DOI: 10.37943/21LMQF2486

Leila Rzayeva
PhD, Associated Professor, Deputy of Head of the Department
of Intelligent Systems and Cybersecurity
l.rzayeva@astanait.edu.kz; orcid.org/0000-0002-3382-4685
Astana IT University, Kazakhstan

Abulkhair Imanberdi
Master of Science, Team Lead of State Technical Service JSC, CTF-Trainer
a.imanberdiyev@astanait.edu.kz; orcid.org/0009-0005-6144-2392
Astana IT University, Kazakhstan

Ivan Opirskyy
Doctor of Technical Sciences, Professor, Head of the Department
of Information Protection
ivan.r.opirskyi@lpnu.ua; orcid.org/0000-0002-8461-8996
Lviv Polytechnic National University, Ukraine

Oleh Harasymchuk
PhD, Associate Professor of the Department of Information Security
oleh.harasymchuk@gmail.com; orcid.org/0000-0002-8742-8872
Lviv Polytechnic National University, Ukraine

Gulnara Abitova
PhD, High-Researcher, Associated Professor of the Department
of Intelligent Systems and Cybersecurity
gulnara.abitova@astanait.edu.kz; orcid.org/0000-0003-3830-6905
Astana IT University, Kazakhstan

ANALYSIS OF TECHNICAL FEATURES OF DATA ENCRYPTION IMPLEMENTATION ON SD CARDS IN THE ANDROID SYSTEM

Abstract: This article provides a detailed analysis of data encryption mechanisms for removable storage devices in the Android operating system. Two main information protection technologies are examined: file-based encryption when using an SD card as portable storage and full-disk encryption when using a memory card as an extension of the device's internal storage (Adoptable Storage). The technical implementation features of each method are investigated, including the encryption algorithms used, the structure of encrypted data, and key storage mechanisms. The research was conducted using Sony Xperia XZ and Xiaomi Redmi 5 Plus devices, employing tools for working with file systems and encryption based on Linux and Android. The analysis has established that full-disk encryption is utilized the dm-crypt kernel module in plain mode with AES-256-CBC-ESSIV:SHA256 cipher. The partition encryption key is stored in the device's internal memory. File-based encryption employs the eCryptFS kernel module. The file structure includes information about the original file size, format marker, flags, number of extents, their size, and the encryption key. Comparative analysis has shown that Adoptable Storage mode provides more comprehensive data protection through full-disk encryption, while Portable Storage mode with file-based encryption offers greater flexibility in use but may be less secure due to the possibility of analyzing the file system structure and file metadata. Research has revealed the implementation of encryption mechanisms depends on the device manufacturer and Android operating system version. The research findings have

Received: 01.02.2025 Accepted: 24.03.2025 Published: 30.03.2025

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 21, MARCH 2025

practical significance for understanding the level of data protection using different modes of removable storage operation in the Android system and are useful for both developers and information security specialists, as well as ordinary users.

Keywords: encryption mechanisms, data encryption, file-based encryption (FBE), full-disk encryption (FDE), information security, encryption algorithms, data protection, android operating system, SD cards

Introduction

With the active development of information technologies and ubiquitous use of mobile devices, the volume of data stored and processed on smartphones, tablets, and other portable devices is growing exponentially. A significant portion of this information is confidential: personal photos, documents, passwords, financial data, and other sensitive information. This makes data protection on mobile devices one of the key aspects of information security in today's digital world [1].

The Android operating system, which holds a leading position in the mobile platform market, provides users with various mechanisms to protect their data. A special place among these mechanisms belongs to encryption technologies for information stored both in devices' internal memory and on removable storage [2]. Given that many users use SD cards to expand their devices' memory, the issue of data protection on removable media becomes particularly relevant.

The Android operating system implements two fundamentally different approaches to working with removable media: Portable Storage mode and Adoptable Storage mode. Each of these modes has its own implementation features for data encryption mechanisms, which directly affects the level of user data protection [3], [4]. It is important to understand not only the general principles of these mechanisms but also the technical details of their implementation, including the encryption algorithms used, key storage methods, and the organization of encrypted data.

