University of Peloponnese
Department of Informatics and Telecommunications

ZenCrypt - Securely Encrypt Files

Orestis Zestas
2022 2020 02006

Supervisor: Nikolaos Tselikas

A thesis submitted in partial fulfillment of the University's requirements for the masters degree.

February 12, 2022

Πανεπιστήμιο Πελοποννήσου
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Ανάπτυξη Εφαρμογής Κινητών Τερματικών για Ασφαλή Κρυπτογράφηση Αρχείων

Ορέστης Ζέστας
2022 2020 02006

Επιβλέπων: Νικόλαος Τσελίκας - Αναπληρωτής Καθηγητής

Διπλωματική Εργασία

12 Φεβρουαρίου 2022

Abstract

The technology advances and every day life requirements of the last decade have led to more and more people replacing computers/laptops with smart phones. As these devices gained increasing capabilities and better hardware/software, many opportunities came to light for novel application development. The Android market has reached an all-time high in application development, with growing demand for applications that focus on day-to-day tasks as well as communal interaction. This ultimately led to social media being widely used for user communication and file sharing every second of every day. Though, in this vast sea of information exchange and data transfers, the security aspect of such complicated tasks is more than often overlooked. In this thesis, we will present a new Android application, ZenCrypt, which was developed having this concern in mind. It stands as a very powerful tool that aims to help the user handle the security aspect of sharing private files online or securely storing them offline, as well as doing so with ease. We will also discuss some fundamental principles when developing an Android application, and analyze the security and functionality of some of the most common encryption algorithms. Finally, we will briefly comment on ZenCrypt's performance, and present some example usages of the application.

Keywords: Encryption Algorithms, Android App Development, Data Security, Kotlin, Kotlin vs Java, Coroutines, Flows, DES, 3DES, RSA, AES, Brute Forcing, Data Padding, Scoped Storage, AES Rounds, AES Cipher Block Chaining, PKCS, Native Android Apps, Android Lifecycle, Android ViewBinding

Revision 0.3

Περίληψη

Η πρόοδος της τεχνολογίας και οι καθημερινές απαιτήσεις της τελευταίας δεκαετίας έχουν οδηγήσει όλο και περισσότερους ανθρώπους να αντικαθιστούν τους υπολογιστές/φορητούς υπολογιστές με έξυπνα τηλέφωνα. Καθώς αυτές οι συσκευές αποκτούν αυξανόμενες δυνατότητες και καλύτερο υλικό/λογισμικό, ήρθαν στο φως πολλές ευκαιρίες για ανάπτυξη νέων εφαρμογών. Η αγορά του Android έχει φτάσει στο υψηλότερο επίπεδο ανάπτυξης λογισμικού όλων των εποχών, με αυξανόμενη ζήτηση για εφαρμογές που επικεντρώνονται σε καθημερινές εργασίες καθώς και στην κοινωνική αλληλεπίδραση. Τελικά, αυτό οδήγησε στο να χρησιμοποιούνται ευρέως τα μέσα κοινωνικής δικτύωσης για την επικοινωνία των χρηστών, καθώς και την κοινοποίηση αρχείων, κάθε δευτερόλεπτο της ημέρας. Όμως, σε αυτήν την αχανή έκταση ανταλλαγής πληροφοριών και μεταφοράς δεδομένων, η πτυχή της ασφάλειας τέτοιων περίπλοκων διαδικασιών συχνά παραβλέπεται. Σε αυτή τη μεταπτυχιακή διπλωματική εργασία, θα παρουσιάσουμε μια νέα εφαρμογή Android, την ZenCrypt, η οποία αναπτύχθηκε έχοντας κατά νου αυτή την ανησυχία. Αποτελεί ένα πολύ ισχυρό εργαλείο που στοχεύει στο να βοηθήσει τον χρήστη να θωρακίσει την ασφάλεια της κοινής χρήσης ιδιωτικών αρχείων στο διαδίκτυο ή της ασφαλούς αποθήκευσης τους εκτός σύνδεσης, με ευκολία. Θα συζητήσουμε, επίσης, ορισμένες θεμελιώδεις αρχές κατά την ανάπτυξη μιας εφαρμογής Android και θα αναλύσουμε την ασφάλεια και λειτουργικότητα ορισμένων από τους πιο κοινούς αλγόριθμους κρυπτογράφησης. Τέλος, θα σχολιάσουμε εν συντομία την αποδοτικότητα του ZenCrypt και θα παρουσιάσουμε μερικά παραδείγματα χρήσεων της εφαρμογής.

Λέξεις Κλειδιά: Αλγόριθμοι Κρυπτογράφησης, Ανάπτυξη λογισμικού στο Android, Προστασία Δεδομένων, Kotlin, Kotlin vs Java, Υπορουτίνες, Flows, DES, 3DES, RSA, AES, Εξαντλητική Επίθεση, Συμπλήρωση Δεδομένων, Εύρος Αποθήκευσης, Γύροι AES, AES Cipher Block Chaining, PKCS, Ανάπτυξη Γηγενών Android Εφαρμογών, Κύκλος Ζωής Android Εφαρμογών, Android ViewBinding

Έκδοση 0.3

Contents

List of Figures

Listings

1 Introduction

1.1 Data Security

The rising abuse of computers and increasing threat to personal privacy through data banks have stimulated much interest in the technical safeguards for data[7]. There are many safeguards, all of which are studied in a field of computer science, called *data security*, that lies under the umbrella of *information security*. Information security is an important issue in our information society. To put it into perspective, due to the increasing amounts of data that users are storing, and the increasingly easier options to carry data around (via memory sticks, hard drives, cloud hosting and mobile phones/tablets), it is critical to protect them from various threats that can lead to fundamental loss. After all, in the current age of advanced digital technology, embezzling digital datasets is truly a click away.

Data security is often confused with similar terms like *data protection* and *data privacy*, because they all refer to ways to secure someone's data. However, the difference between these terms lies in the reasons for securing that data in the first place, as well as the methods for doing so. To elaborate:

- **Data security** refers to protecting data against unauthorized access or use that could result in exposure, deletion, or corruption of that data. An example of data security would be using encryption to prevent malicious activities in the event that the stored information is breached.
- **Data protection** refers to the creation of backups or duplication of data to protect against accidental erasure or loss.
- **Data privacy** refers to concerns regarding how the data is handled; regulatory concerns, notification, and consent of use, etc.

We can further address data security by identifying its three main elements:

- **Confidentiality**: Systems and data are accessible to authorized users only.
- **Integrity**: Systems and data are accurate and complete.
- **Availability**: Systems and data are accessible when they are needed.

There are a lot of appliances for data security, all with different approaches and implementations. This thesis focuses on mobile data security, since mobiles impact nearly every aspect of our lives, from shopping online, to taking care of our medical and banking needs. In the first half of 2019 alone, mobile attacks hit 150 million [5][1], and rose another 30% in 2019, according to McAffe's 2020 Q1 mobile threat report [32]. And as mobile cyber-attacks increase, mobile security becomes an even more critical part of the general data security strategy.

1.2 Cryptography

One of the main approaches to tackle data security is cryptography. Cryptography is the science and study of secret writing [38]. A *cipher* is a secret method of writing, whereby *plaintext* is transformed into *ciphertext*, sometimes called a cryptogram. The process of transforming plaitext into ciphertext is called *encipherment* or *encryption*; the reverse process of transforming ciphertext into plaintext is called *decipherment* or *decryption*. Both processes are controlled by a cryptographic *key* or keys (see figure 1).

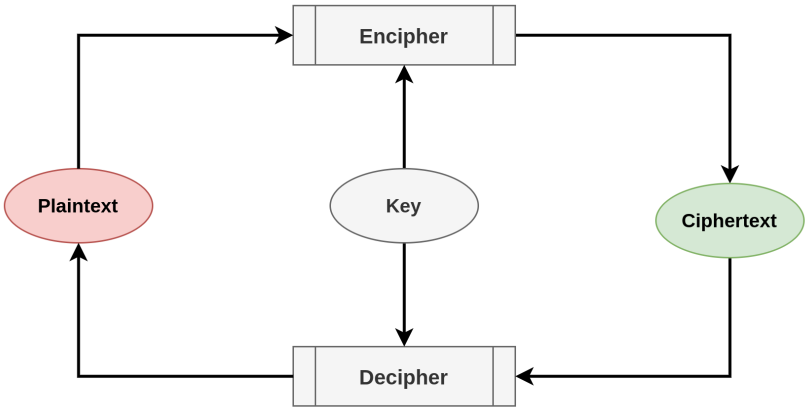


Figure 1: *Secret Writing*

Classical cryptography provided secrecy for information sent over channels where eavesdropping and message interception was possible. The sender selected a cipher and encryption key, and either gave it directly to the receiver or else sent it indirectly over a slow but secure channel.

Modern cryptography protects data transmitted over high-speed electronic lines or stored in computer systems. There are two principal objectives; *secrecy* (or *privacy*), to prevent the unauthorized disclosure of data; and the *authenticity* (or *integrity*), to prevent the unauthorized modification of data.

The problem with information transmitted over lines, is that they are vulnerable to passive wire-tapping, which threatens secrecy, and active wire-tapping, which threatens authenticity. In active wire-tapping, also known as *tampering*, the attacker modifies the message stream in order to make arbitrary changes to it, or to replace specific information, which can lead to false messages.

Encryption protects against message modification and injection of false messages, by making it infeasible for an attacker to create ciphertext that deciphers into meaningful plaintext. However, while this can be used to detect message modifications, it cannot prevent it.

Before continuing to chapter 1.3, there are some terms which should be explained for the better understanding of encryption algorithms. This terminology is critical to

analyze, since it appears in every algorithm description which is going to be discussed.

- **Cryptographic Key**: A key is a numeric or alpha-numeric text that is used for encryption/decryption.
- **Key Size**: The key size is the measure of length of the key in bits.
- **Block Size**: Key cipher works on fixed length string of bits. This fixed length of string in bits is called *block size*. It depends on the applied algorithm.
- **Round**: The round of encryption simply refers the same same function being applied many times over. This is done intentionally to slow down the calculation. For example, one could use 1000 orunds to derive a key from a password. For someone that already knows the password, this would mean waiting 1 second instead of 0.001 seconds. Having to wait 1 second while the system checks your password is not a lot, and most users don't even notice it. But, for an adversary who is trying to brute-force million of passwords per second, using 1000 rounds would slow down his progress significantly. Instead of a million, he could only try 1000 passwords per second. This would make a brute-force attack much less attractive.
- **Encryption Types**: Cryptography systems can be broadly classified into two categories.

  Symmetric encryption algorithms, that use a single key that both the sender and recipient have. This key is kept secret among both parties so that no intruder can intercept the data to be transferred.

  Asymmetric encryption algorithms, or *public-key* systems that use two keys, a public key known to everyone, and a private key that only the recipient of the message knows. Compared to symmetric key encryption, this method provides more security, but is significantly slower [2].

Cryptography has three main goals. *Data privacy* is the most commonly addressed goal, and it ensures that the meaning of a message is concealed by encrypting it with a cryptographic key. *Data integrity* is the second goal, which ensures that the message received is the same as the message that was sent. This is done by hashing the original sent message to create a unique digest, which can be compared later on with the recipient's generated digest from the received message. This technique only protects against unintentional alteration of the message. A variation of this is used to create digital signatures to protect against malicious alteration. Finally, *data authenticity* guarantees that a user or system can prove their identity to another who does not have personal knowledge of their identity. This is accomplished using digital certificates.

1.3 Encryption Algorithms

An algorithm comprises a sequence of unambiguous instructions required to solve a problem, i.e. obtaining a required output for any legitimate input in a finite amount of time. With that said, let's continue to the analysis of each algorithm. Note, however, that there are many encryption algorithms available, and we will only be discussing a selected few.

1.3.1 Data Encryption Standard (DES)

DES was developed by IBM in 1972, and later adopted by the National Bureau of Standard (NBS) as Federal Information Processing Standard (FIPS) in 1977. It stands as the earliest symmetric encryption algorithm, and is considered to be highly insecure in modern cryptography. It uses 64-bit blocks, 8 bits of which are used for parity checks, to verify the key's integrity. The rest 56 bits are directly utilized by the algorithm as key bits and are randomly generated [35]. That means it would take a maximum of 2^{56} attempts to find the correct key. Figure 2 demonstrates the structure of the *Feistel*

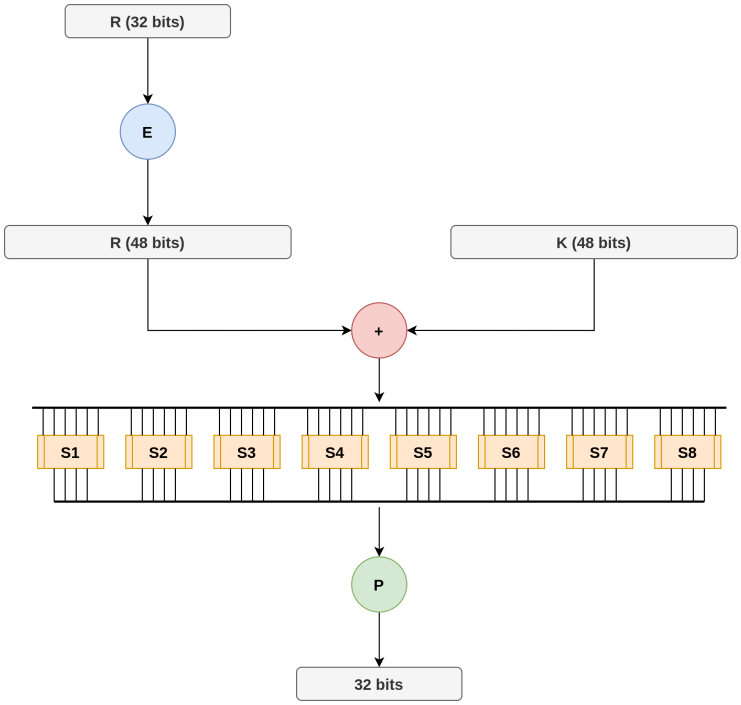


Figure 2: *The Feistel function of DES*

function (F function) of DES [40]. More specifically:

1. Each block of message will be 64 bits. Do initial permutation on 64 bits data and divide it in to two halves. The left half which consists of 32 bits and the right half which is also 32 bits.

2. The right half is expanded to 48 bits.
3. Select 48 bits of the total 64 key bits by permuted choice.
4. The right half 48 bits are XORed with the 48 key bits.
5. Select 32 bits from the previous step by S-box substitution choice.
6. The 32 bits from the previous step are permuted using the P-box.
7. Now, the resulting 32 bits from the right half and the 32 bits form the left half are XORed.
8. The result from the previous step will be the new right half.
9. The old right half from the first step will be the new left half.

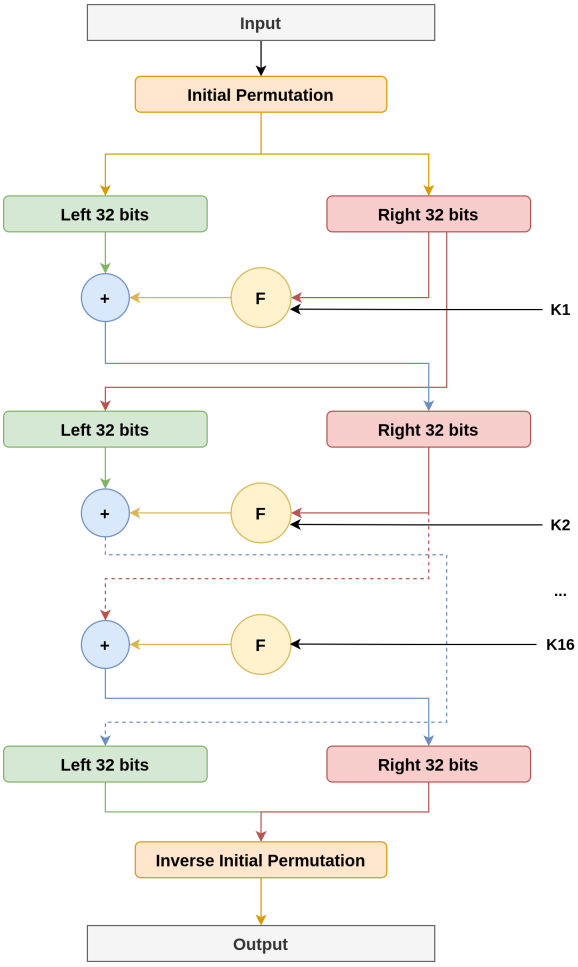


Figure 3: *DES procedure*

A cycle of DES comprises of the above steps. This is one cycle. There will be 16 of them, and after completion a final permutation is done one the data bits to get the decrypted data. Figure 3 graphically demonstrates the whole procedure.

1.3.2 Triple Data Encryption Standard (3DES)

Triple Data Encryption Standard (3DES), also known as Triple Data Encryption Algorithm (TDEA) was also developed by IBM in 1998, and then standardized in ANSI X9.17. It was proposed as a replacement of DES due to the improvement in the key length, and essentially applies the DES encryption algorithm three times to each data block. 3DES uses three times the rounds of simple DES in its computations, resulting

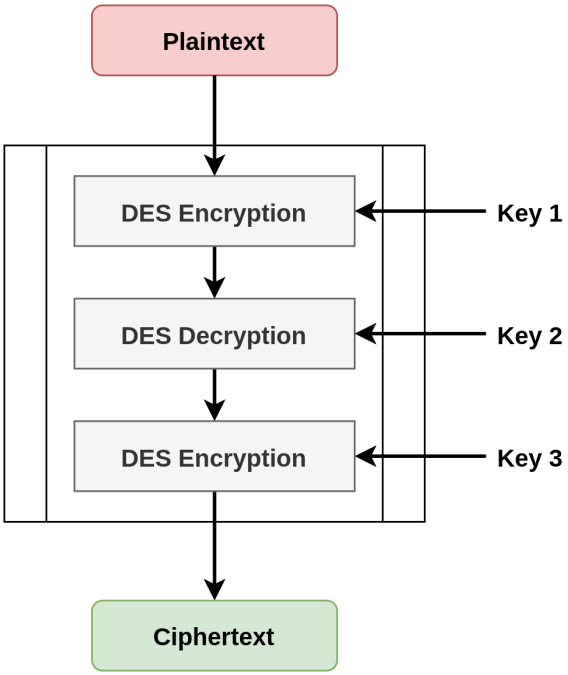


Figure 4: *Structure of 3DES (DES-EDE3)*

in a total of 48 rounds and a key length of 168 bits. Like DES, 3DES also uses a block size of 64 bits for encryption. There are the following modes which 3DES can be used:

- DES-EDE3: Encrypt with key 1, decrypt with key 2, and encrypt again with key 3 (as shown if figure 4).
- DES-EEE3: The encrypt operation is encrypt-encrypt-encrypt, and the decrypt operation is decrypt-decrypt-decrypt, all while using 3 different keys.
- DES-EDE2: There are only 2 keys used here, which remain the same for the first and last encryption.

