDROID UTILIZING THE KOTLIN LANGUAGE

MECHANISMY ZABEZPEČENÍ OS ANDROID S VYUŽITÍM JAZYKA KOTLIN

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. Lukáš Balaževič
SUPERVISOR VEDOUCÍ PRÁCE	Ing. Kryštof Zeman, Ph.D.

BRNO 2020

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study field **Communications and Informatics**

Department of Telecommunications

Student:	Bc. Lukáš Balaževič	***ID:***	186416
Year of study:	2	***Academic year:***	2019/20

TITLE OF THESIS:

Security mechanisms of OS Android utilizing the Kotlin language

INSTRUCTION:

The target of this diploma thesis is a comprehensive study of security mechanisms of an Android OS, together with options on how to secure communication between the OS, users and remote server (i.e., Cloud storage) in mobile applications that are utilizing the Kotlin language. This knowledge will be further implemented in a sample application. The theoretical part will be dedicated to in detail description of security mechanisms theory, with emphasis on novel cryptographic methods (i.e., elliptic curves), together with “best practices” of Android OS mobile development (utilization of Android KeyChain, Dependency Injection, Reactive programming, etc.). Furthermore, this knowledge will be implemented in a sample application, which is serving as verification of their validity. The outcome of this thesis will be an in-detail overview of current security mechanisms together with their ideal implementation procedures and a working sample application for OS Android that demonstrates them.

RECOMMENDED LITERATURE:

[1] Kotlin documentation [online], 2019. [cit. 2019-09-16]. Dostupné z: https://kotlinlang.org/docs/tutorials/kotlin-android.html

[2] JEMEROV, Dmitry a Svetlana ISAKOVA, [2017]. Kotlin in action. Shelter Island, NY: Manning Publications Co. ISBN 978-161-7293-290.

Date of project specification:	3.2.2020	***Deadline for submission:***	1.6.2020

Supervisor: Ing. Kryštof Zeman, Ph.D.

prof. Ing. Jiří Mišurec, CSc.
Subject Council chairman

ABSTRACT

Mobile devices are a novelty in technological history. With technology that is evolving at such a rapid pace and growth in use, it is necessary to pay attention to security. This diploma thesis deals with the analysis of the security mechanisms used in the Android OS and the communication between the Android OS and the remote server. The aim is to examine these mechanisms and test which cryptographic methods and procedures are most advantageous in terms of security with regard to efficiency. This knowledge was used to create a demonstration system that uses selected security mechanisms and cryptographic methods.

KEYWORDS

Android, Android security, MVVM, Cryptography, RSA, EC, Digital signature, JWT, Microservices, gRPC, Kubernetes, Istio, Docker

ABSTRAKT

Mobilné zariadenia sú v rámci technologickej histórie novinka a pri technológii, ktorá sa vyvíja tak rapídnym tempom a rastom používania je nutné dbať na zabezpečenie. Táto diplomová práca sa zaoberá rozborom bezpečnostných mechanizmov používaných v Android OS a komunikáciou medzi OS Android a vzdialeným serverom. Cieľom je preskúmať tieto mechanizmy a otestovať aké kryptografické metódy a postupy je najvýhodnejšie používať z hľadiska bezpečnosti s ohľadom na efektivitu. Tieto znalosti boli použité pre vytvorenie demonštračného systému, ktorý využíva vybrané zabezpečovacie mechanizmy a kryptografické postupy.

KĽÚČOVÉ SLOVÁ

Android, Zabezpečenie Androidu, MVVM, Kryptografia, RSA, EC, Digitálny podpis, JWT, Mikroslužby, gRPC, Kubernetes, Istio, Docker

BALAŽEVIČ, Lukáš. *Security mechanisms of OS Android utilizing the Kotlin language*. Brno, 2020, 107 p. Master's Thesis. Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications. Advised by Ing. Zeman Kryštof, Ph.D.

ROZŠÍŘENÝ ABSTRAKT

Smartfóny sú jedny z najrýchlejšie sa rozvíjajúcich a bežne dostupných technológií. S rýchlym rastom užívateľskej základne a softvérových funkcií sa otázky bezpečnosti stávajú relevantnejšími. Užívatelia používajú smartfóny častejšie na riadenie každodenných úloh, ako sú e-mail, internetové bankovníctvo a mobilné platby. Všetky tieto úlohy si vyžadujú citlivé užívateľské údaje, ktoré sa vo väčšine prípadov ukladajú priamo do smartfónu. Vďaka zariadeniu, ktoré prichádza všade kam používateľ a ktoré obsahuje citlivé údaje, sa bezpečnosť musela vyvinúť nad úroveň zabezpečenia stolného počítača. Výrobcovia mobilných telefónov a vývojári aplikácií musia zaistiť bezproblémové prostredie pre všetkých používateľov (dokonca aj pre tých, ktorí nie sú odborníkmi v oblasti zabezpečenia alebo sa nimi nechcú priamo zaoberať) implementovať bezpečnostné opatrenia na ochranu údajov používateľov, a to aj v situácií pri krádeži smartfónu.

Cieľom tejto diplomovej práce je vysvetliť model zabezpečenia smartfónov v operačnom systéme Android a porovnať existujúce kryptografické protokoly, ktoré Android OS podporuje. Ďalším cieľom je na základe získaných teoretických znalostí a výsledkov z testovania implementovať ukážkový systém, ktorý preukáže reálne použitie týchto bezpečnostných mechanizmov a bude klásť dôraz na overené praktiky pri implementácií.

Prvá kapitola popisuje vývoj smartfónov a predstavuje systém Android, systém verzií systému Android a spôsob distribúcie aplikácií v ekosystéme Android OS.

Druhá kapitola popisuje, z ktorých častí sa model zabezpečenia systému Android skladá, ich účel a spôsob práce s inými systémovými komponentmi, aby poskytovali sofistikované zabezpečenie používateľských údajov. Okrem popisu jednotlivých komponentov a vysvetlení na čo slúžia, je v tejto kapitole kladený dôraz na ukážky aké kryptografické operácie Android podporuje a ako ich je možné použiť.

Tretia časť je založená na teórii opísanej v druhej časti. Sumarizuje výsledky získané z referenčnej aplikácie, v ktorej bolo implementovaných a vykonaných 280 testov. Použité testovacie prípady pokrývajú väčšinu kryptografických operácií, ktoré je možné vykonávať v systéme Android. Ich hlavným cieľom bolo vyhodnotiť výpočtový čas každej kryptografickej operácie. Výsledky boli ďalej spracované a vizualizované v tepelných mapách a stĺpcovom grafe. Na základe výsledkov sa dospelo k záveru, že nie všetky predpoklady boli splnené. Napríklad v niektorých prípadoch bolo šifrovanie RSA rýchlejšie ako šifrovanie AES. Tento prípad je možné vidieť u zariadení Google Pixel XL, Motorola Moto G Plus, LG Nexus 5X a ďalšími. Určité staršie modely zariadení vykonávajú niektoré kryptografické algoritmy rýchlejšie ako novšie zariadenia. Príkladom tohto správania sú zariadenia HTC One M9 (20 nm procesor) a Samsung Galaxy A5 (14nm procesor) ak na týchto dvoch zariadeniach porovnáme vytváranie RSA alebo EC kľúča tak v

každom prípade je HTC One M9 rýchlejší. To je možné vysvetliť hardvérovou podporou určitých kryptografických algoritmov. Po analýze výsledkov a zohľadnením širokého spektra zariadení, ktoré sú podporované boli vybrané algoritmy, ktoré sú vhodné a optimálne pre implementáciu aplikácie, ktorá využíva kryptografické operácie. Ohľad sa bral hlavne na to, aby daný algoritmus bol vykonávaný, čo v najmenšom časovom rozmedzí na všetkých zariadeniach. Vybrané algoritmy sú AES256/GCM/NoPadding pre symetrické šifrovanie, SHA512withRSA2048/PSS pre digitálny podpis a RSA3072/OAEPWithSHA512AndMGF1Padding pre asymetrické šifrovanie.

Posledná štvrtá časť popisuje systém SecNote. Systém SecNote je implementácia kompletného riešenia, ktoré demonštruje bezpečnostné mechanizmy systému Android a osvedčené postupy ako ich implementovať. V tejto kapitole je taktiež popísanej ako implementovať bezpečnú komunikáciu medzi aplikáciou Android a cloudovým systémom. Cloudový systém je riadený pomocou Kubernetes a Istia. Bežia tu tri mikro-služby s tromi databázami. Systém SecNote demonštruje bezpečnostné mechanizmy, ako napríklad:

- **Biometrické overenie** - Aplikácia vyžaduje pre funkčnosť, aby telefónne zariadenie bolo bezpečné, čiže musí mať nastavený minimálne PIN a v ideálnom prípade biometrické overenie. V prípade, že ani jeden z mechanizmov nie je nastavený, aplikácia vyžiada od používateľa, aby si dané overenie nastavil ak chce aplikáciu naďalej používať. Vždy je možné, ako záložný mechanizmus overenia použiť PIN v prípade, že by sa biometrický senzor pokazil alebo inak by znemožil používateľovi sa týmto spôsobom overiť. Po overení používateľa sa otvorí časové okno, v ktorom je možné používať kryptografický materiál aplikácie, ktorý je využívaný na podpisovanie žiadosti, šifrovanie a dešifrovanie poznámok, šifrovanie úložiska.
- **Časovo obmedzené prihlásenie** - Existujú 2 časové okná, a to okno aktívnej interakcie, ktoré využíva prístupový token s dobou platnosti 5 minút a obnovovací token, ktorý platí 7 dní. Po uplynutí piatich minút sa aplikácia pokúsi obnoviť prístupový token pomocou obnovovacieho tokenu. Ak používateľ nepoužíva aplikáciu dlhšie, ako 7 dní je z aplikácie odhlásený a musí sa znova prihlásiť.
- **Spôsob prezentácie identity v cloudovom riešení** - Pri daných požiadavkách používateľ neposiela priamo v parametroch svoje ID, ale namiesto toho si služby získavajú ID užívateľa z JWT tokenu, ktorý je pripojený do kontextu požiadavku. Týmto spôsobom je zaručené, že sa v skutočnosti jedná o daného užívateľa, pretože tokeny sú digitálne podpísané.
- **Digitálny podpis požiadavkou** - Každý požiadavok, ktorý je odoslaný z aplikácie na dátové služby je digitálne podpísaný používateľom v aplikácií.

Podpis sa overuje na strane serveru, aby sa zaručilo, že parametre požiadavku neboli zmenené.

- **Šifrovanie a dešifrovanie údajov** - Používateľ môže jednotlivé poznámky, ktoré sú v aplikácií šifrovať a dešifrovať pomocou kľúčov, ktoré si v aplikácií vytvorí. Tieto kľúče nikdy neopustia TEE systém takže nie je možné, aby poznámky dešifroval dakto iný, než používateľ na svojom zariadení.
- **Vytvorenie bezpečného kanála medzi aplikáciou a cloudom** - Aplikácia komunikuje s cloudovým riešením pomocou gRPC protokolu. Medzi aplikáciou a cloudovým systémom je vytvorený zabezpečený gRPCs kanál, ktorý je chránený pomocou TLS. Certifikát je registrovaný na doménu secnote.space a o jej platnosť a obnovu sa stará zautomatizovaný systém, ktorý beží na cloudovom riešení.

V tejto práci bol popísaný komplexný model zabezpečenie systému Android. Boli rozobraté detailne jednotlivé komponenty tohto systému ich funkcionalita a úloha. Následne bola vytvorená aplikácia, ktorá testuje pomocou 280 testov kryptogragické operácie, ktoré sú podporované v systéme Android. Na základe výsledkov z aplikácie bola vytvorená vizualizácia týchto výsledkov, z ktorej boli odvodené závery aké algoritmy je najvýhodnejšie použiť pri implementácii aplikácie. Na základe rozobranej teórie a výsledkov z aplikácie bol implemtovaný ukážkový systém, tvorený z Android aplikácie a troch mikro-služieb, ktoré bežia na cloude. Tento systém demonštruje jednotlivé bezpečnostné mechanizmy a ukazuje, ako je ich možné implementovať.

ACKNOWLEDGEMENT

I want to thank my supervisor, Mr. Ing. Zeman Kryštof, Ph.D. for professional guidance, consultations, patience, and always fast feedback for my work. I would also like to thank my parents for their continued support and faith in me. And last but not least, I would like to thank my girlfriend who, despite the stressful finishing weeks, always stood by me and helped me as much as possible.

Contents

List of Figures

List of Tables

Listings

Introduction

Smartphones are one of the most rapidly evolving and widely available technologies. With the fast growth in user base and software features, security questions become more relevant. Users are using smartphones more frequently to manage daily tasks such as email, internet banking, and mobile payments. All these tasks require sensitive user data, which in most cases, are saved directly on the smartphone. With the device that comes everywhere where the user goes and also contains sensitive data, security had to evolve beyond a desktop PC's security level. To provide a seamless experience for all users (even the ones who are not proficient in security, or do not want to deal with it directly), mobile manufacturers and application developers need to implement security measures that protect users' data, even in smartphone theft situations.

This thesis aims to explain the security model on smartphones using the Android OS and to compare existing cryptographic protocols. The first chapter describes the evolution of smartphones and presents the Android OS, the Android versioning system, and how applications are distributed within the Android OS ecosystem. The second chapter describes what parts the Android security model is composed of, their purpose, and how they work with other system components to provide sophisticated security for user data. The third chapter visualizes and summarizes the results obtained from the created benchmark application. Last, the fourth chapter describes the SecNote system, which is implemented based on knowledge gained from the theory and results obtained from tests. The SecNote system showcases practical usage of security mechanisms and best practices on how to implement them.

Part I

Theory

1 Mobile devices

A Mobile device, or by some called a handheld computer, is a computing device that can function independently and is small amply to hold and manipulate in hand. Inbuilt LCD or OLED flatscreen with integrated touch input is one of the many standard features from the feature-rich diverse set. Mobile devices usually connect to the Internet and communicate with other devices that can be far from each other. Devices located nearby each other can communicate by Wi-Fi, Bluetooth, or NFC. They can bear single or multiple integrated cameras and, in most cases, are powered by lithium batteries. Security is an essential part of mobile devices, and in order to satisfy current security standards, mobile devices can include biometric sensors and specialized hardware for cryptographic operations. The manufacturer chooses the operating system used on the mobile device, and it can range from small embedded systems to robust feature-rich ecosystems. A smartphone is a special derivation of the mobile device, which this thesis deals with a more significant deal.

1.1 History

IBM made the first smartphone and released it to the public on August 16, 1994. IBM Simon Personal Communicator os simply IBM Simon, was the first PDA that included features for telephony.

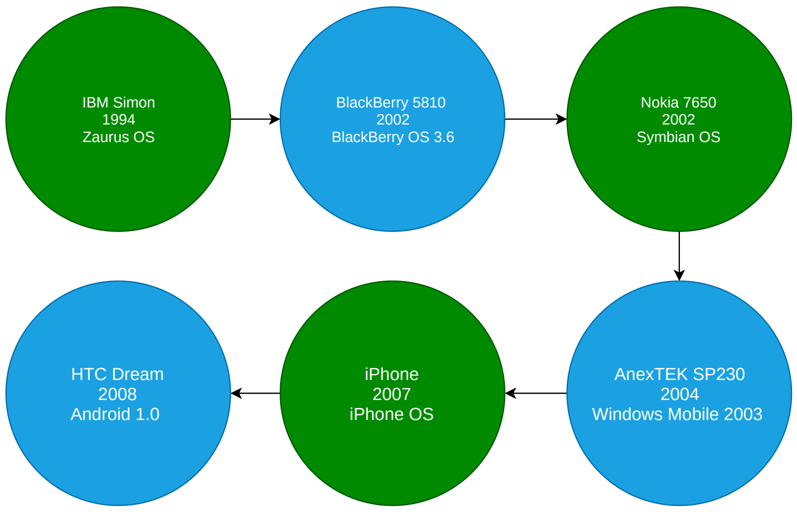


Fig. 1.1: Smartphone OS evolution

From the days of IBM Simon, more than ten years passed before we got to the modern days of IOS or Android smartphones. Figure 1.1 shows evolution history from days of IBM Simon to the days of the first Android smartphone in the year 2008.

Blackberry OS, Symbian OS, and Windows Mobile OS attempted to established a modern standard of smartphone OS yet still failed. The change came in 2007 with the introduction of Apple iPhone. This concept was further followed by the introduction of the first Android phone in 2008. iPhone and Android are, to this day, the most successful operating systems and own majority market share. Sucess of smartphones can be proven by increasing market share 1.2, which has been increasing since the first smartphone appeared. In August 2016, mobile devices' and desktop computers had equal market share. Since September 2016, mobile devices hold dominance in market share besides a few months where the market share was equal or slightly under the desktop market share.

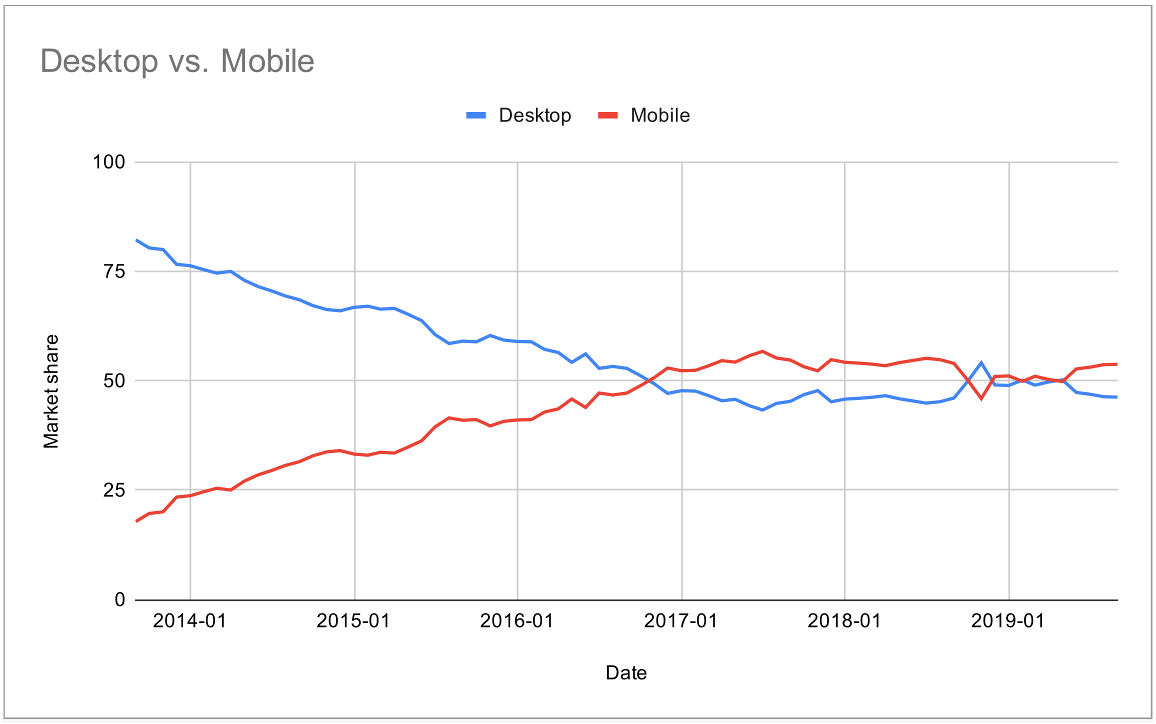


Fig. 1.2: Mobile vs Desktop market share [1]

1.2 Android

Android is an open-source software stack that can run on a broad arrangement of devices from embedded, and smartwatches to mobile phones, tablets, and cars. Android is designed so that there is no central point of failure. The resulted platform is available to manufacturers to create products that improve the mobile experience for users.

1.2.1 Versions

Android defines the version using three things: code name, version number, and API level. Besides the first release of Android, which does not have the code name, all other versions have code names inspired by confectionery and are alphabetically ordered. The release of Android 10 brings many changes. One of the changes lies within the code naming convention. Android 10 brings an end to names inspired from confectionery, and instead, code names are named with Android prefix and version number, so Android 10, Android 11. The version number is used as a software versioning number. API version is increased with every new release of Android API. Diagram 1.3 states the market share of devices using the given Android version. Diagram 1.4 indicates how many devices will be supported by the developed

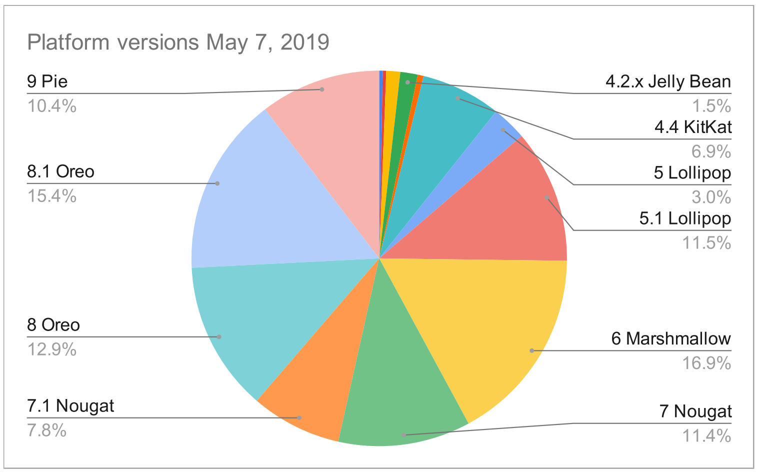


Fig. 1.3: Distribution of Android versions [2]

application if the minimum supported SDK level is chosen as given Android API.

1.2.2 Application distribution

Android application can have two formats: APK and AAB. AAB is a newer format that supports dynamic delivery. The dynamic delivery feature provides the possibility for users to download only part of the application that is important for their device os version, which means a smaller application size for users to download [3].

After the successful upload of APK or AAB to the store, the user can download the application from the store. There are many stores from which users can download applications, but Google officially supports only one store. Google Play is an official application store made and curated by Google. It is by far the safest way to get applications for Android. Every application submitted to the Google Play store is

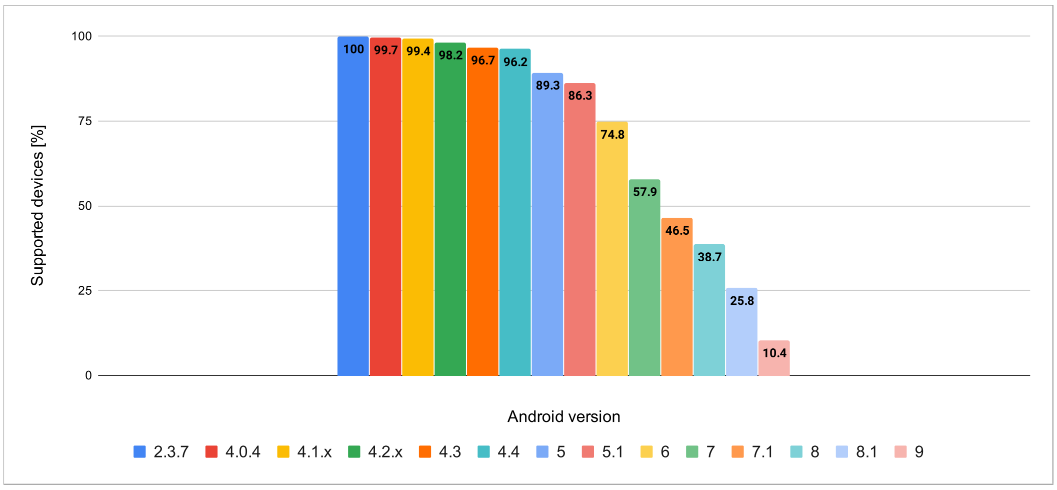


Fig. 1.4: Supported devices with minumum SDK [2]

subjected to automation tests, and if any indication of safety concerns arise, the application is submitted to manual testing. Service that handles application testing is named Google play protect. Google Play Protect provides on-device and cloud-based protection [4].

2 Android security overview

Google works tightly with developers and device manufacturers to incorporate industry-leading security features to the Android and keep the ecosystem safe. This approach resulted in a robust security model implemented in the Android ecosystem [5]. The following sections are targeted at the android security model and principles description. Furthermore, it provides an in-depth description of necessary components together with their purpose and relations between them.

2.1 Authentication

The android ecosystem has to ensure that the user's data are safe and secure. Authentication mechanisms are leveraged to provide this kind of assurance and forbid unauthenticated users' access. Android uses the concept of user-authentication-gated cryptographic keys that require cryptographic key storage, cryptographic key service provider, and user authenticator [6].

2.1.1 Cryptographic key storage and service provider

Cryptographic key storage is responsible for storing cryptographic keys. Android Keystore occupies a position of cryptographic hardware-backed key-storage. A service provider provides standard cryptographic routines on top of keys provided by cryptographic key storage. Keymaster is an implementation of a cryptographic service provider [6].

2.1.2 Authentication mechanisms

The primary purpose of authentication mechanisms is to attest to users' presence and/or successful authentication. These mechanisms are implemented in Gatekeeper, Fingerprint, and BiometricPrompt classes. The Fingerprint class supports fingerprint authentication, but only up to Android P, which corresponds to API below 29. Starting with the Android P, BiometricPrompt should be utilized for user authentication using the fingerprint or additional biometrics. The BiometricPrompt itself is a single integration point for biometric authentication of any kind. All these components communicate their authentication state to the Keystore service through an authenticated channel [6].

2.1.3 Credential enrolment

On the first boot of the device after a factory reset, all authenticators are prepared to receive credential enrollments from the user. A user must initially enroll a PIN, pattern, or password with Gatekeeper. Otherwise, an application that uses security features won't work correctly and prompt the user to enroll with Gatekeeper. Initial enrolment with a Gatekeeper creates a randomly generated, 64-bit user secure identifier (SID) that serves as an identifier for the user and as a binding token for the user's cryptographic routines. The SID is cryptographically bound to the user's security chosen measure [6].

An attacker won't be able to change or access the user's credentials unless he knows explicitly the user's PIN, pattern, or password that depends on what the user has chosen on enrolment. Under normal conditions, the Android framework does not allow an untrusted enroll, but it is possible to force it. In this case, if existing credentials are not provided, the new credentials are enrolled with an entirely random User SID. The attacker can access the device, but keys created under the old user SID are permanently lost. This procedure is known as an untrusted enroll. A user who wants to replace a credential propperly must exhibit an existing credential. If a current credential is verified successfully, the user SID associated with the existing credential is transferred to the new credential, enabling the user to keep accessing keys after changing a credential [6].

2.1.4 Authentication

A user that successfully set up credentials and were provided with user SID can start authenticating. Authentication is a process where the user uses created credentials by providing a PIN, pattern, or password to verify his identity against the Android system. Diagram 2.1 is an example of how the user authenticates with user authenticators using a TEE. An example with TEE is chosen because it is the most common system configuration by the date of writing this thesis.

Every user authenticator has a dedicated deamon. PIN, pattern, or password uses LockSettingsService, which sends requests to gatekeeperd. Biometrics-based authentication on android devices below Android P calls FingerprintService. FingerprintService requests fingerprintd. Biometrics-based authentication on android devices running Android P and above requests BiometricPrompt. BiometricPrompt requests biometric manager, which requests appropriate biometric daemon. A user provides an authentication method, and the associated service requests the associated daemon. The daemon sends data to its counterpart, which generates an AuthToken. Fingerprint deamon listens for fingerprint event. After fingerprintd receives fingerprint event, it sends data to Fingerprint in a TEE. If authentication in TEE succeeds, Fingerprint in TEE sends an auth token signed with auth token HMAC key to fingerprintd in Android OS. For PIN, pattern, or password, the flow is similar with just different used service and deamon. After deamon receives signed token, it passes auth token to the Keystore service through an extension to the Keystore service's Binder interface. The Keystore service passes the auth tokens to Keymaster and verifies them using the key. The key is shared with the Gatekeeper, Fingerprint, and other supported biometric TEE components. Keymaster trusts the timestamp in the token as the last authentication time and bases decision if to allow an app to use the key on it [6].

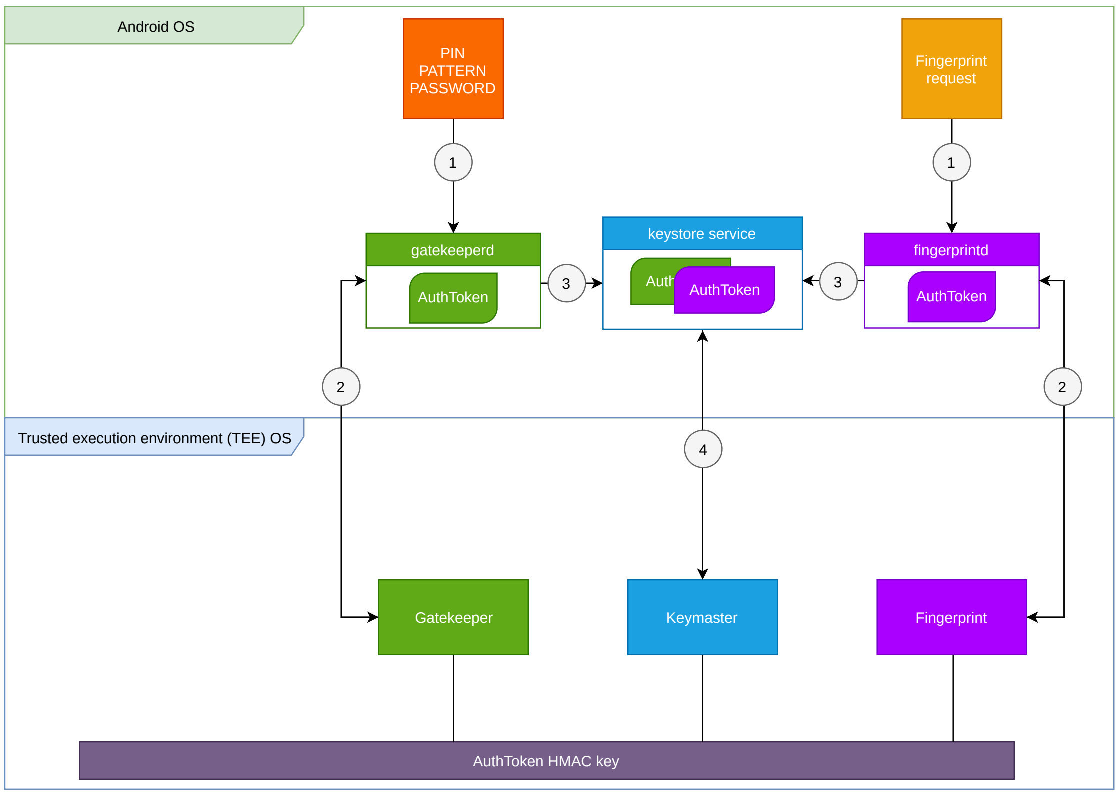


Fig. 2.1: Authentication flow [6]

2.2 Gatekeeper

Gatekeeper, as an authentication notion, consists of two parts. Gatekeeper deamon referred to as gatekeeperd that lives in an Android OS and Gatekeeper that lives in Trusted Execution Environment (TEE). Gatekeeper's primary intent is to verify password, pattern, or PIN via an HMAC that is backed by a secret hardware key. Gatekeeper can refuse to verify a password, pattern, or PIN if it is presented with consequence failed attempts to authenticate. In this case, Gatekeeper throttles authentication requests and timeouts next requests based on the count of the previous failed attempts.

After a success password, pattern, or PIN verification, Gatekeeper uses TEE-derived shared secret to sign an authentication attestation that is sent back to gatekeeperd. Gatekeeperd sends this key to the hardware-backed Keystore. Authentication attestation is a sign for Keystore that the app can use application created keys [7].

2.2.1 Architecture

How gatekeeper elements communicate between themselves is described in diagram 2.2. Lock setting service makes requests via binder interface to gatekeeper deamon, which gives the Android framework APIs access to the HAL implementation. The android framework can use the gatekeeper daemon to communicate with Gatekeeper in TEE via HAL implementation to authenticate the user‘s password, pattern, or PIN [7].