The relevance of studying data encryption mechanisms for removable media in the Android system is also because understanding the technical aspects of their implementation is critically important for assessing the actual level of information security and potential vulnerabilities [5]. This is particularly important in the context of increasing cyber threats and continuous improvement of unauthorized data access methods. A deep understanding of encryption systems' operating principles allows not only to evaluate their effectiveness but also to develop recommendations for their improvement and secure use.

It is worth noting that the study of encryption mechanisms in Android is important not only for developers and information security specialists but also for ordinary users who need to understand how securely their data is protected when using different modes of removable storage operation. This understanding allows you to make informed decisions about choosing methods for storing and protecting important information.

Problem Statement. In the context of rapid mobile technology development and growing requirements for confidential data protection, the study of information encryption mechanisms on mobile devices becomes particularly significant. Although the Android operating system provides built-in data encryption tools for removable media, the technical aspects of their implementation remain insufficiently studied.

The main problem lies in the lack of systematized knowledge about the implementation features of different data encryption methods when using SD cards in the Android system. Encryption mechanisms in Portable Storage and Adoptable Storage modes require detailed study, including analysis of the algorithms used, encrypted data structure, and encryption key storage methods.

DOI: 10.37943/21LMQF2486

Additional complexity arises from the fact that different mobile device manufacturers may use their own modifications of standard encryption mechanisms, which complicates understanding the overall picture of data security on removable media. Furthermore, the lack of documentation regarding some aspects of encryption implementation complicates the process of researching and evaluating the effectiveness of these mechanisms [6].

Solving this problem requires conducting a comprehensive technical analysis of data encryption mechanisms for removable media in the Android operating system, which will allow better understanding of information security levels and identifying potential vulnerabilities in existing data protection methods.

The aim of this paper is to analyze the technical features of the data encryption mechanisms implementation on the removable media in the Android operating system, determine the encryption algorithms use, and evaluate their effectiveness.

To achieve this aim, the following tasks must be accomplished:

1. Investigate the implementation features of file-based encryption when using an SD card in Portable Storage mode.
2. Analyze full-disk encryption mechanisms when using an SD card in Adoptable Storage mode.
3. Identify encryption algorithms and key storage methods used in both modes.
4. Evaluate the effectiveness and reliability of the studied encryption mechanisms.

Materials and Methods

This research focused on the analysis of data encryption mechanisms for removable media within the Android operating system. The study examines two primary approaches to data protection: file-based encryption when using an SD card as portable storage and full-disk encryption when using the memory card as an extension of the device's internal memory (Adoptable Storage).

Research Methods:

1. Technical Specification Analysis: Examination of documentation and technical specifications related to the encryption algorithms used and their implementation in Android.
2. Experimental Verification: Testing encryption methods in practice using various Android devices to analyze data encryption behavior on SD cards.
3. System Analysis: Detailed review of file systems used in different modes of SD card connection and data encryption methods.
4. Instrumental Analysis: Utilization of tools such as ecryptfs-utils for analyzing the structure of encrypted files and crypt setup for investigating the application of dm-crypt.

Research Materials:

- Mobile Devices: Various models of Android smartphones, including Sony Xperia XZ and Xiaomi Redmi 5 Plus.
- Software: Toolsets for working with file systems and encryption based on Linux and Android.
- Technical Databases and Specifications: Documentation and specifications related to the encryption algorithms used and their implementation in Android.

This approach to research allows for an in-depth analysis of various aspects of data encryption on removable media and assessment of the level of data protection when using different modes of removable media operation in the Android system.

Analysis of Recent Research and Publications. The Android operating system offers three options for connecting an SD card as a data storage device: Portable Storage [7], Adoptable Storage [8], and Mixed Storage (or Semi-Adoptable Storage) [9].