- DES-EEE2: Like DES-EDE2, there are only two keys, used for the first and last encryption.

3DES provides up to 2^{112} security, and has been classified as legacy since 2014 [6] by ENISA, which recommends a minimum of 128 bits for encryption. 3DES provides only 112 bits and, whith a 112 bit key, National Institute of Standards and Technology (NIST) suggests that only provides 80 bits of actual security.

1.3.3 Rivest-Shamir-Adleman Algorithm (RSA)

The Rivest-Shamir-Adleman (RSA) algorithm is the most important public-key cryptosystem. It is an asymmetric algorithm, meaning that it uses a *public* key and a *private* key. As their names state, the public key can be shared with everyone, and the

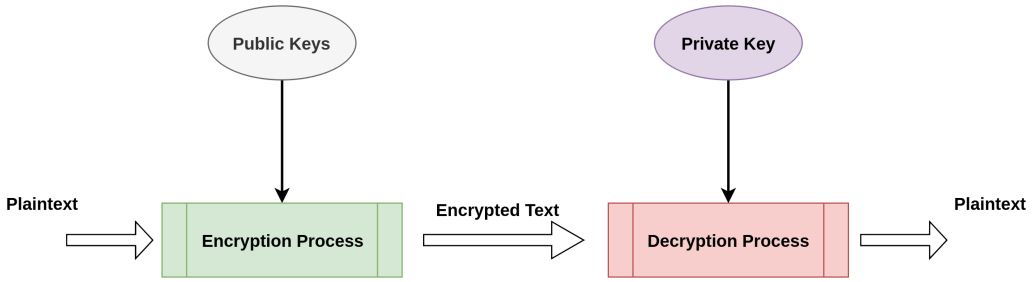


Figure 5: *RSA Algorithm*

private key is kept secret, and should not be shared. This algorithm was developed in 1978 [37], and it is based on the factoring problem.

1. The first step to RSA is to select two large prime numbers, P and Q. They need to be large enough so that it will be difficult for someone else to figure out.
2. Compute $n = P * Q$
3. Compute the *totient* function, $\phi(n) = (P-1) * (Q-1)$
4. Choose an integer e such that $1 < e < \phi(n)$ and *(e,n)* are co-prime. Co-prime are two numbers that their only positive integer that devides them is 1. The pair *(n,e)* is the public key.
5. Compute d such that $(d * e) \mod \phi(n) = 1$. The pair *(n,d)* makes up the private key.

Given a plaintext P, the ciphertext C is calculated as:

$$C = P^e \mod n \tag{1}$$

Conversely, using the private key *(n,d)* the plaintext can be found using:

$$P = C^d \mod n \tag{2}$$

The main disadvantage of RSA is its encryption speed, as is with all the asymmetric key encryption algorithms, due to the use of two asymmetric keys. Thus, it is not recommended for encrypting files because of their size. On the bright side, RSA provides a good level of security.

1.3.4 Advanced Encryption Standard (AES)

In 1997, NIST announced an initiative to choose a cipher to implement a new encryption standard, because of the need for high security and efficiency, since it was time to replace the existing DES and 3DES encryption algorithms. In 2001, Advanced Encryption Standard (AES) was selected, developed by Vincent Rijmen and Joan Daeman. AES is a symmetric block cipher used by the United States government to protect sensitive information, and has since been implemented throughout the world. It stands as the recent generation block cipher and significantly increases in the block size up to 128 bits with the key sizes being 128 bits, 192 bits and 256 bits. Each cipher encrypts and decrypts data in blocks of 128 using cryptographic keys of 128 bits, 192 bits and 256 bits respectively. There are 10 rounds for 128-bit keys, 12 rounds for 192-bits keys and 14 rounds for 256-bit keys. Figure 6 illustrates the AES algorithm.

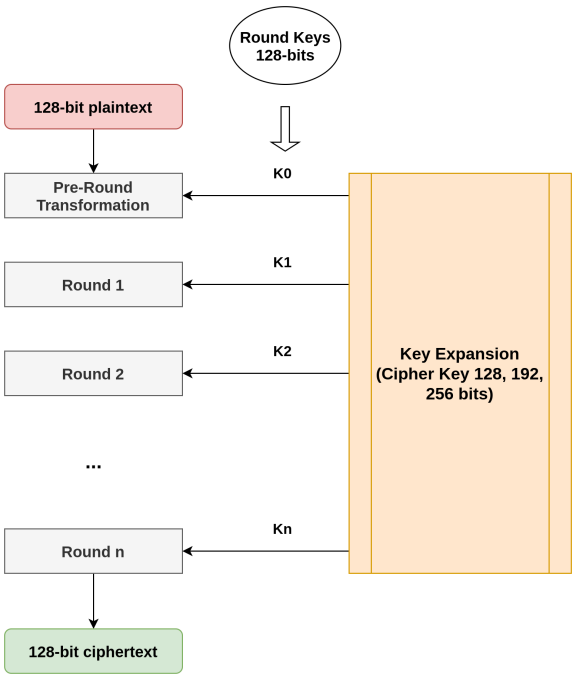


Figure 6: *AES Algorithm*

Ks (words/bytes/bits)	Bs (words/bytes/bits)	Nr	Rks (words/bytes/bits)	Eks (words/bytes)
4/16/128	4/16/128	10	4/16/128	44/176
6/24/192	4/16/128	12	4/16/128	52/208
8/32/256	4/16/128	14	4/16/128	60/240

Table 1: *AES Parameters*

Table 1 mentions the various numbers of AES parameters, based on the key length. Those are Key size (Ks), Block size (Bs), Number of rounds (Nr), Round key size (Rks), and Expanded key size (Eks). Each round (as seen in figure 6) consists of four layers. The substitute byte, shift rows, mix column and add round key. In the first layer, each byte is substituted during the forward encryption process (S-box). The second layer shifts the rows of the state array, and the third layer mixes up the bytes in each column. Finally, the forth layer XORes the subkey bytes with every byte of the state array. This operation will be done repeatedly based on the specified key size.

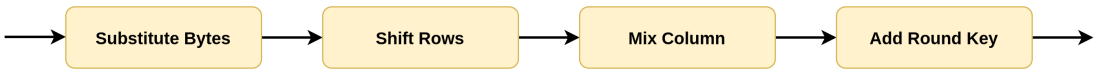


Figure 7: *An AES Round*

AES is known for its speed and small code footprint, regardless of the platform it's running, its simple design, and finally its heavy protection against all known attacks. AES has five modes of operation, which we will briefly discuss below [20]:

A. Electronic Code Book (ECB)

The Electronic Code Book (ECB) mode is the simplest of all. It has known weaknesses, and thus is generally not recommended. The block scheme can be seen in figure 8. The plaintext is divided into blocks, the size of which is the length of an AES block, i.e. 128 bytes. So, this mode needs to pad data until their size is the same as the length of the block. Then, the data is encrypted using the same key and algorithm each time. Two equal plaintexts will result into two equal ciphertexts, meaning that this mode of operations does not provide a lot of security. Additionally, since the encryption and decryption processes are independent of each other, the data can be encrypted/decrypted in parallel. If a block of plaintext or ciphertext is broken, it won't affect the other blocks. ECB was often used in databases, where it encrypted tables, indexes and system catalogs.

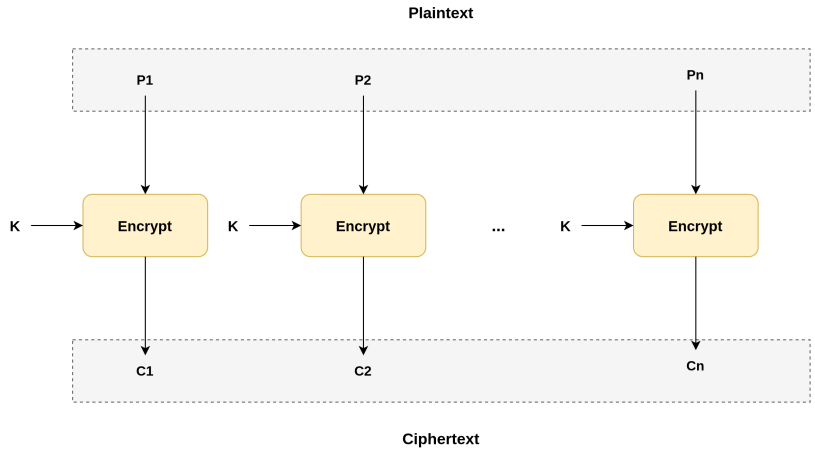


Figure 8: *AES-ECB Block Scheme*

B. Cipher Block Chaining (CBC)

The cipher-block chaining (CBC) mode trumps the ECB mode in hiding away patterns in the plaintext [8]. It achieves this by using an initialization vector (IV), which has the same size as the block that is encrypted. In general, the IV usually is a random number, not a nonce. The IV is XORed with the first plaintext before encryption. Then, CBC involves block chaining in the sense that every subsequent plaintext block is XORed with the ciphertext of the previous block. This is better illustrated in figure 9.

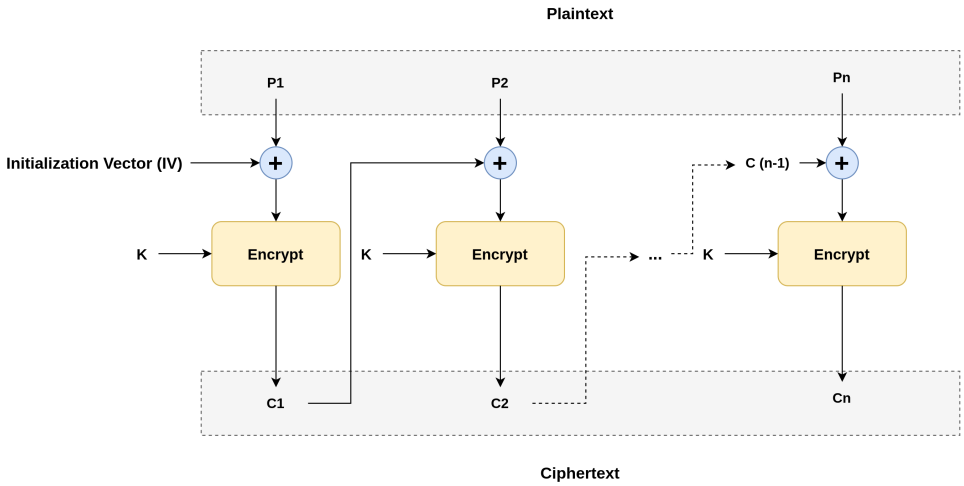


Figure 9: *AES-CBC Block Scheme*

The advantage over ECB is that, with CBC mode, identical blocks do not have the same cipher, due to the addition of the initialization vector. Hence, same blocks in different positions will have different ciphers. This also means that CBC is not tolerant of block losses, and that the process needs to be done sequentially, and not in parallel. However, decryption can be done in parallel, if all ciphertext blocks are available, and can tolerate block losses.

C. Cipher FeedBack (CFB)

The Cipher FeedBack (CFB) mode is similar to CBC in the sense that for the encryption of a block, the cipher of the previous block is required. First, CFB will encrypt the initialization vector, then it will XOR the output with the plaintext to get the ciphertext. This mode does not require data padding, since it will not encrypt plaintext directly. CFB does not use a decryption function. In order to decrypt data, all we have to do is the reverse the plaintext and ciphertext sections, seen in figure 10. This mode is generally faster than CBC, and is also non-deterministic, which means that it does not reveal any patterns the plaintext may have. Its disadvantages are, like CBC, that it can not tolerate block losses, nor can the blocks be encrypted in parallel.

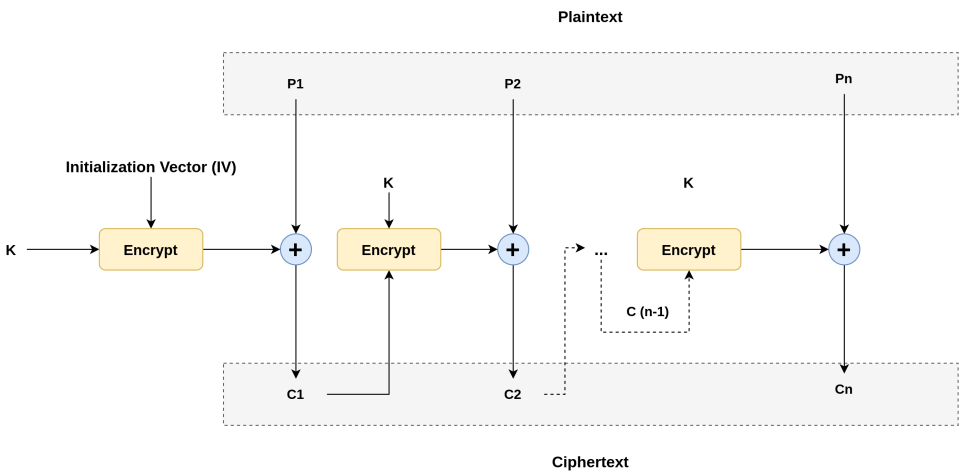


Figure 10: *AES-CFB Block Scheme*

D. Output FeedBack (OFB)

The Output FeedBack (OFB) mode is similar to CFB, with one key difference. OFB relies on XORing plaintext and ciphertext blocks with expanded versions of the initialization vector. Like CFB, it enables a block encryptor to be used as a stream encryptor, and does not need data padding. It also uses a single encryption algorithm for both encryption and decryption. OFB's block scheme can be seen in figure 11.

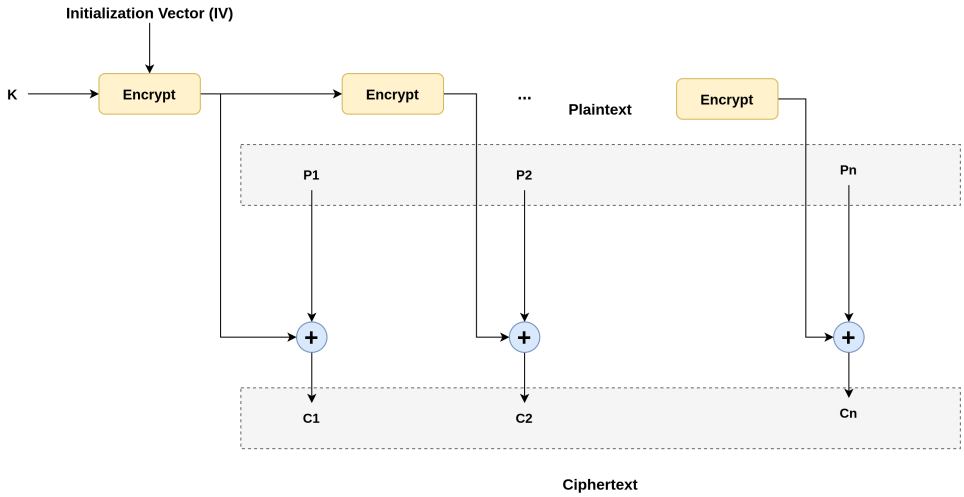


Figure 11: *AES-OFB Block Scheme*

In OFB, both encryption and decryption of blocks can be done in parallel, once the expanded initialization vectors have been generated. The lack of interconnection means that OFB is tolerant to block losses. Though, OFB has a major disadvantage, and that is the repeated encryption of the initialization vector, because it may produce a state that has already been produced before. While a highly unlikely scenario, it remains a significant security flaw if it indeed happens, and the result would be that the plaintext will start to be encrypted with the same data as it was previously.

E. Counter (CTR)

The Counter (CTR) mode enables every step to be done in parallel. It is similar to OFB, since it also involves XORing a sequence of expanded vectors with the plaintext and ciphertext blocks. What sets it apart from OFB, is that these expanded vectors in CTR are generated using the value of a counter as an IV . The counter has the same size as the used block. Just like OFB and CFB, CTR also makes use of a single encryption algorithm for both encryption and decryption. It is considered to be very secure and efficient mode for most purposes. Though, CTR is a synchronous counter that needs to be maintained both by the sender and the recipient. If this counter is not synchronized correctly, it could lead to wrong plaintext recovery. CTR's block scheme can be seen in figure 12.

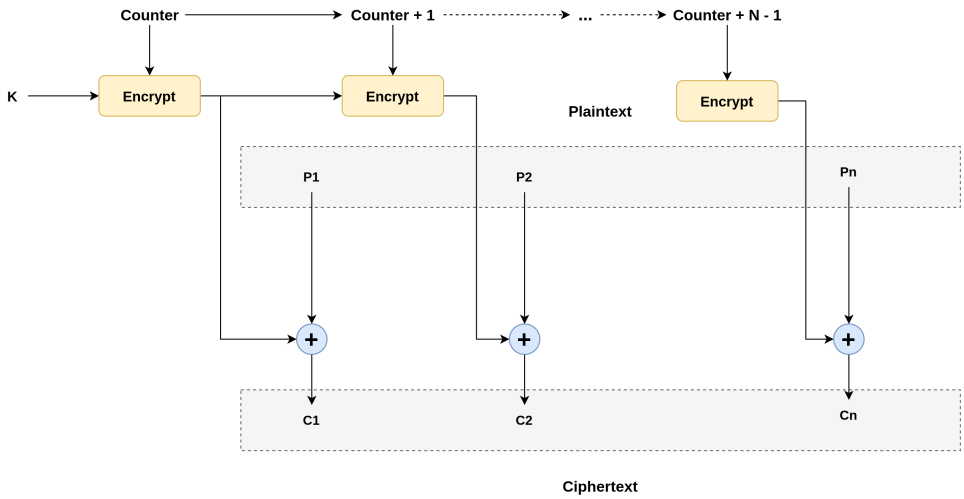


Figure 12: *AES-CTR Block Scheme*

1.4 Data Padding

In cryptography, padding is any of a number of distinct practices which all include adding data to the beginning, middle, or end f a message prior to encryption [43]. Padding may include the addition of phrases that don't make sense, to an alphanumerical string, in order to obscure the fact that many messages end in predictable ways. One padding method available is the PKCS method. The rules for PKCS padding are simple [21]:

- The padding bytes are added to the plaintext before encryption.
- Each padding byte has a value equal to the total number of padding bytes that are added.
- The total number of padding bytes is at least one, and is the number that is required in order to bring the data length up to a multiple of the cipher algorithm block size.

Two very popular modes for PKCS padding are PKCS#5 and PKCS#7. Their difference is that PKCS#5 padding is defined for 8-byte block sizes, while PKCS#7 padding would work for any block size from 1 to 255 bytes. So, fundamentally, PKCS#5 is a subset of PKCS#7 for 8-byte block sizes. Hence, PKCS#5 cannot be used for AES. It was only defined for DES and 3DES encryption algorithms in mind. Note, that neither PKCS#5 nor PKCS#7 is a standard created to describe a padding mechanism. The padding part is only a small subset of the defined functionality. PKCS#5 is a standard for Password Based Encryption (PBE) [23], and PKCS#7 defines the Cryptographic Message Syntax (CMS) [22]. PKCS#7 CMS is a standard for 16-byte block ciphers

such as AES.
In summary:

- PKCS#5: The passing string PS shall consist of $8 - (||M|| \mod 8)$ octets all having value $8 - (||M|| \mod 8)$ [23].
- PKCS#7: For such algorithms, the method shall be to pad the input at the trailing end with $k - (l \mod k)$ octets all having value $k - (l \mod k)$, where l is the length of the input [22].