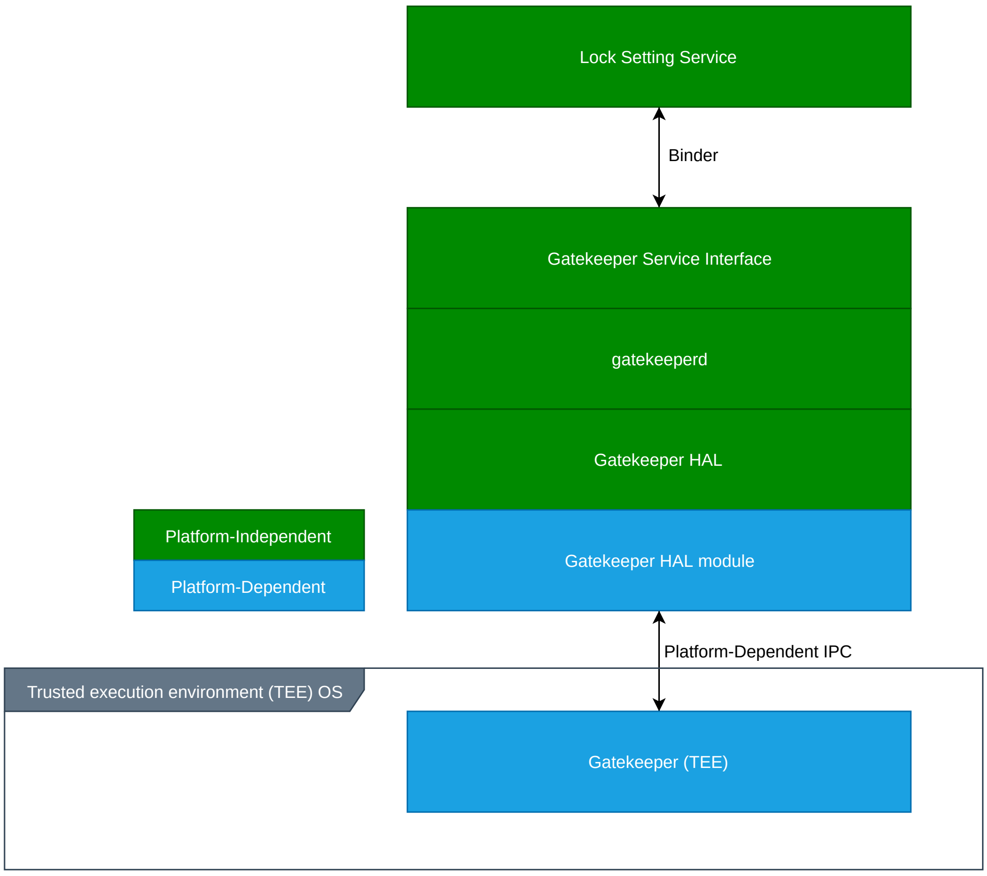


Fig. 2.2: Gatekeeper communication diagram [7]

HAL implementation must be able to enroll and verify a password, pattern, or PIN. Every HAL implementation must fulfill an enroll and verify functions from the gatekeeper header file. Gatekeeper TEE must fulfill the system gatekeeper header. Those are conditions for gatekeeper implementation. Based on these conditions, a device manufacturer or system administrator can alter other parts of the gatekeeper system and can use any TEE OS to implement Gatekeeper as long as the TEE has access to a hardware-backed key and a secure monotonic clock that ticks in suspend. In most cases, phones use The Trusty operating system, which is an open-source implementation of TEE by Google. TEE Gatekeeper, Keymaster, and other TEE components use a shared secret key that they share via the internal IPC system. Sharing the secret key is not dangerous, so Gatekeeper does not save or cache this key and requests it every time from Keymaster via IPC. Key is used to derive an HMAC key to enroll and verify passwords. This derived key is kept solely in Gatekeeper [7].

2.2.2 Request throttling

The request-throttling feature provides additional security measurements against brute-force attacks. Without a request-throttling, an attacker would be able to crack user passwords/PIN if the password/PIN is not complicated enough. Gatekeeper HAL implementation can return timeout in milliseconds. The timeout informs the client not to call Gatekeeper again until after the timeout has elapsed and refuses to serve any incoming request in the timeout period. Before every password verification or enroll, Gatekeeper writes a failure counter. If the password verification succeeds, the Gatekeeper clears the failure counter. The procedure mentioned above prevents attacks that prevent throttling by disabling the fastened MMC (eMMC) after issuing a verify or enroll call. Failure counter is written to secure storage on devices supporting secure storage. If the device does not support file-based encryption or secure storage is too slow, implementation can use Replay Protected Memory Block (RPMB) [7].

2.3 Biometrics

Biometrics allows authenticating users securely in Android OS. The most used biometrics are fingerprint and face detection. Any other type of biometric can be added to biometric authentications in case it meets security specifications and have a deficient rating of false positives. Imposter Accept Rate (IAR), Spoof Accept Rate (SAR), and False Accept Rate (FAR) metrics are measured to determine if biometric meet the requirements [8].

FAR metric defines how often a model mistakenly accepts a randomly chosen incorrect input. Before Android 8.1, that is API 27, FAR was only known and use matric to measure biometric modalities security. Even though it is the most used metric, let alone, it does not provide adequate information to assess how well the model stands up to targeted attacks [9].

Android 8.1 introduces IAR and SAR metrics that aim to improve biometric security. IAR metric is the chance that a biometric model accepts input that is meant to mimic a known good sample. SAR metric is the chance that a biometric model accepts a previously recorded, known good sample. Android defines three base levels of biometric sensor security: strong, weak, and convenience. By default, every sensor is classified as a convenience. Additional requirements need to be fulfilled by the sensor to be classified as weak or strong. These additional requirements are a combination of the three accept rates - FAR, IAR, and SAR [9].

Android open-source project releases compatibility definition with every android API version. Compatibility definition defines requirements that the device must meet in order for devices to be compatible with a specific Android API version. One of the requirements of compatibility definition is for biometric sensors. Table 2.1 provide accept rates for Android 10.

Classification	FAR	IAR	SAR
Convenience	-	-	-
Weak	0.002%	20%	20%
Strong	0.002%	7%	7%

Tab. 2.1: Biometric accept rates [10]

Additional requirements are needed from the sensor to be assigned to a weak or strong category. A weak sensor must have a hardware-backed Keystore implementation and perform all biometric authentication outside Android kernel such as TEE. A strong sensor must have all features of a weak sensor, and additionally must challenge the user for the recommended primary authentication PIN, pattern, or password once every 72 hours or less [10].

2.3.1 Biometric architecture

As the Android platform evolves, security measurements evolve with it, Fingerprint-Manager, BiometricPrompt, BiometricManager are the result of android evolution. FingerprintManager is oldest and deprecated since Android P. His successor is BiometricPrompt that is available on Android P and higher. BiometricManager has been introduced in Android Q and offers a method to check which biometric methods are available to the user. How different biometric classes interact with the Android system is described in diagram 2.3.

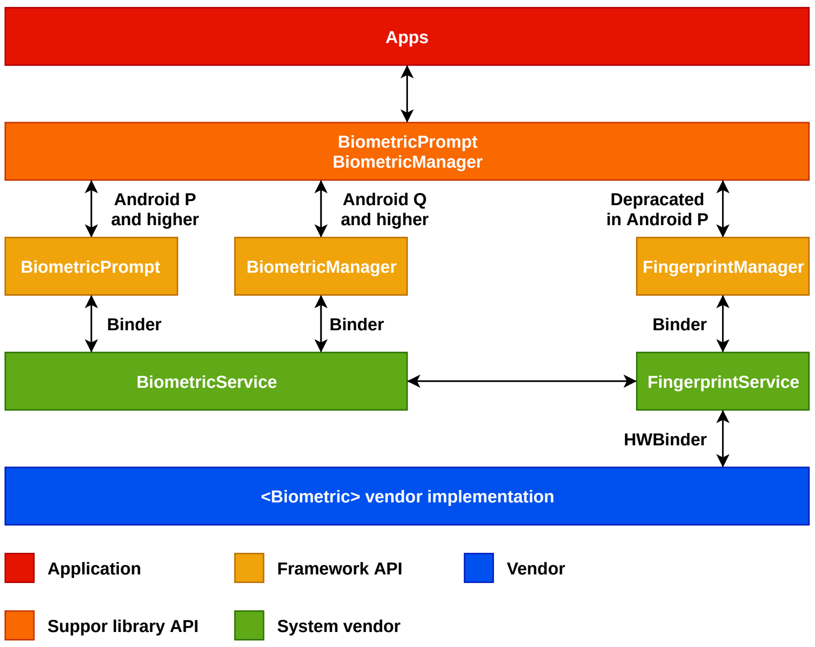


Fig. 2.3: Biometric architecture [10]

FingeprintManager only accepts fingerprints as biometric authentication. As a new biometric methods rose, this was an obstacle to good user experience. Instead of creating a new manager for every biometric, FingeprintManager was marked as deprecated. BiometricPrompt has been introduced to the community as the successor to FingeprintManager as a single entry point for biometric authentication. BiometricPrompt uses a default setting that the user can set in phone settings in the security section.

Android provides interfaces and header files to implement biometric methods. The concrete implementation is the device manufacturer's role. To guarantee that users and developers have a seamless biometric experience, device manufacturers have to integrate biometric stack with BiometricPrompt. Any biometric method that is about to be integrated with BiometricPrompt must meet the strength requirements defined by CDD [8].

2.4 Keystore

Keystore API provides storage for cryptographic keys and certificates. To provide hardware-backed cryptography, Keystore leverages TEE Keymaster implementation[11].

Before Android 6.0, Keystore API had a simple hardware-backed API for signing and verification operations. In Android 6.0, Keymaster features were extended to implement a more comprehensive array of capacities provided by the Keystore. Keystore features added in Android 6.0 are:

- symmetric cryptographic primitives, AES, HMAC,
- access control system for hardware-backed keys,
- a usage control scheme to allow key usage to be limited, to mitigate the risk of security compromise due to misuse of keys,
- an access control scheme to enable restriction of keys to specified users, clients, and a defined time range.

Android 7.0 introduced Keymaster 2, which adds support for key attestation and version binding [12]. In Keymaster 1, apps or remote servers cannot reliably verify if keys are known to be in hardware-backed storage. To mitigate this, Keymaster 2 introduced key attestation. Key attestation provides a way to securely decide if an asymmetric key pair is hardware-backed, what constraints are applied to its usage, and what the properties of the key are [13].

Version binding binds keys to the operating system and patch level version. Version binding ensures that an attacker who discovers a weakness in an old version of the system or TEE software cannot roll a device back to the vulnerable version and use keys created with the newer version. Also, when a key with a given version and patch level is used on a device that has been upgraded to a newer version or patch level, the key is upgraded before it can be used. The previous version of the key is invalidated [12].

Android 8.0 introduced Keymaster 3, which extends Keymaster 2's attestation feature to support ID attestation. ID attestation is optional and provides the possibility to bind keys to the device hardware such as phone ID (IMEI / MEID), device serial number, or a product name. Also, Keymaster 3 transitioned from old-style C-structure HAL to C++ HAL interface generated from a new Hardware Interface Definition Language (HIDL) [12].

Android 9.0 introduced Keymaster 4, which adds support for embedded Secure Elements (SE), secure key import, 3DES encryption, changes to version binding, so it allows independent version updates for boot.img and system.img [12].

2.5 Supported cryptographic primitives

Cryptographic primitives are well-established, low-level cryptographic algorithms [14]. Keystore supports various categories of cryptographic primitives:

- Hash function
- Symmetric key cryptography

- Asymmetric key cryptography

Cryptographic primitives are often used to build cryptographic protocols. Keystore leverages cryptographic primitives to provide feature-rich cryptographic operations, which includes but are not limited to:

- Key generation
- Import and export of asymmetric keys
- Import of raw symmetric keys
- Asymmetric encryption and decryption with appropriate padding modes
- Digital signature and verification
- Symmetric encryption and decryption in appropriate modes, including an AEAD mode
- Generation and verification of symmetric message authentication codes
- Random number generation

The key purpose, *padding*, *access control constraints*, or any other protocol element, is defined on a key generation or import, and it is permanently bound to the key. The protocol elements bound to the key ensures the key cannot be used in any other way. Random number generation is not exposed to the public API, and it is used internally for the generation of keys, initialization vectors, random padding, and other elements of secure protocols that require randomness. Keystore can be utilized as a provider and used with supported algorithms, which are:

- Cipher
- KeyGenerator
- KeyFactory
- KeyPairGenerator
- Mac
- Signature
- SecretKeyFactory

2.5.1 Key generation

To generate a key, KeyGenerator or KeyPairGenerator class can be used. KeyGenerator provides the functionality of the symmetric key generator. KeyPairGenerator provides the functionality of the asymmetric key generator.

KeyGenerator

There are two ways to generate a key with a KeyGenerator: in an algorithm-independent manner, and an algorithm-specific manner. The difference between them is generator initialization. In listing, 2.1 are listed all initialization methods of KeyGenerator. Init methods that do not use AlgorithmParameterSpec are an algorithm-independent. AlgorithmParameterSpec init method is used in situations where a set of algorithm-specific parameters already exists. In case the user does not use any of the provided init methods, the provider specified at creation must supply a default initialization.

```kotlin
// Algorithm-Independent Initialization
fun init(random: SecureRandom)
fun init(keysize: Int)
fun init(keysize: Int, random: SecureRandom)
// Algorithm-Specific Initialization
fun init(params: AlgorithmParameterSpec)
fun init(
   params: AlgorithmParameterSpec,
   random: SecureRandom
)
```

Listing 2.1: KeyGenerator init methods

Supported algorithms of KeyGenerator are listed in table 2.2.

Algorithm	Supported API Levels	Notes
AES	23+	Supported sizes: 128, 192, 256
HmacSHA1	23+	Supported sizes: 8–1024 (inclusive), must be multiple of 8 Default size: 160
HmacSHA224	23+	Supported sizes: 8–1024 (inclusive), must be multiple of 8 Default size: 224
HmacSHA256	23+	Supported sizes: 8–1024 (inclusive), must be multiple of 8 Default size: 256
HmacSHA384	23+	Supported sizes: 8–1024 (inclusive), must be multiple of 8 Default size: 384
HmacSHA512	23+	Supported sizes: 8–1024 (inclusive), must be multiple of 8 Default size: 512

Tab. 2.2: Supported KeyGenerator algorithms with AndroidKeyStore provider

Example of key generation with KeyGenerator

Listing 2.2 shows how to generate AES symmetric key in Galois/Counter Mode (GCM), which's purpose is encryption and decryption with no encryption padding. AndroidKeyStore is defined as the KeyGenerator provider, so the creation of the key occurs in the hardware-backed Keystore. Other constraints can be applied to the KeyGenParameterSpec builder. SetUserAuthenticationRequired and SetUser-AuthenticationValidityDurationSeconds can be applied to the builder, to condition the key retrieval to the time window starting from the last unlock of the phone or user can be prompted directly in application to authorize via LockScreen.

```kotlin
val keyGenerator = KeyGenerator
    .getInstance("AES", "AndroidKeyStore")
val keyGenParameterSpec =
    KeyGenParameterSpec.Builder(
        "AES_KEY_ALIAS",
        KeyProperties.PURPOSE_ENCRYPT
        or
        KeyProperties.PURPOSE_DECRYPT
    )
        .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
        .setEncryptionPaddings(
            KeyProperties.ENCRYPTION_PADDING_NONE
        )
        .setKeySize(256)
        .build()
keyGenerator.init(keyGenParameterSpec)
val secretKey = keyGenerator.generateKey()
```

Listing 2.2: AES Key generation

The key can be retrieved from AndroidKeyStore 2.3. AndroidKeyStore is initiated via a static function in the Keystore object, whereas type is passed AndroidKeyStore after the creation load() method is called on object to initialize KeyStore. Load method loads Keystore using the given LoadStoreParameter, which can be null. After KeyStore is initialized key can be retrieved by calling getKey() method. Key alias passed to getKey() must be the same as an alias during the key creation.

```kotlin
val keystore = KeyStore.getInstance("AndroidKeyStore")
                    .apply { load(null) }
val secretKey = keystore.getKey("AES_KEY_ALIAS", null)
```

Listing 2.3: Retrive AES key from keystore

KeyPairGenerator

As in KeyGenerator, there are two ways to generate a key pair: in an algorithm-independent manner, or an algorithm-specific manner. The difference between them is explained in section 2.5.1. Supported algorithms are listed in the Tab 2.3.

Algorithm	Supported API Levels	Notes
DSA	19-22	
EC	23+	Supported sizes: 224, 256, 384, 521 Supported named curves: P-224 (secp224r1), P-521 (aka secp521r1). P-256 (aka secp256r1 and prime256v1), P-384 (aka secp384r1),
RSA	18+	Supported sizes: 512, 768, 1024, 2048, 3072, 4096 Supported public exponents: 3, 65537 Default public exponent: 65537

Tab. 2.3: Supported KeyPairGenerator algorithms with AndroidKeyStore provider

Example of key pair generation with KeyPairGenerator

Listing 2.4 shows how to generate an EC key pair whose purpose is encryption and decryption. Key can be used only by an authenticated user in time window of 5 minutes from the last successful authentication. AndroidKeyStore is defined as the KeyPairGenerator provider, so the creation of the key pair occurs in the hardware backed Keystore.

```
val kpg = KeyPairGenerator.getInstance(
    KeyProperties.KEY_ALGORITHM_EC,
    "AndroidKeyStore"
)
val parameterSpec: KeyGenParameterSpec =
 KeyGenParameterSpec.Builder(
    "EC_KEY",
    KeyProperties.PURPOSE_ENCRYPT
    or
    KeyProperties.PURPOSE_DECRYPT
    )
    .setUserAuthenticationRequired(true)
    .setUserAuthenticationValidityDurationSeconds(300)
    .build()
kpg.initialize(parameterSpec)
val kp = kpg.generateKeyPair()
```

Listing 2.4: EC Key pair generation

2.5.2 Import and export of asymmetric keys

Keystore supports the import of public and private key pairs in DER-encoded PKCS8 format, without password-based encryption. Export is only supported for public keys in X.509 format. Two different tags are used for origin to distinguish imported keys from securely generated keys. Imported keys use tag *imported*, and securely generated keys use tag *generated*.

Example of RSA private key import

PrivateKey instance and X.509 certificate for the public key corresponding to the private key represented as an X509Certificate instance are needed to import a private key into KeyStore. KeyStore abstraction does not support storing private keys without a certificate. Listing 2.5 shows how to generate RSA private key in DER format and X.509 certificate for public key.

```
openssl genrsa -out private_key.pem 2048
openssl pkcs8 -topk8 -inform PEM -outform DER
                -in private_key.pem
                -out private_key.der
                -nocrypt
openssl req -new -x509 -key private_key.pem
         -out publickey.cer
         -days 365
```

Listing 2.5: RSA key and certificate generation

For demonstration purposes, key and certificate files are directly imported to the raw resources of the application. Listing 2.6 shows how to convert DER encoded private key with X.509 public-key certificate into the PrivateKey and Certificate instances, which are used to import private-key into KeyStore. Even though it is possible to import externally generated keys into Keystore, it is not recommended to do so. Private-key is exposed to the main memory and, therefore, can be abused by an attacker.

```
val privByteArray =
    resources.openRawResource(R.raw.private_key)
        .readBytes()
val spec =
    PKCS8EncodedKeySpec(privByteArray)
val kf = KeyFactory.getInstance("RSA")
val privKey =
    kf.generatePrivate(spec) as RSAPrivateKey
val publicInputStream =
    resources.openRawResource(R.raw.publickey)
val cert = CertificateFactory
    .getInstance("X.509")
    .generateCertificate(
    publicInputStream
)
val ks = KeyStore.getInstance("AndroidKeyStore")
    .apply { load(null) }
ks.setKeyEntry(
    "MyImportedRsaKey",
    privKey,
    null,
    arrayOf(cert))
val privateKey =
    ks.getKey("MyImportedRsaKey", null)
```

Listing 2.6: Import of RSA private key

Example of public key export

The export of the public key is straightforward. Listing 2.7 shows how to get private entry from Keystore and its certificate. Certificate can be converted to byte array or base64 string and sent to the recipient. Note that private key entry contains a private key field, which holds a reference to the private key. No sensitive pieces of information that could let to abuse of the key are not presented in private key entry.

```
val entry = kS.getEntry("EC_KEY", null)
     as KeyStore.PrivateKeyEntry
val certificate = entry.certificate as X509Certificate

val base64cert = Base64.encodeToString(
    certificate.encoded, Base64.NO_WRAP
)
val base64PubKey = Base64.encodeToString(
    certificate.publicKey.encoded, Base64.NO_WRAP
)
```

Listing 2.7: Export of EC public key

2.5.3 Encryption and decryption using an asymmetric key

RSA in different modes and padding settings is the only asymmetric algorithm that can be used on Android to encrypt and decrypt data safely. At the time of writing this thesis, no other asymmetric algorithm is supported. In table 2.4 are listed all combinations of encryption modes and paddings. Additionally, all combinations support all RSA key sizes that can KeyPairGenerator generate, (512, 768, 1024, 2048, 3072, 4096) bits. Cipher class is used to encrypt and decrypt data. Listing 2.8 shows how to generate RSA key for RSA/ECB/PKCS1Padding transformation used in *Cipher*.

```
val kpg = KeyPairGenerator.getInstance(
    KeyProperties.KEY_ALGORITHM_RSA,
    "AndroidKeyStore"
)
val parameterSpec: KeyGenParameterSpec =
    KeyGenParameterSpec.Builder(
        "RSA_KEY",
        KeyProperties.PURPOSE_ENCRYPT
                or
                KeyProperties.PURPOSE_DECRYPT
    )
        .setBlockModes(KeyProperties.BLOCK_MODE_ECB)
        .setEncryptionPaddings(
            KeyProperties.ENCRYPTION_PADDING_RSA_PKCS1
         )
        .build()
kpg.initialize(parameterSpec)
val kp = kpg.generateKeyPair()

val kS = KeyStore.getInstance("AndroidKeyStore")
    .apply { load(null) }

val entry = kS.getEntry("RSA_KEY", null)
        as KeyStore.PrivateKeyEntry
```

Listing 2.8: Generate RSA Key for encryption and decryption

Algorithm	Supported (API Levels)
RSA/ECB/NoPadding	18+
RSA/ECB/PKCS1Padding	18+
RSA/ECB/OAEPWithSHA-1AndMGF1Padding	23+
RSA/ECB/OAEPWithSHA-224AndMGF1Padding	23+
RSA/ECB/OAEPWithSHA-256AndMGF1Padding	23+
RSA/ECB/OAEPWithSHA-384AndMGF1Padding	23+
RSA/ECB/OAEPWithSHA-512AndMGF1Padding	23+
RSA/ECB/OAEPPadding	23+

Tab. 2.4: Supported RSA variants for encryption and decryption

Example of RSA data encryption and decryption

Listing 2.9 shows how to encrypt data with the RSA key in ECB mode with PKCS1 padding. Cipher is initialized with the transformation RSA/ECB/PKCS1Padding, which corresponds to the RSA key. The mode is set to encryption, and the encryption key is the public key of RSA. Data input is in a byte array format. To encrypt data, doFinal(1) method is called. The result is a byte array of encrypted data.

```kotlin
val dataToEncrypt = "VUT THESIS".toByteArray()

val encryptedData: ByteArray =
    Cipher.getInstance("RSA/ECB/PKCS1Padding")
        .run {
            init(
                  Cipher.ENCRYPT_MODE,
                  entry.certificate.publicKey
               )
            doFinal(dataToEncrypt)
        }
```

Listing 2.9: Enrypt data with RSA

```kotlin
val data =
    Cipher.getInstance("RSA/ECB/PKCS1Padding")
        .run {
            init(Cipher.DECRYPT_MODE, entry.privateKey)
            doFinal(encryptedData)
        }.let { String(it) }
```

Listing 2.10: Decrypt data with RSA

The private key of RSA is used to decrypt the data. Listing 2.10 shows the decryption approach. Cipher instance is initialized with the same transformation as which data were encrypted. The mode is set to decryption, and the decryption key is the private key of RSA. Method doFinal(1) is called with passed cryptogram as a parameter. The result is a byte array that can be converted to the string format.

2.5.4 Digital signature and verification of signature

RSA, EC, DSA can be used in different modes and padding settings. Table 2.5 shows all different configurations that can be used for signature and verification. Listing 2.11 shows how to generate an elliptic curve key pair with SHA512 digest for signing and verification.

```kotlin
val kpg = KeyPairGenerator.getInstance(
    KeyProperties.KEY_ALGORITHM_EC,
    "AndroidKeyStore"
)
val parameterSpec: KeyGenParameterSpec =
    KeyGenParameterSpec.Builder(
        "EC_KEY",
         KeyProperties.PURPOSE_SIGN
         or
         KeyProperties.PURPOSE_VERIFY
    )
        .setDigests(KeyProperties.DIGEST_SHA512)
        .build()
kpg.initialize(parameterSpec)
val kp = kpg.generateKeyPair()
val kS = KeyStore.getInstance("AndroidKeyStore")
    .apply { load(null) }
val entry = kS.getEntry("EC_KEY", null)
        as KeyStore.PrivateKeyEntry
```

Listing 2.11: Generate EC Key for sign and verify

Algorithm	Supported (API Levels)
MD5withRSA	18+
NONEwithECDSA	23+
NONEwithRSA	18+
SHA1withDSA	19-22
SHA1withECDSA	19+
SHA1withRSA	18+
SHA1withRSA/PSS	23+
SHA224withDSA	20-22
SHA224withECDSA	20+
SHA224withRSA	20+
SHA224withRSA/PSS	23+
SHA256withDSA	19-22
SHA256withECDSA	19+
SHA256withRSA	18+
SHA256withRSA/PSS	23+
SHA384withDSA	19-22
SHA384withECDSA	19+
SHA384withRSA	18+
SHA384withRSA/PSS	23+
SHA512withDSA	19-22
SHA512withECDSA	19+
SHA512withRSA	18+
SHA512withRSA/PSS	23+

Tab. 2.5: Supported algorithms for signing and verification

Example of ECDSA data signing and verification

Listing 2.12 shows how to sign data with the EC key with SHA512 digest. Signature is initialized with the transformation SHA512withECDSA. The signature key is the private key of EC. Data that should be signed are passed to the method update(1). The sign method is called to sign data. The result is a byte array.

```
val dataToSign = "VUT THESIS".toByteArray()

val signature: ByteArray =
    Signature.getInstance("SHA512withECDSA")
        .run {
            initSign(entry.privateKey)
            update(dataToSign)
            sign()
        }
```

Listing 2.12: Sign data with ECDSA

The certificate of the public key of EC is used to verify the signature. Listing 2.13 shows how to verify a signature. A signature instance is initialized with the same transformation used to sign the data. Method verify(1) is called to verify the signature passed to the method. Data which signature belong to are passed to the method update(1). The result is a boolean that indicates if the signature is valid.

```kotlin
val valid: Boolean =
    Signature.getInstance("SHA512withECDSA")
        .run {
            initVerify(entry.certificate)
            update(dataToSign)
            verify(signature)
        }
```

Listing 2.13: Verify data with ECDSA

2.5.5 Import of raw symmetric keys

The import process of symmetric keys is much more straightforward than the import process of asymmetric keys. The symmetric key is wrapped into SecretKeyEntry and imported directly to the KeyStore. Listing 2.14 shows an example of how to import AES key with additional key properties definition.

```kotlin
fun importAESKey(byteArr: ByteArray) {
    val spec = SecretKeySpec(
        byteArr, 0,
        byteArr.size, "AES")
    val kS = KeyStore.getInstance("AndroidKeyStore")
        .apply { load(null) }
    kS.setEntry(
        "Imported_AES",
        KeyStore.SecretKeyEntry(spec),
        KeyProtection.Builder(
            KeyProperties.PURPOSE_ENCRYPT
                    or
                    KeyProperties.PURPOSE_DECRYPT)
            .setBlockModes(
                KeyProperties.BLOCK_MODE_GCM
            )
            .setEncryptionPaddings(
                KeyProperties.ENCRYPTION_PADDING_NONE
            )
            .build()
    )
}
```

Listing 2.14: Import of AES key

2.5.6 Encryption and decryption using an symetric key

AES in different modes and padding settings is the only symmetric algorithm that can be used on android to encrypt and decrypt data safely. At the time of writing this thesis, no other symmetric algorithm is supported. In table 2.4 are listed all combinations of encryption modes and paddings. Additionally, all combinations support all AES key sizes that can KeyGenerator generate, (128, 192, 256) bits. Cipher class is used to encrypt and decrypt data. Listing 2.2 shows how to generate AES key for AES/GCM/NoPadding transformation used in Cipher.

Listing 2.15 shows how to encrypt data with the AES key in GCM mode with no padding. Cipher is initialized with the transformation AES/GRCM/NoPadding, which corresponds to the AES key purpose. The mode is set to encryption. Data input is in a byte array format. To encrypt data, doFinal(1) method is called. The result is a byte array of encrypted data. To be able to decrypt cryptogram, the initialization vector of Cipher must be saved for later use.

```kotlin
val dataToEncrypt = "VUT THESIS".toByteArray()
val cipher = Cipher.getInstance("AES/GCM/NoPadding")
val encryptedData: ByteArray =
    cipher
        .run {
            init(
                    Cipher.ENCRYPT_MODE,
                    entry.secretKey
                )
            doFinal(dataToEncrypt)
        }
val vector = cipher.iv
```

Listing 2.15: Enrypt data with AES

Listing 2.16 shows the decryption process. Cipher instance is initialized with the same transformation as which data were encrypted. The mode is set to decryption, and the GCMParameterSpec is initialized with the initialization vector and authorization tag. Method doFinal(1) is called with passed cryptogram as a parameter. The result is a byte array that can be converted to the string format.

```kotlin
val spec = GCMParameterSpec(128, vector)
val data =
    Cipher.getInstance("AES/GCM/NoPadding")
        .run {
            init(
                    Cipher.DECRYPT_MODE,
                    entry.secretKey, spec
                )
            doFinal(encryptedData)
        }.let { String(it) }
```

Listing 2.16: Decrypt data with AES

2.6 Key access control

Hardware-based keys let alone are not secure enough, if an attacker could use them at will. Access control was introduced to add another security layer on to the hardware-based keys, and the Keystore must enforce access controls. Authorization list of tag/value pairs is a definition of access controls. Authorization tags are 32-bit integers, and the values are a variety of types. Authorization tags are defined at the key creation, and any attempt to modify tags after creation results in key deprecation so that any cryptographic operation will fail. Some tags can be defined multiple times. If and when can tag be used multiple times is defined in concrete tag definition. After the user-defined authorization tag, the key master adds additional tags, such as whether the key has rollback protection and encodes the final list to the returned key blob.

Some of the authorization tags are KeyPurpose tags to define the purpose of the created key: encrypt, decrypt, verify, e.t.c. The expiration date of the key, key size, user authentication required, and many more can be defined at the creation to define access-list.

2.7 Hardware vs. software enforcement

Secure hardware applications vary in implementations, and not all support the same set of features. To support this variety of approaches, Keymaster distinguishes between secure and non-secure world access control enforcement, or hardware and software enforcement, respectively. Even though not all secure hardware implementations are the same. The base set of features supported on all implementations are:

- Enforce the exact matching of all authorizations. Authorization lists in key blobs exactly match the authorizations returned during key generation, including ordering. Any mismatch causes an error diagnostic.
- Declare the authorizations whose semantic values are enforced.

The API mechanism for declaring hardware-enforced authorizations divides the authorization list into two sub-lists, hardware-enforced and software-enforced. Based on what the secure hardware can enforce, it places the appropriate values in each sub-list.

2.8 Trusty TEE

Trusted Execution Environment is provided by Trusty, which is a secure Operation System. TEE runs parallel to the Android OS, and it is on the same processor as the Android OS, but it is isolated from the Android OS by hardware and software. TEE can use the full power of the primary processor and memory. TEE isolation protects it from malicious applications that could be installed by the user. The Trusty use Trustzone on an ARM processor and Intel's Virtualization Technology on x86 platform.

Part II

Results

3 Cryptographic algorithms comparison

This chapter utilizes information from chapter 2 to create a benchmark application for cryptographic algorithms. The application contains 280 benchmark tests that measure the run time of cryptographic algorithms on 16 different devices listed in Tab 3.1. This chapter presents the processed results from the benchmark application.