1.5 Brute-Forcing

Brute forcing is an attempt to guess a secret, in which the attacker has the ciphertext and the encryption algorithm used, and performs an exhaustive (brute force) search on the key space to determine the key that decrypts the ciphertext to obtain the plaintext. This attack does not require much skill, but requires a powerful enough computer with sufficient hardware, like CPU, RAM and fast hard drive. The exact requirements vary depending on the of the job, and the required completion time. The attack can be executed both online and offline. In the case of an online attack, the attacker needs to interact with the target system to which he is trying to gain access.

Let's take AES-256 for an example, that we discussed in section 1.3.4, to see how long it would take exactly to brute-force this algorithm, without using any padding and complicated mode of operations. Assume that we have a single high performance PC, decrypting AES-256 at around 128MiB/sec per core. On a 4 core machine with hyperthreading (meaning that there are 8 concurrent threads), that equals to 1024MiB/sec or 2^{30} bytes per second. AES uses a 16-byte block size, so this high performance PC can encrypt $2^{30-4} = 2^{26}$ blocks per second, i.e. 2^{26} different encryption keys per second. In a year (31,557,600 seconds), the number of keys tested would be $31,557,600 * 2^{26} = 2,117,794,686,566,400$ keys. Now, let's assume that one can find the answer after searching half the key space. This equals to 2^{255} keys. The total time to perform this attack is $\frac{2^{255}}{2,117,794,686,566,400} = 27$ trillion trillion trillion trillion trillion years.

1.6 Thesis Road-map

The thesis is organized as follows. Section 1 was an introductory chapter with some foundational knowledge. In section 2, we will discuss android app development, along with its more important aspects. Section 3 will illustrate why we decided to develop ZenCrypt, and its implementation. Continuing, section 4 will demonstrate some example usages of the application. In section 5, we will analyze the performance of ZenCrypt, and in section 6 we will link its online presence. Finally, section 7 will present some conclusions and further development.

2 Android App Development

This section will cover some fundamental principles when developing an application for android, which will help in understanding the later sections.

2.1 Mobile Project Types

There are many ways that you can go about and develop a mobile application, each one having its own strengths and weaknesses. There are three basic mobile project types: native apps, web apps, and hybrid apps. Let's break down each one of them.

2.1.1 Native Apps

Native apps are developed using the mobile's default programming language (in android's case, java and/or kotlin). These kind of apps are compiled and executed directly on the device. They have access to a broad range of SKDs (APIs) which let them communicate with the operating system at hand, in order to retrieve device data, sensor values, or load data from an external source using HTTP requests. This puts no limits an the app usage.

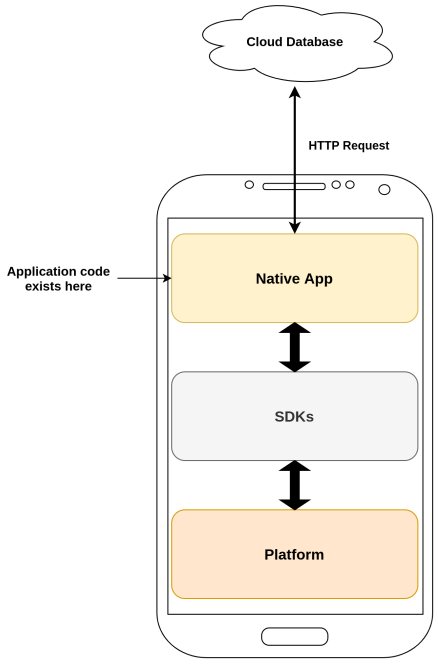


Figure 13: *Native Apps Scheme*

Since native apps take advantage of the native system's APIs, they are tightly connected to it. This leads to a number of benefits, the main of which is the performance.

Native apps tend to deliver the highest levels of performance, compared to the other development types. Since they are also written using native SDKs, it is common for developers to use frameworks in their native apps, which not only makes development easier, but it is very helpful for developers that are already familiar with the languages used. With that said, native apps do not come without drawbacks. For example, they are not cross platform ready. This means, that native apps can be developed for one platform at a time, which can be tedious if the target audience is more than just android phones. They also require a high level of work effort, and advanced knowledge of the platform's APIs and language.

2.1.2 Web Apps

Web based applications behave in a very similar fashion to native apps. They require a certain browser in order to run and are commonly written in CSS, JavaScript and/or HTML5. Simply put, they are websites viewed on the device through a browser, with the exception of being designed to fit a mobile screen size. Many websites these days have one version of their site designed for viewing through a PC, and a one version designed to be viewed through a mobile device. Another popular approach is the responsive web design, in which there is a unique app site, that re-flows the viewed page adjusts to the form factor and screen size of a device, to fit better in both smaller and larger screens.

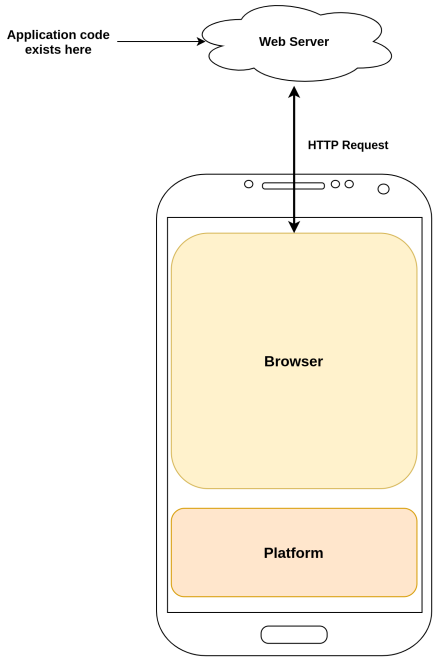


Figure 14: *Web Apps Scheme*

The key selling point of web apps is that they require a minimum of device memory in order to run. End-users can simply access them from any device that has a functioning internet connection. Though this also means that since all of the databases are stored on the server, a poor internet connection equals bad user experience. Web apps are easy to maintain, require no installation and are cross platform ready. On the other hand, they have no access to native API calls, and rely on just the APIs provided by the browser at hand.

2.1.3 Hybrid Apps

Hybrid apps try to implement the best of both worlds. They are essentially web applications created in a native wrapper, meaning they use elements of both native apps and web based applications. This wrapper is responsible for the communication between the native device platform and the web view. This enables the app to have access to the device, and retrieve data and sensor values. The development of such apps is possible using tools that have been specifically developed to act as the "middle man" between the platform and the web view, such as Apache Cordova. They are not officially supported by Android (or iOS for that matter), and are third party solutions.

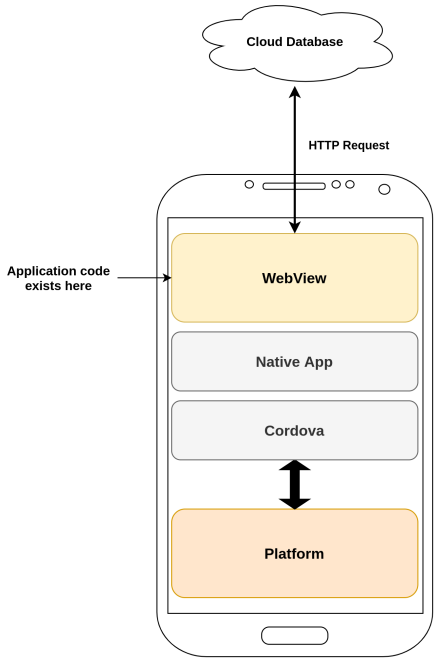


Figure 15: *Hybrid Apps Scheme*

Hybrid apps, like web apps, are cross platform, and do not require more skill than web development. They solve the device accessibility problem using the native wrapper,

and are relatively easy to develop. On the other hand, they are bound by the web view limitations, which results in the lack of speed, performance and overall optimization compared to native apps. Additionally, in order to make use of certain native API calls, the developer must install plugins that support those calls, or even develop one from scratch.

2.2 Understanding the Android Lifecycle

ZenCrypt was developed as a native application, so we will focus on this approach of Android programming. Let's start with a fundamental functionality, the app's lifecycle.

2.2.1 The Role of Lifecycle in Apps

To better understand the lifecycle's role, we must first discuss some basic operating system functionality. After all, Android is an operating system, and more specifically a multi-user *Linux* system. This means that, most of the time, each application runs in its own isolated Linux process. Every time an application component needs to be executed, the OS creates a process to host its functionality. On the other hand, when no such component is is running, the OS kills that specific process in order to free up memory for other applications that need to run. To facilitate this, the Android operating system uses an *importance* hierarchy to find out which of the processes need to be executed or be killed. It is based on a categorization of the processes as unique types, that mainly depend on the app components that are running at the time, as well as their respective state [3].

2.2.2 The Activity Lifecycle

One of the most common app components is called an *Activity*. An app can have one or more of such components. Now, depending on how the end user uses the app, these activities go through different lifecycle states. This is important to understand, because the functionality of different components results in a different process lifetime. Failing to develop correctly these components, can have a major impact on the application, such as the system killing it when it has still work to do. Figure 16 shows the different activity lifecycle states. Those are:

- **Initialized**: The instance of the activity is created along with the initialization of its necessary properties.
- **Created**: The activity's UI is ready to be configured.
- **Started**: The activity is now visible to the user.
- **Resumed**: The activity has focus, meaning that the user is likely interacting with it.

- **Destroyed**: The activity is destroyed.

There are different callbacks between these states, and the operating system invokes these callbacks during an activity's state change. If a developer wants a different functionality than the default one, these methods can be overridden according to the desired implementation.

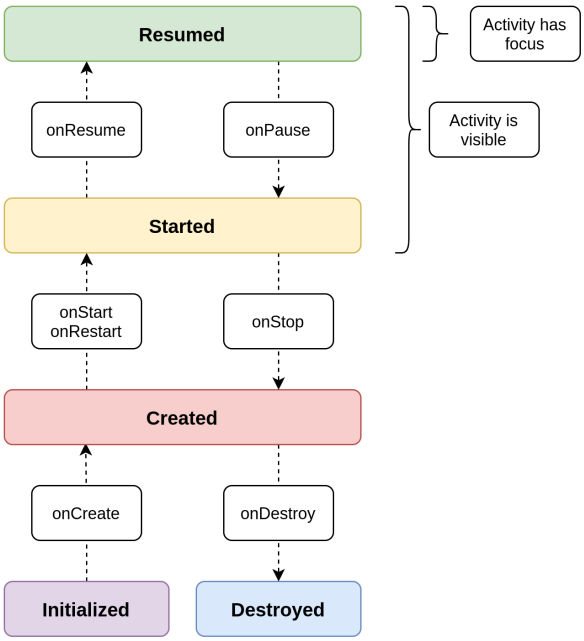


Figure 16: *Activity Lifecycle States*

2.2.3 Activity Lifecycle Callbacks

In order to monitor the activity lifecycle, we must decide which of the lifetime state(s) we want to focus on [4]:

- **Entire Lifetime**: This takes place between the first call to *onCreate()* and the final call to *onDestroy()*. *onCreate* is responsible for for the activity's global setup, and any remaining objects are released in *onDestroy()*.
- **Visible Lifetime**: This takes place between the *onStart()* and *onStop()*. Any resources that were initialized before can be maintained here, if the developer chooses to do so.
- **Foreground Lifetime**: This takes place between the call to *onResume()* and the call to *onPause()*. It is triggered multiple times when, for example, the device goes to sleep. This means that the code that is executed in these methods must be relatively lightweight, and not handle tedious tasks.

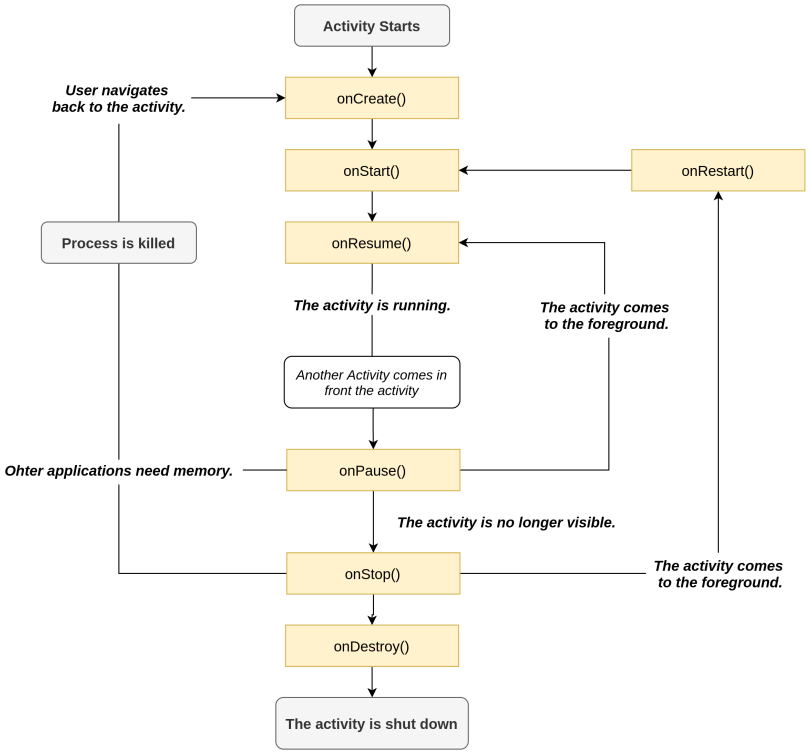


Figure 17: *The Activity Lifecycle*

Figure 17 shows the movement of an activity throughout its lifecycle. More specifically:

- *onCreate()*: When the activity is first created, this methods is called, which initializes most of the activity's variables. It is not a killable process. The next process to be called is always *onStart()*.
- *onRestart()*: This method is called when the activity is stopped, but before it starts again. It is not killable, and next is always *onStart()*.
- *onStart()*: This is called when the activity becomes visible. Next is *onResume()* if the activity is brought into foreground, or *onStop()* if it is hidden from the user. It is not killable.
- *onResume()*: This is called when the activity is ready to be interacted with the user. The activity stack has the current activity on top of its stack. It is not killable, and next is *onPause()*.
- *onPause()*: This is called when the system decides that a previous activity is about to be resumed, or if the user has navigated to a different system screen (for example pressing the home button). Here, unsaved changes/data are persisted. This process is killable on android 2.3 and earlier, and next is *onResume()* if the

activity is brought to the foreground again, or *onStop()* if the activity becomes invisible.

- *onStop()*: This is called when the activity at hand is no longer visible. It's followed by *onRestart()* if the activity is returned to interact with the user, or by *onDestroy()* if the activity goes away. This is a killable process.

- *onDestroy()*: This is the final call before the destruction of the activity. It's called when the developer calls the *finish()* method, or when the system decides to temporarily destroy the activity to claim back some memory space. In order to know which of the two events is happening, the developer can make use of the *isFinishing()* method, which returns true if the activity is finishing, or false if the system is killing it.

Remember that the system can kill specific methods unexpectedly, meaning that *onDestroy()* should not be used to do things that are intended to remain after the process finishes.

2.2.4 The Fragment Lifecycle

Fragments were created to address the problem of putting multiple activities on the screen at the same time, since Android allows only one activity to be displayed. Using fragments, you can make use of the bigger screen sizes available, and not waste any screen real estate, for example in tablets. On top of that, multiple fragments can hold different views to be displayed at the same time, allowing one to develop a more intuitive and appealing application design. Fragments are essentially miniature activities, and every fragment has its own lifecycle. They reside inside activities, and are commonly used to display data to the user. Though, it's important to know that a developer must never perform IO operations on the interface thread (which fragments are running on), which may result in an "Application Not Responsive" dialog box to be shown to the user. Figure 18 demonstrates the fragment lifecycle, which resembles a lot to the activity lifecycle.

2.2.5 Fragment Lifecycle Callbacks

The following is every lifecycle event that can be triggered in the fragment's lifecycle:

- *onCreate()*: This is called when the fragment reaches its *Created* state. It's very similar to the activity's *onCreate()* callback.

- *onCreateView()*: This is called in order to *inflate* or to *create* the fragment's view.

- *onViewCreated()*: The fragment's view is instantiated with a view object that is *NOT* null. The view associated with the fragment can be returned from *getView()*.

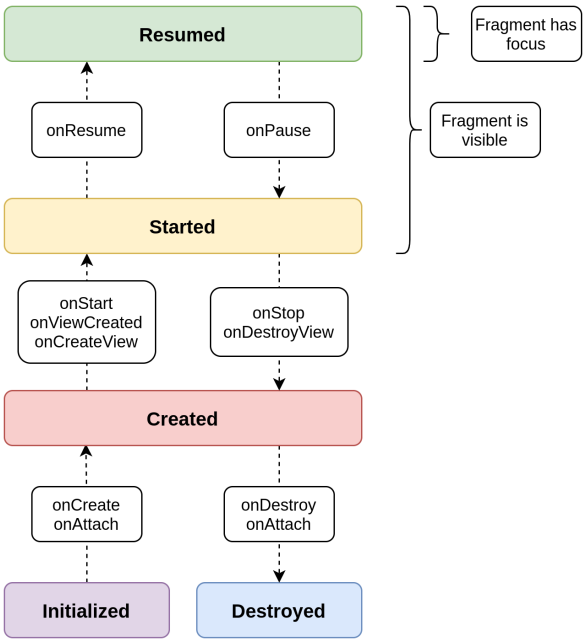


Figure 18: *The Fragment Lifecycle*

- *onStart()*: This is called when the fragment reached its *Started* state. The view is guaranteed to be available and that it's safe to perform a *FragmentTransaction*.
- *onResumed()*: The fragment now enters the *Resumed* state. This is available only after any animation and transition effects have completed their work.
- *onPause()*: This is called when the OS realizes that the user begins to leave the fragment at hand, while it's still visible. The fragment now goes back to the *Started* state,
- *onStop()*: The fragment is no longer visible and returns back to the *Created* state.
- *onDestroyView()*: This is called when the fragment's view has been detached from the screen and all animations and transitions have completed their work. The garbage collector can now collect the fragment's view object.
- *onDestroy()*: This is called when the fragment is removed or when the *FragmentManager* object is destroyed. Now, the fragment enters the *Destroyed* state and has reached the end of its lifecycle.