Manufacturer	Model	Chipset	CPU	RAM [GB]	Android
Google	Pixel 3a	Qualcomm SDM670 Snapdragon 670 (10 nm)	Octa-core (2x2.0 GHz 360 Gold & 6x1.7 GHz Kryo 360 Silver)	4	10
Google	Pixel XL	Qualcomm MSM8996 Snapdragon 821 (14 nm)	Quad-core (2x2.15 GHz Kryo & 2x1.6 GHz Kryo)	4	10
LG	G6	Qualcomm MSM8996 Snapdragon 821 (14 nm)	Quad-core (2x2.35 GHz Kryo & 2x1.6 GHz Kryo)	4	9
LG	Nexus 5X	Qualcomm MSM8992 Snapdragon 808 (20 nm)	Hexa-core (4x1.4 GHz Cortex-A53 & 2x1.8 GHz Cortex-A57)	2	8.1
Samsung	Galaxy A5	Exynos 7880 (14 nm)	Octa-core 1.9 GHz Cortex-A53	3	8
Asus	Zenfone 3 max	Mediatek MT6737M (28 nm)	Quad-core 1.25 GHz Cortex-A53	2	7
HTC	One M9	Qualcomm MSM8994 Snapdragon 810 (20 nm)	Octa-core (4x1.5 GHz Cortex-A53 & 4x2.0 GHz Cortex-A57)	3	7
Huawei	P9 lite	HiSilicon Kirin 650 (16 nm)	Octa-core (4x2.0 GHz Cortex-A53 & 4x1.7 GHz Cortex-A53)	3	7
Samsung	Galaxy S6	Exynos 7420 Octa (14 nm)	Octa-core (4x2.1 GHz Cortex-A57 & 4x1.5 GHz Cortex-A53)	3	7
Motorola	Moto G5 Plus	Qualcomm MSM8953 Snapdragon 625 (14 nm)	Octa-core 2.0 GHz Cortex-A53	4	8.1
Samsung	Note10+	Exynos 9825 (7 nm)	Octa-core (2x2.73 GHz Mongoose M4 & 2x2.4 GHz Cortex-A75 & 4x1.9 GHz Cortex-A55)	12	9
OnePlus	7 Pro	Qualcomm SDM855 Snapdragon 855+ (7 nm)	Octa-core (1x2.96 GHz Kryo 485 & 3x2.42 GHz Kryo 485 & 4x1.78 GHz Kryo 485)	8	10
Huawei	P20lite	HiSilicon Kirin 659 (16 nm)	Octa-core (4x2.36 GHz Cortex-A53 & 4x1.7 GHz Cortex-A53)	4	9
OnePlus	6	Qualcomm SDM845 Snapdragon 845 (10 nm)	Octa-core (4x2.8 GHz Kryo 385 Gold & 4x1.7 GHz Kryo 385 Silver)	8	10
Samsung	Galaxy S9+	Exynos 9810 (10 nm)	Octa-core (4x2.7 GHz Mongoose M3 & 4x1.8 GHz Cortex-A55)	6	9
Samsung	S10e	Exynos 9820 (8 nm)	Octa-core (2x2.73 GHz Mongoose M4 & 2x2.31 GHz Cortex-A75 & 4x1.95 GHz Cortex-A55)	8	9

Tab. 3.1: Devices used for benchmarking

3.1 Creation of asymmetric key

Asymmetric key creation benchmark measures run-time of the creation of an asymmetric key. Measured key types are RSA and EC in different key size variations.

With increasing key size, the computation complexity of the algorithm increase as well. So with the greater key size, longer run times are expected than with smaller key sizes. The heat map A.1 summarizes the results for asymmetric key creation. The results, in most cases, are coherent with the assumption that with increased key size, run time also increases. RSA algorithm results approve this assumption across all devices. With the EC algorithm, if we compare run times of EC224 and EC256 across devices, results show that on six devices, greater key size resulted in shorter run times, which is the opposite of assumption. A small difference between the key size used in algorithms could cause this. Another assumption is that the EC key generation is faster than the RSA key generation due to EC keys can be a lot smaller than RSA keys and still have the same level of security. Results show that overall run times of EC are a lot lower than run times of RSA, which confirms the original assumption.

3.2 Encryption using an asymmetric key

Encryption using asymmetric key is currently supported only for RSA, as mentioned in Section 2.5.3. RSA for encryption can be used in eight different variations. The difference between them is in used padding mode. Android Keystore and also general java Keystore do not implement ECB mode for RSA, so encryption/decryption can be used only on data that are smaller than key size. Interestingly encryption modes have in their name ECB even though it is not implemented. Heat maps A.3 and A.2 indicate that the most consistent run time across devices is achieved by RSA/ECB/PKCS1Padding variation. PKCS1Padding adds the least overhead of all supported padding schemes (at least 11 bytes). OAEP padding adds even more overhead. OAEP padding scheme needs two hash functions with different properties to operate. One hash function should map arbitrary sized input to fixed-size output. Another hash function map arbitrary sized input to arbitrarily sized output. Such a hash function is called "mask generation function" (MFG). OAEP adds at least 42 bytes, which is 31 bytes more than minimum on PKCS1 Padding. Results confirm assumptions, and a general run time of schemes using OAEP padding is longer than in the case of PKCS1 schemes. Samsung Galaxy S6 on all encryption schemes shows significantly slower encryption with used key size 4096bits. Huawei P9 Lite and Asus Zenphone 3 MAX, when used with OAEP padding, results in slower run times than other devices. From a security standpoint, it is recommended to use OAEP padding scheme. Result comparison of algorithms using the OAEP padding scheme shows the best value brings RSA3072/OAEPWithSha-512AndMGF1Padding.

3.3 Decryption using an asymmetric key

Same as in the case of asymmetric key encryption, RSA is the only supported algorithm. The general assumption is that decryption should be slower. Encryption has the advantage that the public exponent is generally relatively small. The private exponent used for decryption is larger, so the decryption of data is slower operation. Results in heat maps A.5 and A.4 summarizes results and confirm the assumption that the data decryption is in the most implementations of RSA is slower than encryption. The same algorithm achieves overall best run time as in case of encryption that is RSA/ECB/PKCS1Padding.

3.4 Digital signature

Unlike asymmetric key encryption and decryption, digital signature is supported by RSA and EC. Benchmark tests are divided into categories by hash function that the

RSA or EC utilizes. Comparing heat maps A.6, A.7 shows that the MD5 hash function adds almost none overhead to the signature algorithm compared to the signature algorithm without hash function. Signature algorithms using SHA hash function, listed in A.8, A.9, A.10,A.11, A.12, have longer run time than signature algorithms using MD5 or none hash function. The run time difference between signature algorithms using the SHA hash function is minimal. Result comparison of algorithms using the SHA hash function shows the best value brings SHA512withRSA2048/PSS.

3.5 Verification of digital signature

Benchmark tests are divided into the same categories as in digital signature. Results are listed in heat maps A.13, A.14, A.15, A.16, A.17, A.18 ,A.19. The difference in run time between verification algorithms using the SHA hash function is the same as in digital signature minimal. Overall, RSA has a lover run time than EC unlike digital signature where EC has the lower run time.

3.6 Creation of symmetric key

Symmetric key creation benchmark measures run-time of the creation of the symmetric key. The measured key type is AES in different key sizes. With increasing key size, the computation complexity of the algorithm increase as well. So with the greater key size, longer run times are expected than with smaller key sizes. The bar chart A.20 summarizes the results for symmetric key creation. Run time difference on devices Google Pixel 3A, Huawei P20 Lite, LG Nexus 5X, Google Pixel XL are expected. On other devices, run times are equal, or run time is not increasing with increased key size. Hardware optimizations can cause this, or it can be a statistical error.

3.7 Encryption using an symmetric key

AES is the only algorithm supported for symmetric encryption in different key sizes and variants. Results in heat map A.21 show that run time stays consistent on the device across all variants and key sizes. Compared to the RSA encryption run times, AES encryption run times are slower in most cases. This can be caused by small data size used to benchmark encryption (smaller than RSA key size). Based on results and security standpoint AES256/GCM/NoPadding brings the best value.

3.8 Decryption using an symmetric key

Same as in the case of symmetric key encryption, AES is the only supported algorithm. The assumption is that decryption and encryption should be roughly equal due to the same key used for both operations. Compared to the RSA decryption, AES decryption should be faster. Results in heat map A.22 show overall slower run times than symmetric encryption run times, which do not confirm the assumption of equal run times. Compared to the RSA decryption run time, AES decryption run time is faster, and this confirms the assumption.

4 Hands-on implementation

The hands-on implementation chapter leverages the theory described in chapter 2 and results from chapter 3, to showcase modern security features that can be used to implement a secure Android application. To properly showcase how these security features can be used, the SecNote system is implemented. The system consists of the Android application and cloud solution. How independent elements in the system communicate with each other is described in Section 4.1. After an overview of the system, the separate components are thoroughly described together with their security mechanisms.

4.1 System overview

The system's mission is to deliver user notes between the Android application and cloud solution securely. The system comprises the Android application, *Authentication service*, *Permission service*, and *Note service*. Communication diagram is shown on Fig 4.1. Diagram showcases where separate components belong and how

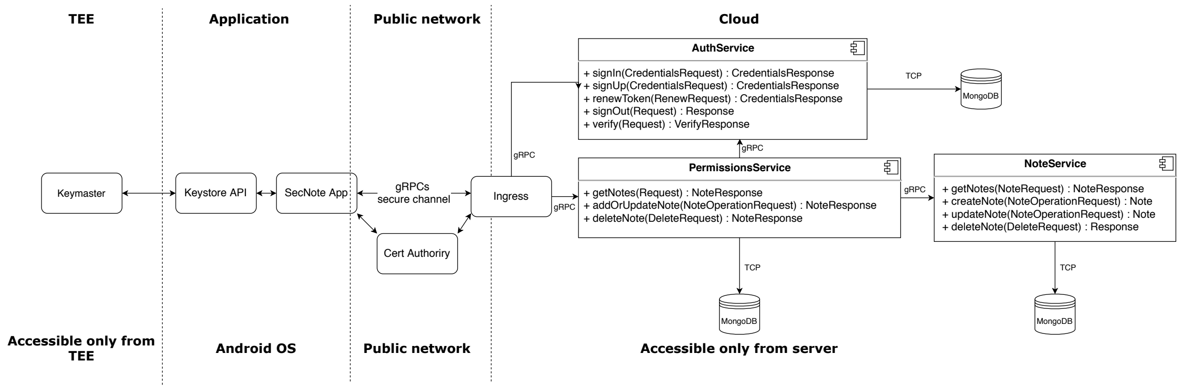

TEE
Application
Public network
Cloud
AuthService
+ signIn(CredentialsRequest) : CredentialsResponse
+ signUp(CredentialsRequest) : CredentialsResponse
+ renewToken(RenewRequest) : CredentialsResponse
+ signOut(Request) : Response
+ verify(Request) : VerifyResponse
TCP
MongoDB
gRPC
gRPC
PermissionsService
+ getNotes(Request) : NoteResponse
+ addOrUpdateNote(NoteOperationRequest) : NoteResponse
+ deleteNote(DeleteRequest) : NoteResponse
NoteService
+ getNotes(NoteRequest) : NoteResponse
+ createNote(NoteOperationRequest) : Note
+ updateNote(NoteOperationRequest) : Note
+ deleteNote(DeleteRequest) : Response
gRPC
Keymaster
Keystore API
SecNote App
gRPCs secure channel
Ingress
gRPC
Cert Authoriry
TCP
TCP
MongoDB
MongoDB
Accessible only from TEE
Android OS
Public network
Accessible only from server

Fig. 4.1: Secnote system overview

accessible they are. The cloud solution is based on microservice architecture. It allows the system to be rapidly developed and scaled as needed. To deploy and manage microservices on cloud effectively, *Kubernetes* with the *Istio* service mesh is used. The application communicates with the cloud gateway through the channel. Channel is secured with TLS, and it is terminated on ingress gateway. Due to the single cluster solution, for faster response times, it is easier to terminate TLS on the gateway and let communication inside the cluster be unencrypted. Communication between application and cloud and also between microservices itself is based on *gRPC*. *gRPC* is a modern RPC framework that uses HTTP/2 for transport and serializes data with the protocol buffers. Protocol buffers are Google's language-neutral mechanism for serializing structured data.

4.2 Service architecture

This section describes common architectural patterns between authentication, permission, and note services. All three services are written in go language as *gRPC* microservice. The file structure of the project is kept consistent, so navigation between different project source code is reasonably similar. The runnable portion of the code, the main, is located in the cmd folder. The main is exactly the same on all three services besides the provided service implementation. Fig 4.2 shows the code execution of the main. This process describes the startup of the service. To

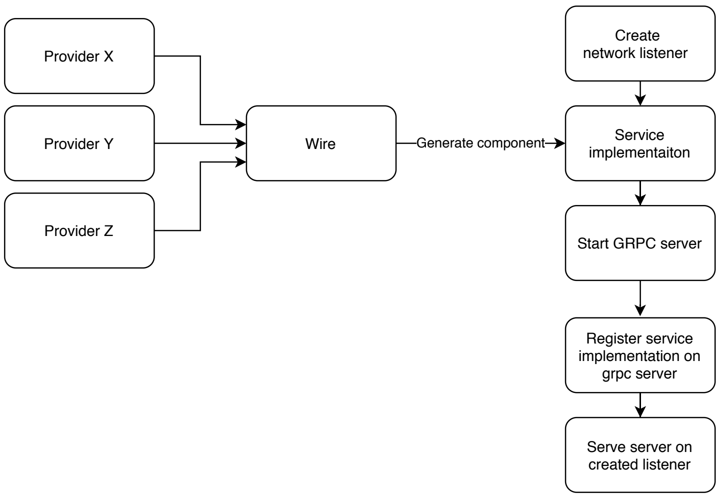


Fig. 4.2: Service startup

implement the *gRPC* server, proto file needs to be defined so the interface of the service can be generated and implemented. Proto files are located in the API folder. The internal folder encapsulates all code that is related to service and is not meant to be included in other services. Concrete implementation of methods that are generated from the proto file is located in the internal/service folder. Dependencies are provided through the Wire. The Wire is GO library used to manage dependency injection at compile time. All components that are intentioned to be provided have module file in which provide method is implemented. This method tells to Wire how to provide that particular dependency. After the service is implemented, the docker image is built. A docker deamon needs the definition of the image, which is provided with the Dockerfile. Dockerfile of the service is located in the build folder.

Features described in this section are common between all three services used in the SecNote system, so the next section will describe what methods service contains and how those methods are implemented.

4.3 Authentication service

The authentication service is used to authenticate the user in the SecNote system. The user can use this service to log in to the service or create a new account. In addition to these features that are visible to the end-user, the service is used to extend the user's login and to verify the user's requests. The ability to verify user requests increases system security and prevents the so-called man-in-the-middle attack. All features are shown in Fig 4.3. How the individual features are implemented will be described in this section.

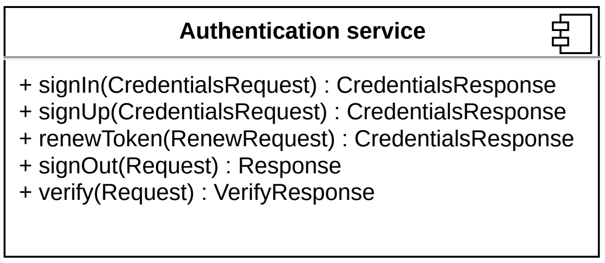

Authentication service
+ signIn(CredentialsRequest) : CredentialsResponse
+ signUp(CredentialsRequest) : CredentialsResponse
+ renewToken(RenewRequest) : CredentialsResponse
+ signOut(Request) : Response
+ verify(Request) : VerifyResponse

Fig. 4.3: Authentication service

4.3.1 Sing In

The RPC method is used to sign-in a user to an existing account. The parameter of the method is a structure called CredentialsRequest. The fields of the structure are email, password, and key. Email and password are used to verify that the user has the correct access data and thus is allowed access to the system. The key is an RSA public key in PEM format encoded with base64 and is tied to the user at login. By associating the key with the user account when signing-in, we protect the system from account sharing. If the user logs in under the same account on another device, then after the access token expires, the user will be logged out of the device because the recovery of the access token using the recovery token fails due to a different RSA key associated with the account. This is a useful feature for systems that offer their service free of charge for an individual, but if you want to use the service with more people, the service is charged. If the login details are correct and the key is in the correct format, the user's CredentialsResponse structure is returned. The

CredentialsResponse structure contains a JWT structure. The JWT token serves as a representation of the user when interacting with the SecNote system. How a JWT token is created and used is described in Section 4.3.6.

4.3.2 Sign Up

The RPC method is used to create a user account. The procedure is similar to sign-in, but instead of verifying that the credentials match the credentials saved in the database, it verifies that the user does not exist and if not, a new user account is created. As in the case of the sign-in method, the key is tied to the user account, and the use case is the same. The return type of the method is the same as in the case of login.

4.3.3 Sign Out

The sign-out is one of three methods with which the user interacts directly. Unlike sign-in and sign-up, when using the sign-out method, the user must provide access token in addition to the input parameters in the context of the RPC method. This ensures that an unauthorized user cannot sign-out the user from the system. The input parameters are empty in this case. The id of the user to be signed-out is obtained from the access token. After verification of the validity of the access token, the key that is bound to the user is deleted, and an empty structure is returned. The empty response indicates to the application that the logout was successful.

4.3.4 Renew token

The RPC method is used to create a new access token for the user. The access token is valid for 5 minutes. After expiration, it is necessary to renew the token so that the user is not signed out of the application. The token can be renewed using a renewal token, which is valid for seven days and is intended for renewal of the access token. The RSA key that is part of the renewal token is compared to the key that is bound to the user, and if the token is valid and the keys match, new access token and renew token are created and returned to the user using the CredentialsResponse structure.

4.3.5 Request verification

The Verify RPC method is used to verify the signature of the request. A signature, token, and digest are extracted from the request context to verify the request signature. This RPC method is used primarily by other cloud services. The user can also verify his request, but within the business logic, it does not make sense, and this method is not used for this purpose. Before the service calls the Verify RPC method, it verifies that the digest matches the digest created from the request. If the digests do not match, the error is returned to the user, and the Verify method is not executed. How exactly the signature verification is implemented is explained in Section 4.3.6.

4.3.6 Security mechanisms

This section describes in more detail the security techniques used in the RPC methods of the Authentication Service. These techniques include creating a JWT token, verifying the token, and verifying the request signature. Where these techniques are used has been described in the sections mentioned above.

Token creation

As mentioned, the JWT token represents the user in the system. The token is digitally signed by the server, so it is not possible to change data placed in the token. When the user sign-in, sign-up or renew an access token, then an access token and a renewal tokens are created. The JWT token contains claims that are divided into public and private claims, and the creation of those claims is on Listing 4.1. One of the fields in the private claims is the key, and this is the public key of the user, which was used as a method parameter when logging in or creating the account.

```go
currentTime := time.Now()

accessPublicClaims := jwt.Claims{
	Issuer:    IssuerValue,
	IssuedAt:  jwt.NewNumericDate(currentTime),
	Expiry:    jwt.NewNumericDate(currentTime.Add(ExpirationTimeAccessValue)),
}

accessPrivateClaims := map[string]interface{}{
	UserIdKey:        id,
	AccessTokenKey:   true,
	RefreshTokenKey:  false,
	PublicKeyKey:     key,
}
```

Listing 4.1: JWT Claims creation

As mentioned, the JWT tokens are signed by the server in order for the token to be signed, the server must generate an elliptic curve key. Listing 4.2 shows how a key is generated using elliptic curves and then used to create a JwtCrypto structure. This structure is later used to obtain a signer. After creating the structure and claims, the server can create a token. The process of creation of token from claims using signer is shown in Listing 4.3. The renewal token is created in the same way. The difference is in the type of token that is set in the private claims and the expiration time.

```go
func ProvideSigned() *model.JwtCrypto {
   ecdsaKey, _ := ecdsa.GenerateKey(elliptic.P521(), rand.Reader)
   key := jose.SigningKey{Algorithm: jose.ES512, Key: ecdsaKey}
   var options = jose.SignerOptions{}
   options.WithType("JWT")
   ecSigner, err := jose.NewSigner(key, &options)
   if err != nil {
      panic(err)
   }

   return &model.JwtCrypto{
      Signer: ecSigner,
      Key:    ecdsaKey,
   }
}
```

Listing 4.2: Eliptic curve key creation

```go
builder := jwt.Signed(repository.JwtCrypto.Signer)
   .Claims(accessPublicClaims)
   .Claims(accessPrivateClaims)
accessToken, err := builder.CompactSerialize()
```

Listing 4.3: JWT token creation

Token verification

Token verification is performed at each request where user data is handled. Before any token operation is performed, the token must be parsed from the string into the JSONWebToken structure. How to convert a token from a string to a structure is shown in Listing 4.4.

```go
func (repository *CryptoRepository) ParseJWTToken(token string) (*jwt.JSONWebToken, error) {
   parsedJWT, err := jwt.ParseSigned(token)
   if err != nil {
      err = tools.ConvertError(codes.Unauthenticated, "Can‘t parse JWT token", err)
      return nil, err
   }
   return parsedJWT, nil
}
```

Listing 4.4: Parsing of JWT token

After the token is parsed, the server tries to verify the token signature. The signature is verified using the public key of the elliptic curve. If the signature is valid, claims are obtained from the token. This process is shown in Listing 4.5.

```go
type JWTClaims map[string]interface{}
func (repository *CryptoRepository) GetJWTClaims(token *jwt.JSONWebToken) (JWTClaims, error) {
   claims := make(JWTClaims)
   err := token.Claims(&repository.JwtCrypto.Key.PublicKey, &claims)
   if err != nil {
      err = tools.ConvertError(codes.Aborted, "Can‘t deserialize claims from JWT token", err)
      return nil, err
   }
   return claims, nil
}
```

Listing 4.5: JWT token signature verification

The procedure that has been performed so far ensures that the data in the token have not been altered. The server can now verify that the token is of the correct type and that the token is still valid, in other words, that the expiration time has not exceeded the current time. If these conditions succeed, the token is evaluated as valid. Otherwise, an error is returned to the user, and the action the user tried to perform is invalid. The whole procedure is shown in Listing 4.6.

```go
func (repository *CryptoRepository) VerifyJWT(token *jwt.JSONWebToken, isRefreshToken bool) error {
   claims, err := repository.GetJWTClaims(token) // Returns formatted error
   if err != nil {
      return err
   }
   if claims[RefreshTokenKey] != isRefreshToken {
      return tools.CreateError(codes.InvalidArgument, "Token is in invalid type")
   }

   expiration := int64(claims[ExpirationTimeKey].(float64))
   if time.Now().After(time.Unix(expiration, 0)) {
      return status.Error(codes.Unauthenticated, "token is expired")
   }

   return nil
}
```

Listing 4.6: JWT token expiration verification

Request signature verification

Up to this point, the server has determined which user it is according to the token and that the data in the token has not been modified. However, the server has not yet ruled out whether the token was stolen, and now the attacker is trying to retrieve or change the user's data. In order for the server to prevent this scenario, each request is protected by the user's digital signature.

```go
func (repository *CryptoRepository) VerifyRequestSignature(
      token *jwt.JSONWebToken, encodedMessage string,
      signature string, encodedToken string) (bool, error) {
   headers, err := repository.GetJWTClaims(token) // Returns formatted error
   if err != nil {
      return false, err
   }
   pubKeyRaw := headers[PublicKeyKey].(string)
   pubKey, err := repository.ParseKey(pubKeyRaw) // Returns formatted error
   if err != nil {
      return false, err
   }
   sign, err := base64.StdEncoding.DecodeString(signature)
   if err != nil {
      err = tools.ConvertError(codes.InvalidArgument, "Invalid signature base64 encoding", err)
      return false, err
   }
   messageByte := []byte(encodedMessage)
   tokenByte := []byte(encodedToken)
   sha512Hash := sha512.New()
   sha512Hash.Write(append(messageByte, tokenByte...))
   hash := sha512Hash.Sum(nil)
   err = rsa.VerifyPKCS1v15(pubKey, crypto.SHA512, hash, sign)
   if err != nil {
      err = tools.ConvertError(codes.Unauthenticated, "Invalid signature, err)
      return false, err
   }
   return true, nil
}
```

Listing 4.7: Request signature verification

A signature, token, and digest are obtained from the context of the RPC method. Before the signature verification method is called in the authentication service, the service that wants to verify the signature validates whether the digest in context is identical to the digest created from the request body. An RSA public key is obtained from the token to verify the digital signature. The body of the signed message is created from a digest and a token. This ensures that the request comes from the token owner. Such a message is then hashed with the SHA512 algorithm. If the result after the signature verification is equal to the created hash, the message is valid, and the signature is valid. The whole process can be seen in Listing 4.7.

4.4 Note service

The service provides CRUD operations on notes. The service does not know who the individual notes belong to, and service is used primarily to perform CRUD operations on the notes. The RPC methods that the service provides in the cloud system also correspond to this. The user does not have direct access to this service, and they use it through the permission service. This means that the service cannot be accessed outside the cloud. The methods that the service provides are shown in Listing 1.0, and the service primarily only manages database queries and formats errors in the event of a mistake, for these reasons, the specific methods will not be described further in the text.

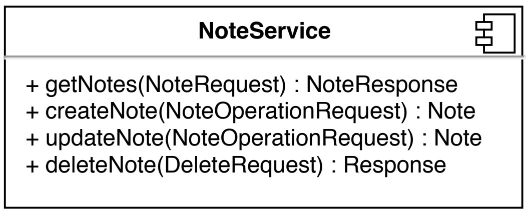

NoteService
+ getNotes(NoteRequest) : NoteResponse
+ createNote(NoteOperationRequest) : Note
+ updateNote(NoteOperationRequest) : Note
+ deleteNote(DeleteRequest) : Response

Fig. 4.4: Note service

4.5 Permission service

The permission service serves as the primary point for the user in the SecNote system with which the user interacts. The service stores information about the user's mapping to notes and communicates with authentication services and note services. The user can use this service to manage their notes. The following sections describe how the individual methods are implemented and how they use other services in the system to provide information for the user.

4.5.1 Get notes

In order for the service to provide all the notes that belong to the user, the permission service must communicate with the authentication services and the note services. The permission service first verifies that the digest from the request context equals the digest created from the request body. If this check passes, then the service sends a request to the authentication service to verify the token and signature of the request. After confirming the identity of the applicant and the request credibility, the service will start the process for obtaining notes. Notes are stored in the note service, but the information which notes belong to whom is stored in the permission service. So, the permission service first obtains information about which notes belong to the user from its database. According to the received note identifiers, it gathers data from the note service and provides it to the user.

4.5.2 Add or update note

As with the getNotes() method, the token and signature of the request are verified using the authentication service. The input parameter for AddOrUpdateNote() is the NoteOperationRequest structure that wraps the note. The server checks the id of the note in the NoteOperationRequest, and if the id is empty, the server will assume that it is a new note, and it will try to create it. Otherwise, if the id is specified, it will be assumed that the note already exists and tries to edit it. In case of note creation createNote() is called on note service, and after success, execution permission service creates a mapping of the user to the newly created note. In case of note update, permission service check if the user owns the note and if the user owns it, and it executes updateMethod() on note service. In either case, after the action is executed AddOrUpdateNote() returns the newly modified or created note to the user.

4.5.3 Delete note

When deleting a note, the procedure is very similar to updating a note. The server checks the JWT token and the signature of the request, and if this information is valid, the server checks whether the user owns the note he wants to delete. If all checks are successful, a request is sent to the note service for the note to be deleted, and the ownership record in the permission service is also removed.

4.5.4 Security mechanisms

This section describes the implementation details of the security mechanism implemented in the SecNote application. These techniques include application lock, binding the keys to the user, encryption, decryption, and digital signature of requests. Previous chapters described the application screen by screen for a better picture where these security mechanisms come into play and how to leverage them to make the application more secure.

Digest verification

The digest check is related to the verification of the signature of the request. The server must verify the digest so that the user does not create a request signature and then changes the data in the request body just before sending it. This use case would evaluate the signature as valid, but in reality, the signature would not match the request body. So to verify digest, the permission service converts the request to a byte array and passes the result to the ProxyMedataFromContext() method. In Listing 4.8 is shown implementation of the ProxyMedataFromContext method. The request that was converted to a byte array is used to create a hash using the SHA512 algorithm, which is subsequently encoded using base64. The string created in this way is compared with the digest that was obtained from the context of the request. In case the digests are equal, the metadata of request is proxied to the verify method of authentication service to verify the signature of the request and JWT validity.

```go
func ProxyMedataFromContext(context context.Context, request []byte) (context.Context, error) {
  md, ok := metadata.FromIncomingContext(context)
  md = md.Copy()
  digest := md.Get(DigestKey)
  if len(digest) == 0 {
    return nil, CreateError(codes.FailedPrecondition, "Can't get metadata from context")
  }

  hashed := sha512.New()
  hashed.Write(request)
  output := hashed.Sum(nil)
  requestDigest := base64.StdEncoding.EncodeToString(output)

  if digest[0] != requestDigest {
    return nil, CreateError(codes.FailedPrecondition, "Digest is not equal to request body")
  }

  if !ok {
    return nil, CreateError(codes.Unauthenticated, "Can't get metadata from context")
  }
  return metadata.NewOutgoingContext(context, md), nil
}
```

Listing 4.8: Digest verification

4.6 SecNote application

SecNote is native Android application where security is the priority, and the application allows users to manage notes, add different categories to the notes, and also encrypt them. Different security mechanisms are implemented to ensure that users' notes are appropriately protected. Besides local security mechanism, notes are synchronized into cloud so secure communication between mobile device and cloud system is also implemented. The application is the entry point for the end-user into the SecNote system. Principles that are universal and applicable to all screens will be explained inside the Architecture Section 4.6.1. Flow of the application is shown in Fig 4.5. In the following sections, the application's functionality will be divided based on its screens and further described.

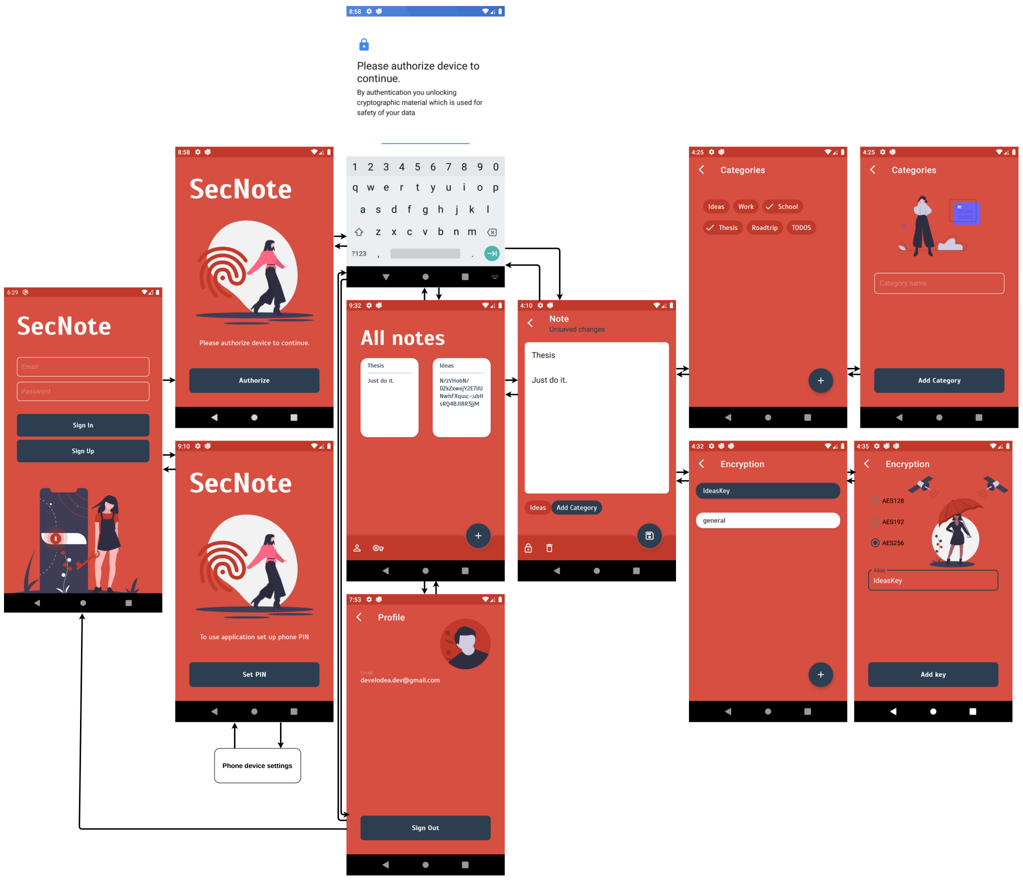


Fig. 4.5: Secnote application flow

4.6.1 Architecture

The application is based on MVVM [15] architecture and uses the Arkitekt library[16], which fallows MVVM architecture and adds base components that are then used to implement the application. Arkitekt is an open-source library with MIT Licence. Fig 4.6 showcase abstract application architecture. Every screen in the application is built upon principles shown in Fig 4.6, and every component will be briefly described depending on their use. All components create together architecture, which allows easy navigation through the code and supports the robustness of resulting application. The application can run on any Android device with Android 6.0 or higher. The application needs to communicate with the remote API so notes can be synchronized to the server. How application established connection and what kind of communication is used is explained in the networking Section 4.6.2.

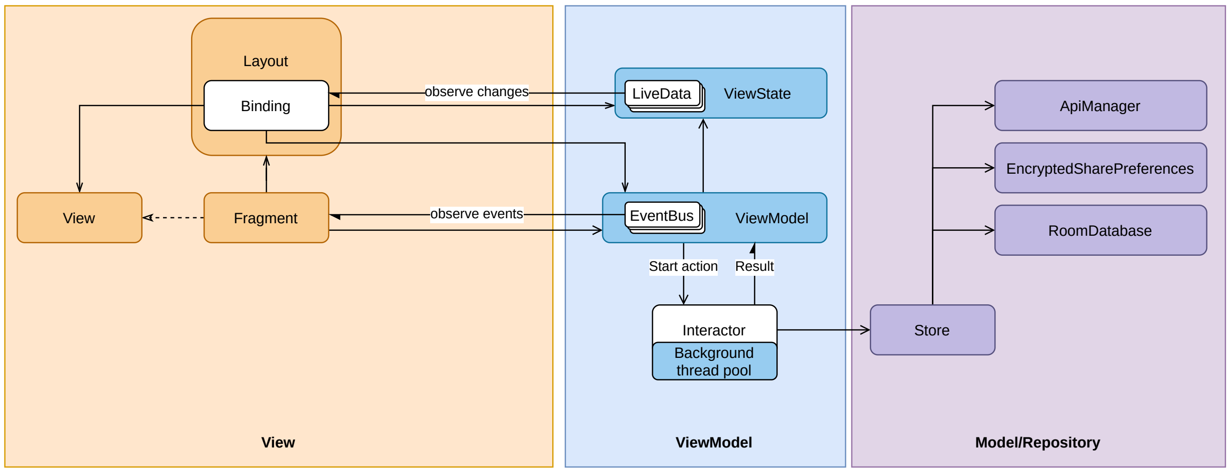


Fig. 4.6: Application architecture

Fragment

The Fragment represents the UI portion of the application and also inherits from BaseBindingFragment from Arkitekt library. BaseBindingFragment sets up data-binding from Fragment layout and also sets up dagger dependencies to be injected. If the screen has defined events, the Fragment can observe the events and react depending on the event type. The Fragment has direct access to the view-model.

View

The view is an interface that represents actions that can be called on the Fragment. This is useful when a fragment contains a recycler view, and the view holder requires action to be executed after on click on it. In that case, the view can be provided by the dagger to the adapter, which provides it to the view holder. This way, actions can be called from view holders on a fragment without the need to provide the full accessibility of Fragment.

ViewModel

The role of ViewModel is to encapsulate business logic from the Fragment. Besides its architectural intent, it also implements BaseCrViewModel from Arkitekt library, which provides coroutine scope bound to view-model scope. This scope is used by interactors and ensures the clean up of resources and calls in case of scope destroy event.

ViewState

ViewState is a representation of the screen. All data that are displayed on screen or are related to the screen are persisted in ViewState. Data representation is presented in live data, which can be observed in the Fragment. ViewState is persisted in ViewModel, so the ViewState is not destroyed on lifecycle changes of Fragment.

Events

In the MVVM architecture, ViewModel should not be able to communicate directly with the Fragment. ViewModel should be clueless, which Fragment is using the ViewModel. Based on these principles, events can be defined and send from the ViewModel. The Fragment can than observe the view-model events and react based on the type of event.

Interactors

Interactors encapsulate domain logic. There are two base classes CoroutineInteractor and FlowInteractor. CoroutineInteractor emits a single result, and FlowInteractor emits a stream of results. Interactors are executed on background executors, which frees the main thread from heavy background work. Concrete interactors are injected into the view-model and executed at will.

4.6.2 Networking

SecNote application is based on the premise that the user can get access to the notes if he has a knowledge of authentication credentials. The application needs to communicate with the remote API to implement these features. The SecNote application communicates with the Authentication 4.3 and Permission 4.5 service. These services run on *gRPC* protocol, that is based on HTTP/2, which means that all communication is multiplexed through the single channel. Listing 4.9 shows the declaration of the channel that is located in NetworkModule. In line seven is used useTransportSecurity() method. This method enforces that all communication through this channel needs to be under the TLS. TSL secures the communication channel, so the communication between the application and remote API is secure. *gRPC* service is defined by a proto file. Proto file is a file that declares all the messages that can be sent to the service and all the RPC that service contains. So, in contrast with REST API, *gRPC* is a type-safe. On build type compiler generates *gRPC* client builder that can be used to build a client for service. To build a client, application needs to provide a channel that was created on Listing 4.9.

```
@Provides
@Singleton
fun provideChannel(@ApplicationContext context: Context): Channel = AndroidChannelBuilder
    .forTarget(Constants.Network.URL)
    .context(context)
    .idleTimeout(Constants.Network.IDLE_TIMEOUT, TimeUnit.MINUTES)
    .useTransportSecurity()
    .build()
```

Listing 4.9: gRPC Android channel

Listing 4.10 shows how to use generated builder to build stub for services. After a stub is built, the application can call an RPC that is defined in the proto file.

```
@Provides
@Singleton
fun provideAuthStub(channel: Channel) =
    AuthServiceGrpcKt.AuthServiceCoroutineStub(channel)

@Provides
@Singleton
fun providePermissionStub(channel: Channel) =
    PermissionServiceGrpcKt.PermissionServiceCoroutineStub(channel)
```

Listing 4.10: gRPC Stub

4.6.3 Login

Users can create a new account with sign up or sign in into an existing account. Fig 4.7 shows login UI where the user enters login credentials, which are composed of email and password, and selects which action would like to perform.

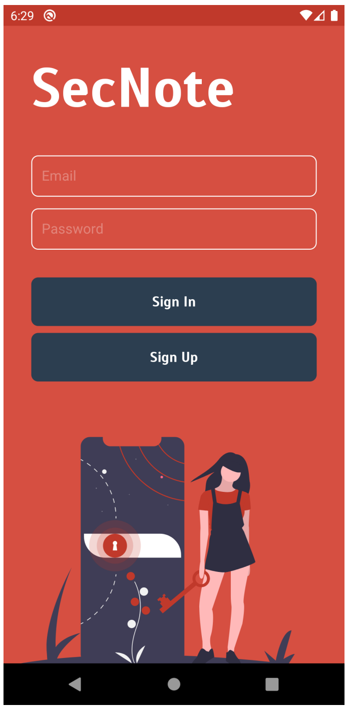
Validation

Before sign in or sign up action is executed, in both cases, the input is validated with the same method validateInput() shown on Listing 4.11.

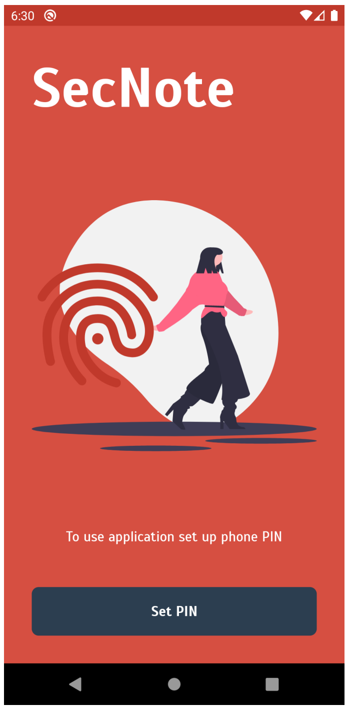
```
private fun validateInput(action: () -> Unit) {
    val isEmailValid = Patterns.EMAIL_ADDRESS.matcher(viewState.email.value).matches()
    val isPasswordValid = viewState.password.value.isNotBlank()

    viewState.emailError.value = if (isEmailValid.not())
          resources.getString(R.string.general_email_error) else ""
    viewState.passwordError.value = if (isPasswordValid.not())
          resources.getString(R.string.general_password_error) else ""

    if (isEmailValid && isPasswordValid) {
        action()
    }
}
```

Listing 4.11: Email and password validation

Email and password edit text fields are bound to the ViewState so the method can access the values inserted by the user. Email is checked if it matches the pattern,



(a) Login screen



(b) Pin set up screen

Fig. 4.7: Login and pin set up screens

and the password is checked if it is not empty. If both conditions are valid than the provided action is executed. Otherwise, an error is shown to the user.

Device security

After input validation, application checks if the device is secure. By secure, it means the device has set up PIN/PATTERN/PASSWORD. This can be validated with the KeyguardManager that has method isDeviceSecure() 4.12. If the device is not secure, NavigateToPinEvent is sent from the view-model with the value of PinState.PIN_SET. After the event is observed in the Fragment, the PinFragment is started with the PIN_SET argument 4.6.4. In the case that the device is secured, the method proceeds with the action, which can be SignIn or SingUp.

```
    private fun checkIfDeviceIsSecure(action: () -> Unit) {
        if (!keyguardManager.isDeviceSecure) {
            sendEvent(NavigateToPinEvent(PinState.PIN_SET))
        } else {
            action()
        }
    }
```

Listing 4.12: Device security validation

Sign Up

The sign-up action creates a new user account. After input and device security validation, SignUpInteractor is executed. Interactor generates new RSA key pair. This RSA key is used for request signing and verification. Before the password is sent to the server, salt is added to the password. Salt is a sequence of characters that is added to a user's password to increase security against dictionary attacks. After the salt is added, the result is hashed with the SHA512 hash function. To create new account, signUp() RPC is executed on Authentication service. Parameters for the call are email, password, and public RSA key. On successful account creation, access and renew tokens are returned as a response. Tokens are saved into encrypted shared preferences. Interactor on success callback is called, and NavigateToPinEvent is sent with the argument PinState.AUTHORISE. After the event is observed in the Fragment, the PinFragment is started with the AUTHORISE argument 4.6.4.

Sign In

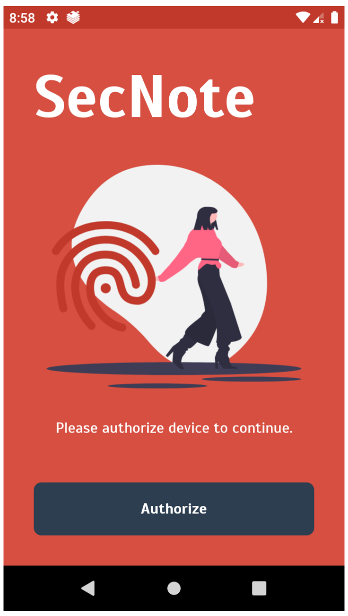
The sign-in action gets the existing user account. Flow with sign-in is the same as in sign-up. The difference is that the SignInInteractor is executed, and inside the interactor signIn() RPC is executed on Authentication service. Other flow from key creation to the event sent is the same.

4.6.4 Pin

The screen has three states consisting of *PIN_SET*, *AUTHORIZE*, *REAUTHORIZE*. The initial state is set by an argument with which the screen is started. Screens in all three states can be seen in Fig 4.8.

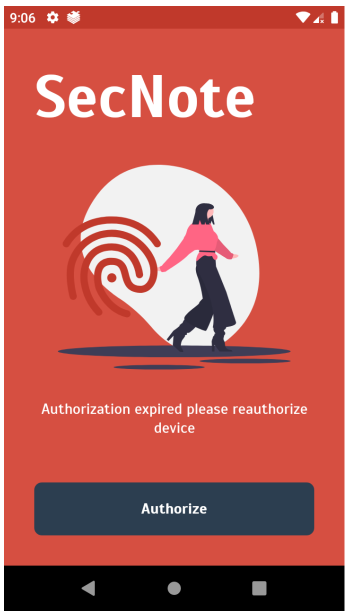
The *PIN_SET* state screen informs the user that the lock screen is not set on this device. It needs to be set up so the application can function correctly. The "Set PIN" button navigates the user to the device settings where the user can set the lock screen of the device. Only the login screen can start a pin-screen with an initial state of *PIN_SET*. In other use cases, if the user removes a lock screen from the device after the login, app-generated keys will be invalidated and unusable. The application will automatically log out the user from the application and show him a login screen where on login attempt user will be prompt to the pin-screen. If the user sets up the lock screen, application will return the user to the login screen where he can continue with the authentication process.

The *AUTHORIZE* state screen initially opens the lock screen. The security mechanism informs the user that he needs to authorize to unlock cryptographic material. By authorizing, the user unlocks keys saved in Keystore for a 5-minute



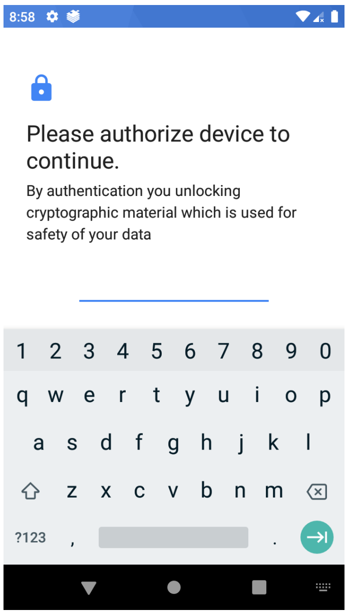
Please authorize device to continue.

(a) State AUTHORIZE



Authorization expired please reauthorize device

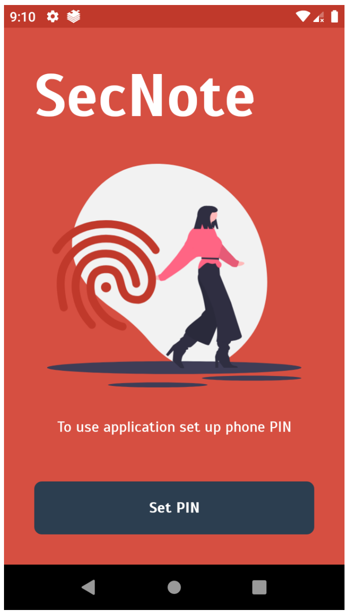
(b) State REAUTHORIZE



Please authorize device to continue.

By authentication you unlocking cryptographic material which is used for safety of your data

(c) Lock screen



To use application set up phone PIN

(d) State PIN_SET

Fig. 4.8: Pin-screen in all three states with lock screen

time window. If a user tries to go back or fail the authorization, the app will show infographics to the user that he needs to authorize to continue. Screen with the initial state of *AUTHORIZE* is invoked right after login or if the user is already signed in after app startup.

The *REAUTHORIZE* state screen informs the user that a 5-minutes time window had expired, and he needs to authorize again to unlock cryptographic material.

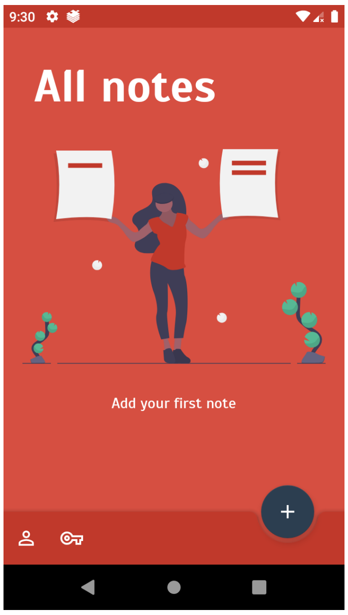
The difference between *AUTHORIZE* and *REAUTHORIZE* state is that the *REAUTHORIZE* state does not open initially lock screen but instead, informs the user what happened and what he needs to do next in order to use application. This behavior is implemented due to the use case of how a screen with state *REAUTHORIZE* is opened. If the user tries to do some in-app action after the expiration of a 5-minutes window pin-screen with the initial state of *REAUTHORIZE* is opened. Users can invoke the lock screen with the Authorize button and, after successful authorization, continue where left off in the application.

4.6.5 Notes

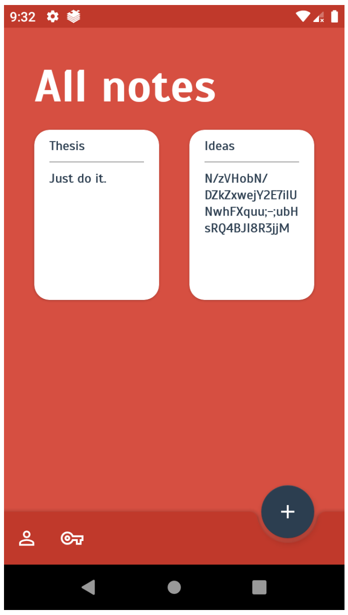
The notes-screen displays all notes that belong to the signed-in user, and also serves as the primary navigation component from where the user can navigate to the other screens. Notes are loaded from the room database and synchronized from remote permission service into the room database. Synchronization is invoked by onResume call from fragment lifecycle call. If the user interacts with the application longer than 5 mins and tries to invoke the synchronization, pin-screen with an argument of REAUTHORIZATION will be shown. From the notes screen, the user can navigate to note detail, profile, and preview of encryption keys. At the bottom right of the screen *floating action button*, navigates the user to note screen without id argument, which corresponds to note creation. Notes are displayed in raw form, which means when a note is encrypted user sees only the encrypted text body and readable title. Fig 4.9 shows the empty state of notes screen and also screen with content where one of the notes is encrypted, and the other one is not. By clicking on the note, the application navigates to the note screen.

4.6.6 Note

In the note-screen Fig. 4.10, the user can create a new note or edit an existing note. In which of two states screen opens depends on noteId argument. If noteId argument is provided, the corresponding note will be preloaded on the screen. The note is observed from the local room database. Any update to note will propagate to the UI immediately. If the note is encrypted, the application will try to decrypt it and show the results to the user. The decryption of note is conditioned by user authorization and can be done only in authorize 5-minute window. Otherwise, pin-screen with argument REAUTHORIZATION will be shown to the user. Changes made by users are not immediately saved. Instead, the user is notified that he made some changes and can save the note with the *floating action button*. If the user tries to navigate back without saving dialog is open that informs the user if he leaves this screen, all unsaved changes will be discarded. Notes can be added to categories



(a) Empty state



(b) Content

Fig. 4.9: Notes screen

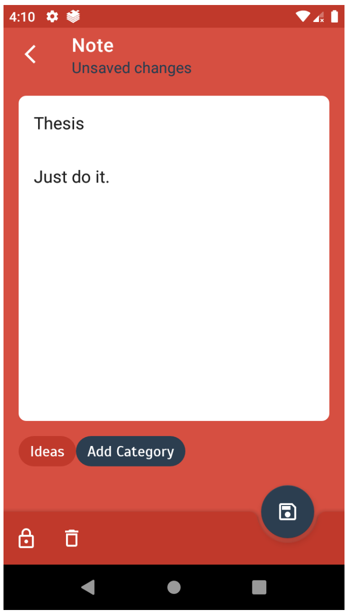
that are created by the user. On the bottom of the screen is a chip with the text "Add Category". By clicking on the *chip*, application navigates to the categories screen. As mentioned earlier, the user can encrypt the note. Encryption can be done by clicking on the lock icon in the bottom app bar, which opens the encryption key selection screen. Right next to the lock icon is the bin icon, which deletes the current note and returns the user to the notes-screen.

4.6.7 Categories

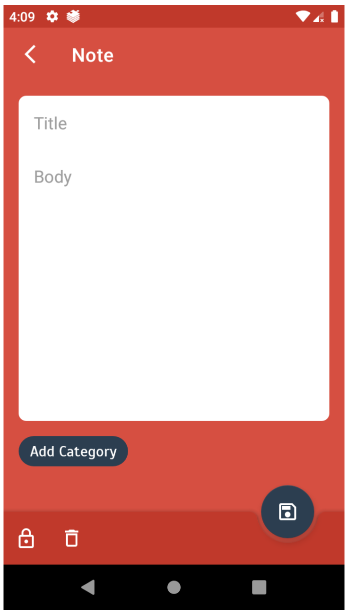
The categories screen displays all categories created by the user. The user can manage categories corresponding to the note by selecting and deselecting individual categories. The new category can be added by clicking on the *floating action button* on the bottom of the screen. The categories screen is started with argument selectedCategories. The argument serves for categories synchronization and pre-selection of categories.

Synchronization of categories

Upon view model creation, the onStart lifecycle call is called. In this call, SyncCategoryInteractor is executed. The categories screen is started with the argument selectedCategories that contains selected categories for currently selected note. This argument is also a mandatory parameter for the SyncCategoryInteractor, as listed



(a) Note with pending changes



(b) New note

Fig. 4.10: Note screen

in Listing 4.13. The SyncCategoryInteractor check if provided categories are created in the room database, and if not, it creates them after successful categories synchronization application can guarantee that selected categories are created and can be shown in the UI.

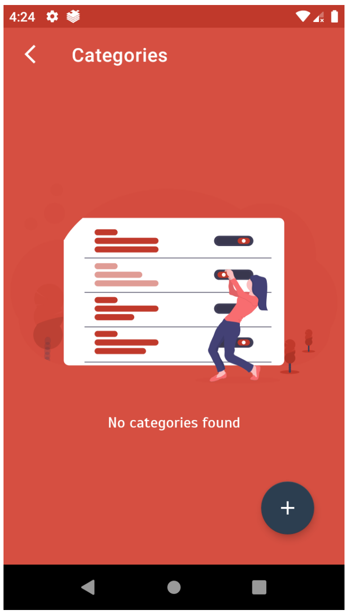
```
    override fun onStart() {
        if (viewState.selectedCategories.value.isNotEmpty()) {
            syncCategoryInteractor.init(viewState.selectedCategories.value.toList()).execute {
                getCategories()
            }
        } else {
            getCategories()
        }
    }
```

Listing 4.13: Category synchronization interactor call

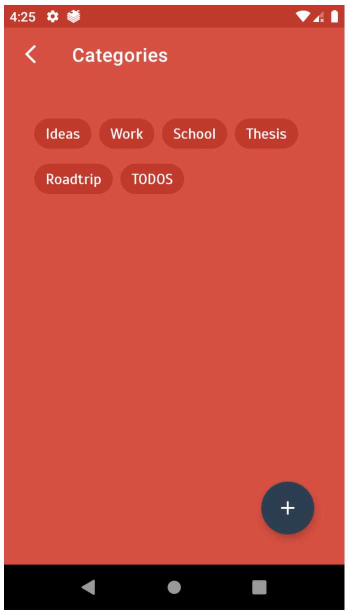
The application needs to load them from the room database to the ViewState to show categories in the UI. This is the purpose of GetCategoriesInteractor executed in the success callback of SyncCategoryInteractor Listing 4.14.

```
    private fun getCategories() {
        getCategoriesInteractor.execute(
            onNext = {
                viewState.categories.value = it
                viewState.loading.value = false
            }
        )
    }
```

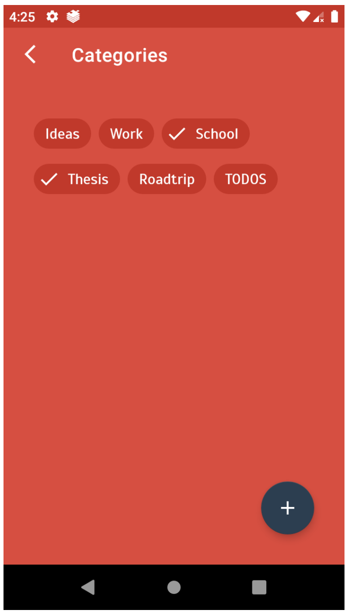
Listing 4.14: Category synchronization interactor call



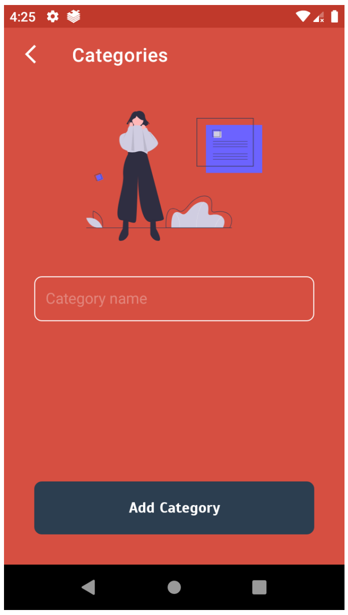
(a) Empty state



(b) Content



(c) Selected



(d) New

Fig. 4.11: Categories screen

SyncCategoryInteractor observers categories saved in the room database and changes are propagated to the UI by calling onNext callback on every change to the categories. This guarantees that newly created categories are shown into the UI without the need for result mechanism implementation from the create categories screen.

Category selection and deselection

Category can be selected or deselected by clicking on the chip. Depending on the state of the selected chip, the state needs to be inverted and added or removed from note selected categories. By clicking on the chip, invertState() method from view-model is called. Implementation of invertState() is shown on Listing 4.15. After the modification of selected categories, the ChangeCategoriesEvent is sent to the CategoriesFragment to set new categories for the selected note.

```
    fun invertState(category: CategorySelection) {
        viewState.selectedCategories.value = if (category.selected) {
            viewState.selectedCategories.value.apply { remove(category.name) }
        } else {
            viewState.selectedCategories.value.apply { add(category.name) }
        }
        sendEvent(ChangeCategoriesEvent(
            NoteCategories(
                viewState.selectedCategories.value.toList()
            )
        ))
    }
```

Listing 4.15: Invert state of category

In the CategoriesFragment event is observed Listing 4.17, and newly selected categories are saved as a result under the *CATEGORIES_CHANGE* key for the previous fragment in back stack.

```
    observeEvent(ChangeCategoriesEvent::class) {
         setResult(Constants.Note.CATEGORIES_CHANGE, it.result)
      }
```

Listing 4.16: Set new categories for note

New category creation

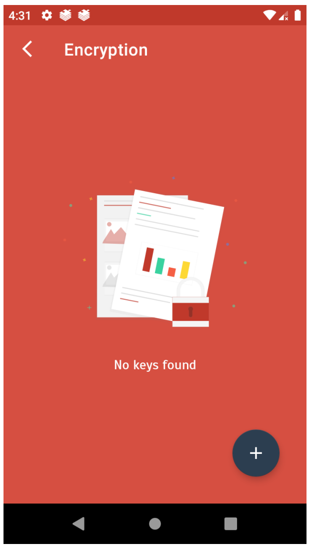
To create a new category, the user enters the category name into the edit text input. Edit text input is bi-directionally bound to the live data in ViewState. To confirm the category creation user clicks on the "Add Category" button, which calls the createCategory() method 4.17 in the view model. The CreateCategoryInteractor creates a new category with the name entered into the input. After the successful creation of the category user is returned back to the category-screen.

```
    fun createCategory() {
        val name = viewState.name.value
        if (name.isNotBlank()) {
            createCategoryInteractor.init(name).execute(
                onSuccess = {
                    sendEvent(NavigateBack)
                }
            )
        }
    }
```

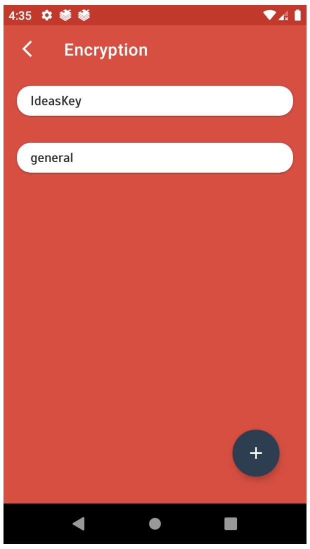
Listing 4.17: Set new categories for note

4.6.8 Encryption

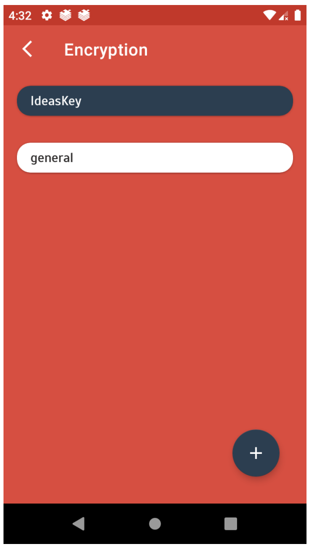
Encryption screen helps the user to manage AES keys, which can be used to encrypt and decrypt notes. Usage of a key is conditioned by the authorization window opened by user authorization for five minutes. Keys are saved and loaded from Keystore. By clicking on a key, the user select and deselect key for encryption and decryption of the note. The user can choose the length of the AES key and alias. The encryption screen is started with argument alias. The alias argument serves for the pre-selection of the key.



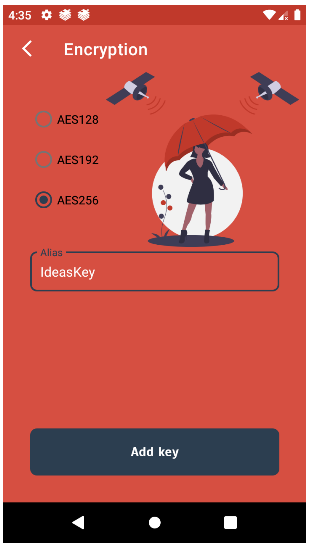
(a) Empty state



(b) Content



(c) Selected key



(d) Key creation

Fig. 4.12: Encryption screen

Keystore aliases

For the user to be able to select the key from the Keystore, the application needs to be able to list all saved keys by defined aliases. Keystore API provides method aliases(), which returns the list of all saved aliases in the Keystore. The application needs to filter out from this list device key, which is used for request signing and encrypted shared preferences key, which is used to encrypt and decrypt data saved in shared preferences. Implementation is shown on Listing 4.18.

```
fun getKeystoreAliases() = keystore.aliases().toList()
    .filter {
        it != Constants.Security.DEVICE_USER_KEY &&
            it != MasterKeys.AES256_GCM_SPEC.keystoreAlias
    }
```

Listing 4.18: List all aliases from keystore

Key selection and deselection

The key can be selected or deselected by clicking on the key alias in the list. Depending on the state of the selected key, the state needs to be inverted and set as a result of the previous note fragment. By clicking on the key onKeySelection() method from view-model is called Listing 4.19. The KeySelectionEvent is sent to the EncryptionFragment with the name of alias with which note should be encrypted. If the alias is the empty string, that means encryption is removed from the note.

```
fun onKeySelection(alias: String) {
    viewState.selected.value = if (viewState.selected.value == alias) {
        ""
    } else {
        alias
    }
    sendEvent(KeySelectionEvent(viewState.selected.value))
}
```

Listing 4.19: Key selection

New key creation

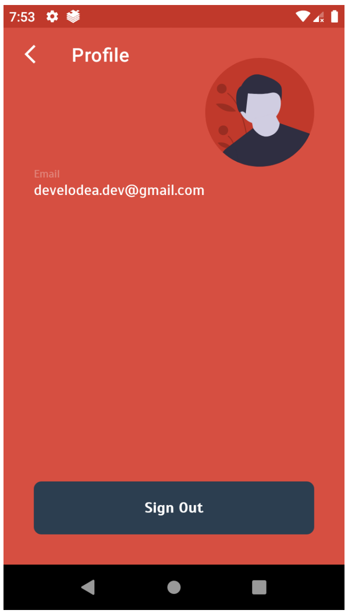
The user can generate AES keys for encryption and decryption of notes and can choose between AES128, AES192, AES256. To tell the difference and also tell the application which key should be used for different notes, the user assign aliases to all keys on their creation. Listing 4.20 shows how keys are generated in code. The mode chosen for the AES key is GCM with no padding. By setting randomize encryption to true, KeyStore ensures that each time such a key is used, a new randomized initialization vector is generated and used for encryption. This vector must be provided to the KeyStore during decryption for decryption to be successful.

```kotlin
    fun generateEncryptionKey(alias: String, keySize: Int) {
        val keyGenerator = KeyGenerator
            .getInstance(Constants.Security.AES, Constants.Security.KEYSTORE)
        val keyGenParameterSpec =
            KeyGenParameterSpec.Builder(alias, KeyProperties.PURPOSE_ENCRYPT
                                                or KeyProperties.PURPOSE_DECRYPT)
                .apply {
                    setBlockModes(KeyProperties.BLOCK_MODE_GCM)
                    setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
                    setRandomizedEncryptionRequired(true)
                    setUserAuthenticationRequired(true)
                    setUserAuthenticationValidityDurationSeconds(
                        Constants.Security.DEVICE_AUTHORIZATION_WINDOW
                    )
                }
                .setKeySize(keySize)
                .build()
        keyGenerator.init(keyGenParameterSpec)
        keyGenerator.generateKey()
    }
```

Listing 4.20: Key creation

4.6.9 Profile

The profile screen 4.13 shows the email of the signed-in user and provides a sign-out button for the user to sign-out from the application. On the click of the sign-out button, the method signOut() in the view model is called. In the signOut() method SignOutInteractor is executed. The SignOutInteractor uses the authentication service to call the signOut() method. Upon successful response from the server, the access and renew tokens are removed from the application and the user is redirected to the login screen.