2.2.6 Store UI Data with ViewModels

The ViewModel object is a class designed to store and manage UI-related data in a lifecycle conscious way [18]. This information is retained through configuration changes,

such as screen rotations. They are very helpful in a handful of situations. For example, imagine that the system destroys or re-creates a UI-controler (such as an activity or a fragment). Any transient UI-related data that is stored in the will be forever lost. For simple data, an activity can make use of the *onSaveInstanceState()* method, and later restore its data from the bundle in *onCreate()*. Though, this approach is only applicable when the data set is very small, and when the data can be serialized and de-serialized. Additionally, it is very common that a UI controller needs to make some asynchronous calls, which could potentially finish after a while. The controller needs to monitor these calls and make sure that the OS cleans them up after the main object is destroyed, in order to avoid serious memory leaks. This requires a lot of code maintenance, and could potentially break the application in the event of a screen rotation, since the object will be re-created and restart calls that were already made before. Since UI controllers are intended to primarily display UI data, and not data loading, for example, from a database, a developer must not burden them with such tasks, since it bloats the class. This is where ViewModels come into play, allowing to separate the view data

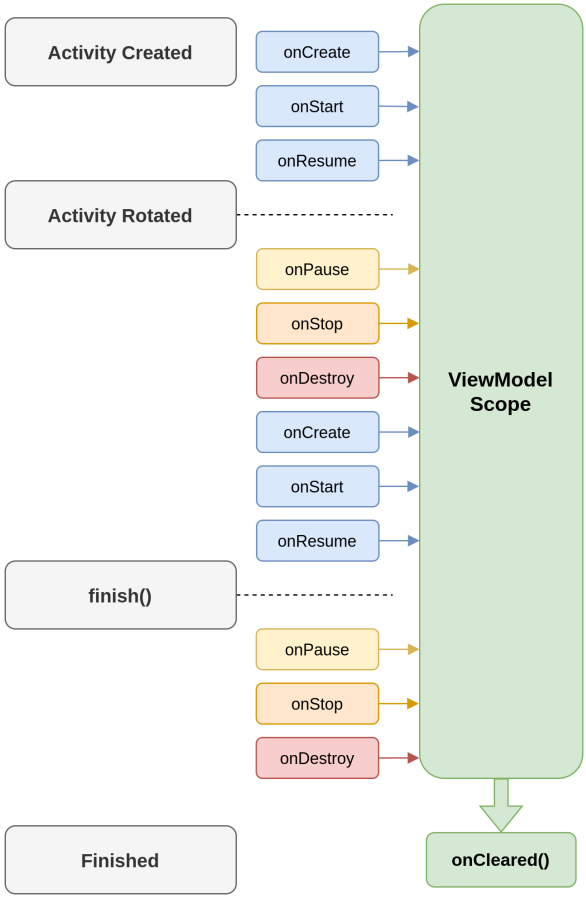


Figure 19: *The ViewModel's Lifecycle*

from the UI controller logic. ViewModel objects are automatically retained during configuration changes, and the information stored in them is available for the next UI controller instance. Figure 19 shows the lifecycle of a ViewModel object. These objects are scoped to the Lifecycle of the *ViewModelProvider*. They remain in the memory until their Lifecycle scope goes away permanently. This happens when an activity is finished, or when a fragment is detached.

2.3 Data Storage

There are many occasions where android applications need to to implement data persistence, either to store user preferences that the app offers, or important/sensitive information such as files, passwords, account credentials etc. For example, a note taking app wouldn't be useful at all if it didn't save the notes between device restarts, or even application restarts. This where the Android's framework, along with the preferred programming language (Kotlin/Java) come into play. Together they provide a set of tools to help the developer tackle the problem that is data storage.

2.3.1 Storage Options

The first thing to consider when developing an app that requires data persistence, is the kind of information that it needs to store. Android provides a variety of places where one can store data, and its usage depends on how the app is meant to be operated. In the end, it all comes down to whether or not the app interacts with other applications, and needs to save information in a public space, where it is easily accessible and readable. For example, a photo viewing app would possibly store downloaded photos in a place where they can be also viewed by other applications on the user's device. On the other hand, an app that stores sensitive data, such as user credentials, would encrypt them and place them in a secure, local storage environment, where they are inaccessible by other installed services. Android provides the following storage options:

- **Shared preferences**: This stores private data using key-value pairs in XML files, located in the app's private working directory (/data/data/app). This information is inaccessible by other apps and is commonly used to store the user's app-related preferences, such as colors/themes, various inputs, favorites, etc. It is not advised to store sensitive data here, at least unencrypted, since these files can be easily read from an unlocked/compromised device with root privileges granted, like any other Linux based system.

- **Jetpack DataStore**: This is a data storage solution much like the shared preferences described above, but has a few major enhancements. It's still a key-value pair system, but it stores these pairs or typed objects using *protocol buffers*. Protocol buffers are Google's language-neutral, platform0neutral, extensible mechanism for serializing structured data [16]. It's faster than XML, smaller and simpler. The data structure is generated only once, and then the developer can

use any special generated source code to easily write and read them. It is implemented using Kotlin's coroutines and Flow (more on that later).

- **Internal storage**: This is a location for storing files on the device's main storage. By default, information stored here is private to the application, and not accessible by other services. Note, however, that once the application is uninstalled, these data are deleted as well. This solution is commonly used to store files that do not contain sensitive data, and are only used by the application at hand. A developer can override the default behavior, and make the files accessible by other apps, such as a file manager. Though file deletion on app uninstall cannot be de-activated.

- **Local cache**: Sometimes an application needs to store cached data, such as a downloaded image that it is not needed once the app is closed (a common example are profile photos). These files can be deleted at any time with the help of *getCacheDir()* method, available in *Context* objects in Android.

- **External storage**: All Android versions support a shared storage for files, like SD cards, or non-removable internal storage. All the files stored here are *public*, and can be modified by any person or application. They provide no security what so ever, and can be read by a computer using a USB cable. In order to start working with external storage, the developer must make use of the *Environment* object, which provides methods such as *getExternalStorageState()* and *GetExternalFilesDir()* to help with the task. This is where it gets a little tricky though. In order to enforce security, and not allow malicious apps to arbitrarily delete/modify the user's stored data, Google has introduced *Scoped Storage* in Android 11 onwards, which cripples an app's capability of writing and modifying files on the external storage. This will be further discussed in section 2.3.2.

- **SQLite database**: This is the common implementation of the well-known SQLite database that is available across all platforms. It is fully supported by Android, and all of its functionality is available through Java/Kotlin.

- **Content provider**: A content provider is a wrapper that encapsulates data and provides it to applications. They are offered by a single *ContentResolver* interface, and the content provider is only required if it is needed to share data between multiple applications [13]. A common usage of content providers is to read and write application data which can be stored in files, or even SQLite databases. When a request is made through a *ContentResolver* instance, the system inspects the authority of the given URI (Uniform Resource Identifier), and then redirects the request to the provider responsible for handling requests of that specific authority.

- **Remote storage**: This refers to any storage that is not stored on the device, and is located in a remote data source that is accessible. Files from here can be

retrieved by any means necessary, such as the HTTP protocol.

An overview of the android's storage handling can be seen in figure 20.

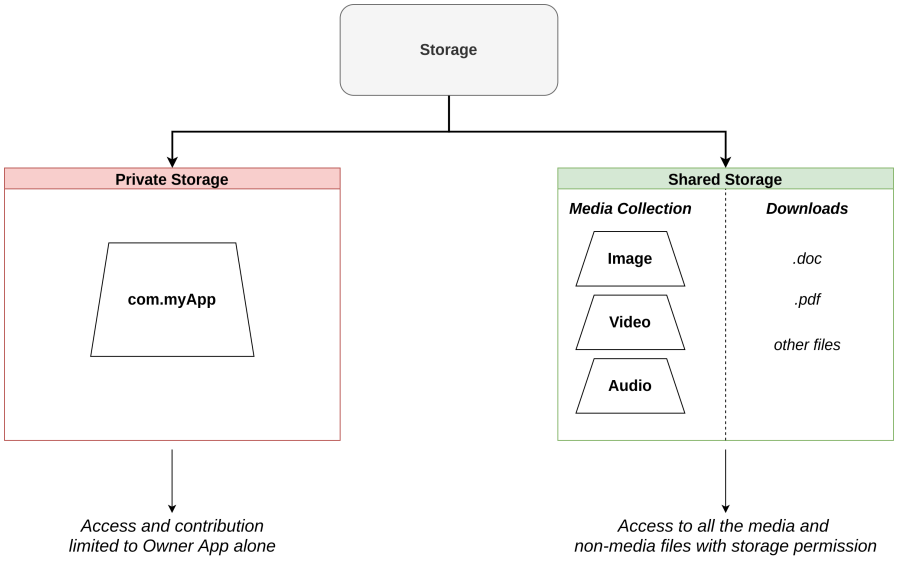


Figure 20: *Android's Storage Overview*

Moreover, figure 21 displays the above more in depth [31]. The */data/data* represents the app's internal storage, while the */data/media* path provides access to the shared directory */storage/emulated* where an application can store and modify files (given permission).

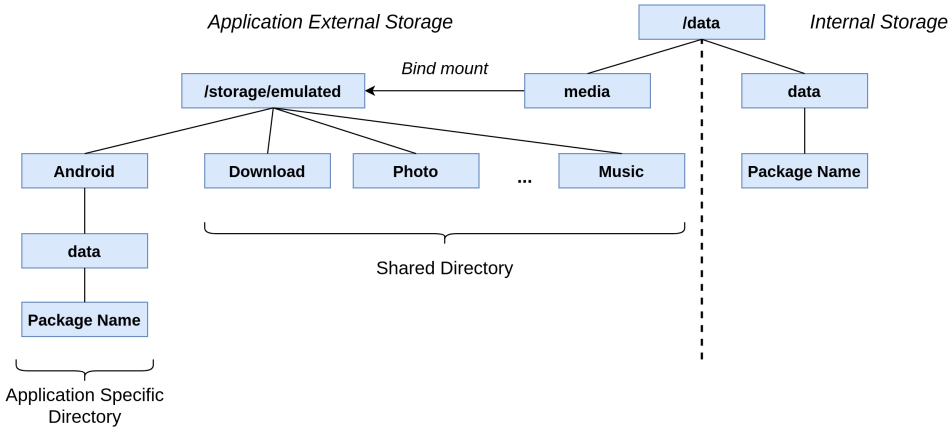


Figure 21: *Application Storage Structure*

2.3.2 Scoped Storage

Scoped storage was originally (and experimentally) implemented in Android 10, and later was fully adapted and mandatory in Android 11. Scoped storage aims to provide improved app and user data protection in external storage. It involves two major restrictions of file sharing/editing:

1. All application files stored into the app's specific directory are treated as *private* files.

2. All application files stored into shared directories are treated as *public* files, which need the user's consent via the Android's permission system to be read/shared. For writing files, an explicit user consent is needed, via the *all-files access* permission, which is only granted if the app is vetted before publication via the Google Play Store.

By default, all applications targeting Android 11, can read and write only their *own* files. Additional permissions are needed in order for an application to modify any file, but only one file at a time, and only for public files that are stored in shared directories.

With all of that said, applications currently have three ways to handle files inside scoped storage. The existing APIs that Android provides, which result in Posix calls, the *MediaStore* API, and the *Storage Access Framework (SAF)* API.

- **MediaStore APIs**: Like the *File* API, MediaStore APIs provide comparable file control access to that available in pre-scoped storage, but with scoped storage there is a file access limit, especially for writing. More specifically, the "Write External Storage" permission is considered deprecated in scoped storage, thus resulting in problems in shared directories via the File API. Third party apps cannot ask users to modify/write files which are owned by other applications for example. The only exception to this rule is, as discussed above, the *all-file access* permission. In order to write a public file which is owned by another app, the developer must use the new MediaStore API and obtain user consent. Note, however, that this approach has some limitations. For starters, it cannot be used with directories that are considered internal storage by other applications, as seen in figure 21. Also, it cannot be used when the target file is not a media file, like documents (.pdf).

- **Storage Access Framework (SAF)**: This framework was first introduced in Android 4.4 (API level 19), and will provide file access to paths that are chosen by the user [14]. An application that requires to implement functionality for sharing local and/or cloud files, can do so easily by implementing the *DocumentsProvider* class, which is capable of handling app file requests. In pre-scoped storage, SAF allowed directory level sharing from external storage, but since Android 11, scoped storage induced limitations to the SAF. For example, applications can no longer access the external storage's *Download* directory (/storage/emulated/0/Download).

Scoped storage is implemented using an adoption of the FUSE (Filesystem in USERspace) system for the external storage. It consists of two main entities; the kernel space and the user space. The kernel space (driver) is invoked by the operating system when an app tries to access files stored on the external storage, which then passes the request to the user space. The user daemon then retrieves the said request and tries to find the file owner from the *MediaProvider*, in order to check for sufficient permissions. If all goes well, the operation is performed on the EXT4 file system and the application is finally able to retrieve the result. This process is demonstrated visually in figure 22.

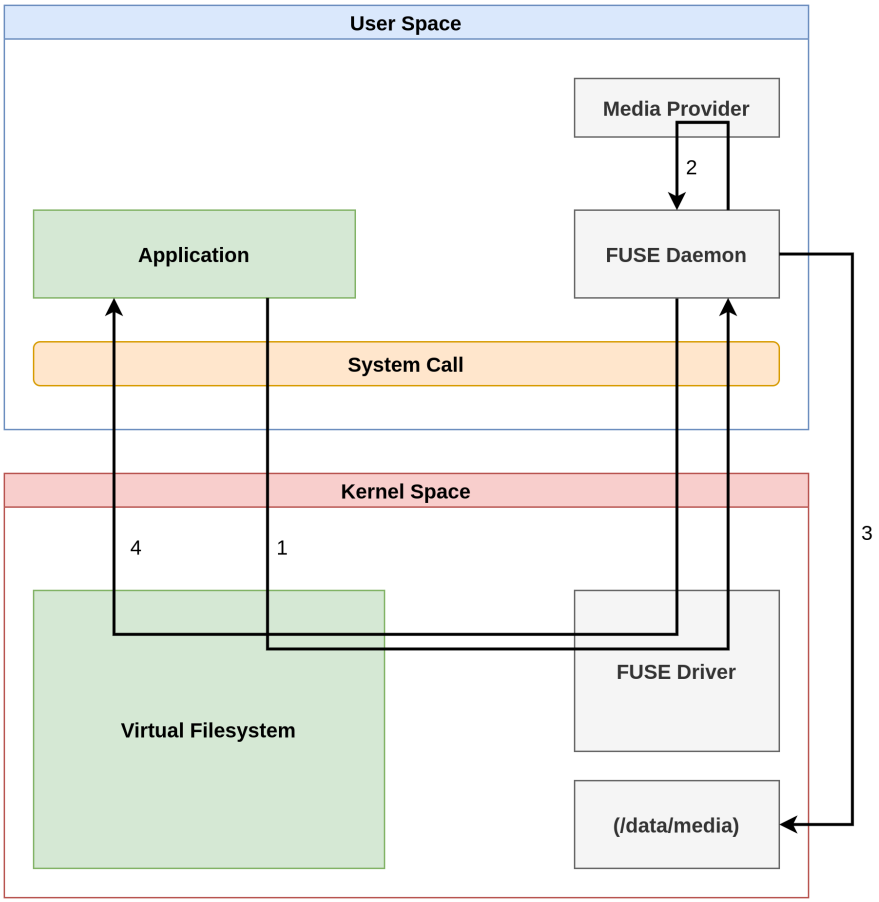


Figure 22: *Storage Storage's FUSE Implementation*

The bind mount operation is responsible for blocking third-party application access, using UNIX-DAC (Discretionary Access Control) permissions [31]. This method refers to file access in app-specific directories. To facilitate access enforcement for shard directories, a more tedious process takes place. Whenever an application is granted write access to a shared directory, and a new file is created, the *MediaProvider* takes this file and processes its owner and MIME (Multipurpose Internet Mail Extension) type,

which then stores in its database. The only requirement here is that the MIME type must strictly match the target folder's *intended* MIME type (for example, pictures must be stored in the Pictures folder). If this is not the case, then the operation will be stopped. *MediaProvider* can then be used by other apps to check for file instances, and if they own the specific file, the *MediaProvider* can grant them access. On the other hand, if the requested file is stored on a shared directory, it must either already have the "Read External Store" permission granted for reading, or prompt the user to allow the "Read External Storage" permission for both reading and writing.

Keep in mind that applications that heavily rely on storing files on the external shared storage, i.e. require the *all-file access* permission, are currently at a dead-end, since Google has paused app vetting on the Google Play Store. The only real alternative here is to comply with the scoped storage policy and store files on the internal app-specific storage.

2.4 Native Languages

There are many languages that can be used when developing Android applications. These include Basic, Kotlin, Java, Lua, C, C++ and C#. Though, in this thesis we will only focus on Kotlin and some Java.

2.4.1 Java

Java is the main language used by Android to communicate with the OS. It allows the developers to create programs that run on almost any platform, and it works seamlessly with most types of devices. Its robust code is built on top of object-oriented technology, and it prevents corruption or compromise of data from other applications. Java was originally crated by *Sun Microsystems* (which is now owned by Oracle), and is open-source. This enables the developer to take advantage of the many third party libraries available without worrying about safety, since many of them are maintained by trusted corporations like Apache and Google. While Kotlin is more robust and complex, Java is still very easy to use and compiles without hassle. This is why Java was the preferred Android language for years.

Though, there are some shortcomings when using Java, which led to many developers preferring Kotlin. For starters, Java is not able to access certain functionality when the user's device and/or equipment is incompatible. It is also not easy to take advantage of the functionality that the newer Java versions offer, since many of those are not available in mobile development. Additionally, a common issue that many Java developers face, is that Java is very cumbersome when it comes to API design, and there are many occasions where it will cause problems that take a lot of time to troubleshoot. Finally, Java is slower than most other programming languages, and is very memory hungry. This, especially in mobile development, is a major issue.

2.4.2 Kotlin

Kotlin is a programming language that simplifies the work of developers by making it less verbose and less prone to bugs. It compiles code to a byte code, which can be executed in the JVM (like Java). This means that any libraries and frameworks written in Java are ready to be implemented and run in a Kotlin application. It is also safe against *NullPointerException*, faster to compile, and very lightweight while maintaining small application size. Kotlin also offers coroutines, which can have a significant impact on the application's fluidity and speed, and make the user interaction more pleasant when heavy tasks need to run in the background.

There are only a few downsides to using Kotlin as the main application language, and all of them are based on the fact that Kotlin is a relatively newly developed language. This results in a development community that is limited compared to Java's community for example. There are fewer tutorials and libraries available, and has a steep learning curve because of its condensed syntax, which can be a real challenge for someone not familiar with these types of languages. This also reflects the fact that companies and start-ups need to search quite a bit before finding experienced Kotlin developers.

A brief comparison between the two can be seen in table 2 [12].

Parameter	Java	Kotlin
Compilation	Bytecodes	Virtual Machine
Null Safety	-	+
Lambda Expression	-	+
Invariant Array	-	+
Non-private Fields	+	-
Smart Casts	-	+
Static Members	+	-
Wildcard Types	+	-
Singletons Objects	+	+

Table 2: *Java vs Kotlin*

2.5 Kotlin Development

Kotlin is the clear winner when it comes to Java vs Kotlin [19], mainly because of its backwards compatibility, declarative style with less code, and the fact that it interworks with Java by default. ZenCrypt was developed mainly in Kotlin, so that's what we will focus on. Let's start by discussing some Kotlin basics, and then moving on to coroutines.