Email
develodea.dev@gmail.com

Fig. 4.13: Profile screen

4.6.10 Security mechanisms

This section describes the implementation details of the security mechanism. Previous chapters described the application screen by screen for a better picture where these security mechanisms come into play and how to leverage them to make the application more secure.

Application lock

Application lock is useful when only the user that sets the device lock mechanism should be able to enter the application content or when the application needs to bind cryptographic keys with the application lock. Both use cases are used in SecNote. If the user doesn't have set PIN/PATTERN/PASSWORD application needs to confront the user that it is mandatory to set it before continuing. Optionally, the user can set biometric unlock, and then PIN/PATTERN/PASSWORD serves as a backup. How biometric works on Android is described in Section 2.3. The conclusion of Section 2.3 is that the application needs to use two different mechanisms for biometric authentication, FingerprintManager, and BiometricManager. Instead of implementing two different approaches in SecNote, Google's androidx.biometric library (in short xbiometric) can be used instead. Xbiometric encapsulates the logic for both FingerprintManager and BiometricPrompt, and depends on the device Android version. The library executes the appropriate code. Xbiometric is not part of the standard Android library and needs to be added to the application. This can be done by adding Gradle dependency as shown on Listing 4.21.

```
implementation("androidx.biometric:biometric:1.0.1")
```

Listing 4.21: Biometric library dependency

To show BiometricPrompt to the user, application needs to provide BiometricPromptInfo, BiometricCallback, and ExecutorService. This dependencies are provided by PinFragmentModule that is shown on Listing 4.22.

```
    @Provides
    fun callback(fragment: PinFragment) = BiometricCallback(fragment)

    @Provides
    fun biometricPromptInfo(resources: Resources) = BiometricPrompt.PromptInfo.Builder()
        .setTitle(resources.getString(R.string.general_lock_screen_title))
        .setSubtitle(resources.getString(R.string.general_lock_screen_subtitle))
        .setDeviceCredentialAllowed(true)
        .build()

    @Provides
    fun executor() = Executors.newSingleThreadExecutor()

    @Provides
    fun biometricPrompt(fragment: PinFragment,
                    callback: BiometricCallback, executor: ExecutorService) =
        BiometricPrompt(fragment, executor, callback)
```

Listing 4.22: BiometricPrompt dependencies in PinFragmentModule

The BiometricPromptInfo sets the dialog title message, body message, and by setting setDeviceCredentialAllowed() to true dialog will offer the possibility to authenticate with the backoff mechanism in case the biometric authentication is not available for the user at the moment. The ExecutorService manages on which thread will be BiometricCallback called. The BiometricCallback provides a way of communication with the calling PinFragment. In the case of authentication success, error, or failure, the appropriate method is called to inform the application about the result.

PinFragment implements BiometricCallbackInterface, which is used in BiometricCallback to proxy calls from BiometricCallbackInterface to the AuthenticationCallback from xbiometric library. AuthenticationCallback can't be directly extended in PinFragment due to the reason AuthenticationCallback is an abstract class, and PinFragment is already extending BaseBindingFragment from Arkitekt library.

```
    override fun onAuthenticationError(errorCode: Int, errString: CharSequence) =
        runOnUIThread {
            viewState.loading.value = false
        }

    override fun onAuthenticationSucceeded(result: BiometricPrompt.AuthenticationResult) =
        runOnUIThread {
            viewModel.checkState()
        }

    override fun onAuthenticationFailed() = runOnUIThread {
        viewState.loading.value = false
    }
```

Listing 4.23: BiometricPrompt result

In PinFragment, when AuthenticateDeviceEvent is observed, BiometricPrompt and BiometricPromptInfo are lazily injected and used to show lock mechanism appropriate for the Android version as shown on Listing 4.24. Listing 4.23 shows callback implementation in PinFragment. One of these methods is called to notify PinFragment about the result from BiometricPrompt.

```
class PinFragment : BaseBindingFragment<PinViewModel, PinViewState, FragmentPinBinding>(),
                    PinView, BiometricCallbackInterface {
    ...
    @Inject lateinit var biometricPrompt: BiometricPrompt
    @Inject lateinit var promptInfo: BiometricPrompt.PromptInfo

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        observeEvent(AuthenticateDeviceEvent::class) {
                biometricPrompt.authenticate(promptInfo)
        }
        ...
    }
    ...
}
```

Listing 4.24: BiometricPrompt

Request signing

Section 4.6.3 describes how user sign-in or sing-up into the application and how the public key of the generated RSA key pair is sent with the credentials to the server.

On successful response, the JWT access token and renew token is obtained in the application. This JWT tokens have embedded public key into them that was sent with the login credentials to the server. The public key is used for verification of the digital signature. Tokens are digitally signed by the server, so any modifications to them would result in a failed response from the server. This ensures that the JWT tokens are bind to the device that was used to sign-in or sign-up. Attempt to use this tokens on another device would fail with the cause of the missing private RSA key that is stored in the device Keystore. So to summarize, JWT is used to identify the user on the server and also provides users public key to the server with which server can verify provided request signatures created by the user. Listing 4.25 shows how the application generates the RSA key for digital signature and how the public key is extracted and converted to PEM format encoded into base64. To use this key user needs to be authenticated, authentication is enforced by setUserAuthenticationRequired() on line nine, which means he needs to use the application lock. After unlocking the device and becoming authenticated application opens a 5-minutes time window where the user can use the key. How long the window is defined on line ten with setUserAuthenticationValidityDurationSeconds(). The key generation is invoked on every attempt to sign-up or sign-in.

```
1      fun generateKey(): String {
2          val params = KeyGenParameterSpec
3              .Builder(
4                       Constants.Security.DEVICE_USER_KEY,
5                       KeyProperties.PURPOSE_SIGN or KeyProperties.PURPOSE_VERIFY
6                 )
7              .setDigests(KeyProperties.DIGEST_SHA512)
8              .setKeySize(Constants.Security.DEVICE_USER_KEY_SIG_SIZE)
9              .setUserAuthenticationRequired(true)
10             .setUserAuthenticationValidityDurationSeconds(
11                             Constants.Security.DEVICE_AUTHORIZATION_WINDOW)
12             .setSignaturePaddings(KeyProperties.SIGNATURE_PADDING_RSA_PKCS1)
13             .build()
14
15         keyPairGenerator.initialize(params)
16         val kP = keyPairGenerator.generateKeyPair()
17         val encoded = BaseEncoding.base64().encode(kP.public.encoded)
18         val pem = "$PEM_KEY_RREFIX\n$encoded\n$PEM_KEY_POSFIX"
19
20         return encodeBase64(pem.toByteArray())
21     }
```

Listing 4.25: Digital signature key generation

After the key is successfully generated, and the user is successfully signed-in into the application, all onwards requests to the server are digitally signed. Listing 4.26 shows getNotes() method in PermissionServiceManager.

```
    suspend fun getNotes() = executeApiCall {
        val request = Request.newBuilder().build()
        val digest = cryptoHelper.hashMessage(request.toByteArray())
        client
            .executeWithMetadata(digest)
            .getNotes(request)
    }
```

Listing 4.26: PermissionServiceManager - getNotes() method

In the method getNotes() request body is hashed with the SHA512 algorithm and encoded with base64 encoding, this process is shown on Listing 4.27. Hash is passed as a parameter to the method executeWithMetadata(), which implementation is shown in Listing 4.28.

```
    fun hashMessage(message: ByteArray): String =
        MessageDigest.getInstance(Constants.Security.HASH_ALG).run {
            update(message)
            encodeBase64(digest())
        }
```

Listing 4.27: CryptoHelper - hashMessage() method

```
    suspend fun <T : AbstractStub<T>> T.executeWithMetadata(request: String): T {

        val token = tokenStore.getAccessToken() ?: ""
        val toBeSign = request.toByteArray() + token.toByteArray()
        val signature = cryptoHelper.signAndEncodeDataBase64(toBeSign)

        val header = Metadata()
        header.put(Metadata.Key.of("Authorization", Metadata.ASCII_STRING_MARSHALLER), token)
        header.put(Metadata.Key.of("Digest", Metadata.ASCII_STRING_MARSHALLER), request)
        header.put(Metadata.Key.of("Signature", Metadata.ASCII_STRING_MARSHALLER), signature)

        MetadataUtils.attachHeaders(
            this,
            header
        )
    }
```

Listing 4.28: ServiceManager - executeWithMetadata() method

At this point, the application has all the necessary information to create the metadata. Individual information is gradually added to the metadata, and this process can be seen in Listing 4.28. First, the Authorization value is added to the metadata. Below the Authorization field is the value of the JWT token. Subsequently, the Digest value is added, which represents the value of the hashed request body. Signature is added as the last value. Signature is a combination of digest and JWT token. These two values are combined together, and then a hash is created from them, which is digitally signed by the RSA private key. Creation of signature is shown on Listing 4.29.

```
    fun signAndEncodeDataBase64(data: ByteArray): String {
        val entry = keystore.getKey(Constants.Security.DEVICE_USER_KEY, null) as PrivateKey
        return Signature.getInstance(Constants.Security.DEVICE_USER_KEY_SIG_ALG)
            .run {
                initSign(entry)
                update(data)
                encodeBase64(sign())
            }
    }
```

Listing 4.29: Signature creation

Note encryption and decryption

Section 4.6.8 describes how encryption and decryption keys are created and selected to encrypt and decryp the note. Trigger for note encryption is the selection of the key and on click on the save button in the note screen. The save action executes CreateOrUpdateNoteInteractor. Before the note is sent to the server, the note body is encrypted with the key that is saved in the Keystore under provided alias. Listing 4.30 shows implementation of CreateOrUpdateNoteInteractor build method.

```kotlin
    override suspend fun build(): NoteResponse {
        if (encrypted == true) {
            body = cryptoHelper.encryptData(alias, body)
        }
        val response = permissionServiceManager
                .createOrUpdateNote(prototype, id, title, body, categories, encrypted, alias)
        val notes = response.notesList.map { it.convertToRoomNote() }
        noteStore.syncNotes(notes)
        return response
    }
```

Listing 4.30: CreateOrUpdateNoteInteractor build method

The method encryptData() is responsible for note encryption. Parameters of the method are alias of the key and note body. The implementation of the method is shown on Listing 4.31.

```kotlin
    fun encryptData(alias: String, data: String): String {
        val key = keystore.getKey(alias, null)
        return Cipher.getInstance(Constants.Security.AES_ALG).run {
            init(
                Cipher.ENCRYPT_MODE,
                key
            )
            val encrypted = doFinal(data.toByteArray())
            val vector = iv
            "${encodeBase64(encrypted)}${ENCRYPTION_DELIMITER}${encodeBase64(vector)}"
        }
    }
```

Listing 4.31: CryptoHelper encryptData method

Method encryptData() encrypt the note with the "AES/GCM/NoPadding" algorithm. As mentioned when creating the key, one of the constraints set to the key is to enforce the random initialization vector generation on every encryption. The same initialization vector needs to be provided in case of a decryption of the note. Otherwise, decryption will fail. For that reason, the body of the note is saved as the concatenated text of encrypted text, divider, and vector. After that, the note is sent to the remote API and persisted on the server.

```kotlin
    fun decryptData(alias: String, data: String): String {
        val split = data.split(ENCRYPTION_DELIMITER)
        val encryptedData = decodeBase64(split.first())
        val vector = decodeBase64(split.second())
        val key = keystore.getKey(alias, null) ?: throw IllegalStateException("Missing key")
        val spec = GCMParameterSpec(128, vector)
        return Cipher.getInstance(Constants.Security.AES_ALG).run {
            init(
                Cipher.DECRYPT_MODE,
                key,
                spec
            )
            String(doFinal(encryptedData))
        }
    }
```

Listing 4.32: CryptoHelper decryptData method

The decryption of the note is occurring in GetNoteInteractor. Listing 4.32 shows decryptData() method that is started in GetNoteInteractor. In case the note is encrypted, the interactor decrypts the body and replaces the encrypted body with the decrypted version. The body is split with the encryption delimiter to obtain encrypted body and initialization vector. Before using any of the information, it needs to be decoded from base64 to ByteArray after which the raw initialization vector and body are passed to Cipher to obtain the decrypted body of the note.

5 Conclusion

This thesis aimed to describe the security mechanisms used in the Android OS and the communication between the Android OS and the remote server. The goal was divided into several parts of the diploma thesis. The first section, where the history of mobile devices and the Android system is presented, serves as an introduction to the reader as to why it is necessary to fiddle with security. The thesis introduces the components that make up the security model of Android OS and the role of individual components after stating why it is necessary to address the security of applications. This theory regarding what makes up the Android security model is presented in the second section. In addition to the introduction of components, this section also presents code snippets that describe how to use cryptographic algorithms and procedures that the Android system provides.

The third section is based on the theory described in the second section. It summarizes the results obtained from the benchmark application in which 280 tests have been implemented. The utilized test cases cover most of the cryptographic operations that can be performed on the Android system. Their primary goal was to evaluate each cryptographic operation's computational time and its suitability for application use. The results were further processed and visualized in heatmaps and a bar graph. Based on the results, it was concluded that not all assumptions were met. For example, RSA encryption was, in some cases, faster than AES encryption. Some older models of devices with the older processor units perform some of the cryptographic algorithms faster than newer devices with the newer processor units. This could be explained with the hardware acceleration for concrete cryptographic algorithms. By taking into account the various devices that are supported, and after analyzing the result, specific cryptographic algorithms were selected that are suitable and optimal for the implementation of an application that utilizes cryptographic operations. Selected algorithms are AES256/GCM/NoPadding for symmetric encryption, SHA512withRSA2048/PSS for digital signature, and RSA3072/OAEPWithSHA512AndMGF1Padding for asymmetric encryption.

Last, the fourth section describes the SecNote system. The SecNote system is an implementation of a complete solution that demonstrates the Android system's security mechanisms and best practices on how to implement them, as well as how to implement secure communication between the Android application and the cloud system. The cloud system is managed using Kubernetes and Istio service mesh and runs three micro-services with three databases. The SecNote system demonstrates security mechanisms such as biometric user authentication, time-limited login, how to present identity in a cloud solution, creating a secure channel between the application and the cloud, data encryption and decryption, and more.

Overall, this work offers a theoretical evaluation of the android security model, a comparison of options in a test environment, and a complete solution where various security mechanisms can be seen in practice. The primary area for improvement is in the cloud solution. It would be possible to set cluster-wide authentication policy rules instead of a specific service. This would reduce the amount of information transmitted and queries to the authentication service. Instead of ordinary JWT tokens, a more complex system could be used to authenticate and authorize users, such as OpenID. Currently, the solution assumes that it is deployed in a single cluster. The work could be extended to a multicluster solution where it would be necessary to add mTLS between individual services.

Bibliography

[1] StatCounter [online] [cit. 2019-12-20]. Desktop vs mobile market share worldwide. URL: `https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide/#monthly-201309-201909`.

[2] Android Developers [online] [cit. 2019-12-20]. Distribution dashboard. URL: `https://developer.android.com/about/dashboards`.

[3] Android Developers [online] [cit. 2019-12-20]. About android app bundles. URL: `https://developer.android.com/guide/app-bundle`.

[4] Google Developers [online] [cit. 2019-12-20]. Play protect. URL: `https://developers.google.com/android/play-protect`.

[5] Android Open Source Project [online] [cit. 2019-12-20]. Secure an android device. URL: `https://source.android.com/security/`.

[6] Android Open Source Project [online] [cit. 2019-12-20]. Authentication. URL: `https://source.android.com/security/authentication`.

[7] Android Open Source Project [online] [cit. 2019-12-20]. Gatekeeper. URL: `https://source.android.com/security/authentication/gatekeeper`.

[8] Android Open Source Project [online] [cit. 2019-12-20]. Biometrics. URL: `https://source.android.com/security/biometric`.

[9] Android Open Source Project [online] [cit. 2019-12-20]. Measuring biometric unlock security. URL: `https://source.android.com/security/biometric/measure#metrics`.

[10] Android Open Source Project [online] [cit. 2019-12-20]. Android 10 compatibility definition. URL: `https://source.android.com/compatibility/android-cdd`.

[11] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The android platform security model. *arXiv preprint arXiv:1904.05572*, 2019.

[12] Android Open Source Project [online] [cit. 2019-12-20]. Hardware-backed keystore. URL: `https://source.android.com/security/keystore`.

[13] Android Open Source Project [online] [cit. 2019-12-20]. Key and id attestation. URL: `https://source.android.com/security/keystore/attestation`.

[14] Alfred J Menezes, Jonathan Katz, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography.* CRC press, 1996.

[15] M Brett, Pollice Gary, and West David. Head first object-oriented analysis and design. *O'Reilly*, 338:349, 2006.

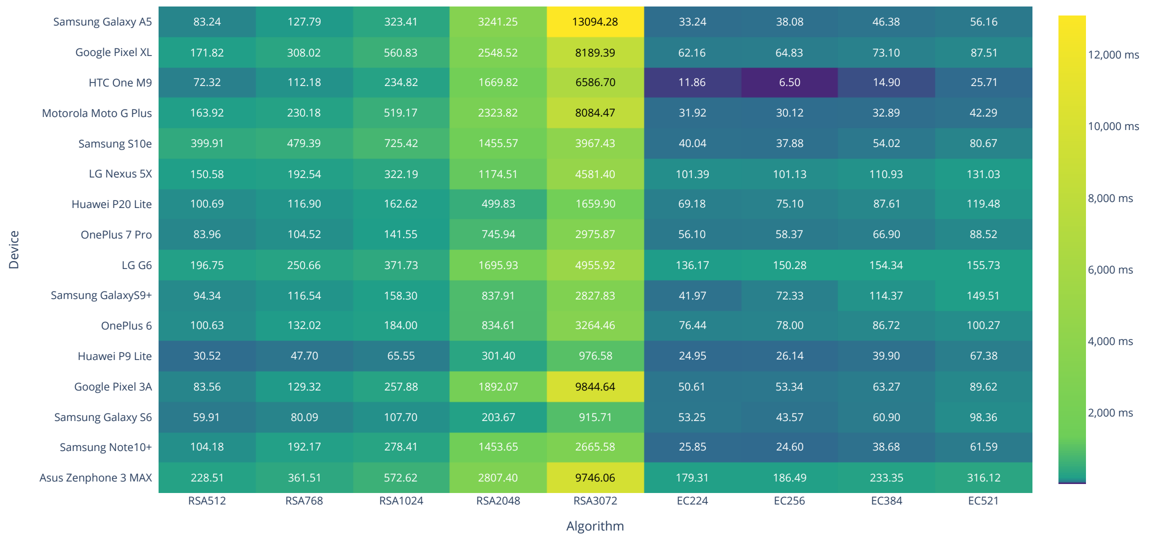
[16] Futuredapp. futuredapp/arkitekt, Mar 2020. URL: `https://github.com/futuredapp/arkitekt`.

List of symbols, physical constants and abbreviations

ICP	Interprocess communication
PIN	Personal identification number
SID	User Secure ID
TEE	Trusted execution environment
HMAC	Keyed-hash message authentication code
OS	Operating system
eMMC	Embedded Multi Media Card
RPMB	Replay Protected Memory Block
IAR	Imposter Accept Rate
SAR	Spoof Accept Rate
FAR	False Accept Rate
CDD	Android Compatibility Definition Document
HAL	Hardware Abstraction Layer
HIDL	Hardware Interface Definition Language
IMEI	International Mobile Equipment Identity
MEID	Mobile equipment identifier
TLS	Transport Layer Security
HTTP	Hypertext Transfer Protocol
gRPC	gRPC Remote Procedure Calls
MVVM	Model–view–viewmodel
API	Application Programming Interface
UI	User interface
REST	Representational state transfer
JSON	JavaScript Object Notation
JWT	JSON Web Token
CRUD	Create, Read, Update, Delete

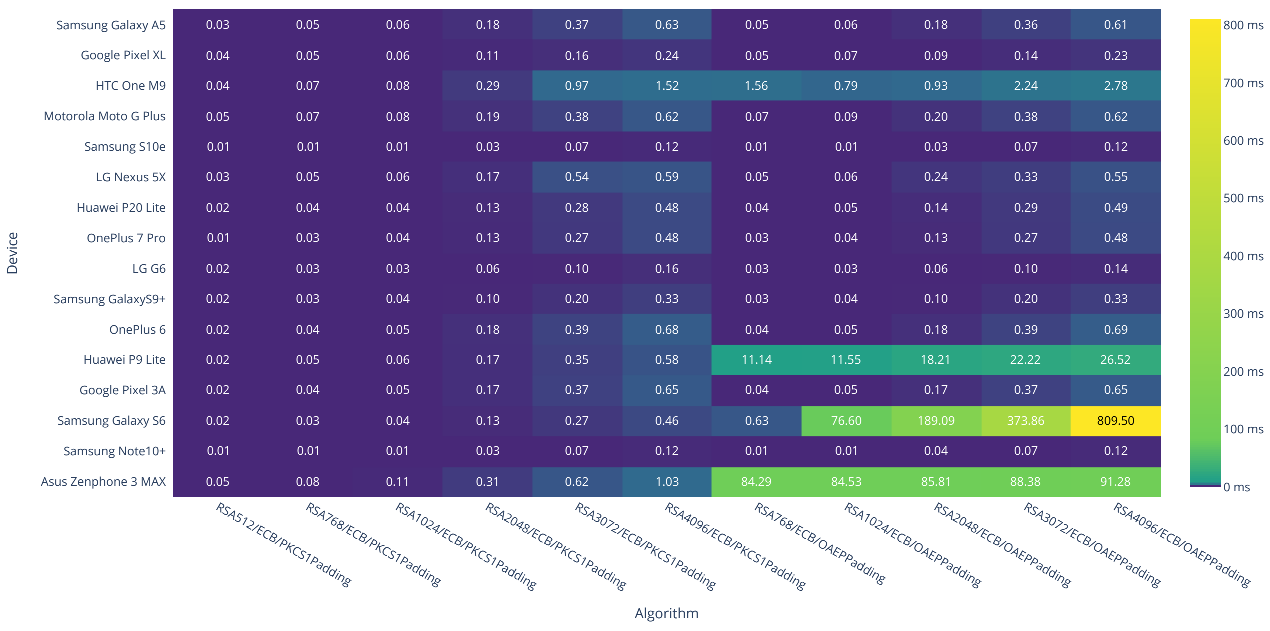
List of appendices

A Benchmark results



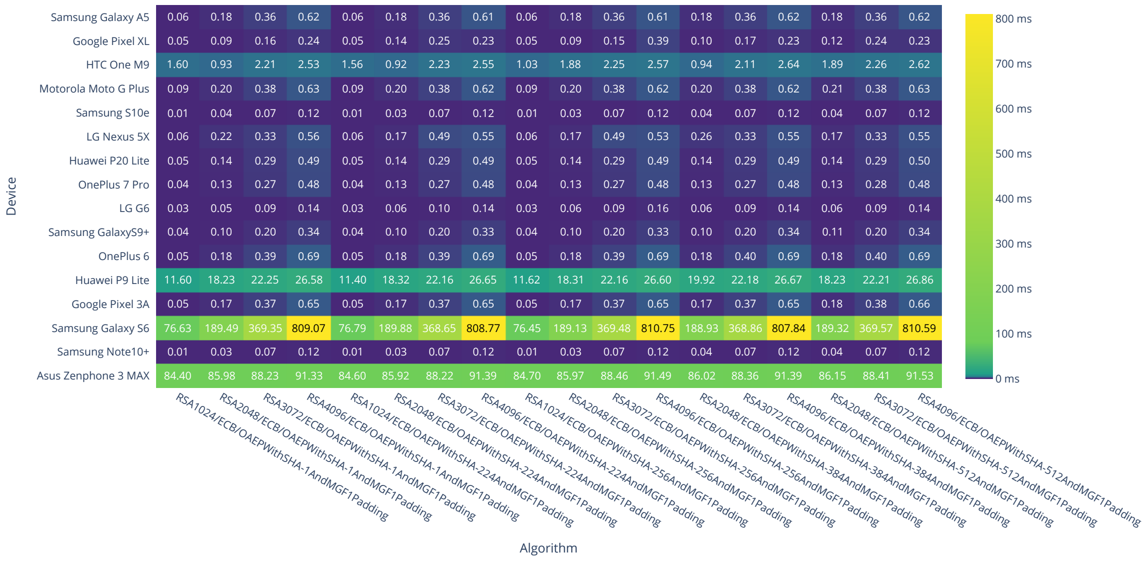
Device	RSA512	RSA768	RSA1024	RSA2048	RSA3072	EC224	EC256	EC384	EC521
Samsung Galaxy A5	83.24	127.79	323.41	3241.25	13094.28	33.24	38.08	46.38	56.16
Google Pixel XL	171.82	308.02	560.83	2548.52	8189.39	62.16	64.83	73.10	87.51
HTC One M9	72.32	112.18	234.82	1669.82	6586.70	11.86	6.50	14.90	25.71
Motorola Moto G Plus	163.92	230.18	519.17	2323.82	8084.47	31.92	30.12	32.89	42.29
Samsung S10e	399.91	479.39	725.42	1455.57	3967.43	40.04	37.88	54.02	80.67
LG Nexus 5X	150.58	192.54	322.19	1174.51	4581.40	101.39	101.13	110.93	131.03
Huawei P20 Lite	100.69	116.90	162.62	499.83	1659.90	69.18	75.10	87.61	119.48
OnePlus 7 Pro	83.96	104.52	141.55	745.94	2975.87	56.10	58.37	66.90	88.52
LG G6	196.75	250.66	371.73	1695.93	4955.92	136.17	150.28	154.34	155.73
Samsung GalaxyS9+	94.34	116.54	158.30	837.91	2827.83	41.97	72.33	114.37	149.51
OnePlus 6	100.63	132.02	184.00	834.61	3264.46	76.44	78.00	86.72	100.27
Huawei P9 Lite	30.52	47.70	65.55	301.40	976.58	24.95	26.14	39.90	67.38
Google Pixel 3A	83.56	129.32	257.88	1892.07	9844.64	50.61	53.34	63.27	89.62
Samsung Galaxy S6	59.91	80.09	107.70	203.67	915.71	53.25	43.57	60.90	98.36
Samsung Note10+	104.18	192.17	278.41	1453.65	2665.58	25.85	24.60	38.68	61.59
Asus Zenphone 3 MAX	228.51	361.51	572.62	2807.40	9746.06	179.31	186.49	233.35	316.12

Fig. A.1: Asymmetric key creation



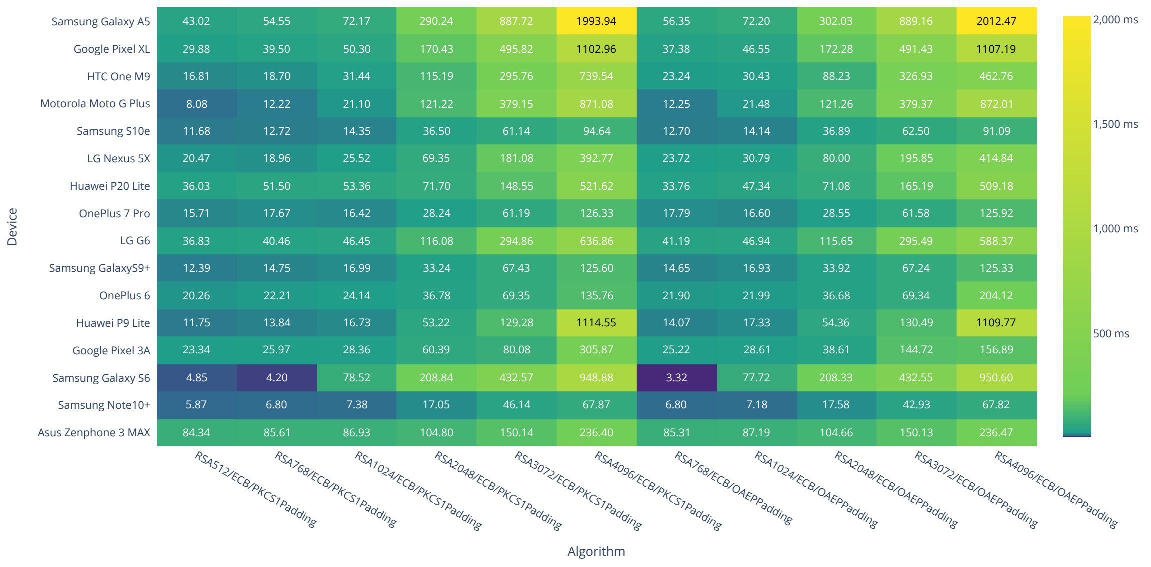
Device	RSA512/ECB/PKCS1Padding	RSA768/ECB/PKCS1Padding	RSA1024/ECB/PKCS1Padding	RSA2048/ECB/PKCS1Padding	RSA3072/ECB/PKCS1Padding	RSA4096/ECB/PKCS1Padding	RSA768/ECB/OAEPPadding	RSA1024/ECB/OAEPPadding	RSA2048/ECB/OAEPPadding	RSA3072/ECB/OAEPPadding	RSA4096/ECB/OAEPPadding
Samsung Galaxy A5	0.03	0.05	0.06	0.18	0.37	0.63	0.05	0.06	0.18	0.36	0.61
Google Pixel XL	0.04	0.05	0.06	0.11	0.16	0.24	0.05	0.07	0.09	0.14	0.23
HTC One M9	0.04	0.07	0.08	0.29	0.97	1.52	1.56	0.79	0.93	2.24	2.78
Motorola Moto G Plus	0.05	0.07	0.08	0.19	0.38	0.62	0.07	0.09	0.20	0.38	0.62
Samsung S10e	0.01	0.01	0.01	0.03	0.07	0.12	0.01	0.01	0.03	0.07	0.12
LG Nexus 5X	0.03	0.05	0.06	0.17	0.54	0.59	0.05	0.06	0.24	0.33	0.55
Huawei P20 Lite	0.02	0.04	0.04	0.13	0.28	0.48	0.04	0.05	0.14	0.29	0.49
OnePlus 7 Pro	0.01	0.03	0.04	0.13	0.27	0.48	0.03	0.04	0.13	0.27	0.48
LG G6	0.02	0.03	0.03	0.06	0.10	0.16	0.03	0.03	0.06	0.10	0.14
Samsung GalaxyS9+	0.02	0.03	0.04	0.10	0.20	0.33	0.03	0.04	0.10	0.20	0.33
OnePlus 6	0.02	0.04	0.05	0.18	0.39	0.68	0.04	0.05	0.18	0.39	0.69
Huawei P9 Lite	0.02	0.05	0.06	0.17	0.35	0.58	11.14	11.55	18.21	22.22	26.52
Google Pixel 3A	0.02	0.04	0.05	0.17	0.37	0.65	0.04	0.05	0.17	0.37	0.65
Samsung Galaxy S6	0.02	0.03	0.04	0.13	0.27	0.46	0.63	76.60	189.09	373.86	809.50
Samsung Note10+	0.01	0.01	0.01	0.03	0.07	0.12	0.01	0.01	0.04	0.07	0.12
Asus Zenphone 3 MAX	0.05	0.08	0.11	0.31	0.62	1.03	84.29	84.53	85.81	88.38	91.28

Fig. A.2: Encryption using RSA/ECB with PKCS1 or OAEP Padding



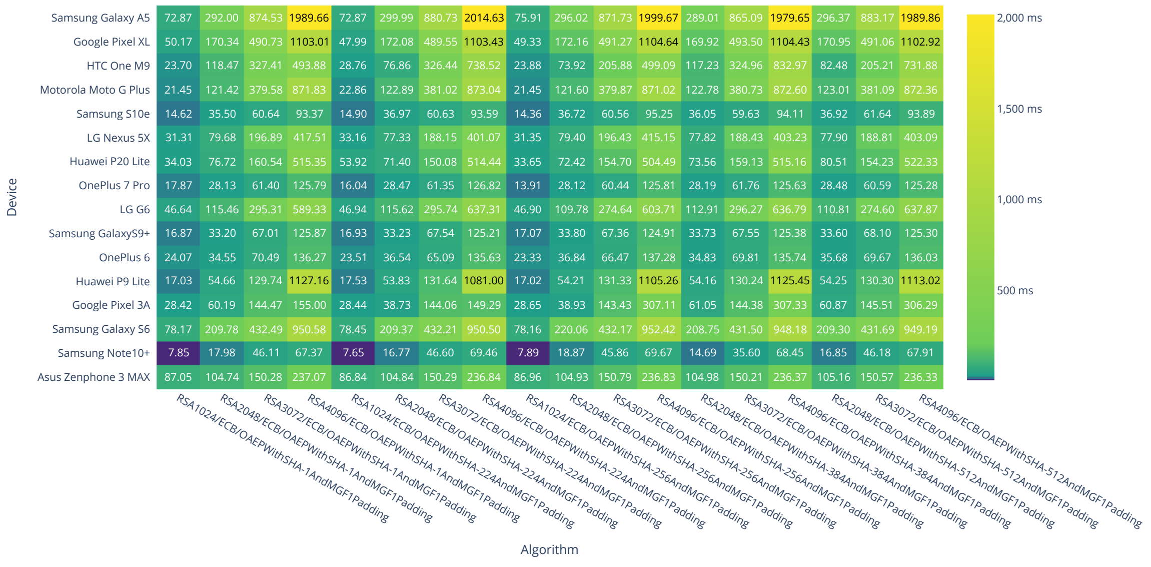
Device	RSA1024/ECB/OAEPWithSHA-1AndMGF1Padding	RSA2048/ECB/OAEPWithSHA-1AndMGF1Padding	RSA3072/ECB/OAEPWithSHA-1AndMGF1Padding	RSA4096/ECB/OAEPWithSHA-1AndMGF1Padding	RSA1024/ECB/OAEPWithSHA-224AndMGF1Padding	RSA2048/ECB/OAEPWithSHA-224AndMGF1Padding	RSA3072/ECB/OAEPWithSHA-224AndMGF1Padding	RSA4096/ECB/OAEPWithSHA-224AndMGF1Padding	RSA1024/ECB/OAEPWithSHA-256AndMGF1Padding	RSA2048/ECB/OAEPWithSHA-256AndMGF1Padding	RSA3072/ECB/OAEPWithSHA-256AndMGF1Padding	RSA4096/ECB/OAEPWithSHA-256AndMGF1Padding	RSA2048/ECB/OAEPWithSHA-384AndMGF1Padding	RSA3072/ECB/OAEPWithSHA-384AndMGF1Padding	RSA4096/ECB/OAEPWithSHA-384AndMGF1Padding	RSA2048/ECB/OAEPWithSHA-512AndMGF1Padding	RSA3072/ECB/OAEPWithSHA-512AndMGF1Padding	RSA4096/ECB/OAEPWithSHA-512AndMGF1Padding
Samsung Galaxy A5	0.06	0.18	0.36	0.62	0.06	0.18	0.36	0.61	0.06	0.18	0.36	0.61	0.18	0.36	0.62	0.18	0.36	0.62
Google Pixel XL	0.05	0.09	0.16	0.24	0.05	0.14	0.25	0.23	0.05	0.09	0.15	0.39	0.10	0.17	0.23	0.12	0.24	0.23
HTC One M9	1.60	0.93	2.21	2.53	1.56	0.92	2.23	2.55	1.03	1.88	2.25	2.57	0.94	2.11	2.64	1.89	2.26	2.62
Motorola Moto G Plus	0.09	0.20	0.38	0.63	0.09	0.20	0.38	0.62	0.09	0.20	0.38	0.62	0.20	0.38	0.62	0.21	0.38	0.63
Samsung S10e	0.01	0.04	0.07	0.12	0.01	0.03	0.07	0.12	0.01	0.03	0.07	0.12	0.04	0.07	0.12	0.04	0.07	0.12
LG Nexus 5X	0.06	0.22	0.33	0.56	0.06	0.17	0.49	0.55	0.06	0.17	0.49	0.53	0.26	0.33	0.55	0.17	0.33	0.55
Huawei P20 Lite	0.05	0.14	0.29	0.49	0.05	0.14	0.29	0.49	0.05	0.14	0.29	0.49	0.14	0.29	0.49	0.14	0.29	0.50
OnePlus 7 Pro	0.04	0.13	0.27	0.48	0.04	0.13	0.27	0.48	0.04	0.13	0.27	0.48	0.13	0.27	0.48	0.13	0.28	0.48
LG G6	0.03	0.05	0.09	0.14	0.03	0.06	0.10	0.14	0.03	0.06	0.09	0.16	0.06	0.09	0.14	0.06	0.09	0.14
Samsung GalaxyS9+	0.04	0.10	0.20	0.34	0.04	0.10	0.20	0.33	0.04	0.10	0.20	0.33	0.10	0.20	0.34	0.11	0.20	0.34
OnePlus 6	0.05	0.18	0.39	0.69	0.05	0.18	0.39	0.69	0.05	0.18	0.39	0.69	0.18	0.40	0.69	0.18	0.40	0.69
Huawei P9 Lite	11.60	18.23	22.25	26.58	11.40	18.32	22.16	26.65	11.62	18.31	22.16	26.60	19.92	22.18	26.67	18.23	22.21	26.86
Google Pixel 3A	0.05	0.17	0.37	0.65	0.05	0.17	0.37	0.65	0.05	0.17	0.37	0.65	0.17	0.37	0.65	0.18	0.38	0.66
Samsung Galaxy S6	76.63	189.49	369.35	809.07	76.79	189.88	368.65	808.77	76.45	189.13	369.48	810.75	188.93	368.86	807.84	189.32	369.57	810.59
Samsung Note10+	0.01	0.03	0.07	0.12	0.01	0.03	0.07	0.12	0.01	0.03	0.07	0.12	0.04	0.07	0.12	0.04	0.07	0.12
Asus Zenphone 3 MAX	84.40	85.98	88.23	91.33	84.60	85.92	88.22	91.39	84.70	85.97	88.46	91.49	86.02	88.36	91.39	86.15	88.41	91.53

Fig. A.3: Encryption using RSA/ECB/OAEPwithSHA and MGF1 Padding



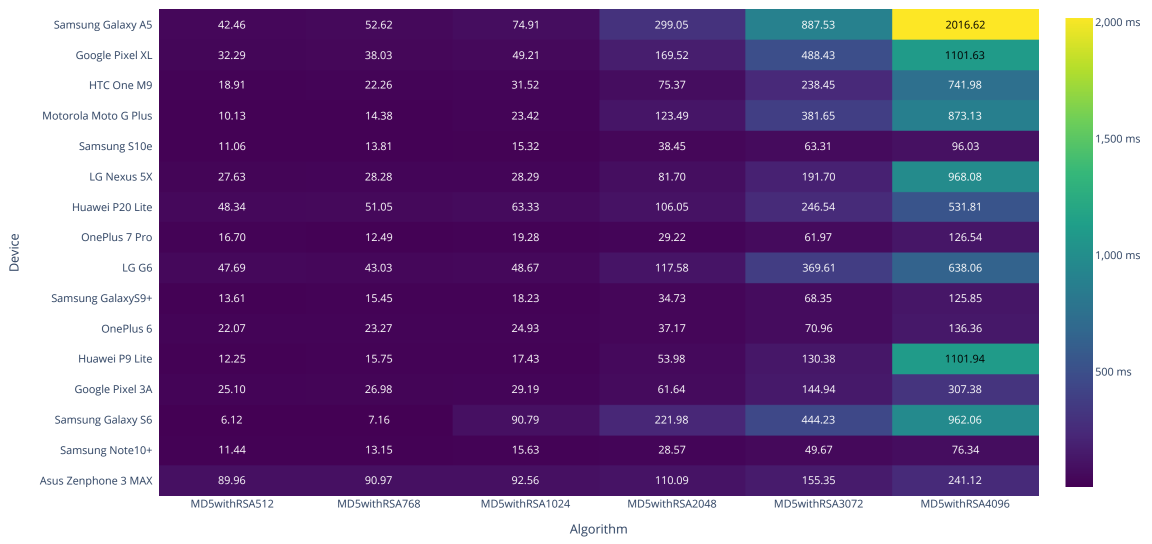
Device	RSA512/ECB/PKCS1Padding	RSA768/ECB/PKCS1Padding	RSA1024/ECB/PKCS1Padding	RSA2048/ECB/PKCS1Padding	RSA3072/ECB/PKCS1Padding	RSA4096/ECB/PKCS1Padding	RSA768/ECB/OAEPPadding	RSA1024/ECB/OAEPPadding	RSA2048/ECB/OAEPPadding	RSA3072/ECB/OAEPPadding	RSA4096/ECB/OAEPPadding
Samsung Galaxy A5	43.02	54.55	72.17	290.24	887.72	1993.94	56.35	72.20	302.03	889.16	2012.47
Google Pixel XL	29.88	39.50	50.30	170.43	495.82	1102.96	37.38	46.55	172.28	491.43	1107.19
HTC One M9	16.81	18.70	31.44	115.19	295.76	739.54	23.24	30.43	88.23	326.93	462.76
Motorola Moto G Plus	8.08	12.22	21.10	121.22	379.15	871.08	12.25	21.48	121.26	379.37	872.01
Samsung S10e	11.68	12.72	14.35	36.50	61.14	94.64	12.70	14.14	36.89	62.50	91.09
LG Nexus 5X	20.47	18.96	25.52	69.35	181.08	392.77	23.72	30.79	80.00	195.85	414.84
Huawei P20 Lite	36.03	51.50	53.36	71.70	148.55	521.62	33.76	47.34	71.08	165.19	509.18
OnePlus 7 Pro	15.71	17.67	16.42	28.24	61.19	126.33	17.79	16.60	28.55	61.58	125.92
LG G6	36.83	40.46	46.45	116.08	294.86	636.86	41.19	46.94	115.65	295.49	588.37
Samsung GalaxyS9+	12.39	14.75	16.99	33.24	67.43	125.60	14.65	16.93	33.92	67.24	125.33
OnePlus 6	20.26	22.21	24.14	36.78	69.35	135.76	21.90	21.99	36.68	69.34	204.12
Huawei P9 Lite	11.75	13.84	16.73	53.22	129.28	1114.55	14.07	17.33	54.36	130.49	1109.77
Google Pixel 3A	23.34	25.97	28.36	60.39	80.08	305.87	25.22	28.61	38.61	144.72	156.89
Samsung Galaxy S6	4.85	4.20	78.52	208.84	432.57	948.88	3.32	77.72	208.33	432.55	950.60
Samsung Note10+	5.87	6.80	7.38	17.05	46.14	67.87	6.80	7.18	17.58	42.93	67.82
Asus Zenphone 3 MAX	84.34	85.61	86.93	104.80	150.14	236.40	85.31	87.19	104.66	150.13	236.47

Fig. A.4: Decryption using RSA/ECB with PKCS1 or OAEP Padding



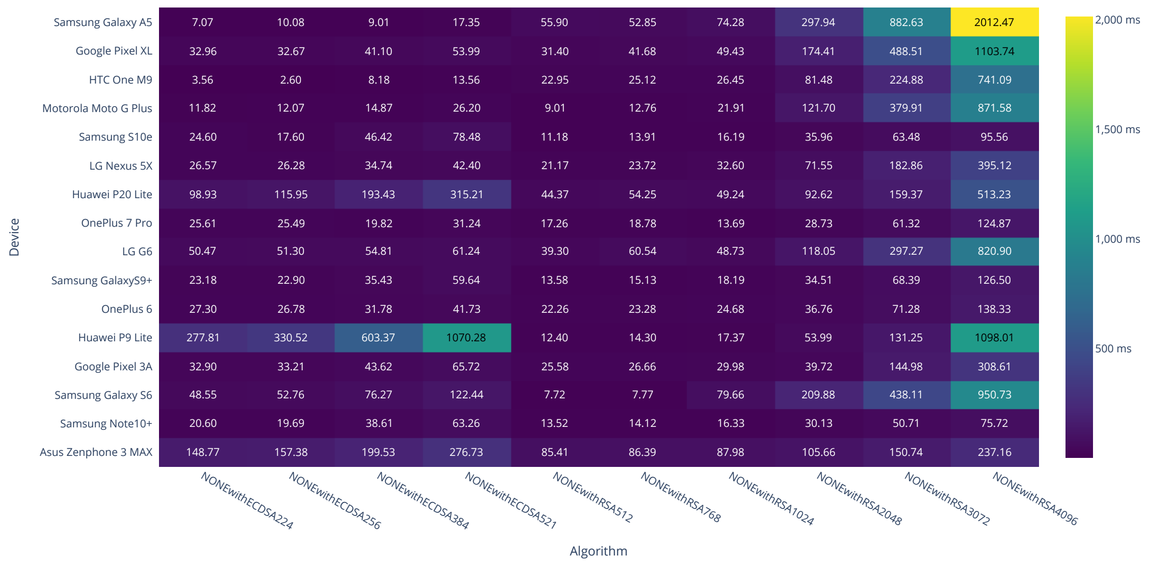
Samsung Galaxy A5	72.87	292.00	874.53	1989.66	72.87	299.99	880.73	2014.63	75.91	296.02	871.73	1999.67	289.01	865.09	1979.65	296.37	883.17	1989.86
Google Pixel XL	50.17	170.34	490.73	1103.01	47.99	172.08	489.55	1103.43	49.33	172.16	491.27	1104.64	169.92	493.50	1104.43	170.95	491.06	1102.92
HTC One M9	23.70	118.47	327.41	493.88	28.76	76.86	326.44	738.52	23.88	73.92	205.88	499.09	117.23	324.96	832.97	82.48	205.21	731.88
Motorola Moto G Plus	21.45	121.42	379.58	871.83	22.86	122.89	381.02	873.04	21.45	121.60	379.87	871.02	122.78	380.73	872.60	123.01	381.09	872.36
Samsung S10e	14.62	35.50	60.64	93.37	14.90	36.97	60.63	93.59	14.36	36.72	60.56	95.25	36.05	59.63	94.11	36.92	61.64	93.89
LG Nexus 5X	31.31	79.68	196.89	417.51	33.16	77.33	188.15	401.07	31.35	79.40	196.43	415.15	77.82	188.43	403.23	77.90	188.81	403.09
Huawei P20 Lite	34.03	76.72	160.54	515.35	53.92	71.40	150.08	514.44	33.65	72.42	154.70	504.49	73.56	159.13	515.16	80.51	154.23	522.33
OnePlus 7 Pro	17.87	28.13	61.40	125.79	16.04	28.47	61.35	126.82	13.91	28.12	60.44	125.81	28.19	61.76	125.63	28.48	60.59	125.28
LG G6	46.64	115.46	295.31	589.33	46.94	115.62	295.74	637.31	46.90	109.78	274.64	603.71	112.91	296.27	636.79	110.81	274.60	637.87
Samsung GalaxyS9+	16.87	33.20	67.01	125.87	16.93	33.23	67.54	125.21	17.07	33.80	67.36	124.91	33.73	67.55	125.38	33.60	68.10	125.30
OnePlus 6	24.07	34.55	70.49	136.27	23.51	36.54	65.09	135.63	23.33	36.84	66.47	137.28	34.83	69.81	135.74	35.68	69.67	136.03
Huawei P9 Lite	17.03	54.66	129.74	1127.16	17.53	53.83	131.64	1081.00	17.02	54.21	131.33	1105.26	54.16	130.24	1125.45	54.25	130.30	1113.02
Google Pixel 3A	28.42	60.19	144.47	155.00	28.44	38.73	144.06	149.29	28.65	38.93	143.43	307.11	61.05	144.38	307.33	60.87	145.51	306.29
Samsung Galaxy S6	78.17	209.78	432.49	950.58	78.45	209.37	432.21	950.50	78.16	220.06	432.17	952.42	208.75	431.50	948.18	209.30	431.69	949.19
Samsung Note10+	7.85	17.98	46.11	67.37	7.65	16.77	46.60	69.46	7.89	18.87	45.86	69.67	14.69	35.60	68.45	16.85	46.18	67.91
Asus Zenphone 3 MAX	87.05	104.74	150.28	237.07	86.84	104.84	150.29	236.84	86.96	104.93	150.79	236.83	104.98	150.21	236.37	105.16	150.57	236.33

Fig. A.5: Decryption using RSA/ECB/OAEPwithSHA and MGF1 Padding



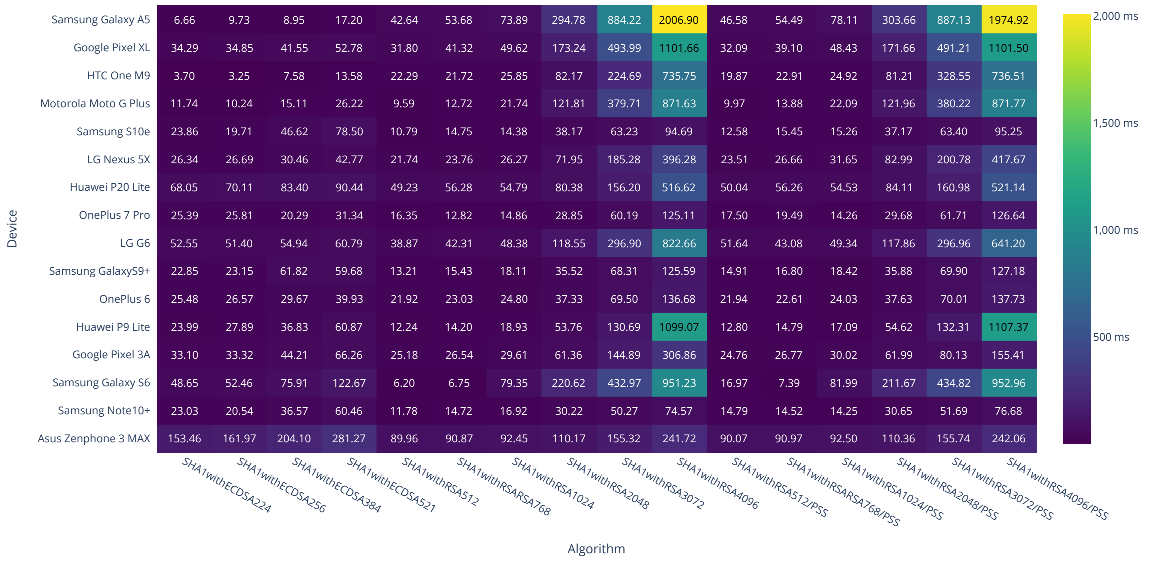
Device	MD5withRSA512	MD5withRSA768	MD5withRSA1024	MD5withRSA2048	MD5withRSA3072	MD5withRSA4096
Samsung Galaxy A5	42.46	52.62	74.91	299.05	887.53	2016.62
Google Pixel XL	32.29	38.03	49.21	169.52	488.43	1101.63
HTC One M9	18.91	22.26	31.52	75.37	238.45	741.98
Motorola Moto G Plus	10.13	14.38	23.42	123.49	381.65	873.13
Samsung S10e	11.06	13.81	15.32	38.45	63.31	96.03
LG Nexus 5X	27.63	28.28	28.29	81.70	191.70	968.08
Huawei P20 Lite	48.34	51.05	63.33	106.05	246.54	531.81
OnePlus 7 Pro	16.70	12.49	19.28	29.22	61.97	126.54
LG G6	47.69	43.03	48.67	117.58	369.61	638.06
Samsung GalaxyS9+	13.61	15.45	18.23	34.73	68.35	125.85
OnePlus 6	22.07	23.27	24.93	37.17	70.96	136.36
Huawei P9 Lite	12.25	15.75	17.43	53.98	130.38	1101.94
Google Pixel 3A	25.10	26.98	29.19	61.64	144.94	307.38
Samsung Galaxy S6	6.12	7.16	90.79	221.98	444.23	962.06
Samsung Note10+	11.44	13.15	15.63	28.57	49.67	76.34
Asus Zenphone 3 MAX	89.96	90.97	92.56	110.09	155.35	241.12

Fig. A.6: Signature using RSA with MD5



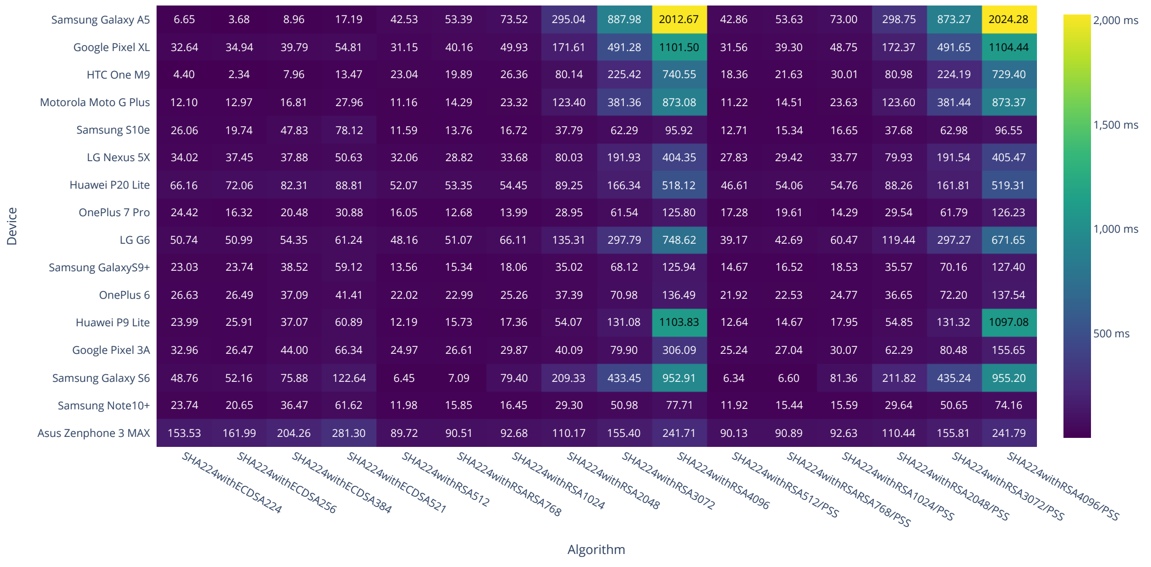
Device	NONEwithECDSA224	NONEwithECDSA256	NONEwithECDSA384	NONEwithECDSA521	NONEwithRSA512	NONEwithRSA768	NONEwithRSA1024	NONEwithRSA2048	NONEwithRSA3072	NONEwithRSA4096
Samsung Galaxy A5	7.07	10.08	9.01	17.35	55.90	52.85	74.28	297.94	882.63	2012.47
Google Pixel XL	32.96	32.67	41.10	53.99	31.40	41.68	49.43	174.41	488.51	1103.74
HTC One M9	3.56	2.60	8.18	13.56	22.95	25.12	26.45	81.48	224.88	741.09
Motorola Moto G Plus	11.82	12.07	14.87	26.20	9.01	12.76	21.91	121.70	379.91	871.58
Samsung S10e	24.60	17.60	46.42	78.48	11.18	13.91	16.19	35.96	63.48	95.56
LG Nexus 5X	26.57	26.28	34.74	42.40	21.17	23.72	32.60	71.55	182.86	395.12
Huawei P20 Lite	98.93	115.95	193.43	315.21	44.37	54.25	49.24	92.62	159.37	513.23
OnePlus 7 Pro	25.61	25.49	19.82	31.24	17.26	18.78	13.69	28.73	61.32	124.87
LG G6	50.47	51.30	54.81	61.24	39.30	60.54	48.73	118.05	297.27	820.90
Samsung GalaxyS9+	23.18	22.90	35.43	59.64	13.58	15.13	18.19	34.51	68.39	126.50
OnePlus 6	27.30	26.78	31.78	41.73	22.26	23.28	24.68	36.76	71.28	138.33
Huawei P9 Lite	277.81	330.52	603.37	1070.28	12.40	14.30	17.37	53.99	131.25	1098.01
Google Pixel 3A	32.90	33.21	43.62	65.72	25.58	26.66	29.98	39.72	144.98	308.61
Samsung Galaxy S6	48.55	52.76	76.27	122.44	7.72	7.77	79.66	209.88	438.11	950.73
Samsung Note10+	20.60	19.69	38.61	63.26	13.52	14.12	16.33	30.13	50.71	75.72
Asus Zenphone 3 MAX	148.77	157.38	199.53	276.73	85.41	86.39	87.98	105.66	150.74	237.16

Fig. A.7: Signature using RSA or EC without hash function



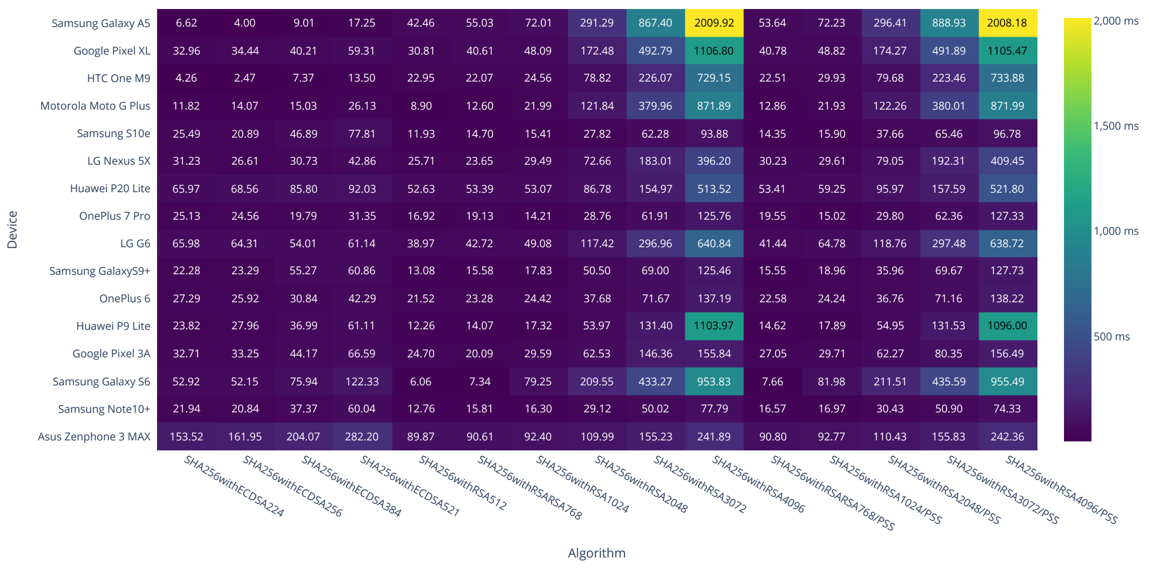
Device	SHA1withECDSA224	SHA1withECDSA256	SHA1withECDSA384	SHA1withECDSA521	SHA1withRSA512	SHA1withRSARSA768	SHA1withRSA1024	SHA1withRSA2048	SHA1withRSA3072	SHA1withRSA4096	SHA1withRSA512/PSS	SHA1withRSARSA768/PSS	SHA1withRSA1024/PSS	SHA1withRSA2048/PSS	SHA1withRSA3072/PSS	SHA1withRSA4096/PSS
Samsung Galaxy A5	6.66	9.73	8.95	17.20	42.64	53.68	73.89	294.78	884.22	2006.90	46.58	54.49	78.11	303.66	887.13	1974.92
Google Pixel XL	34.29	34.85	41.55	52.78	31.80	41.32	49.62	173.24	493.99	1101.66	32.09	39.10	48.43	171.66	491.21	1101.50
HTC One M9	3.70	3.25	7.58	13.58	22.29	21.72	25.85	82.17	224.69	735.75	19.87	22.91	24.92	81.21	328.55	736.51
Motorola Moto G Plus	11.74	10.24	15.11	26.22	9.59	12.72	21.74	121.81	379.71	871.63	9.97	13.88	22.09	121.96	380.22	871.77
Samsung S10e	23.86	19.71	46.62	78.50	10.79	14.75	14.38	38.17	63.23	94.69	12.58	15.45	15.26	37.17	63.40	95.25
LG Nexus 5X	26.34	26.69	30.46	42.77	21.74	23.76	26.27	71.95	185.28	396.28	23.51	26.66	31.65	82.99	200.78	417.67
Huawei P20 Lite	68.05	70.11	83.40	90.44	49.23	56.28	54.79	80.38	156.20	516.62	50.04	56.26	54.53	84.11	160.98	521.14
OnePlus 7 Pro	25.39	25.81	20.29	31.34	16.35	12.82	14.86	28.85	60.19	125.11	17.50	19.49	14.26	29.68	61.71	126.64
LG G6	52.55	51.40	54.94	60.79	38.87	42.31	48.38	118.55	296.90	822.66	51.64	43.08	49.34	117.86	296.96	641.20
Samsung GalaxyS9+	22.85	23.15	61.82	59.68	13.21	15.43	18.11	35.52	68.31	125.59	14.91	16.80	18.42	35.88	69.90	127.18
OnePlus 6	25.48	26.57	29.67	39.93	21.92	23.03	24.80	37.33	69.50	136.68	21.94	22.61	24.03	37.63	70.01	137.73
Huawei P9 Lite	23.99	27.89	36.83	60.87	12.24	14.20	18.93	53.76	130.69	1099.07	12.80	14.79	17.09	54.62	132.31	1107.37
Google Pixel 3A	33.10	33.32	44.21	66.26	25.18	26.54	29.61	61.36	144.89	306.86	24.76	26.77	30.02	61.99	80.13	155.41
Samsung Galaxy S6	48.65	52.46	75.91	122.67	6.20	6.75	79.35	220.62	432.97	951.23	16.97	7.39	81.99	211.67	434.82	952.96
Samsung Note10+	23.03	20.54	36.57	60.46	11.78	14.72	16.92	30.22	50.27	74.57	14.79	14.52	14.25	30.65	51.69	76.68
Asus Zenphone 3 MAX	153.46	161.97	204.10	281.27	89.96	90.87	92.45	110.17	155.32	241.72	90.07	90.97	92.50	110.36	155.74	242.06