2.5.1 Basics

Kotlin offers a variety of tools that make the development of an application easy while maintaining a readable and straight-forward source. We will take a look at some of these tools below:

A. Extension Functions

Kotlin allows the developer to extend the functionality of a class without using any design patterns or inheritance for that matter [28]. For example, one can implement new functions for a class that is not modifiable. In order to write an extension function, we must first declare the *receiver* type as its name prefix, followed by the function name. An example *swap()* extension method of *MutableList<Int >* can be seen in listing 1.

```
//----
fun MutableList<Int>.swap(first: Int, second: Int) {
 val tmp = this[first] // 'this' points to the list
 this[first] = this[second]
 this[second] = tmp
}
//---- which can later be used as:
val my_list = mutableListOf(5, 6, 7)
my_list.swap(1, 0) // 'this' inside 'swap()' will hold the value of 'my_list'
```

Listing 1: Extension example.

Extension functions can also have a nullable receiver, as seen in listing 2. That way, the extension method is responsible for the null check, and is ready to be called directly.

```
fun Any?.toString(): String {
 if (this == null) return "null"
 // after the null check, 'this' is autocast to a non-null type,
 // so the toString() below
 // resolves to the member function of the Any class
 return toString()
}
```

Listing 2: Nullable extension example.

B. Smart Casts

Kotlin allows the developer to safely and automatically cast immutable values and inserts casts without needing to use explicit cast operators [30]. For example:

```
fun demo(x: Any) {
 if (x is String) {
  print(x.length) // x is automatically cast to String
 }
}
```

Listing 3: Smart cast example 1.

Additionally, the compiler is clever enough when a cast is safe, even if a negative check needs to be returned:

```
// x is automatically cast to String on the right-hand side of `||`
if (x !is String || x.length == 0) return

// x is automatically cast to String on the right-hand side of `&&`
if (x is String && x.length > 0) {
 print(x.length) // x is automatically cast to String
}
```

Listing 4: Smart cast example 2.

C. Type Inference

Kotlin allows to define variable types both explicitly and not explicitly. The compiler automatically identifies the data type of each variable, by the initializer. That means, if we initialize the value in the declaration, it is not needed to define the data type explicitly. But, if we want to initialize the value of a variable later on, then the data type must be defined explicitly, and the concept of type inference does not apply. This can be seen in the example below:

```
//---- not explicitly defined
fun main(args: Array <String> ) {
 val text = 10
 println(text)
}
//---- explicitly defined
fun main(args: Array <String> ) {
 val text: Int = 10
 println(text)
}
```

Listing 5: Type inference example.

D. Functional Programming

Functional programming is a programming paradigm where programs are constructed by applying and composing functions [42], i.e. writing applications using only pure functions and immutable values. This helps squash bugs easily, since we know what task each functions performs. Pure functions help in achieving a safer way of programming [11]. This is one of the most important things Kotlin has to offer, since it greatly increases the performance when compared to Java. As an example, let's say that we need to filter out negative numbers from a given collection:

```
fun main(args: Array <String> ) {
 val numbers = arrayListOf(15, -5, 11, -39)
 val nonNegativeNumbers = numbers.filter
 {
  it >= 0
 }
 println(nonNegativeNumbers)
}
```

Listing 6: Functional programming example 1.

The output of listing 6 will be: 15,11.

Additionally, we can create functions that take other functions as a parameter and also return function. Consider the following:

```
fun alphaNum(func: () -> Unit) {}
```

Listing 7: Functional programming example 2.

Here, the *func* is the parameter's name and *() ->Unit* is the type of the function. In other words, we are defining a function *func* that won't receive any parameters, and will also no return any value.

We can also take the above a step further and use a **Lambda expression**, i.e. a *function literal*, to calculate a result. Lambda expressions are functions which are not declared somewhere, but can be passed immediately as an expression:

```
val sum: (Int, Int) -> Int = {
 x,
 y -> x + y
}
```

Listing 8: Functional programming example 3.

Listing 8 simply declares a variable *sum* that has two integers as input, and returns their total as an integer. The above can also be declared in an anonymous function:

```
fun(x: Int, y: Int): Int = x + y
//---- OR
fun(a: Int, b: int): Int {
 return a + b
}
```

Listing 9: Functional programming example 4.

E. Null Safety

Kotlin is safe against *NullPointerExceptions*. This is great because Kotlin simply will not compile if it detects that somewhere in the source code a null parameter is assigned or returned:

```
// the following won't compile,
// since a null parameter tries to be assigned.
val name: String = nullbasics
// this also won't compile,
// since a null parameter tries to be returned.
fun getName(): String = null
```

Listing 10: Null safety example.

2.5.2 Coroutines

Kotlin, by default, provides very few low-level APIs in its standard library. In order to utilize coroutines, we must make use of other libraries that provide keywords such as *await*,*async*,*launch* and others. A library that provides rich coroutine functionality is *kotlinx.coroutines* which is developed by JetBrains. Kotlin also provides suspending functions that offer safe and asynchronous operations.

Let's dive further in. Coroutines are essentially instances of a suspendable computation [27]. It is very similar to *threading*, in such a way that it executes a block of code concurrently with the rest of the code. The only difference is that a coroutine can pause and resume its execution on different threads, and not a single particular one. Listing 11 demonstrates a simple Kotlin coroutine:

```
import kotlinx.coroutines.*

fun main() = runBlocking { // this: CoroutineScope
 launch { // launch a new coroutine and continue
  delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
  println("World!") // print after delay
 }
 println("Hello") // main coroutine continues while a previous one is delayed
}
```

Listing 11: A simple coroutine example.

The result of the above will be:

Hello
World!

In essence, this is what listing 11 does:

- *runBlocking*: runBlocking ensures that the thread that is executing the code inside the curly braces, gets blocked for the duration of the call. Once all the coroutines inside this block finish their task, then the thread gets unblocked. Here, everything will be executed in the *main* thread. Keep in mind, though, that threads need to be blocked as few as possible in the application's lifetime, since they are one expensive resource. runBlocking is a coroutine builder.

- *launch*: This launches a newly created coroutine *in parallel* with the rest of the application code. It is also a coroutine builder. If the *runBlocking* is removed, then *launch* is not recognized as a valid keyword, since it is missing its declared scope, which is the *coroutineScope*.

- *delay*: This *suspends* the coroutine for a given amount of time. It's called a *suspending function* that does not block the thread, while allowing other coroutines to be executed on said thread.

Another example of a *coroutineScope* builder can be seen in listing 12. It can reside in any suspending function and can be used to execute as many coroutines as needed (in parallel).

```
import kotlinx.coroutines.*

// my_function will run followed by "Finished"
fun main() = runBlocking {
 doWorld()
 println("Finished")
}

// Concurrently executes both sections
suspend fun my_function() = coroutineScope {
 launch {
  delay(3000L)
  println("My Fynction 2")
 }
 launch {
  delay(2000L)
  println("My Function 1")
 }
 println("Started"
}
```

Listing 12: Two coroutines in parallel.

The output of the above will be:

Started
My Function 1
My Function 2
Finished

Since both *launch* blocks are executed concurrently, the first to finish will print *My Function 1* after 2 seconds, then after a total of three seconds from the start, the second *launch* block will finish and print *My Function 2*. Then, once the *my_function* returns, the text *Finished* will be printed.

Kotlin coroutines are surprisingly light weight. Consider the following code:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
 repeat(100_000) { // launch a lot of coroutines
  launch {
   delay(5000L)
   print(".")
  }
 }
}
```

Listing 13: 100K coroutines.

What listing 13 does, is that it executes 100.000 coroutines, each one printing a dot after 5 seconds. This will run without any problems or out-of-memory errors.

2.5.3 Coroutine context and dispatchers

Coroutines in Kotlin are always executed in a context. This context is always represented using a value of the *CoroutineContext* type [26]. Each coroutine context comprises of various elements, the main of which are its *job* and its *dispatcher*. The *job* refers to a background job which is usually cancellable and is created with the *launch* builder that we described above. The *dispatcher* is what determines the thread or threads that a coroutine will use for its execution, and it can:

- Execute a coroutine in a specific thread
- Dispatch the coroutine to a thread pool
- Let the coroutine run unconfined

Dispatchers are a parameter that can be defined in all coroutine builders. For example:

```
1  import kotlinx.coroutines.*
2
3  fun main() = runBlocking<Unit> {
4   launch { // context of the parent, main runBlocking coroutine
5    println("main runBlocking: Running in ${Thread.currentThread().name}")
6   }
7   launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
8    println("Unconfined: Running in ${Thread.currentThread().name}")
9   }
10  launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher
11   println("Default: Running in ${Thread.currentThread().name}")
12  }
13  launch(newSingleThreadContext("my_thread")) { // will get its own new thread
14   println("newSingleThreadContext: Running in " +
15    + "${Thread.currentThread().name}")
16  }
17 }
```

Listing 14: Coroutine dispatcher example.

The output of listing 14 will be the following:

Unconfined: I'm working in thread main
Default: Running in DefaultDispatcher-worker-1
newSingleThreadContext: Running in MyOwnThread
main runBlocking: Running in main

The order of which the messages are printed might be different on separate executions.

The difference between a confined and an unconfined dispatcher is: the *Dispatchers.Unconfined* dispatcher will execute a coroutine in the caller thread, until the first suspension point. After that, the coroutine will be resumed in a thread which is decided by the invoked suspending function. On the other hand, a confined dispatcher (the coroutine's context was inherited from *runBlocking* for example), will reside on the *main* thread, and will continue to execute in that thread until it finishes.

2.5.4 Flows

In Kotlin's coroutines, suspend functions can only return a single value. If we wanted to implement a functionality that were to emit more than one values sequentially, then we would have to use a *flow* type [15]. Essentially, flows are streams of data which are computed in a asynchronous fashion. There are three entities in a stream of data, as seen in figure 23:

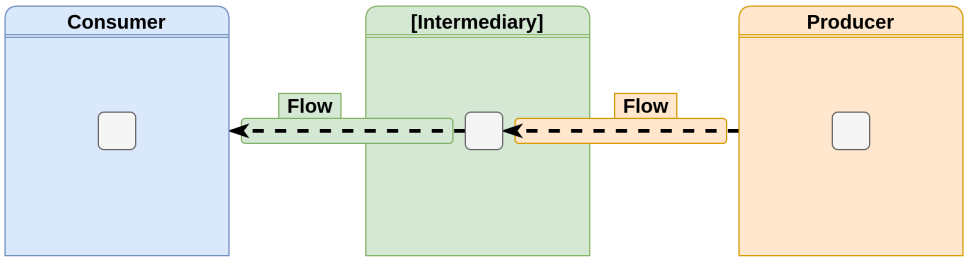


Figure 23: *Entities in a stream of data*

- The *producer* produces the necessary data which is added to the data stream. This can also be done asynchronously with the help of coroutines.
- There may also be *intermediaries*, which will take the data, modify them if necessary, and return them to the stream. They may also modify the data stream itself.
- Finally, the *consumer* is able to read and consume the data from the data stream.

```
1  import kotlinx.coroutines.*
2  import kotlinx.coroutines.flow.*
3
4
5  fun my_function(): Flow<Int> = flow { // flow builder
6   for (i in 5..7) {
7    delay(100) // some background work
8    emit(i) // emit next value
9   }
10 }
11
12 fun main() = runBlocking<Unit> {
13  // Launch a concurrent coroutine to check if the main thread is blocked
14  launch {
15   for (k in 5..7) {
16    println("Not blocked $k")
17    delay(100)
18   }
19  }
20  // Collect the flow
21  my_function().collect { value -> println(value) }
22 }
```

Listing 15: Flow example.

An example of flow usage in Kotlin can be seen in listing 15 [25]. Here, we can notice a few things. First of all, the function that builds a *Flow* is called *flow*. All the code inside it (flow { ... }), is able to suspend, which is why the *simple()* function does not need the *suspend* modifier. Finally, the values are emitted using the *emit()* function, and later on collected using the *collect()* function. The output of listing 15 will be:

Not blocked 5
5
Not blocked 6
6
Not blocked 7
7

Each number is printed every 100ms, all while the main thread remains unblocked. We can verify that this is the case, using a separate coroutine on the main thread, that prints the "Not blocked" message.

2.5.5 Flow Context

Flow collections will always happen in the calling coroutine's context. For example:

```
withContext(context) {
 my_function().collect { v ->
  println(v) // this runs in the specified context
 }
}
```

Listing 16: Flow context example.

The above code will will run the *simple* flow in the developer's specified *context*, even if the *simple* flow says otherwise. This is called *context preservation* [25], which means by default, *flow* { ... } will run in the same context as the one provided by the collector of that specific flow. Consider the following example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun log(message: String) = println("[${Thread.currentThread().name}] $message")

fun my_function(): Flow<Int> = flow {
 log("Started flow")
 for (i in 5..7) {
  emit(i)
 }
}

fun main() = runBlocking<Unit> {
 my_function().collect { v -> log("Finished processing $v") }
}
```

Listing 17: Flow context example 2.

If we were to run listing's 17 code, we would have produced the output:

main@coroutine#1 Started flow
main@coroutine#1 Finished processing 5
main@coroutine#1 Finished processing 6
main@coroutine#1 Finished processing 7

This is because the *collect()* method is being called form the main thread, just like the code inside *simple()* function's body. This results in fast executions and asynchronous code, that do not block the calling thread.

2.5.6 Flow Buffering

A flow can be executed in separate coroutines in order to reduce the overall time it takes to be collected. Such functionality is extremely important, especially when a long running asynchronous task needs to run. Let's demonstrate this. Take, for example, listing 18:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun my_function(): Flow<Int> = flow {
 for (i in 5..7) {
  delay(100) // waiting for 100 miliseconds
  emit(i) // emit next value
 }
}

fun main() = runBlocking<Unit> {
 val ms = measureTimeMillis {
  my_function().collect { value ->
   delay(300) // some background work
   println(value)
  }
 }
 println("Finished in $ms ms")
}
```

Listing 18: Flow example without buffering.

This code features a slow *simple* flow, emitting values every 100ms, and a slow collector, which processes the values every 300ms. The output is:

5
6
7
Finished in 1222 ms

A total of about 1200 ms to complete, which is correct since there were three numbers, with 400ms for each one. Now, we can use a *buffer* operator inside a flow, which will

emit the values from the *simple* flow in parallel with the collecting code (instead of sequentially):

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun my_function(): Flow<Int> = flow {
 for (i in 5..7) {
  delay(100) // waiting for 100 miliseconds
  emit(i) // emit next value
 }
}

fun main() = runBlocking<Unit> {
 val ms = measureTimeMillis {
  my_function()
  .buffer() // buffer emissions, don't wait
  .collect { value ->
   delay(300) // some background work
   println(value)
  }
 }
 println("Finished in $ms ms")
}
```

Listing 19: Flow example with buffering.

The output now is:

5
6
7
Finished in 1070 ms

Essentially, we have created a faster processing pipeline, where we only have to wait 100ms for the first number to be printed, followed by 300ms for processing each next number.

3 Developing ZenCrypt

3.1 Decision

As we discussed in section 1.1, data security is very important, and failing to achieve it stands as a very real threat to a person's finances, privacy, as well as well-being. The unwanted disclosure of personal information (especially on social media) can lead to identity theft and/or other personal harm, and there is no sign that this trend will slow.

We decided to develop ZenCrypt having taken into consideration the above concerns. An application that will allow users to easily and safely encrypt images, passwords, voice recordings, documents and, in general, any sort of file that they seem fit, in order to exchange them without worrying about the sharing platform "collecting" that private information. At the time of writing, there are no applications available on the Google Play Store that will cover *all* of the following criteria:

1. Strong encryption with the latest secure standards for Android.
2. File encryption and not only text/password inputs.
3. Storing of encrypted files in a fashion that is easy to display to the user, and decrypt with one-click.
4. Password analyzer with hints, for stronger passwords used when encrypting files [36].
5. Encryption using fingerprints.
6. Friendly, slick, and intuitive user interface, respecting Android's material design guidelines.
7. Lifecycle aware development with minimum memory usage.
8. Fast encryption with very little overhead.
9. No internet permission required, so that the user feels comfortable with his/hers inputs to the app not being sent anywhere.
10. Free & Open-source.

ZenCrypt offers all of the above packaged in a single modern application, which the user can download and use straight away.

3.2 Implementation

An application that facilitates such functionality needs to be designed in such a way so that the user needs the minimum amount of interaction in order to encrypt and share a file. ZenCrypt is designed exactly this way, and its implementation will be discussed in this section. It will be broken down into segments, depending whether the functionality at hand is about UI elements or core application actions.

3.2.1 Core Functionality

Let's start by analyzing ZenCrypt's kernel functions.

A. Encryption

The most important function of the application is the encrypt function (listing 20) [41]. ZenCrypt uses the AES-256 cipher with CBC mode of operations (section 1.3.4), along with PKCS#7 block padding (section 1.4) for encryption.

```
@JvmOverloads
fun <T> encrypt(@NotNull input: T,
@NotNull password: String,
@NotNull erl: ECResultListener,
@NotNull outputFile: File = File(Constants.DEF_ENCRYPTED_FILE_PATH)) {
 GlobalScope.async(Dispatchers.Default) {

  val tPass = password.trim()

  when (input) {

   is String -> encrypt(input.asByteArray(), password, erl, outputFile)

   is CharSequence ->
   encrypt(input.toString().asByteArray(), password, erl, outputFile)

   is ByteArrayInputStream -> encrypt(input.readBytes(), password, erl, outputFile)

   is File -> {
    if (!input.exists() || input.isDirectory) {
     erl.onFailure(Constants.ERR_NO_SUCH_FILE, NoSuchFileException(input))
     return@async
    }
    val encryptedFile =
    if (outputFile.absolutePath == Constants.DEF_ENCRYPTED_FILE_PATH)
    File(input.absolutePath + Constants.ECRYPT_FILE_EXT)
    else outputFile
    encrypt(input.inputStream(), password, erl, encryptedFile)
   }

   else -> performEncrypt.invoke(input, tPass, cipher,
   { pass, salt -> getKey(pass, salt) }, erl, outputFile)
  }
 }
}
```

Listing 20: Encryption method.

The above method symmetrically encrypts the input data using AES algorithm in CBC mode with PKCS7Padding padding, and posts the response to *[ECResultListener.onSuccess]* if successful or posts error to *[ECResultListener.onFailure]* if failed. Encryption progress is posted to *[ECResultListener.onProgress]*. The Result can be a String or a File depending on the data type of [input] and parameter [outputFile]. The parameter T can be either of [String], [CharSequence], [ByteArray], [InputStream], [FileInputStream], or [File]. The rest of the parameters are:

- *input* data to be encrypted.
- *password* string used to encrypt input.
- *erl* listener interface of type [ECResultListener] where result and progress will be posted.
- *outputFile* optional output file. If provided, result will be written to this file.