Fig. A.8: Signature using RSA or EC with SHA1



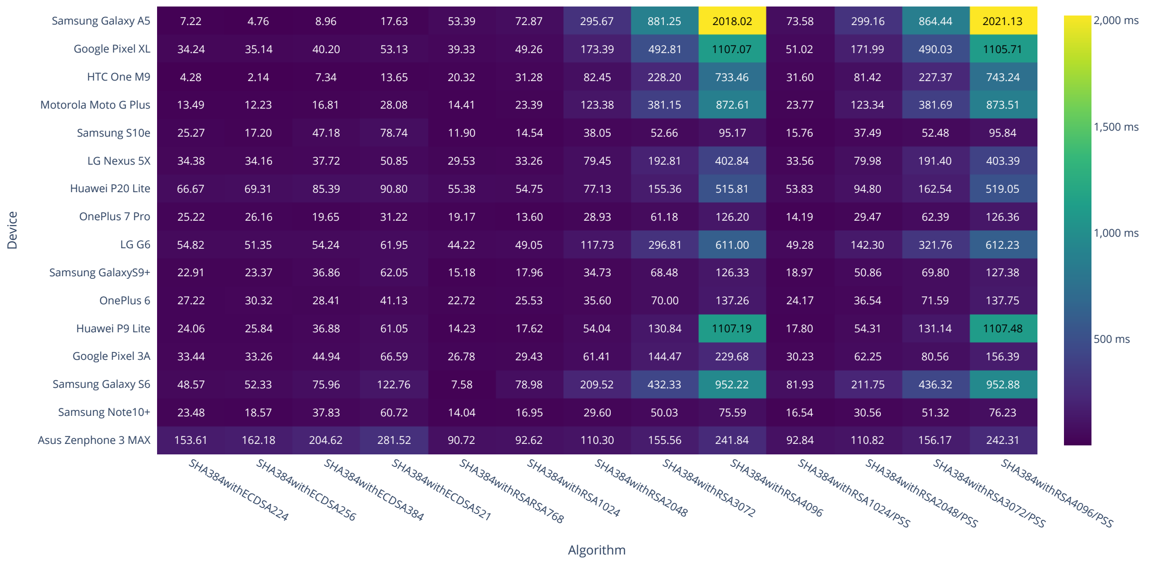
Device	SHA224withECDSA224	SHA224withECDSA256	SHA224withECDSA384	SHA224withECDSA521	SHA224withRSA512	SHA224withRSARSA768	SHA224withRSA1024	SHA224withRSA2048	SHA224withRSA3072	SHA224withRSA4096	SHA224withRSA512/PSS	SHA224withRSARSA768/PSS	SHA224withRSA1024/PSS	SHA224withRSA2048/PSS	SHA224withRSA3072/PSS	SHA224withRSA4096/PSS
Samsung Galaxy A5	6.65	3.68	8.96	17.19	42.53	53.39	73.52	295.04	887.98	2012.67	42.86	53.63	73.00	298.75	873.27	2024.28
Google Pixel XL	32.64	34.94	39.79	54.81	31.15	40.16	49.93	171.61	491.28	1101.50	31.56	39.30	48.75	172.37	491.65	1104.44
HTC One M9	4.40	2.34	7.96	13.47	23.04	19.89	26.36	80.14	225.42	740.55	18.36	21.63	30.01	80.98	224.19	729.40
Motorola Moto G Plus	12.10	12.97	16.81	27.96	11.16	14.29	23.32	123.40	381.36	873.08	11.22	14.51	23.63	123.60	381.44	873.37
Samsung S10e	26.06	19.74	47.83	78.12	11.59	13.76	16.72	37.79	62.29	95.92	12.71	15.34	16.65	37.68	62.98	96.55
LG Nexus 5X	34.02	37.45	37.88	50.63	32.06	28.82	33.68	80.03	191.93	404.35	27.83	29.42	33.77	79.93	191.54	405.47
Huawei P20 Lite	66.16	72.06	82.31	88.81	52.07	53.35	54.45	89.25	166.34	518.12	46.61	54.06	54.76	88.26	161.81	519.31
OnePlus 7 Pro	24.42	16.32	20.48	30.88	16.05	12.68	13.99	28.95	61.54	125.80	17.28	19.61	14.29	29.54	61.79	126.23
LG G6	50.74	50.99	54.35	61.24	48.16	51.07	66.11	135.31	297.79	748.62	39.17	42.69	60.47	119.44	297.27	671.65
Samsung GalaxyS9+	23.03	23.74	38.52	59.12	13.56	15.34	18.06	35.02	68.12	125.94	14.67	16.52	18.53	35.57	70.16	127.40
OnePlus 6	26.63	26.49	37.09	41.41	22.02	22.99	25.26	37.39	70.98	136.49	21.92	22.53	24.77	36.65	72.20	137.54
Huawei P9 Lite	23.99	25.91	37.07	60.89	12.19	15.73	17.36	54.07	131.08	1103.83	12.64	14.67	17.95	54.85	131.32	1097.08
Google Pixel 3A	32.96	26.47	44.00	66.34	24.97	26.61	29.87	40.09	79.90	306.09	25.24	27.04	30.07	62.29	80.48	155.65
Samsung Galaxy S6	48.76	52.16	75.88	122.64	6.45	7.09	79.40	209.33	433.45	952.91	6.34	6.60	81.36	211.82	435.24	955.20
Samsung Note10+	23.74	20.65	36.47	61.62	11.98	15.85	16.45	29.30	50.98	77.71	11.92	15.44	15.59	29.64	50.65	74.16
Asus Zenphone 3 MAX	153.53	161.99	204.26	281.30	89.72	90.51	92.68	110.17	155.40	241.71	90.13	90.89	92.63	110.44	155.81	241.79

Fig. A.9: Signature using RSA or EC with SHA224



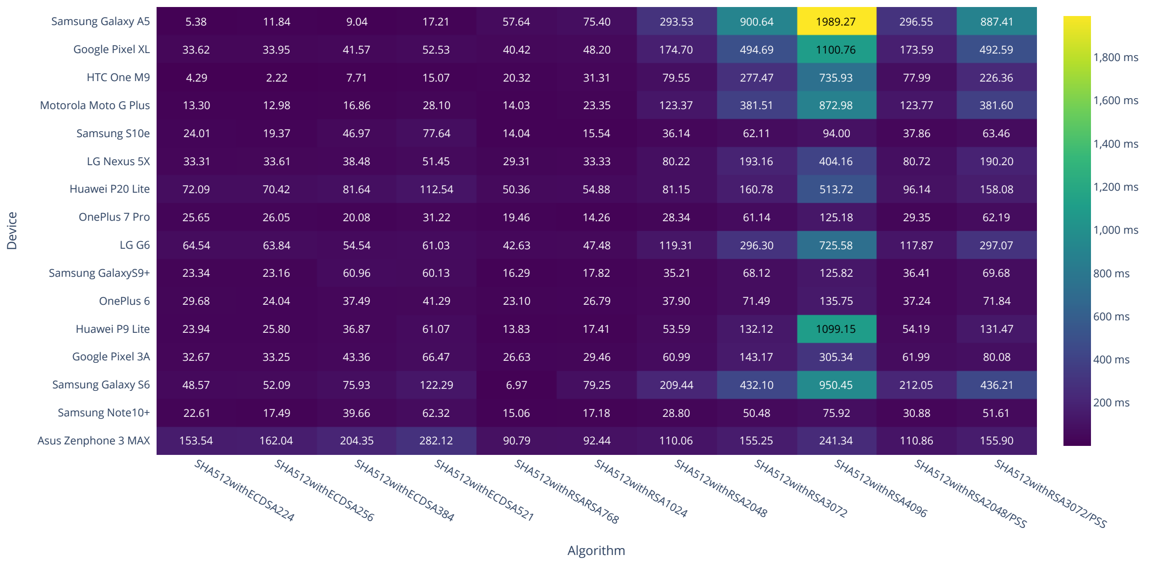
Device	SHA256withECDSA224	SHA256withECDSA256	SHA256withECDSA384	SHA256withECDSA521	SHA256withRSA512	SHA256withRSARSA768	SHA256withRSA1024	SHA256withRSA2048	SHA256withRSA3072	SHA256withRSA4096	SHA256withRSARSA768/PSS	SHA256withRSA1024/PSS	SHA256withRSA2048/PSS	SHA256withRSA3072/PSS	SHA256withRSA4096/PSS
Samsung Galaxy A5	6.62	4.00	9.01	17.25	42.46	55.03	72.01	291.29	867.40	2009.92	53.64	72.23	296.41	888.93	2008.18
Google Pixel XL	32.96	34.44	40.21	59.31	30.81	40.61	48.09	172.48	492.79	1106.80	40.78	48.82	174.27	491.89	1105.47
HTC One M9	4.26	2.47	7.37	13.50	22.95	22.07	24.56	78.82	226.07	729.15	22.51	29.93	79.68	223.46	733.88
Motorola Moto G Plus	11.82	14.07	15.03	26.13	8.90	12.60	21.99	121.84	379.96	871.89	12.86	21.93	122.26	380.01	871.99
Samsung S10e	25.49	20.89	46.89	77.81	11.93	14.70	15.41	27.82	62.28	93.88	14.35	15.90	37.66	65.46	96.78
LG Nexus 5X	31.23	26.61	30.73	42.86	25.71	23.65	29.49	72.66	183.01	396.20	30.23	29.61	79.05	192.31	409.45
Huawei P20 Lite	65.97	68.56	85.80	92.03	52.63	53.39	53.07	86.78	154.97	513.52	53.41	59.25	95.97	157.59	521.80
OnePlus 7 Pro	25.13	24.56	19.79	31.35	16.92	19.13	14.21	28.76	61.91	125.76	19.55	15.02	29.80	62.36	127.33
LG G6	65.98	64.31	54.01	61.14	38.97	42.72	49.08	117.42	296.96	640.84	41.44	64.78	118.76	297.48	638.72
Samsung GalaxyS9+	22.28	23.29	55.27	60.86	13.08	15.58	17.83	50.50	69.00	125.46	15.55	18.96	35.96	69.67	127.73
OnePlus 6	27.29	25.92	30.84	42.29	21.52	23.28	24.42	37.68	71.67	137.19	22.58	24.24	36.76	71.16	138.22
Huawei P9 Lite	23.82	27.96	36.99	61.11	12.26	14.07	17.32	53.97	131.40	1103.97	14.62	17.89	54.95	131.53	1096.00
Google Pixel 3A	32.71	33.25	44.17	66.59	24.70	20.09	29.59	62.53	146.36	155.84	27.05	29.71	62.27	80.35	156.49
Samsung Galaxy S6	52.92	52.15	75.94	122.33	6.06	7.34	79.25	209.55	433.27	953.83	7.66	81.98	211.51	435.59	955.49
Samsung Note10+	21.94	20.84	37.37	60.04	12.76	15.81	16.30	29.12	50.02	77.79	16.57	16.97	30.43	50.90	74.33
Asus Zenphone 3 MAX	153.52	161.95	204.07	282.20	89.87	90.61	92.40	109.99	155.23	241.89	90.80	92.77	110.43	155.83	242.36

Fig. A.10: Signature using RSA or EC with SHA256



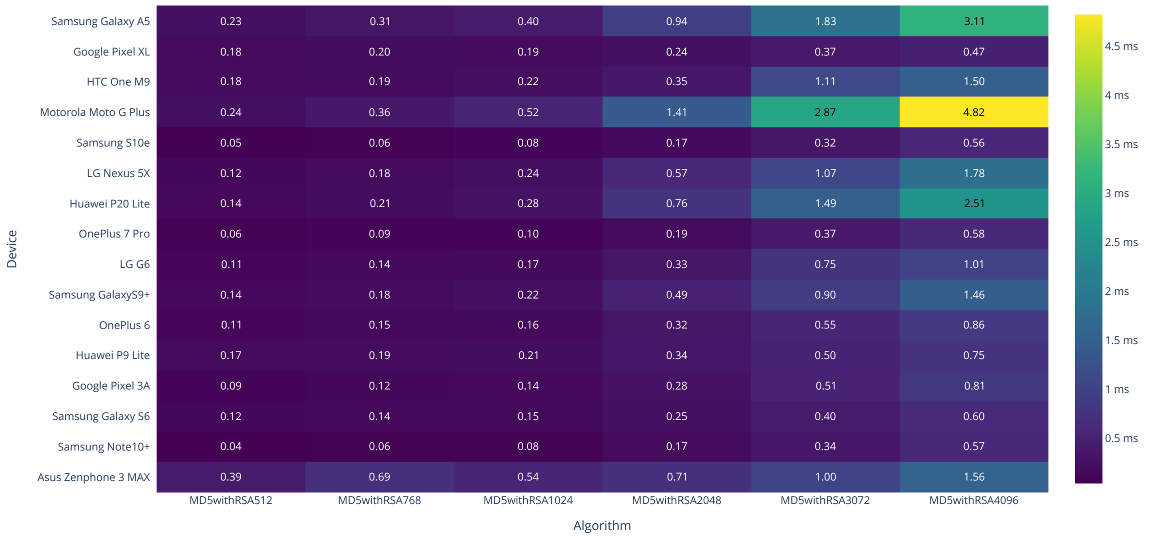
Device	SHA384withECDSA224	SHA384withECDSA256	SHA384withECDSA384	SHA384withECDSA521	SHA384withRSARSA768	SHA384withRSA1024	SHA384withRSA2048	SHA384withRSA3072	SHA384withRSA4096	SHA384withRSA1024/PSS	SHA384withRSA2048/PSS	SHA384withRSA3072/PSS	SHA384withRSA4096/PSS
Samsung Galaxy A5	7.22	4.76	8.96	17.63	53.39	72.87	295.67	881.25	2018.02	73.58	299.16	864.44	2021.13
Google Pixel XL	34.24	35.14	40.20	53.13	39.33	49.26	173.39	492.81	1107.07	51.02	171.99	490.03	1105.71
HTC One M9	4.28	2.14	7.34	13.65	20.32	31.28	82.45	228.20	733.46	31.60	81.42	227.37	743.24
Motorola Moto G Plus	13.49	12.23	16.81	28.08	14.41	23.39	123.38	381.15	872.61	23.77	123.34	381.69	873.51
Samsung S10e	25.27	17.20	47.18	78.74	11.90	14.54	38.05	52.66	95.17	15.76	37.49	52.48	95.84
LG Nexus 5X	34.38	34.16	37.72	50.85	29.53	33.26	79.45	192.81	402.84	33.56	79.98	191.40	403.39
Huawei P20 Lite	66.67	69.31	85.39	90.80	55.38	54.75	77.13	155.36	515.81	53.83	94.80	162.54	519.05
OnePlus 7 Pro	25.22	26.16	19.65	31.22	19.17	13.60	28.93	61.18	126.20	14.19	29.47	62.39	126.36
LG G6	54.82	51.35	54.24	61.95	44.22	49.05	117.73	296.81	611.00	49.28	142.30	321.76	612.23
Samsung GalaxyS9+	22.91	23.37	36.86	62.05	15.18	17.96	34.73	68.48	126.33	18.97	50.86	69.80	127.38
OnePlus 6	27.22	30.32	28.41	41.13	22.72	25.53	35.60	70.00	137.26	24.17	36.54	71.59	137.75
Huawei P9 Lite	24.06	25.84	36.88	61.05	14.23	17.62	54.04	130.84	1107.19	17.80	54.31	131.14	1107.48
Google Pixel 3A	33.44	33.26	44.94	66.59	26.78	29.43	61.41	144.47	229.68	30.23	62.25	80.56	156.39
Samsung Galaxy S6	48.57	52.33	75.96	122.76	7.58	78.98	209.52	432.33	952.22	81.93	211.75	436.32	952.88
Samsung Note10+	23.48	18.57	37.83	60.72	14.04	16.95	29.60	50.03	75.59	16.54	30.56	51.32	76.23
Asus Zenphone 3 MAX	153.61	162.18	204.62	281.52	90.72	92.62	110.30	155.56	241.84	92.84	110.82	156.17	242.31

Fig. A.11: Signature using RSA or EC with SHA384



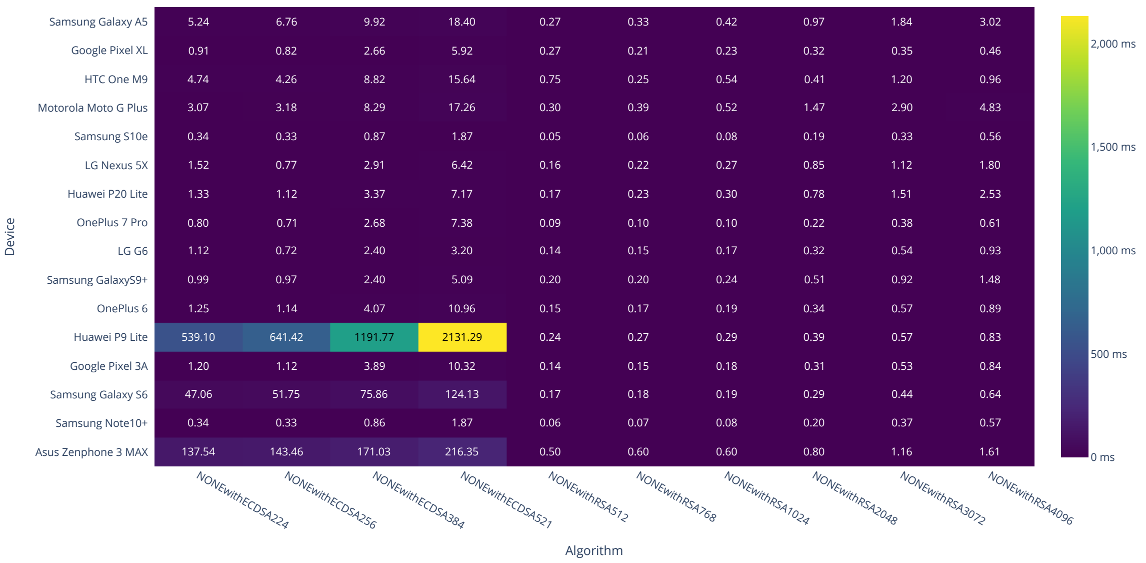
Device	SHA512withECDSA224	SHA512withECDSA256	SHA512withECDSA384	SHA512withECDSA521	SHA512withRSARSA768	SHA512withRSA1024	SHA512withRSA2048	SHA512withRSA3072	SHA512withRSA4096	SHA512withRSA2048/PSS	SHA512withRSA3072/PSS
Samsung Galaxy A5	5.38	11.84	9.04	17.21	57.64	75.40	293.53	900.64	1989.27	296.55	887.41
Google Pixel XL	33.62	33.95	41.57	52.53	40.42	48.20	174.70	494.69	1100.76	173.59	492.59
HTC One M9	4.29	2.22	7.71	15.07	20.32	31.31	79.55	277.47	735.93	77.99	226.36
Motorola Moto G Plus	13.30	12.98	16.86	28.10	14.03	23.35	123.37	381.51	872.98	123.77	381.60
Samsung S10e	24.01	19.37	46.97	77.64	14.04	15.54	36.14	62.11	94.00	37.86	63.46
LG Nexus 5X	33.31	33.61	38.48	51.45	29.31	33.33	80.22	193.16	404.16	80.72	190.20
Huawei P20 Lite	72.09	70.42	81.64	112.54	50.36	54.88	81.15	160.78	513.72	96.14	158.08
OnePlus 7 Pro	25.65	26.05	20.08	31.22	19.46	14.26	28.34	61.14	125.18	29.35	62.19
LG G6	64.54	63.84	54.54	61.03	42.63	47.48	119.31	296.30	725.58	117.87	297.07
Samsung GalaxyS9+	23.34	23.16	60.96	60.13	16.29	17.82	35.21	68.12	125.82	36.41	69.68
OnePlus 6	29.68	24.04	37.49	41.29	23.10	26.79	37.90	71.49	135.75	37.24	71.84
Huawei P9 Lite	23.94	25.80	36.87	61.07	13.83	17.41	53.59	132.12	1099.15	54.19	131.47
Google Pixel 3A	32.67	33.25	43.36	66.47	26.63	29.46	60.99	143.17	305.34	61.99	80.08
Samsung Galaxy S6	48.57	52.09	75.93	122.29	6.97	79.25	209.44	432.10	950.45	212.05	436.21
Samsung Note10+	22.61	17.49	39.66	62.32	15.06	17.18	28.80	50.48	75.92	30.88	51.61
Asus Zenphone 3 MAX	153.54	162.04	204.35	282.12	90.79	92.44	110.06	155.25	241.34	110.86	155.90

Fig. A.12: Signature using RSA or EC with SHA512



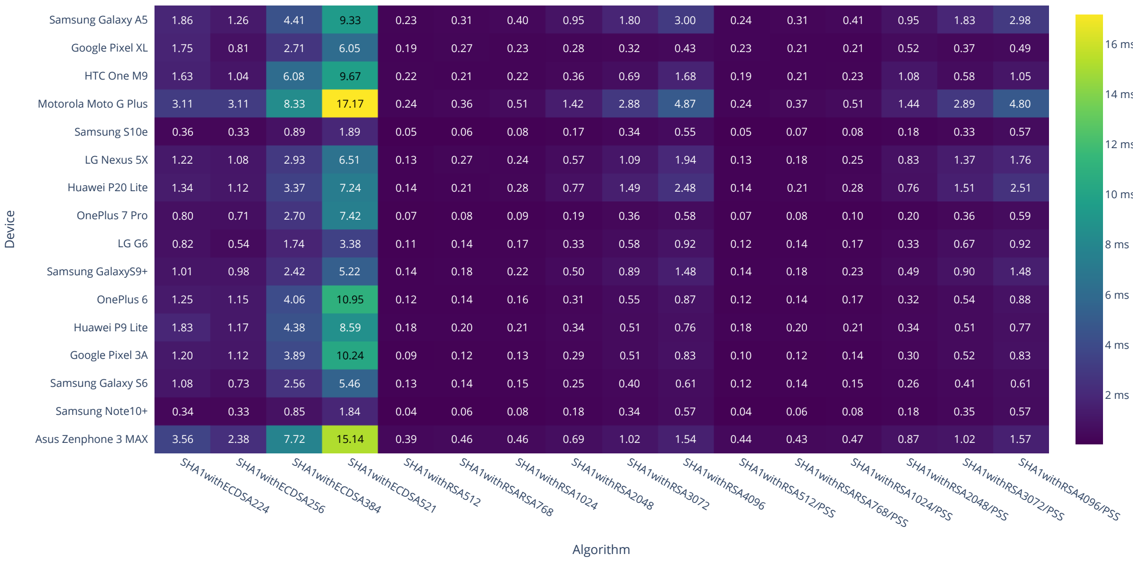
Device	MD5withRSA512	MD5withRSA768	MD5withRSA1024	MD5withRSA2048	MD5withRSA3072	MD5withRSA4096
Samsung Galaxy A5	0.23	0.31	0.40	0.94	1.83	3.11
Google Pixel XL	0.18	0.20	0.19	0.24	0.37	0.47
HTC One M9	0.18	0.19	0.22	0.35	1.11	1.50
Motorola Moto G Plus	0.24	0.36	0.52	1.41	2.87	4.82
Samsung S10e	0.05	0.06	0.08	0.17	0.32	0.56
LG Nexus 5X	0.12	0.18	0.24	0.57	1.07	1.78
Huawei P20 Lite	0.14	0.21	0.28	0.76	1.49	2.51
OnePlus 7 Pro	0.06	0.09	0.10	0.19	0.37	0.58
LG G6	0.11	0.14	0.17	0.33	0.75	1.01
Samsung GalaxyS9+	0.14	0.18	0.22	0.49	0.90	1.46
OnePlus 6	0.11	0.15	0.16	0.32	0.55	0.86
Huawei P9 Lite	0.17	0.19	0.21	0.34	0.50	0.75
Google Pixel 3A	0.09	0.12	0.14	0.28	0.51	0.81
Samsung Galaxy S6	0.12	0.14	0.15	0.25	0.40	0.60
Samsung Note10+	0.04	0.06	0.08	0.17	0.34	0.57
Asus Zenphone 3 MAX	0.39	0.69	0.54	0.71	1.00	1.56

Fig. A.13: Verification using RSA with MD5



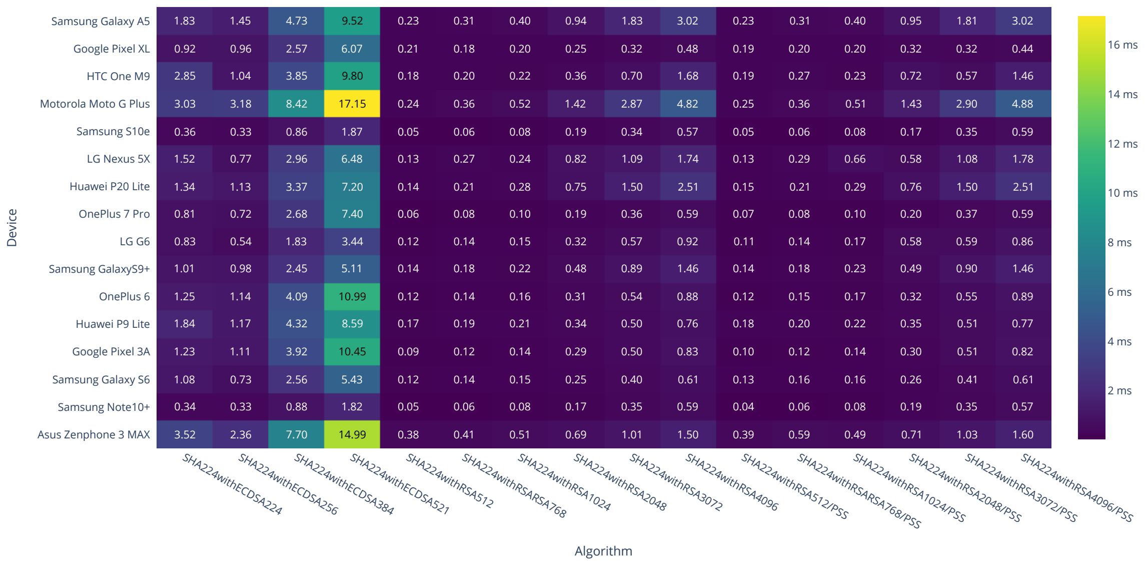
Device	NONEwithECDSA224	NONEwithECDSA256	NONEwithECDSA384	NONEwithECDSA521	NONEwithRSA512	NONEwithRSA768	NONEwithRSA1024	NONEwithRSA2048	NONEwithRSA3072	NONEwithRSA4096
Samsung Galaxy A5	5.24	6.76	9.92	18.40	0.27	0.33	0.42	0.97	1.84	3.02
Google Pixel XL	0.91	0.82	2.66	5.92	0.27	0.21	0.23	0.32	0.35	0.46
HTC One M9	4.74	4.26	8.82	15.64	0.75	0.25	0.54	0.41	1.20	0.96
Motorola Moto G Plus	3.07	3.18	8.29	17.26	0.30	0.39	0.52	1.47	2.90	4.83
Samsung S10e	0.34	0.33	0.87	1.87	0.05	0.06	0.08	0.19	0.33	0.56
LG Nexus 5X	1.52	0.77	2.91	6.42	0.16	0.22	0.27	0.85	1.12	1.80
Huawei P20 Lite	1.33	1.12	3.37	7.17	0.17	0.23	0.30	0.78	1.51	2.53
OnePlus 7 Pro	0.80	0.71	2.68	7.38	0.09	0.10	0.10	0.22	0.38	0.61
LG G6	1.12	0.72	2.40	3.20	0.14	0.15	0.17	0.32	0.54	0.93
Samsung GalaxyS9+	0.99	0.97	2.40	5.09	0.20	0.20	0.24	0.51	0.92	1.48
OnePlus 6	1.25	1.14	4.07	10.96	0.15	0.17	0.19	0.34	0.57	0.89
Huawei P9 Lite	539.10	641.42	1191.77	2131.29	0.24	0.27	0.29	0.39	0.57	0.83
Google Pixel 3A	1.20	1.12	3.89	10.32	0.14	0.15	0.18	0.31	0.53	0.84
Samsung Galaxy S6	47.06	51.75	75.86	124.13	0.17	0.18	0.19	0.29	0.44	0.64
Samsung Note10+	0.34	0.33	0.86	1.87	0.06	0.07	0.08	0.20	0.37	0.57
Asus Zenphone 3 MAX	137.54	143.46	171.03	216.35	0.50	0.60	0.60	0.80	1.16	1.61

Fig. A.14: Verification using RSA or EC without hash function



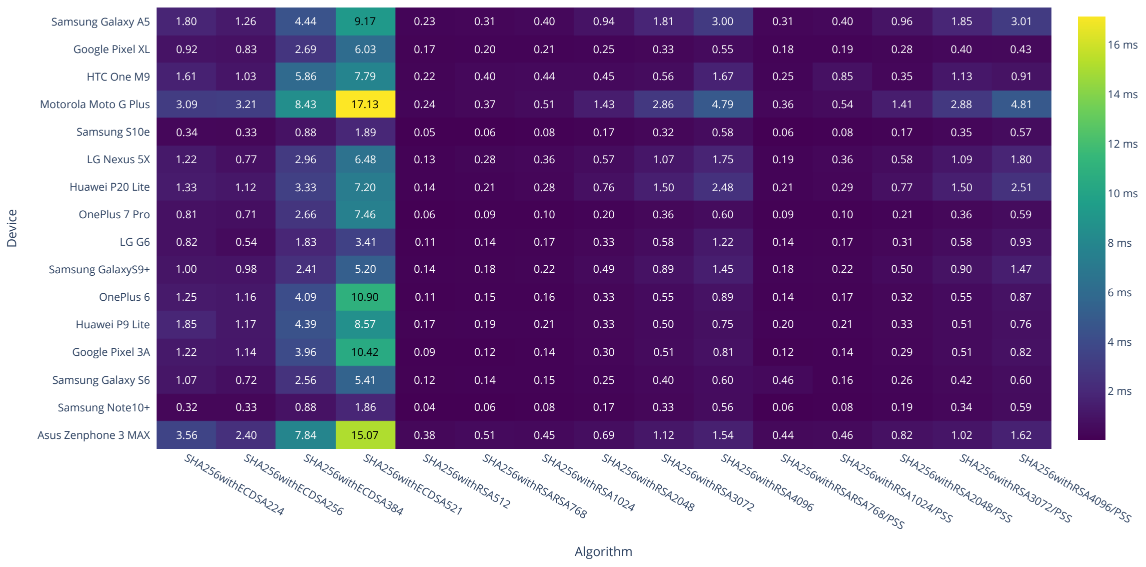
Device	SHA1withECDSA224	SHA1withECDSA256	SHA1withECDSA384	SHA1withECDSA521	SHA1withRSA512	SHA1withRSARSA768	SHA1withRSA1024	SHA1withRSA2048	SHA1withRSA3072	SHA1withRSA4096	SHA1withRSA512/PSS	SHA1withRSARSA768/PSS	SHA1withRSA1024/PSS	SHA1withRSA2048/PSS	SHA1withRSA3072/PSS	SHA1withRSA4096/PSS
Samsung Galaxy A5	1.86	1.26	4.41	9.33	0.23	0.31	0.40	0.95	1.80	3.00	0.24	0.31	0.41	0.95	1.83	2.98
Google Pixel XL	1.75	0.81	2.71	6.05	0.19	0.27	0.23	0.28	0.32	0.43	0.23	0.21	0.21	0.52	0.37	0.49
HTC One M9	1.63	1.04	6.08	9.67	0.22	0.21	0.22	0.36	0.69	1.68	0.19	0.21	0.23	1.08	0.58	1.05
Motorola Moto G Plus	3.11	3.11	8.33	17.17	0.24	0.36	0.51	1.42	2.88	4.87	0.24	0.37	0.51	1.44	2.89	4.80
Samsung S10e	0.36	0.33	0.89	1.89	0.05	0.06	0.08	0.17	0.34	0.55	0.05	0.07	0.08	0.18	0.33	0.57
LG Nexus 5X	1.22	1.08	2.93	6.51	0.13	0.27	0.24	0.57	1.09	1.94	0.13	0.18	0.25	0.83	1.37	1.76
Huawei P20 Lite	1.34	1.12	3.37	7.24	0.14	0.21	0.28	0.77	1.49	2.48	0.14	0.21	0.28	0.76	1.51	2.51
OnePlus 7 Pro	0.80	0.71	2.70	7.42	0.07	0.08	0.09	0.19	0.36	0.58	0.07	0.08	0.10	0.20	0.36	0.59
LG G6	0.82	0.54	1.74	3.38	0.11	0.14	0.17	0.33	0.58	0.92	0.12	0.14	0.17	0.33	0.67	0.92
Samsung GalaxyS9+	1.01	0.98	2.42	5.22	0.14	0.18	0.22	0.50	0.89	1.48	0.14	0.18	0.23	0.49	0.90	1.48
OnePlus 6	1.25	1.15	4.06	10.95	0.12	0.14	0.16	0.31	0.55	0.87	0.12	0.14	0.17	0.32	0.54	0.88
Huawei P9 Lite	1.83	1.17	4.38	8.59	0.18	0.20	0.21	0.34	0.51	0.76	0.18	0.20	0.21	0.34	0.51	0.77
Google Pixel 3A	1.20	1.12	3.89	10.24	0.09	0.12	0.13	0.29	0.51	0.83	0.10	0.12	0.14	0.30	0.52	0.83
Samsung Galaxy S6	1.08	0.73	2.56	5.46	0.13	0.14	0.15	0.25	0.40	0.61	0.12	0.14	0.15	0.26	0.41	0.61
Samsung Note10+	0.34	0.33	0.85	1.84	0.04	0.06	0.08	0.18	0.34	0.57	0.04	0.06	0.08	0.18	0.35	0.57
Asus Zenphone 3 MAX	3.56	2.38	7.72	15.14	0.39	0.46	0.46	0.69	1.02	1.54	0.44	0.43	0.47	0.87	1.02	1.57