Furthermore, this method can throw the following exceptions:

- *InvalidKeyException* if password is null or blank.
- *NoSuchFileException* if input is a File which does not exists or is a Directory.
- *InvalidParameterException* if input data type is not supported.
- *IOException* if cannot read or write to a file.
- *FileAlreadyExistsException* if output file is provided and already exists.
- *IllegalBlockSizeException* if this cipher is a block cipher, no padding has been requested (only in encryption mode), and the total input length of the data processed by this cipher is not a multiple of block size; or if this encryption algorithm is unable to process the input data provided.

This method is essentially a coroutine, which uses the *async* method from the *GlobalScope* scope (section 2.5.2), running on the *Dispatchers.Default* context (section 2.5.3). The [ECResultListener] is a very simple interface shown in listing 21.

```
1  /**
2  * Interface to listen for result from encryption, decryption, or hashing
3  */
4  interface ECResultListener {
5   /**
6   * @param newBytes count processed after last block
7   * @param bytesProcessed count from total input
8   */
9   fun onProgress(newBytes: Int, bytesProcessed: Long, totalBytes: Long) {}
10
11  /**
12  * @param result on successful execution of the calling method
13  */
```

```
14    fun <T> onSuccess(result: T)
15
16    /**
17    * @param message on failure
18    * @param e exception thrown by called method
19    */
20    fun onFailure(message: String, e: Exception)
21  }
```

Listing 21: ECResultListener interface.

B. Decryption

The decryption method is quite similar to that of encryption (listing 22):

```
1   @JvmOverloads
2   fun <T> decrypt(@NotNull input: T,
3   @NotNull password: String,
4   @NotNull erl: ECResultListener,
5   @NotNull outputFile: File = File(Constants.DEF_DECRYPTED_FILE_PATH)) {
6    GlobalScope.async(Dispatchers.Default) {
7
8     val tPass = password.trim()
9
10    when (input) {
11
12     is String -> {
13      try {
14       decrypt(input.fromBase64().inputStream(), password, erl, outputFile)
15      } catch (e: IllegalArgumentException) {
16       erl.onFailure(Constants.ERR_BAD_BASE64, e)
17      }
18     }
19
20     is CharSequence -> {
21      try {
22       decrypt(input.toString().fromBase64().inputStream(),
23       password, erl, outputFile)
24      } catch (e: IllegalArgumentException) {
25       erl.onFailure(Constants.ERR_BAD_BASE64, e)
26      }
27     }
28
29     is ByteArray -> decrypt(input.inputStream(), password, erl, outputFile)
30
31     is File -> {
32
33      if (!input.exists() || input.isDirectory) {
34       erl.onFailure(Constants.ERR_NO_SUCH_FILE, NoSuchFileException(input))
35       return@async
36      }
37
38      val decryptedFile =
39      if (outputFile.absolutePath == Constants.DEF_DECRYPTED_FILE_PATH)
40      File(input.absoluteFile.toString().removeSuffix(Constants.ECRYPT_FILE_EXT))
41      else outputFile
42
43      decrypt(input.inputStream(), password, erl, decryptedFile)
44     }
45
46     else -> performDecrypt.invoke(input, tPass, cipher,
47     { pass, salt -> getKey(pass, salt) }, erl, outputFile)
48    }
49   }
```

```
50  }
```

Listing 22: Decryption method.

Apart from the different implementations, this method can throw an additional exception:

- *BadPaddingException* if this cipher is in decryption mode, and (un)padding has been requested, but the decrypted data is not bounded by the appropriate padding bytes.

C. Fingerprint Authentication [24]

ZenCrypt enables (as a PRO feature) users to optionally encrypt files through a biometric authentication, in this case, their fingerprint.

```
1   private fun startFingerprintAuth() {
2    val biometricPromptCompat = BiometricPromptCompat.Builder(
3    BiometricAuthRequest(BiometricApi.AUTO, BiometricType.BIOMETRIC_FINGERPRINT),
4    this
5    )
6    .setTitle(if (intentAction == ZenCryptConstants.ACTION_ENCRYPT)
7      getString(R.string.encrypt) else getString(R.string.decrypt))
8    .setSubtitle(resultName)
9    .setNegativeButton(getString(android.R.string.cancel), null)
10   .build()
11
12   biometricPromptCompat.authenticate(object : BiometricPromptCompat.Result {
13    override fun onSucceeded(confirmed : Set<BiometricType>) {
14     startZenCryptAction()
15    }
16
17    override fun onCanceled() {
18     enableButtons()
19    }
20
21    override fun onFailed(reason: AuthenticationFailureReason?) {
22     enableButtons()
23     SnackBarHelper.showSnackBarError("Error : $reason", this@ActionActivity)
24     if (ZenCryptSettingsModel.vibration.value) vibrate()
25    }
26   })
27  }
```

Listing 23: Fingerprint authentication.

Listing 23 demonstrates the authentication functionality. Its based on Google's BiometricPrompt API, which is a new approach declaring that the system takes care of a unified way to use different biometric identification methods.

D. Settings Model

One of the main features that nearly every application should have, is a way for the user to customize the way the app works and feels. ZenCrypt comprises many customization options, such as the application's theme (dark-PRO/light), whether or not

to delete the original unencrypted files after encryption, change the output file's extension, enable fingerprint authentication (PRO), and more. So, an implementation is needed that not only hosts this functionality, but ensures that these options are retained after each app exit, and that won't reset on launch. To achieve this, ZenCrypt uses the Jetpack DataStore (discussed in section 2.3.1) [33], and the model is as follows (listing 24):

```
object ZenCryptSettingsModel :
 SettingsModel(DataStoreStorage(name = "zencrypt_settings")) {
 //-------
 val darkTheme by boolPref(false)
 val vibration by boolPref(true)
 val extension by stringPref(".zen")
 val fingerprint_auth by boolPref(false)
 val custom_pass_hash by stringPref("")
 val delete_original_unencrypted by boolPref(true)
 val isProUser by boolPref(false)
 val versionCode by intPref(-1)
 //-------
}
```

Listing 24: Settings DataStore.

E. Avoiding Memory Leaks Using ViewBinding

A major issue with Android applications is *views* leaking their content when the app is not visible to the user, or even when the user has navigated to a different page within the app. This happens by default in Android's views/contexts, and it needs to be addressed by the developer. One way to go by doing this, is to use *View Binding* (section 2.2.6) [39]. This is a feature that generates a *binding class* for each XML layout file present in that module [17]. Though, the developer must manage the ViewBinding lifecycle and clear reference it in order to prevent memory leaks. A Sample usage can be seen in listing 25:

```
class ProfileFragment : Fragment(R.layout.profile) {

 // Using reflection API under the hood and ViewBinding.bind
 private val viewBinding: ProfileBinding by viewBinding()

 // Using reflection API under the hood and ViewBinding.inflate
 private val viewBinding: ProfileBinding
    by viewBinding(createMethod = CreateMethod.INFLATE)

 // Without reflection
 private val viewBinding by viewBinding(ProfileBinding::bind)
}
```

Listing 25: ViewBinding leak prevention example.

The *viewBinding* object can now access all the child views in a safe and memory leak-free way. Note that the *viewBinding* variable must be set to true in the app's *build.gradle* file, inside the *buildFeatures* section.

F. In-app Billing v4

Some of the features that ZenCrypt offers, namely the dark theme and the fingerprint encryption, require for the user to pay a small amount of money in order to unlock them. To facilitate this, we need to implement Google's in-app billing library [34], and code the different outcomes that may occur from the new purchase (such as error, success, item already owned, and more). For example, a successful purchase leads to enabling the PRO version, as shown in listing 26

```
fun initBillingConnector(activity: AppCompatActivity): BillingConnector {
 val billingConnector = BillingConnector(
 activity,
 BuildConfig.ApiKey)
 .setNonConsumableIds(listOf("zencrypt_pro"))
 .autoAcknowledge()
 .autoConsume()
 .enableLogging()
 .connect()

 billingConnector.setBillingEventListener(object : BillingEventListener {
  override fun onProductsFetched(@NonNull skuDetails: List<SkuInfo>) {
   if ( billingConnector.isPurchased(skuDetails.first() ) == PurchasedResult.YES )
   activity.lifecycleScope.launch {
    ZenCryptSettingsModel.isProUser.update(true)
   }
  }

  override fun onPurchasedProductsFetched(@NonNull purchases: List<PurchaseInfo>) {
   /*Provides a list with fetched purchased products*/
  }

  override fun onProductsPurchased(@NonNull purchases: List<PurchaseInfo>) {
   activity.lifecycleScope.launch {
    ZenCryptSettingsModel.isProUser.update(true)
   }
   SnackBarHelper.showSnackBarLove("Thank you for purchasing ZenCrypt pro!")
   FragmentHelper.replaceFragmentWithDelay(SettingsFragment())

  }
```

Listing 26: In-app purchases implementation.

After a successful purchase has been made, we need to update the PRO status of the user within the app. Since we are using DataStorage for storing options, this needs to be done in a way that respects the app's lifecycle, i.e. use Kotlin's flows (section 2.5.4). This is why we launch a coroutine using the main activity's lifecycle scope, and use the *update()* method to store the respective setting to true. On the other hand, if an error occurs, we need to handle it properly (listing 27).

```
override fun onBillingError(
@NonNull billingConnector: BillingConnector,
@NonNull response: BillingResponse
) {
 /*Callback after an error occurs*/
 when (response.errorType) {
  ErrorType.CLIENT_NOT_READY -> { }
  ErrorType.CLIENT_DISCONNECTED -> { }
```

```
    ErrorType.SKU_NOT_EXIST -> { }
    ErrorType.CONSUME_ERROR -> { }
    ErrorType.ACKNOWLEDGE_ERROR -> {
     activity.lifecycleScope.launch {
      ZenCryptSettingsModel.isProUser.update(false)
     }
    }
    ErrorType.ACKNOWLEDGE_WARNING -> {
     activity.lifecycleScope.launch {
      ZenCryptSettingsModel.isProUser.update(false)
     }
    }
    ErrorType.FETCH_PURCHASED_PRODUCTS_ERROR -> { }
    ErrorType.BILLING_ERROR -> {
     SnackBarHelper.showSnackBarError("Billing error!")
     activity.lifecycleScope.launch {
      ZenCryptSettingsModel.isProUser.update(false)
     }
    }
    ErrorType.USER_CANCELED -> SnackBarHelper.showSnackBarError("Payment cancelled.")
    ErrorType.SERVICE_UNAVAILABLE -> { }
    ErrorType.BILLING_UNAVAILABLE -> { }
    ErrorType.ITEM_UNAVAILABLE -> { }
    ErrorType.DEVELOPER_ERROR -> { }
    ErrorType.ERROR -> {
     SnackBarHelper.showSnackBarError("Something went wrong.")
     activity.lifecycleScope.launch {
      ZenCryptSettingsModel.isProUser.update(false)
     }
    }
    ErrorType.ITEM_ALREADY_OWNED -> {
     activity.lifecycleScope.launch {
      ZenCryptSettingsModel.isProUser.update(true)
     }
    }
    ErrorType.ITEM_NOT_OWNED -> {
     activity.lifecycleScope.launch {
      ZenCryptSettingsModel.isProUser.update(false)
     }
    }
    null -> { }
   }
  }
```

Listing 27: In-app purchase error handling.

G. Launch a File Picker Instance

Finally, to enable the user to pick a file for encryption/decryption (respecting what we discussed in section 2.3.2), we need to create a file picking intent, as seen below (listing 28):

```
 private fun selectFile() {
  val intent = Intent(Intent.ACTION_OPEN_DOCUMENT)
  intent.addCategory(Intent.CATEGORY_OPENABLE)
  intent.type = "*/*"
  startForResult.launch(intent)
 }
```

Listing 28: Launch a file picking intent.

3.2.2 User Interface Handling

The interface of ZenCrypt is rather simple, in order to be as straight-forward as possible. It comprises several fragments and activities, as well as a simple animated bottom navigation.

A. Fragment Transactions

An interesting thing to address, is how the transaction between fragments is made possible (listing 29).

```
fun replaceFragmentWithDelay(fragment: Fragment, timeMillis: Long = 350) {
mActivity.lifecycleScope.launchWhenStarted {
 delay(timeMillis)
 val backStateName = fragment.javaClass.name
 val fragmentPopped =
   mActivity.supportFragmentManager.popBackStackImmediate (backStateName, 0)

 if (!fragmentPopped &&
   mActivity.supportFragmentManager.findFragmentByTag(backStateName) == null) {
  // fragment not in back stack, create it.
  val fragmentTransaction = mActivity.supportFragmentManager.beginTransaction()
  fragmentTransaction.replace(R.id.container, fragment);
  fragmentTransaction.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
  fragmentTransaction.addToBackStack(backStateName);
  fragmentTransaction.commit();
 }
}
}
```

Listing 29: Fragment replacement method.

Keep in mind that it's very important to use *launchWhenStarted* {...} here. If we were to use *launch* {...}, then the *isStateSaved* variable would be false and the transaction would be considered NOT safe to commit. This results in app crashes due to the *illegalStateException* exception. Using *launchWhenStarted* {} ensures that the state is saved and the transaction is safe to commit.

B. File Listing

The main concept of the app is to act as a private and secure file "vault", which the user can see its content and read them. So, a basic file *List View* must be implemented, which will be populated with either encrypted or decrypted files, depending on what page the user has navigated to. List views need to be attached to an adapter so that they can be later populated with information. To further reduce memory consumption, we use a *RecyclerViewAdapter* (listing 30):

```
private fun loadDataAndPopulateCardView() {
 buildProgressDialog()
 lifecycleScope.launch {
  whenStarted {
   progressDialog.show()
   val data = withContext(Dispatchers.IO) {
    val encryptedFileItems: ArrayList<FileItem> = ArrayList()
```

```
8       externalFilesDir.walkTopDown().filter { file -> !file.isDirectory }.forEach
9       { file -> ... }
10      return@withContext encryptedFileItems
11    }
12
13    ...
14
15    val recyclerView: RecyclerView = binding.cardListRecyclerView
16    recyclerView.setHasFixedSize(true)
17    val layoutManager: RecyclerView.LayoutManager = LinearLayoutManager(context)
18    recyclerView.layoutManager = layoutManager
19    recyclerView.adapter = EncryptedFilesExpandableRecyclerAdapter(data)
20
21    ...
22
23    progressDialog.dismiss()
24   }
25  }
26 }
```

Listing 30: Populate the list view.

C. Settings Menu

To create the settings menu view with ease, we have used the *MaterialPreferences* library [33], which offers a DSL for defining view settings elements. For example, to create a switch that toggles the dark mode throughout the app, while checking if the user has purchased the PRO version (which is required to toggle this option), we can do the following (listing 31):

```
1  ...
2
3  switch(ZenCryptSettingsModel.darkTheme) {
4   title = getString(R.string.dark_theme).asText()
5   icon = R.drawable.ic_baseline_style_24.asIcon()
6   summary = getString(R.string.choose_between_light_and_dark).asText()
7   badge = "PRO".asBatch()
8   canChange = {
9    if ( !ZenCryptSettingsModel.isProUser.value )
10   SnackBarHelper.showSnackBarError(getString(R.string.zencrypt_pro_is_required))
11   ZenCryptSettingsModel.isProUser.value
12  }
13  onChanged = {
14   println("Dark Theme Settings Listener called: $it")
15   //recreate()
16   MainActivity.themeChanged()
17   AppCompatDelegate.setDefaultNightMode(if (it) AppCompatDelegate.MODE_NIGHT_YES
18    else AppCompatDelegate.MODE_NIGHT_NO)
19  }
20 }
21
22 ...
```

Listing 31: Settings switch to toggle dark mode.

As a side note, toggling between *AppCompatDelegate.MODE_NIGHT_NO* and *AppCompatDelegate.MODE_NIGHT_YES* will not work smoothly across all Android devices and versions available. For example, devices running Android API 29 (version 10) and newer, can force a system-wide dark mode which affects all apps, even if a

specific app does not support it. This leads to unexpected graphical glitches, and even text being colored gray on a dark background, which makes it unreadable. To counter this problem, we can define the following code inside the *styles.xml* file for both the night and light themes (listing 32):

```
...

// <!-- fix for force dark environmental variable-->
<item name="android:forceDarkAllowed" tools:targetApi="29">false</item>

...
```

Listing 32: XML attribute to counter the force dark option.

D. Splash Screen

There are many ways to create a splash screen for an application, the most popular being:

1. Using Timers **(the wrong way)**
2. Using a Launcher Theme **(the right way)**

The first approach is not ideal. It will give rise to cold starts, which inevitably makes the application slow and thus the user experience bad. These cold starts occur since the application takes time to load the layout file of the Splash Activity. To tackle this problem, we can use the second approach, which is the absolute correct way to implement a splash screen. The *Application theme* is instantiated before the layout is created, so the overall procedure is quite fast. The first step is to declare the theme style inside *res/values/themes.xml* (listing 33):

```
...

<style name="SplashTheme" parent="Theme.MaterialComponents.DayNight.NoActionBar">
 <item name="android:windowBackground">@drawable/splash_background</item>
 <item name="colorPrimary">@color/colorPrimary</item>
 <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
 <item name="colorAccent">@color/ZenCryptPrimary</item>
 <item name="android:statusBarColor">@color/ZenCryptDark</item>
 <item name="android:navigationBarColor">@color/ZenCryptDark</item>
</style>

...
```

Listing 33: Splash screen theme style.

After the overall customization, we have to set this specific style for the Splash Activity, inside the *AndroidManifest.xml* (listing 34):

```
...

 <activity
android:name=".activities.SplashActivity"
android:theme="@style/SplashTheme"
android:exported="true">
 <intent-filter>
```

```
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>

  ...
```

Listing 34: Set Splash Activity style.

Finally, we can implement the Splash Activity class (listing 35):

```
...

@SuppressLint("CustomSplashScreen")
class SplashActivity : AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  val intent = Intent(this@SplashActivity, MainActivity::class.java)
  startActivity(intent)
  finish()
 }
}
```

Listing 35: The Splash Activity.

We suppress the lint *"CustomSplashScreen"*, because the latest Android version 12 offers an in-built functionality specifically designed for creating splash screens. So, the system detects that we have implemented our own way for coding a splash screen, and throws this warning. Ultimately, when migrating to Android 12, this way will be considered as deprecated.