Fig. A.15: Verification using RSA or EC with SHA1



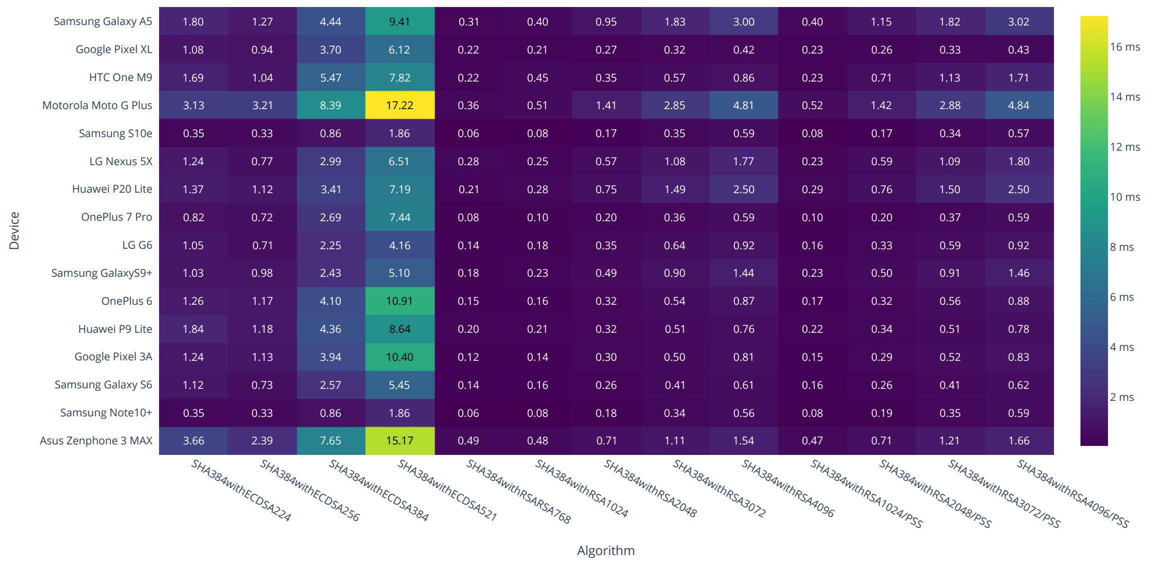
Device	SHA224withECDSA224	SHA224withECDSA256	SHA224withECDSA384	SHA224withECDSA521	SHA224withRSA512	SHA224withRSARSA768	SHA224withRSA1024	SHA224withRSA2048	SHA224withRSA3072	SHA224withRSA4096	SHA224withRSA512/PSS	SHA224withRSARSA768/PSS	SHA224withRSA1024/PSS	SHA224withRSA2048/PSS	SHA224withRSA3072/PSS	SHA224withRSA4096/PSS
Samsung Galaxy A5	1.83	1.45	4.73	9.52	0.23	0.31	0.40	0.94	1.83	3.02	0.23	0.31	0.40	0.95	1.81	3.02
Google Pixel XL	0.92	0.96	2.57	6.07	0.21	0.18	0.20	0.25	0.32	0.48	0.19	0.20	0.20	0.32	0.32	0.44
HTC One M9	2.85	1.04	3.85	9.80	0.18	0.20	0.22	0.36	0.70	1.68	0.19	0.27	0.23	0.72	0.57	1.46
Motorola Moto G Plus	3.03	3.18	8.42	17.15	0.24	0.36	0.52	1.42	2.87	4.82	0.25	0.36	0.51	1.43	2.90	4.88
Samsung S10e	0.36	0.33	0.86	1.87	0.05	0.06	0.08	0.19	0.34	0.57	0.05	0.06	0.08	0.17	0.35	0.59
LG Nexus 5X	1.52	0.77	2.96	6.48	0.13	0.27	0.24	0.82	1.09	1.74	0.13	0.29	0.66	0.58	1.08	1.78
Huawei P20 Lite	1.34	1.13	3.37	7.20	0.14	0.21	0.28	0.75	1.50	2.51	0.15	0.21	0.29	0.76	1.50	2.51
OnePlus 7 Pro	0.81	0.72	2.68	7.40	0.06	0.08	0.10	0.19	0.36	0.59	0.07	0.08	0.10	0.20	0.37	0.59
LG G6	0.83	0.54	1.83	3.44	0.12	0.14	0.15	0.32	0.57	0.92	0.11	0.14	0.17	0.58	0.59	0.86
Samsung GalaxyS9+	1.01	0.98	2.45	5.11	0.14	0.18	0.22	0.48	0.89	1.46	0.14	0.18	0.23	0.49	0.90	1.46
OnePlus 6	1.25	1.14	4.09	10.99	0.12	0.14	0.16	0.31	0.54	0.88	0.12	0.15	0.17	0.32	0.55	0.89
Huawei P9 Lite	1.84	1.17	4.32	8.59	0.17	0.19	0.21	0.34	0.50	0.76	0.18	0.20	0.22	0.35	0.51	0.77
Google Pixel 3A	1.23	1.11	3.92	10.45	0.09	0.12	0.14	0.29	0.50	0.83	0.10	0.12	0.14	0.30	0.51	0.82
Samsung Galaxy S6	1.08	0.73	2.56	5.43	0.12	0.14	0.15	0.25	0.40	0.61	0.13	0.16	0.16	0.26	0.41	0.61
Samsung Note10+	0.34	0.33	0.88	1.82	0.05	0.06	0.08	0.17	0.35	0.59	0.04	0.06	0.08	0.19	0.35	0.57
Asus Zenphone 3 MAX	3.52	2.36	7.70	14.99	0.38	0.41	0.51	0.69	1.01	1.50	0.39	0.59	0.49	0.71	1.03	1.60

Fig. A.16: Verification using RSA or EC with SHA224



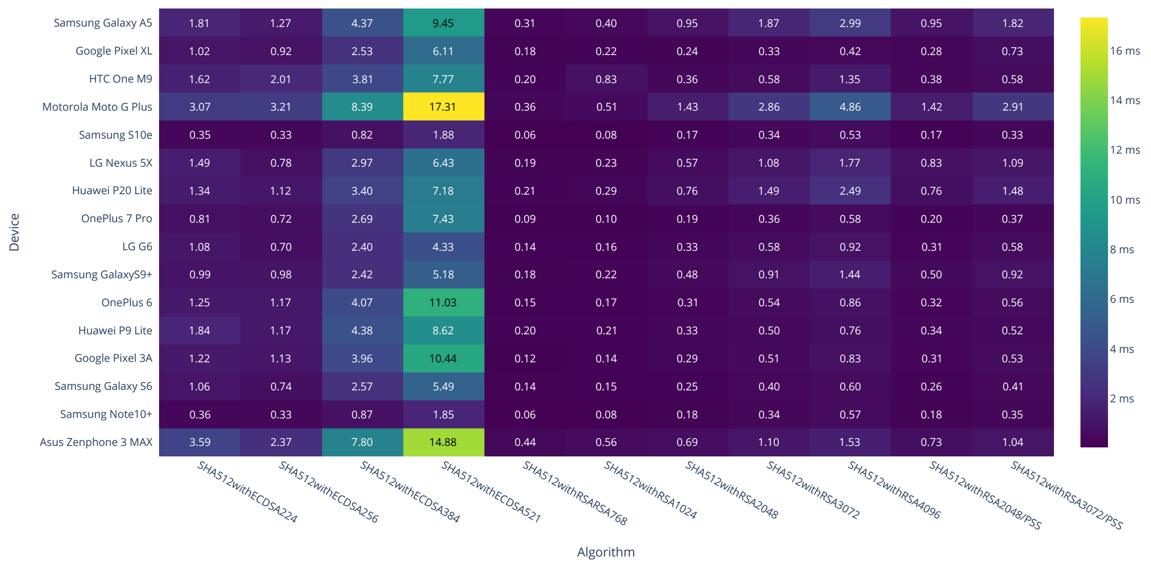
Device	SHA256withECDSA224	SHA256withECDSA256	SHA256withECDSA384	SHA256withECDSA521	SHA256withRSA512	SHA256withRSARSA768	SHA256withRSA1024	SHA256withRSA2048	SHA256withRSA3072	SHA256withRSA4096	SHA256withRSARSA768/PSS	SHA256withRSA1024/PSS	SHA256withRSA2048/PSS	SHA256withRSA3072/PSS	SHA256withRSA4096/PSS
Samsung Galaxy A5	1.80	1.26	4.44	9.17	0.23	0.31	0.40	0.94	1.81	3.00	0.31	0.40	0.96	1.85	3.01
Google Pixel XL	0.92	0.83	2.69	6.03	0.17	0.20	0.21	0.25	0.33	0.55	0.18	0.19	0.28	0.40	0.43
HTC One M9	1.61	1.03	5.86	7.79	0.22	0.40	0.44	0.45	0.56	1.67	0.25	0.85	0.35	1.13	0.91
Motorola Moto G Plus	3.09	3.21	8.43	17.13	0.24	0.37	0.51	1.43	2.86	4.79	0.36	0.54	1.41	2.88	4.81
Samsung S10e	0.34	0.33	0.88	1.89	0.05	0.06	0.08	0.17	0.32	0.58	0.06	0.08	0.17	0.35	0.57
LG Nexus 5X	1.22	0.77	2.96	6.48	0.13	0.28	0.36	0.57	1.07	1.75	0.19	0.36	0.58	1.09	1.80
Huawei P20 Lite	1.33	1.12	3.33	7.20	0.14	0.21	0.28	0.76	1.50	2.48	0.21	0.29	0.77	1.50	2.51
OnePlus 7 Pro	0.81	0.71	2.66	7.46	0.06	0.09	0.10	0.20	0.36	0.60	0.09	0.10	0.21	0.36	0.59
LG G6	0.82	0.54	1.83	3.41	0.11	0.14	0.17	0.33	0.58	1.22	0.14	0.17	0.31	0.58	0.93
Samsung GalaxyS9+	1.00	0.98	2.41	5.20	0.14	0.18	0.22	0.49	0.89	1.45	0.18	0.22	0.50	0.90	1.47
OnePlus 6	1.25	1.16	4.09	10.90	0.11	0.15	0.16	0.33	0.55	0.89	0.14	0.17	0.32	0.55	0.87
Huawei P9 Lite	1.85	1.17	4.39	8.57	0.17	0.19	0.21	0.33	0.50	0.75	0.20	0.21	0.33	0.51	0.76
Google Pixel 3A	1.22	1.14	3.96	10.42	0.09	0.12	0.14	0.30	0.51	0.81	0.12	0.14	0.29	0.51	0.82
Samsung Galaxy S6	1.07	0.72	2.56	5.41	0.12	0.14	0.15	0.25	0.40	0.60	0.46	0.16	0.26	0.42	0.60
Samsung Note10+	0.32	0.33	0.88	1.86	0.04	0.06	0.08	0.17	0.33	0.56	0.06	0.08	0.19	0.34	0.59
Asus Zenphone 3 MAX	3.56	2.40	7.84	15.07	0.38	0.51	0.45	0.69	1.12	1.54	0.44	0.46	0.82	1.02	1.62

Fig. A.17: Verification using RSA or EC with SHA256



Device	SHA384withECDSA224	SHA384withECDSA256	SHA384withECDSA384	SHA384withECDSA521	SHA384withRSARSA768	SHA384withRSA1024	SHA384withRSA2048	SHA384withRSA3072	SHA384withRSA4096	SHA384withRSA1024/PSS	SHA384withRSA2048/PSS	SHA384withRSA3072/PSS	SHA384withRSA4096/PSS
Samsung Galaxy A5	1.80	1.27	4.44	9.41	0.31	0.40	0.95	1.83	3.00	0.40	1.15	1.82	3.02
Google Pixel XL	1.08	0.94	3.70	6.12	0.22	0.21	0.27	0.32	0.42	0.23	0.26	0.33	0.43
HTC One M9	1.69	1.04	5.47	7.82	0.22	0.45	0.35	0.57	0.86	0.23	0.71	1.13	1.71
Motorola Moto G Plus	3.13	3.21	8.39	17.22	0.36	0.51	1.41	2.85	4.81	0.52	1.42	2.88	4.84
Samsung S10e	0.35	0.33	0.86	1.86	0.06	0.08	0.17	0.35	0.59	0.08	0.17	0.34	0.57
LG Nexus 5X	1.24	0.77	2.99	6.51	0.28	0.25	0.57	1.08	1.77	0.23	0.59	1.09	1.80
Huawei P20 Lite	1.37	1.12	3.41	7.19	0.21	0.28	0.75	1.49	2.50	0.29	0.76	1.50	2.50
OnePlus 7 Pro	0.82	0.72	2.69	7.44	0.08	0.10	0.20	0.36	0.59	0.10	0.20	0.37	0.59
LG G6	1.05	0.71	2.25	4.16	0.14	0.18	0.35	0.64	0.92	0.16	0.33	0.59	0.92
Samsung GalaxyS9+	1.03	0.98	2.43	5.10	0.18	0.23	0.49	0.90	1.44	0.23	0.50	0.91	1.46
OnePlus 6	1.26	1.17	4.10	10.91	0.15	0.16	0.32	0.54	0.87	0.17	0.32	0.56	0.88
Huawei P9 Lite	1.84	1.18	4.36	8.64	0.20	0.21	0.32	0.51	0.76	0.22	0.34	0.51	0.78
Google Pixel 3A	1.24	1.13	3.94	10.40	0.12	0.14	0.30	0.50	0.81	0.15	0.29	0.52	0.83
Samsung Galaxy S6	1.12	0.73	2.57	5.45	0.14	0.16	0.26	0.41	0.61	0.16	0.26	0.41	0.62
Samsung Note10+	0.35	0.33	0.86	1.86	0.06	0.08	0.18	0.34	0.56	0.08	0.19	0.35	0.59
Asus Zenphone 3 MAX	3.66	2.39	7.65	15.17	0.49	0.48	0.71	1.11	1.54	0.47	0.71	1.21	1.66

Fig. A.18: Verification using RSA or EC with SHA384



Device	SHA512withECDSA224	SHA512withECDSA256	SHA512withECDSA384	SHA512withECDSA521	SHA512withRSARSA768	SHA512withRSA1024	SHA512withRSA2048	SHA512withRSA3072	SHA512withRSA4096	SHA512withRSA2048/PSS	SHA512withRSA3072/PSS
Samsung Galaxy A5	1.81	1.27	4.37	9.45	0.31	0.40	0.95	1.87	2.99	0.95	1.82
Google Pixel XL	1.02	0.92	2.53	6.11	0.18	0.22	0.24	0.33	0.42	0.28	0.73
HTC One M9	1.62	2.01	3.81	7.77	0.20	0.83	0.36	0.58	1.35	0.38	0.58
Motorola Moto G Plus	3.07	3.21	8.39	17.31	0.36	0.51	1.43	2.86	4.86	1.42	2.91
Samsung S10e	0.35	0.33	0.82	1.88	0.06	0.08	0.17	0.34	0.53	0.17	0.33
LG Nexus 5X	1.49	0.78	2.97	6.43	0.19	0.23	0.57	1.08	1.77	0.83	1.09
Huawei P20 Lite	1.34	1.12	3.40	7.18	0.21	0.29	0.76	1.49	2.49	0.76	1.48
OnePlus 7 Pro	0.81	0.72	2.69	7.43	0.09	0.10	0.19	0.36	0.58	0.20	0.37
LG G6	1.08	0.70	2.40	4.33	0.14	0.16	0.33	0.58	0.92	0.31	0.58
Samsung GalaxyS9+	0.99	0.98	2.42	5.18	0.18	0.22	0.48	0.91	1.44	0.50	0.92
OnePlus 6	1.25	1.17	4.07	11.03	0.15	0.17	0.31	0.54	0.86	0.32	0.56
Huawei P9 Lite	1.84	1.17	4.38	8.62	0.20	0.21	0.33	0.50	0.76	0.34	0.52
Google Pixel 3A	1.22	1.13	3.96	10.44	0.12	0.14	0.29	0.51	0.83	0.31	0.53
Samsung Galaxy S6	1.06	0.74	2.57	5.49	0.14	0.15	0.25	0.40	0.60	0.26	0.41
Samsung Note10+	0.36	0.33	0.87	1.85	0.06	0.08	0.18	0.34	0.57	0.18	0.35
Asus Zenphone 3 MAX	3.59	2.37	7.80	14.88	0.44	0.56	0.69	1.10	1.53	0.73	1.04

Fig. A.19: Verification using RSA or EC with SHA512

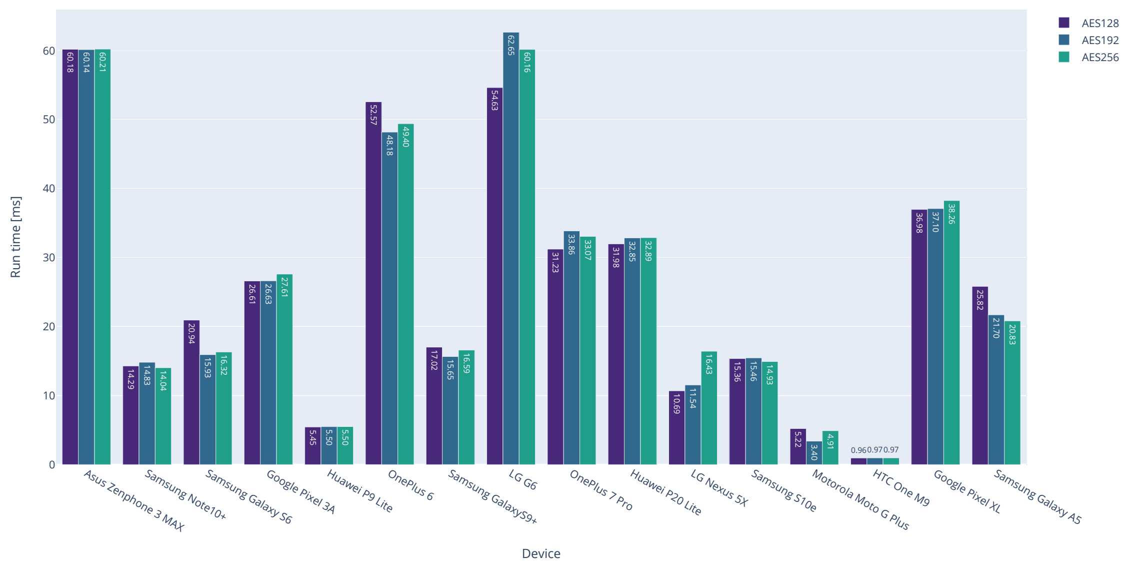
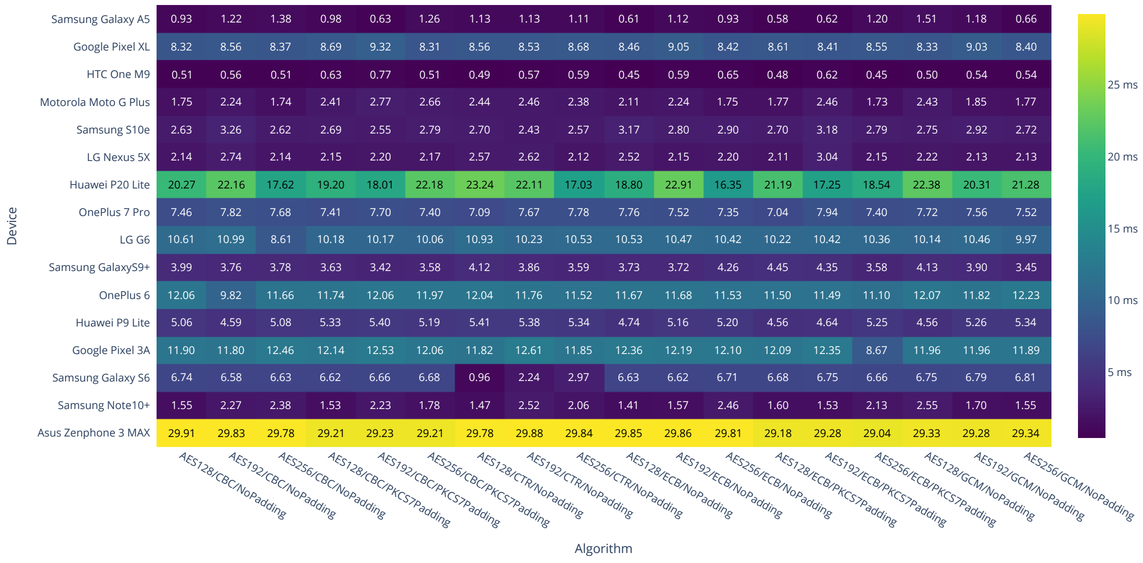


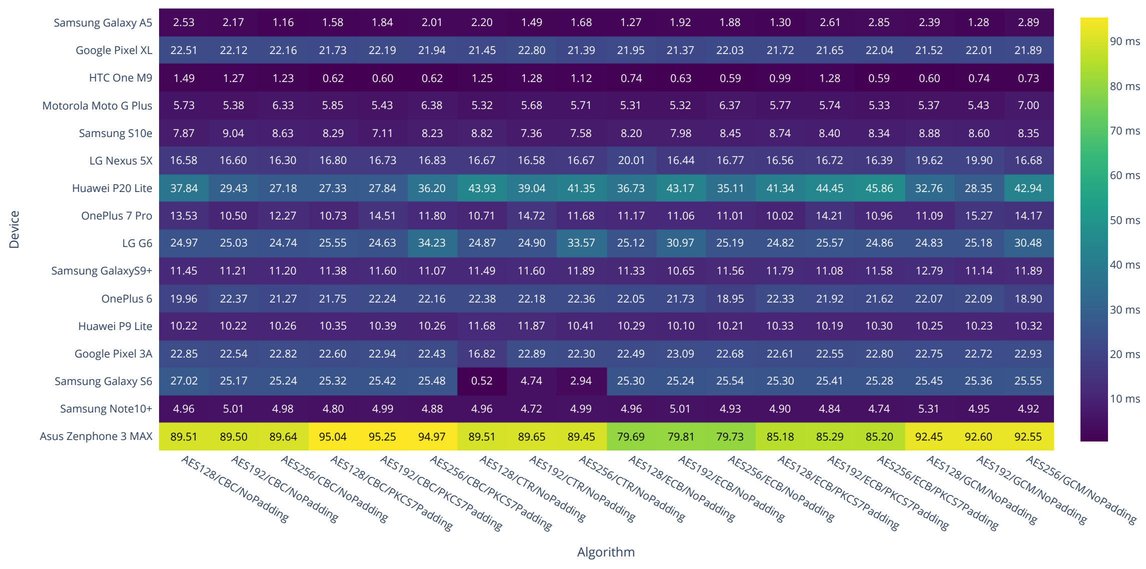
Fig. A.20: Symmetric key creation



Device	AES128/CBC/NoPadding	AES192/CBC/NoPadding	AES256/CBC/NoPadding	AES128/CBC/PKCS7Padding	AES192/CBC/PKCS7Padding	AES256/CBC/PKCS7Padding	AES128/CTR/NoPadding	AES192/CTR/NoPadding	AES256/CTR/NoPadding	AES128/ECB/NoPadding	AES192/ECB/NoPadding	AES256/ECB/NoPadding	AES128/ECB/PKCS7Padding	AES192/ECB/PKCS7Padding	AES256/ECB/PKCS7Padding	AES128/GCM/NoPadding	AES192/GCM/NoPadding	AES256/GCM/NoPadding
Samsung Galaxy A5	0.93	1.22	1.38	0.98	0.63	1.26	1.13	1.13	1.11	0.61	1.12	0.93	0.58	0.62	1.20	1.51	1.18	0.66
Google Pixel XL	8.32	8.56	8.37	8.69	9.32	8.31	8.56	8.53	8.68	8.46	9.05	8.42	8.61	8.41	8.55	8.33	9.03	8.40
HTC One M9	0.51	0.56	0.51	0.63	0.77	0.51	0.49	0.57	0.59	0.45	0.59	0.65	0.48	0.62	0.45	0.50	0.54	0.54
Motorola Moto G Plus	1.75	2.24	1.74	2.41	2.77	2.66	2.44	2.46	2.38	2.11	2.24	1.75	1.77	2.46	1.73	2.43	1.85	1.77
Samsung S10e	2.63	3.26	2.62	2.69	2.55	2.79	2.70	2.43	2.57	3.17	2.80	2.90	2.70	3.18	2.79	2.75	2.92	2.72
LG Nexus 5X	2.14	2.74	2.14	2.15	2.20	2.17	2.57	2.62	2.12	2.52	2.15	2.20	2.11	3.04	2.15	2.22	2.13	2.13
Huawei P20 Lite	20.27	22.16	17.62	19.20	18.01	22.18	23.24	22.11	17.03	18.80	22.91	16.35	21.19	17.25	18.54	22.38	20.31	21.28
OnePlus 7 Pro	7.46	7.82	7.68	7.41	7.70	7.40	7.09	7.67	7.78	7.76	7.52	7.35	7.04	7.94	7.40	7.72	7.56	7.52
LG G6	10.61	10.99	8.61	10.18	10.17	10.06	10.93	10.23	10.53	10.53	10.47	10.42	10.22	10.42	10.36	10.14	10.46	9.97
Samsung GalaxyS9+	3.99	3.76	3.78	3.63	3.42	3.58	4.12	3.86	3.59	3.73	3.72	4.26	4.45	4.35	3.58	4.13	3.90	3.45
OnePlus 6	12.06	9.82	11.66	11.74	12.06	11.97	12.04	11.76	11.52	11.67	11.68	11.53	11.50	11.49	11.10	12.07	11.82	12.23
Huawei P9 Lite	5.06	4.59	5.08	5.33	5.40	5.19	5.41	5.38	5.34	4.74	5.16	5.20	4.56	4.64	5.25	4.56	5.26	5.34
Google Pixel 3A	11.90	11.80	12.46	12.14	12.53	12.06	11.82	12.61	11.85	12.36	12.19	12.10	12.09	12.35	8.67	11.96	11.96	11.89
Samsung Galaxy S6	6.74	6.58	6.63	6.62	6.66	6.68	0.96	2.24	2.97	6.63	6.62	6.71	6.68	6.75	6.66	6.75	6.79	6.81
Samsung Note10+	1.55	2.27	2.38	1.53	2.23	1.78	1.47	2.52	2.06	1.41	1.57	2.46	1.60	1.53	2.13	2.55	1.70	1.55
Asus Zenphone 3 MAX	29.91	29.83	29.78	29.21	29.23	29.21	29.78	29.88	29.84	29.85	29.86	29.81	29.18	29.28	29.04	29.33	29.28	29.34

Algorithm

Fig. A.21: Encryption with AES



Device	AES128/CBC/NoPadding	AES192/CBC/NoPadding	AES256/CBC/NoPadding	AES128/CBC/PKCS7Padding	AES192/CBC/PKCS7Padding	AES256/CBC/PKCS7Padding	AES128/CTR/NoPadding	AES192/CTR/NoPadding	AES256/CTR/NoPadding	AES128/ECB/NoPadding	AES192/ECB/NoPadding	AES256/ECB/NoPadding	AES128/ECB/PKCS7Padding	AES192/ECB/PKCS7Padding	AES256/ECB/PKCS7Padding	AES128/GCM/NoPadding	AES192/GCM/NoPadding	AES256/GCM/NoPadding
Samsung Galaxy A5	2.53	2.17	1.16	1.58	1.84	2.01	2.20	1.49	1.68	1.27	1.92	1.88	1.30	2.61	2.85	2.39	1.28	2.89
Google Pixel XL	22.51	22.12	22.16	21.73	22.19	21.94	21.45	22.80	21.39	21.95	21.37	22.03	21.72	21.65	22.04	21.52	22.01	21.89
HTC One M9	1.49	1.27	1.23	0.62	0.60	0.62	1.25	1.28	1.12	0.74	0.63	0.59	0.99	1.28	0.59	0.60	0.74	0.73
Motorola Moto G Plus	5.73	5.38	6.33	5.85	5.43	6.38	5.32	5.68	5.71	5.31	5.32	6.37	5.77	5.74	5.33	5.37	5.43	7.00
Samsung S10e	7.87	9.04	8.63	8.29	7.11	8.23	8.82	7.36	7.58	8.20	7.98	8.45	8.74	8.40	8.34	8.88	8.60	8.35
LG Nexus 5X	16.58	16.60	16.30	16.80	16.73	16.83	16.67	16.58	16.67	20.01	16.44	16.77	16.56	16.72	16.39	19.62	19.90	16.68
Huawei P20 Lite	37.84	29.43	27.18	27.33	27.84	36.20	43.93	39.04	41.35	36.73	43.17	35.11	41.34	44.45	45.86	32.76	28.35	42.94
OnePlus 7 Pro	13.53	10.50	12.27	10.73	14.51	11.80	10.71	14.72	11.68	11.17	11.06	11.01	10.02	14.21	10.96	11.09	15.27	14.17
LG G6	24.97	25.03	24.74	25.55	24.63	34.23	24.87	24.90	33.57	25.12	30.97	25.19	24.82	25.57	24.86	24.83	25.18	30.48
Samsung GalaxyS9+	11.45	11.21	11.20	11.38	11.60	11.07	11.49	11.60	11.89	11.33	10.65	11.56	11.79	11.08	11.58	12.79	11.14	11.89
OnePlus 6	19.96	22.37	21.27	21.75	22.24	22.16	22.38	22.18	22.36	22.05	21.73	18.95	22.33	21.92	21.62	22.07	22.09	18.90
Huawei P9 Lite	10.22	10.22	10.26	10.35	10.39	10.26	11.68	11.87	10.41	10.29	10.10	10.21	10.33	10.19	10.30	10.25	10.23	10.32
Google Pixel 3A	22.85	22.54	22.82	22.60	22.94	22.43	16.82	22.89	22.30	22.49	23.09	22.68	22.61	22.55	22.80	22.75	22.72	22.93
Samsung Galaxy S6	27.02	25.17	25.24	25.32	25.42	25.48	0.52	4.74	2.94	25.30	25.24	25.54	25.30	25.41	25.28	25.45	25.36	25.55
Samsung Note10+	4.96	5.01	4.98	4.80	4.99	4.88	4.96	4.72	4.99	4.96	5.01	4.93	4.90	4.84	4.74	5.31	4.95	4.92
Asus Zenphone 3 MAX	89.51	89.50	89.64	95.04	95.25	94.97	89.51	89.65	89.45	79.69	79.81	79.73	85.18	85.29	85.20	92.45	92.60	92.55

Fig. A.22: Decryption with AES

B Content of the enclosed CD

```
/ ........................................ root folder of enclosed CD
  Android ................................ source code of Android applications
    Benchmarks ........................... source code of benchmark application
    SecNote .............................. source code of SecNote application
  Benchmark results ...................... python scrips for processing and results
  Server ................................. source code of microservices and deployment configuration
    Authservice .......................... source code of Authentication service
    Noteservice .......................... source code of Note service
    Permissionservice .................... source code of Permission service
    Deployment ........................... Kubernetes and Istio yaml files
  SecNote.apk ............................ Runnable application for Android OS
  Instructions ........................... Instructions how to try application
```