F. Exiting the App

Correctly exiting an application in Android is a very debatable subject in its community. ZenCrypt tries to approach this task without interfering with the OS and its order of killing processes. In other words, ZenCrypt exits respecting the Android's app lifecycle, and letting the system decide when it is time for the application process to be killed. This means faster opening times, and no brute-forcing techniques for app exiting. Listing 36 demonstrates the *exit()* method that ZenCrypt uses when the user decides to leave the app.

```
private fun exit() {
 val _intentOBJ = Intent(Intent.ACTION_MAIN)
 _intentOBJ.addCategory(Intent.CATEGORY_HOME)
 _intentOBJ.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP)
 _intentOBJ.flags = Intent.FLAG_ACTIVITY_NEW_TASK
 startActivity(_intentOBJ)
}
```

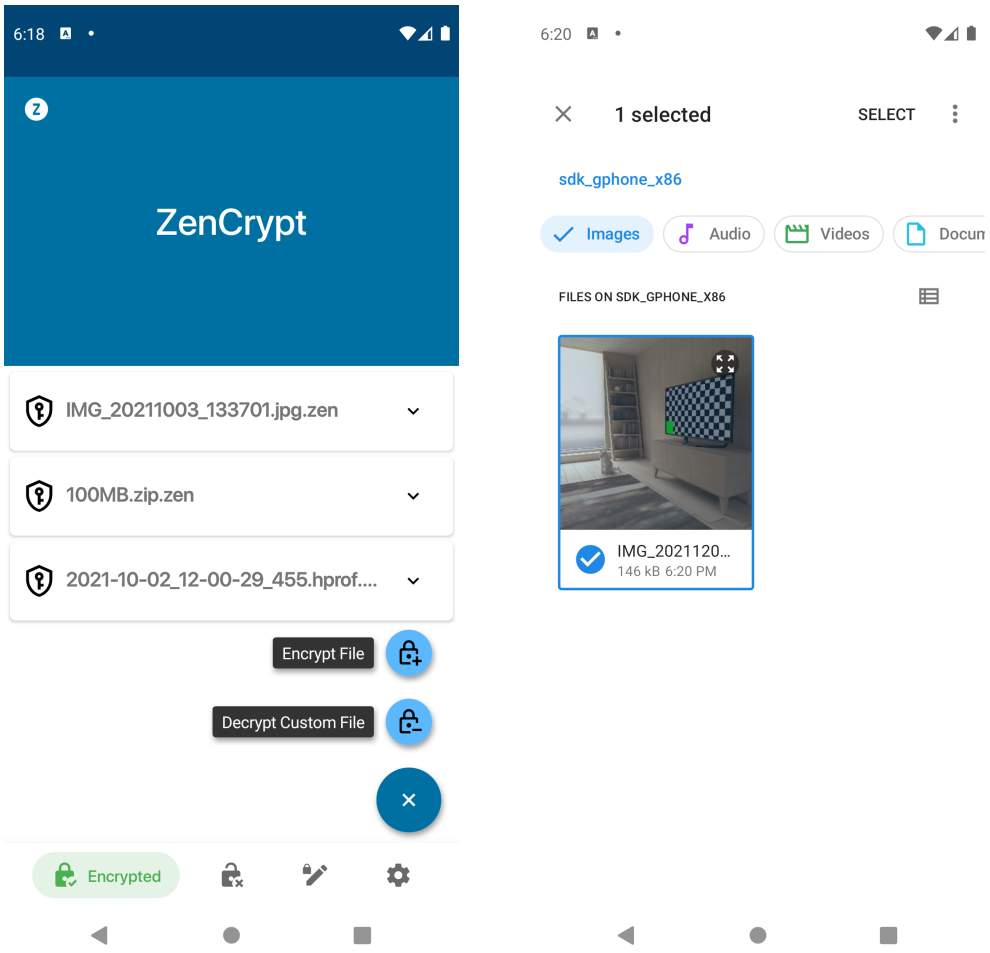
Listing 36: Exit function.

The above approach clears the top of the activity stack, essentially moving the application task to the back. This mimics the "home" button press, and does not kill the process. It's recommended to use this approach instead of, for example, *System.exit(0)*. The *System.exit()* method is very intrusive and forces the application to be killed instantly by the OS. This has many drawbacks, one of which being that it may not facilitate the desired functionality that the user wants when leaving the app. For example, say that the user is encrypting a file, i.e. a coroutine is launched that is doing some heavy work. If the back button is pressed, then the application will be killed while the process is still running and did not finish its task. This can lead to major issues with the integrity of the user's private files, like significant data loss.

4 Example Usage

4.1 Encrypting & Sharing a File

ZenCrypt is a very powerful tool that utilizes the latest secure standards for Android, in order to encrypt files. Let's demonstrate how an end user can use the app in order to securely share a file that contains personal information. After launching ZenCrypt, the first step is to click on the "+" icon located at the bottom right of the main page (figure 24a). After clicking on "Encrypt File", a file picker instance will fire up asking the user to select a file (figure 24b). As an example, we will encrypt a test image taken from the Android emulator.

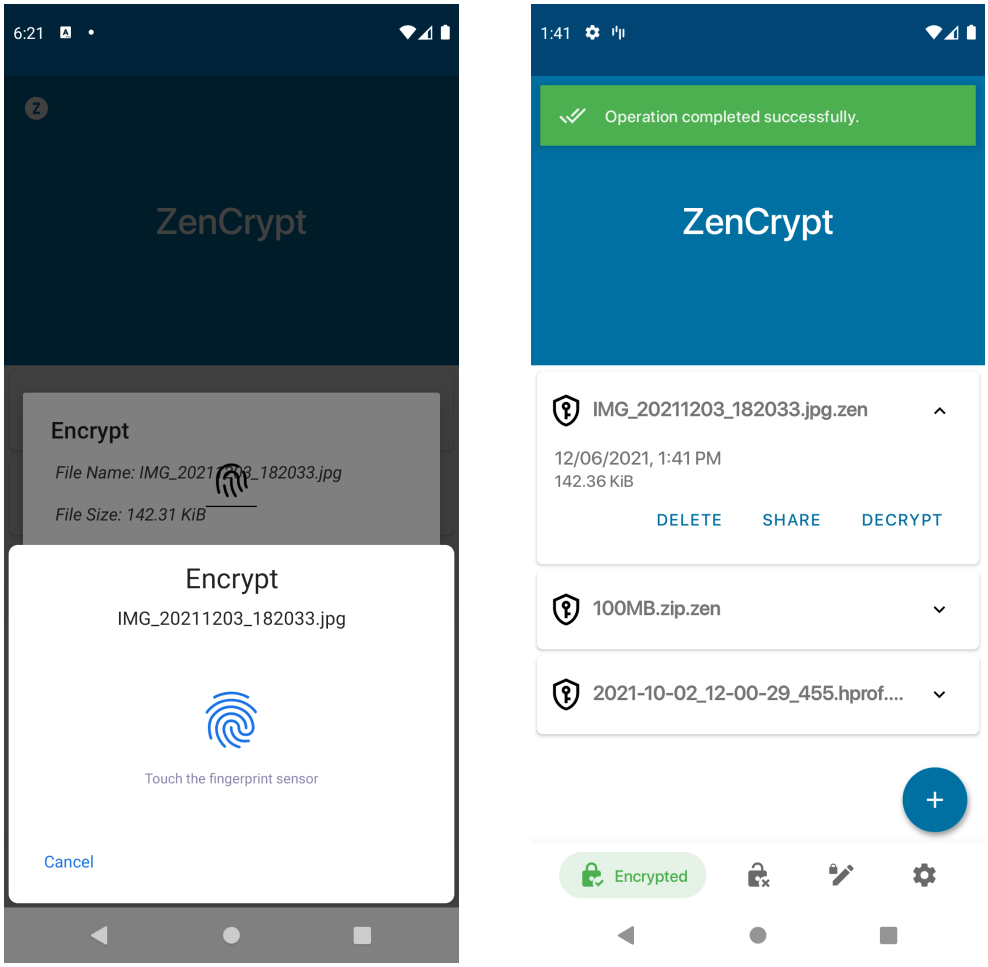


(a) *ZenCrypt's main page*

(b) *File picker instance*

Figure 24: *Selecting a file to encrypt*

Next, ZenCrypt will ask the user the means with witch he wishes to encrypt the file. This is either of two things, a fingerprint or a password. In this example, fingerprint encryption is enabled, so a dialog asking to verify the user's fingerprint will show (figure 25a). After a successful authentication, the encryption process will start. If all goes well, a *"Operation completed successfully"* message will appear, and the newly encrypted file will be added to the list. This now gives us three options, to delete, share, or decrypt the file. If we click on "share", the Android's default share dialog will open, and we can select any app with which we can share that specific file.



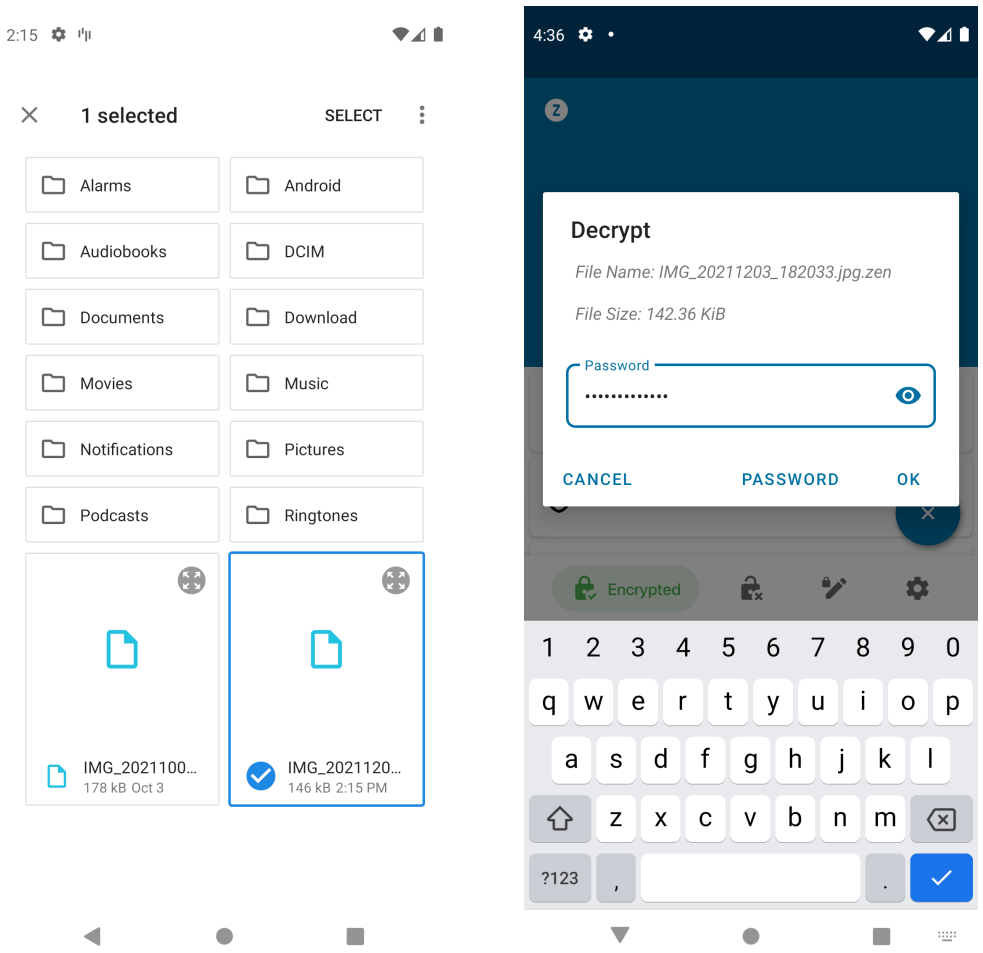
(a) *Fingerprint authentication dialog*

(b) *Main page after successful encryption*

Figure 25: *Successful encryption*

4.2 Decrypting a Shared File

After a user has received an encrypted file (which was previously encrypted using ZenCrypt), he can decrypt it with the password that both parties agreed. This can be done, again, by clicking on the "+" button, but this time selecting the "Decrypt Custom File" option. Then, the file picker instance will be shown, where the user can select the previously shared file (figure 26a). Keep in mind, that ZenCrypt offers picking files from cloud providers as well (such as Google Drive). Finally, the input password dialog is displayed (figure 26b).

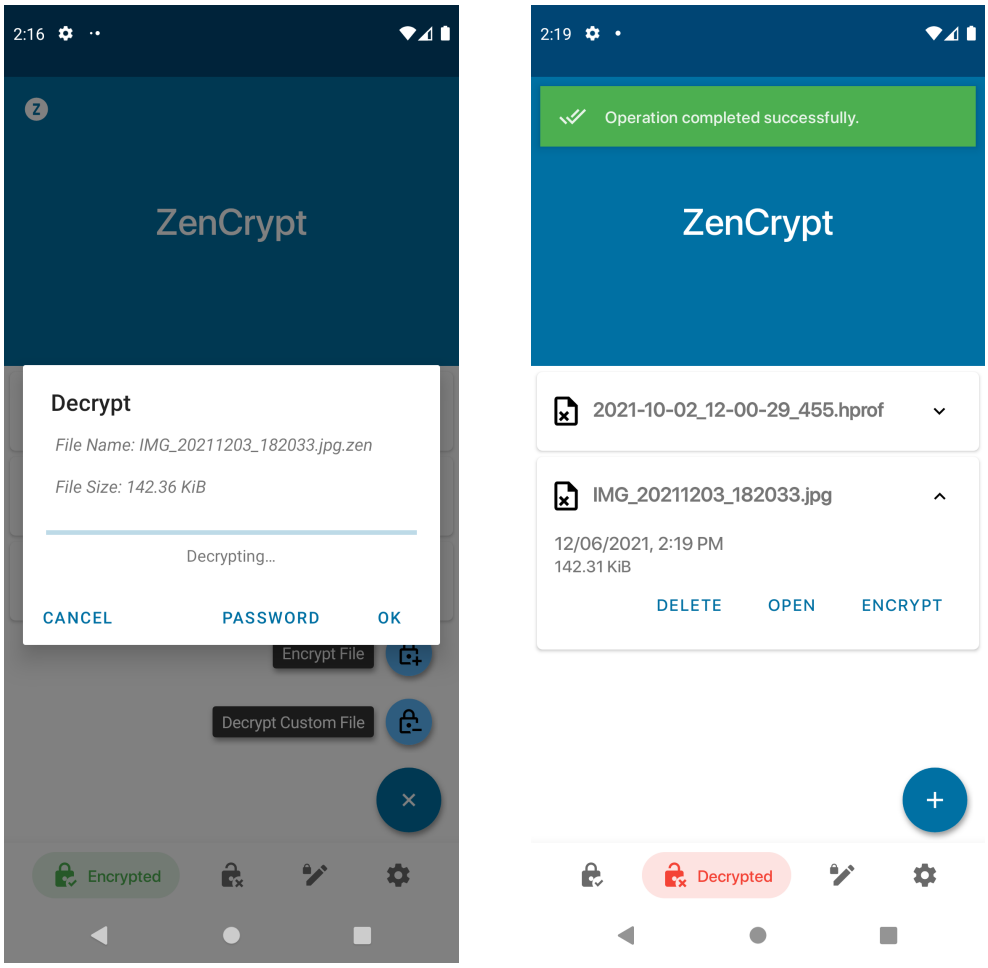


(a) *File picker instance*

(b) *Password input*

Figure 26: *Selecting a file to decrypt*

If all goes well, i.e. the password used to encrypt that file is the same as the input used for decryption, a success message will appear, and the newly decrypted file will be listed in the "decrypted" fragment view. Again, there are three options when selecting this file entry. The user can delete the file, open and view it with an application of choice, or encrypt it again.



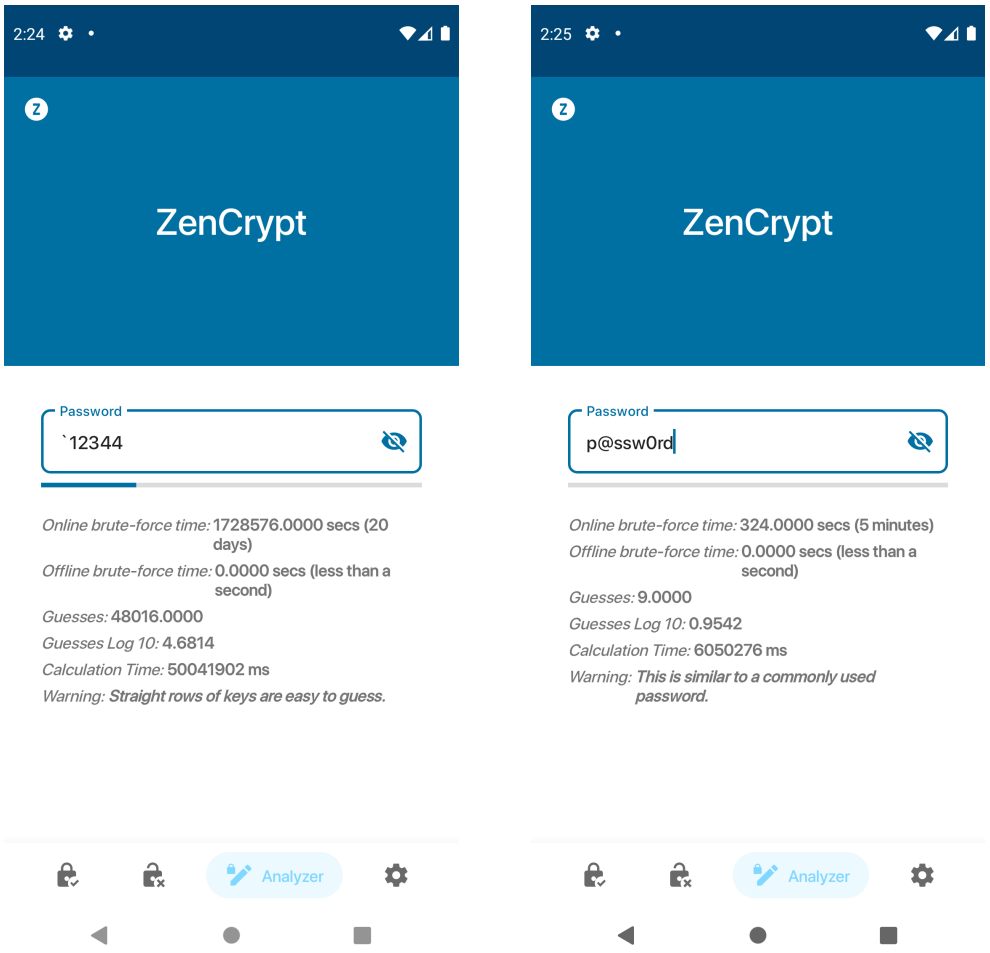
(a) *Decrypting dialog*

(b) *Main page after successful decryption*

Figure 27: *Successful decryption*

4.3 Password Analyzer

An additional functionality that ZenCrypt has baked in, is the *password analyzer*. This is a particularly helpful tool that can analyze in real-time a given input password, and display to the user various information about it, such as online and offline brute-force times, warnings etc. It will also display how much time it took to calculate this data for that password. For example, a weak password analysis can be seen in figure 28.

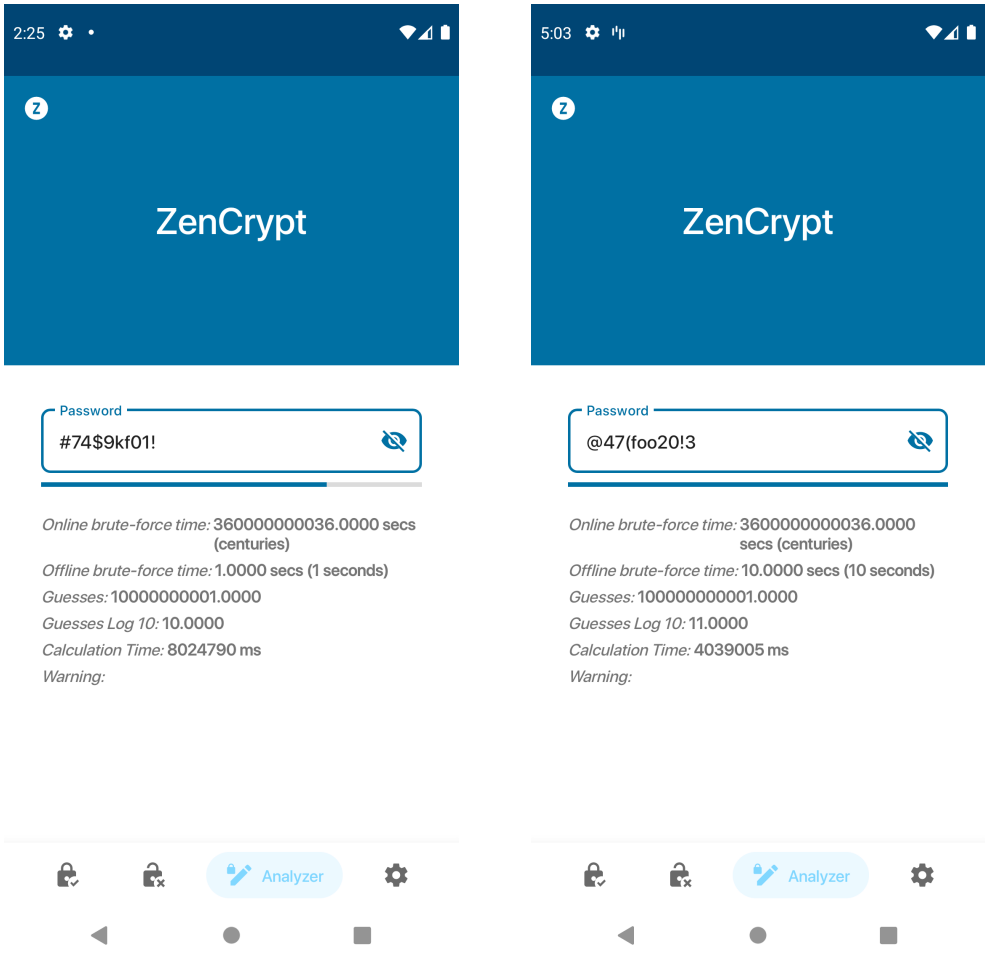


(a) *Straight row password*

(b) *Common password*

Figure 28: *Weak passwords analysis*

On the other hand, a strong password analysis can be seen if figure 29. These passwords include letters, symbols and numbers, while being mostly random and not easy to guess. The progress bar below the password input graphically illustrates the password strength, just as the online and offline brute-force times do in textual form.



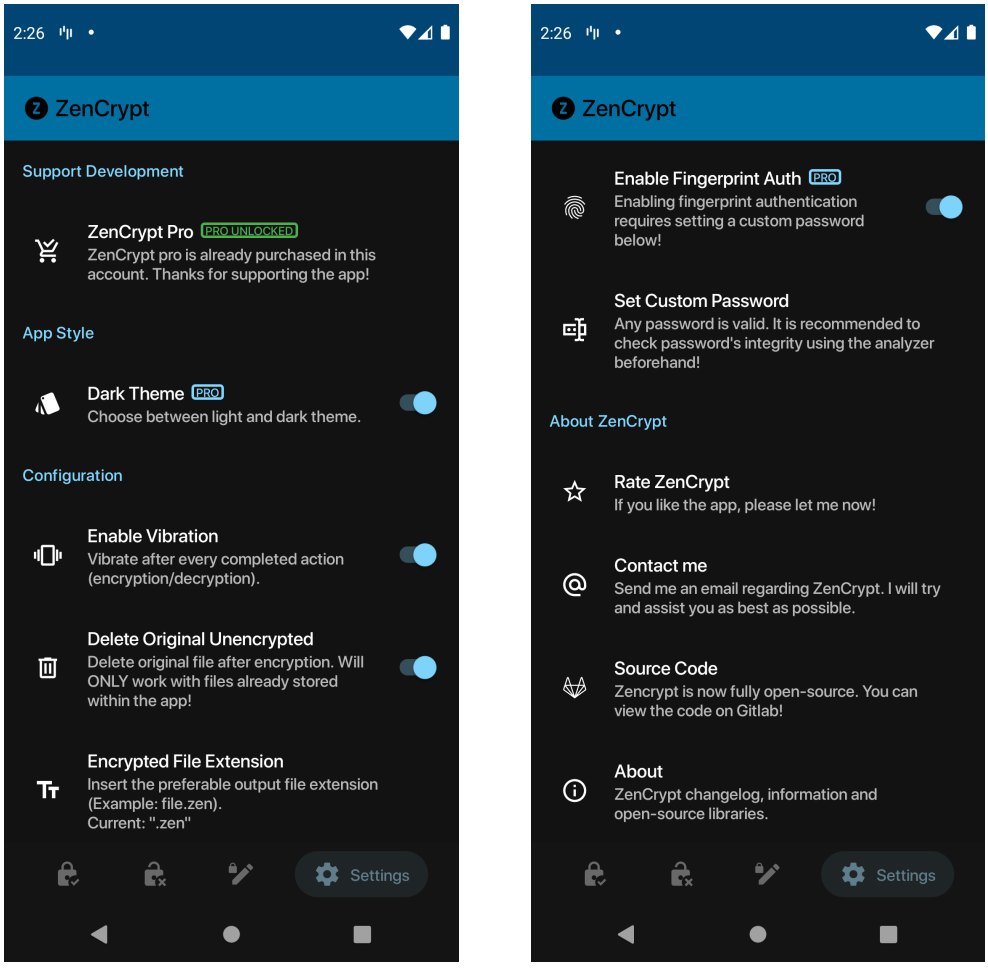
(a) *Strong password example*

(b) *Even stronger password example*

Figure 29: *Strong passwords analysis*

4.4 Various Options

Closing this chapter, we can briefly take a look at the various options that the user can tinker with while using ZenCrypt. Starting off, the first option enables the user to optionally pay a small amount of real money to unlock extra functionality throughout the app. This includes a dark theme, and the option to encrypt using a fingerprint. Next, there are various configurations such as a toggle for vibration, which by default occurs after any successful/failed action (encryption/decryption), an option to delete the original unencrypted file after it has been successfully encrypted, and configurable file extension name. Finally, there are the fingerprint related settings, such as the toggle for fingerprint encryption itself, and an input to be used as the password. Note that this text input is not parsed in raw format, but is rather hashed using the SHA-256 hashing function.



(a) *Cosmetic & configuration*

(b) *Fingerprint & about*

Figure 30: *Settings page*

5 Performance

In order to measure ZenCrypt's performance, we can take advantage of Kotlin's *measureTimeMillis* function [29]. It essentially executes the given block of code and returns elapsed time in milliseconds. An example is shown in listing 37:

```
...

val timeInMillis = measureTimeMillis {
 //Start the encryption process
 startEncrypting()
 //Alternatively, we could measure the decryption time
 //startDecrypting()
}

println("(The operation took $timeInMillis ms)")

...
```

Listing 37: The measureTimeMillis method.

Furthermore, the only parameters that must be taken in consideration while measuring such tasks are, the hardware, and the file size. For all of our testing, we used the following:

1. Various file ranges: 5 MiB, 10 MiB, 20 MiB, 100 MiB, 250 MiB, 500 MiB, 1 GiB.
2. Snapdragon 865 CPU.
3. 8GB RAM LPDDR4X.
4. UFS 3.1 2-LANE storage.
5. Final application built with *release* flag instead of *debug*, for extra speed.

The results can be seen in figure 31 and table 3 below:

File Size	Encryption time (seconds)	Decryption time (seconds)
5 MiB	0.0445	0.347
10 MiB	0.0841	0.688
20 MiB	0.1709	1.279
100 MiB	0.843	6.865
250 MiB	1.804	14.113
500 MiB	3.806	31.599
1 GiB	6.907	75.894

Table 3: *Encryption/Decryption performance (table)*

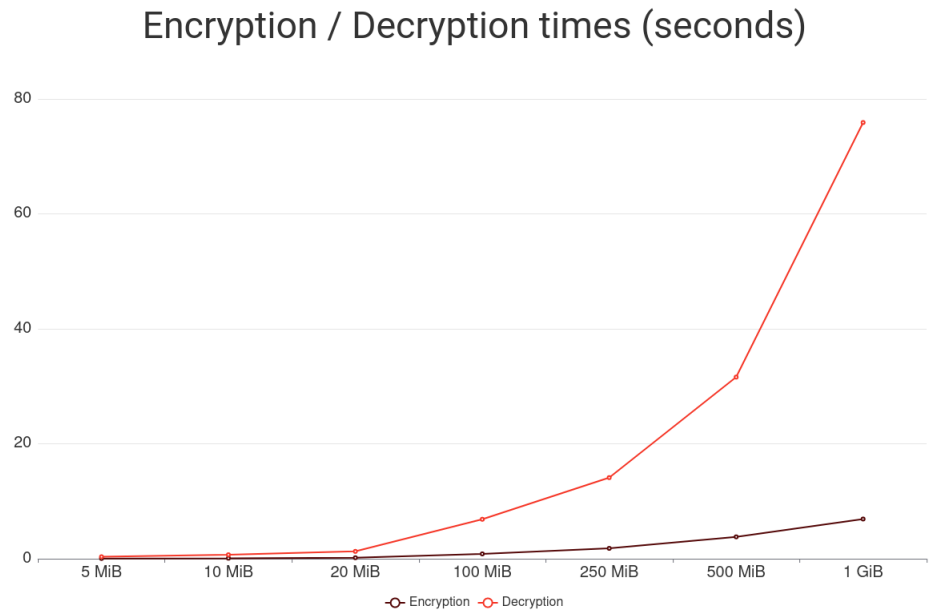


Figure 31: *Encryption/Decryption performance*

Taking a look at the results, we can immediately notice that the encryption process is much faster than the decryption process. This is a very interesting find, and raises the question as to why is this exactly happening. Aren't AES encryption and decryption processes identical, with just the steps in reverse order? Well, it seems that there is a valid reason behind this. AES block encryption is actually faster than AES block decryption, because of the following [9]:

- *MixColumns*[44] uses a matrix which has smaller coefficients than *InvMixColumns*[44], this is simpler to compute. This is particularly true for purely software implementations, hardware implementations sometime use the same number of cycles.
- During encryption, subkeys are needed in the order they are produced from the key, but during decryption that order is reversed, therefor on implementations (including hardware) that start decryption with the pristine key as input, some preliminary work is necessary before decryption can start (there is actually no known shortcut [10]).

6 Online Presence

6.1 Google Play Store

ZenCrypt's binary can be directly downloaded from the Google Play Store:

Updates are pushed on the stable channel, and there is no way to modify the way the app behaves.
(`https://play.google.com/store/apps/details?id=com.zestas.cryptmyfiles`)

6.2 GitLab

Additionally, ZenCrypt's source code is available on GitLab:

The GitLab version does not contain the binary, and it needs to be built from scratch. Though, this version contains the bleeding edge changes before they appear on the Play Store, and since the source is open and downloadable, a developer can clone the project and modify it to accommodate the desired functionality.
(`https://gitlab.com/Kelsios/zencrypt`)

7 Conclusions & Further Development

The era of mobile device interaction started in the past years and has now overwhelmed our every day lives. The vigorous development around Android has led to applications evolving at a meteor pace providing users with more and faster ways to fulfill their every day social (and personal) needs. One of the main concerns that comes to mind when using a mobile device, is the security of any actions taken. From logging in to website or bank, to storing personal information and expecting that its impossible for a third party to read, the safety of such tasks is very important to a person's well-being. ZenCrypt is a tool that helps the user to easily encrypt personal information such as sensitive files, passwords, voice messages, or any given input in general, and either securely store it offline on the device, or share the encrypted file in any way that he/she seems fit. This thesis presented some basic encryption analysis, along with some of the most common algorithms from the bibliography. We also demonstrated the basics of developing an Android application, such as respecting the app's lifecycle, followed by the actual development behind ZenCrypt. Furthermore, we presented some example usages of what is possible when working with this application, and how to easily encrypt and share a single file. In the end, ZenCrypt's performance was measured, which led to some interesting results. The app is encrypting at a very rapid rate, but the decryption process is relatively slow; this mainly happens due to the encryption matrix (*MixColumns*) having smaller coefficients than that of the decryption (*InvMixColumns*) matrix. All in all, ZenCrypt is a well optimized, memory efficient, leak-free Android application that can securely encrypt any sensitive data. Though, there is still room for improvement. For example, it only allows the encryption/decryption of a single item at a time, and requires user interaction to proceed to the next one. This can be dealt with in the future, simply by launching a file picker instance that allows multiple file selection, and implementing the necessary methods to host the desired functionality. Finally, the application could possibly circumvent the severe limitations that come with scoped storage, once Google re-enables app vetting on the Google Play Store and we can submit ZenCrypt for approval.

Bibliography

[1] auth0. What is data security, top threats and best practices. https://auth0.com/blog/what-is-data-security/.

[2] Rajdeep Bhanot and Rahul Hans. A review and comparative analysis of various encryption algorithms. *International Journal of Security and Its Applications*, 9:289–306, 04 2015.

[3] Denis Buketa. Android lifecycle. https://www.raywenderlich.com/21382977-android-lifecycle.

[4] Michael Burton. *Android App Development For Dummies, 3rd Edition*. Wiley Brand, USA, 2015.

[5] Businesswire. Mobile fraud reaches 150 million global attacks in first half of 2018. https://www.businesswire.com/news/home/20180912005231/en/Mobile-Fraud-Reaches-150-Million-Global-Attacks.

[6] cryptosense. Is triple des secure. https://cryptosense.com/blog/is-triple-des-secure.

[7] Dorothy E. Denning and Peter J. Denning. Data security. *ACM Comput. Surv.*, 11(3):227–249, September 1979.

[8] Anusheh Zohair Mustafeez Educative. What is cbc. https://www.educative.io/edpresso/what-is-cbc.

[9] Stack Exchange. Aes decryption vs encryption speed. https://crypto.stackexchange.com/questions/27872/aes-decryption-vs-encryption-speed.

[10] Stack Exchange. On-the-fly computation of aes round keys for decryption. https://crypto.stackexchange.com/questions/5603/on-the-fly-computation-of-aes-round-keys-for-decryption.

[11] Navdeep Singh Gill. Functional programming. https://www.xenonstack.com/insights/functional-programming.

[12] Navdeep Singh Gill. Kotlin vs java: Which is better for android app development. https://www.xenonstack.com/blog/kotlin-andriod.

[13] Google. Content provider. https://developer.android.com/reference/android/content/ContentProvider.

[14] Google. Document provider. https://developer.android.com/guide/topics/providers/document-provider.

[15] Google. Kotlin flows on android. https://developer.android.com/kotlin/flow.

[16] Google. Protocol buffers. https://developers.google.com/protocol-buffers.

[17] Google. View binding. https://developer.android.com/topic/libraries/view-binding.

[18] Google. Viewmodel overview. https://developer.android.com/topic/libraries/architecture/viewmodel.

[19] Daniela Gotseva, Yavor Tomov, and Petko Danov. Comparative study java vs kotlin. In *2019 27th National Conference with International Participation (TELECOM)*, pages 86–89, 2019.

[20] Shawn Wang High Go. The difference in five modes in the aes encryption algorithm. https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/.

[21] IBM. Pkcs padding method. https://www.ibm.com/docs/en/zos/2.4.0?topic=rules-pkcs-padding-method.

[22] B. Kaliski. Pkcs5, cryptographic message syntax. https://www.ietf.org/rfc/rfc2315.txt.

[23] B. Kaliski. Pkcs5, password-based cryptography specification. https://www.ietf.org/rfc/rfc2898.txt.

[24] Sergey Komlach. Advanced biometricpromptcompat. https://github.com/sergeykomlach/AdvancedBiometricPromptCompat.

[25] kotlinlang. Asynchronous flow. https://kotlinlang.org/docs/flow.html.

[26] kotlinlang. Coroutine context and dispatchers. https://kotlinlang.org/docs/coroutine-context-and-dispatchers.html.

[27] kotlinlang. Coroutines basics. https://kotlinlang.org/docs/coroutines-basics.html#your-first-coroutine.

[28] Kotlinlang. Extensions. `https://kotlinlang.org/docs/extensions.html`.

[29] kotlinlang. measuretimemillis. `https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.system/measure-time-millis.html`.

[30] Kotlinlang. Type checks and casts. `https://kotlinlang.org/docs/typecasts.html`.

[31] Yu-Tsung Lee, Haining Chen, and Trent Jaeger. Demystifying android's scoped storage defense. *IEEE Security Privacy*, 19(5):16–25, 2021.

[32] McAffe. Mcafee mobile threat report. `https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf`.

[33] MFlisar. Materialpreferences. `https://github.com/MFlisar/MaterialPreferences`.

[34] moisoni97. Google in-app billing library v4. `https://github.com/moisoni97/google-inapp-billing`.

[35] Muhammad Mushtaq, Sapiee Jamel, Abdulkadir Disina, Zahraddeen Pindar, Nur Shakir, and Mustafa Mat Deris. A survey on the cryptographic encryption algorithms. *International Journal of Advanced Computer Science and Applications*, 8:333–344, 11 2017.

[36] nulab. zxcvbn4j. `https://github.com/nulab/zxcvbn4j`.

[37] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[38] Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., USA, 1982.

[39] Kirill Rozov. Viewbindingpropertydelegate. `https://github.com/androidbroadcast/ViewBindingPropertyDelegate`.

[40] Vijayalakshmi v, Ii Mahalakshmi, and Iii Thamizharasan. Data encryption hiding technique in non-standard cover files. *Advanced Research in Computer Science and Technology*, 03 2014.

[41] Priyank Vasa. Easycrypt. `https://github.com/pvasa/EasyCrypt`.

[42] Wikipedia. Functional programming. `https://en.wikipedia.org/wiki/Functional_programming`.

[43] Wikipedia. Padding (cryptography). `https://en.wikipedia.org/wiki/Padding_(cryptography)`.

[44] Wikipedia. Rijndael mixcolumns. `https://en.wikipedia.org/wiki/Rijndael_MixColumns#MixColumns`.