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 21, MARCH 2025

When connecting a card as Portable Storage, a FAT32 or exFAT file system is created on the card (if not already present), after which the file system is mounted at /storage/XXXX-XXXX, where XXXX-XXXX is serial number of the file system. With this connection type, only user files can be stored on the memory card; application storage is not possible due to the limitations of FAT32 and exFAT file systems compared to f2fs [10] and ext4 [11] file systems used for device internal storage. This connection method remains the only option for many devices that still feature SD card slots. This method is optimal for most users since such SD cards can be removed from the device and connected to a computer for data transfer.

When connecting a card as Adoptable Storage, a partition is created with the same file system used for the device's /data partition. The created file system is unified with file system of the device in a manner similar to creating a composite volume in Windows OS. This connection method allows not only user data storage but also the transfer of applications, system information, etc. Using FAT and NTFS family file systems for storing application data is impossible since these file systems do not support file attributes used by Linux kernel-based operating systems, including Android. This connection method has not gained popularity due to its inconvenience and because not all device manufacturers allowed its use. Currently, using Adoptable Storage is impractical due to the increase in device internal storage to acceptable levels and the removal of SD card slots from new devices. Among the main advantages of Adoptable Storage mode were seamless internal memory expansion and reliable data encryption.

Additionally, in Android 6.0, some manufacturers (such as Motorola) offered a Mixed Storage option. This hybrid mode was an attempt to find a compromise between fully portable and fully adoptable modes of SD card usage in Android. Its implementation and the reasons for limited adoption will be examined in detail.

Semi-Adoptable Storage involves dividing the SD card into two logical partitions. The first partition functions as adoptable storage, integrating with the device's internal memory and allowing application installation. The second partition functions as regular portable storage, available for file storage and transfer between devices.

When setting up hybrid mode, the system performs the following actions:

1. Creates two partitions on the card with different file systems: ext4 for the adoptable portion and FAT32/exFAT for the portable portion.
2. Encrypts the adoptable partition, integrating it with Android's system file structure.
3. Establish separate access rules and data management for each partition.
4. Configure system services to work simultaneously with both storage types.

This mode did not become popular and was quickly abandoned by manufacturers and developers for several reasons:

Technical Complexity: Supporting two different operation modes on a single memory card creates an additional system overhead and complicates memory management. This can lead to decreased performance and increased power consumption.

User Complexity: The concept of split storage proved too complicated for average users to understand. Many found it difficult to determine which data was stored in which partition, leading to confusion.

Reliability Issues: Using hybrid mode increases the risk of data corruption due to the more complex file system structure and increased number of data operations.

Limited Manufacturer Support: Most smartphone manufacturers have preferred implementing simpler and more straightforward SD card operation modes, resulting in limited support for hybrid mode at the device level.

Alternative Solutions: With the emergence of devices with large internal storage capacity and the development of cloud storage, the need has decreased for complex SD card operation modes.

DOI: 10.37943/21LMQF2486

Although hybrid mode has offered an interesting technical solution, the complexity of its implementation and use, along with changes in user needs, has resulted in its failure to gain widespread adoption in the Android ecosystem.

Results and Discussion

Data Encryption With Connecting a Card as Portable Storage

Data encryption verification in Portable Storage connection mode was conducted using a Sony Xperia XZ device (Android 8.1.0, official firmware, version 41.3.A.2.192).

The test SD card was formatted using the Android system tools and connected as external storage. Within copying data, it was saved in its original form without encryption. In the framework of enabling the encryption option (Figure 1), files were unencrypted that already remained on the SD card. After conducting the data encryption, nine (9) photos with a total size of 55.6 megabytes were copied to the memory card. The copying process took approximately 25 seconds. This result indicates a significant input-output speed limitation for operations involving data encryption. This could be caused by either insufficient hardware capabilities of the device or specific features of the software version, which operates with considerable delays and device heating. After copying, the photos have remained accessibility for viewing through the device's tools, as well as within connecting the device to a PC in MTP mode. By connecting the memory card directly to a PC, the files retain their names and extensions but do not preserve metadata and become inaccessible for viewing and editing.

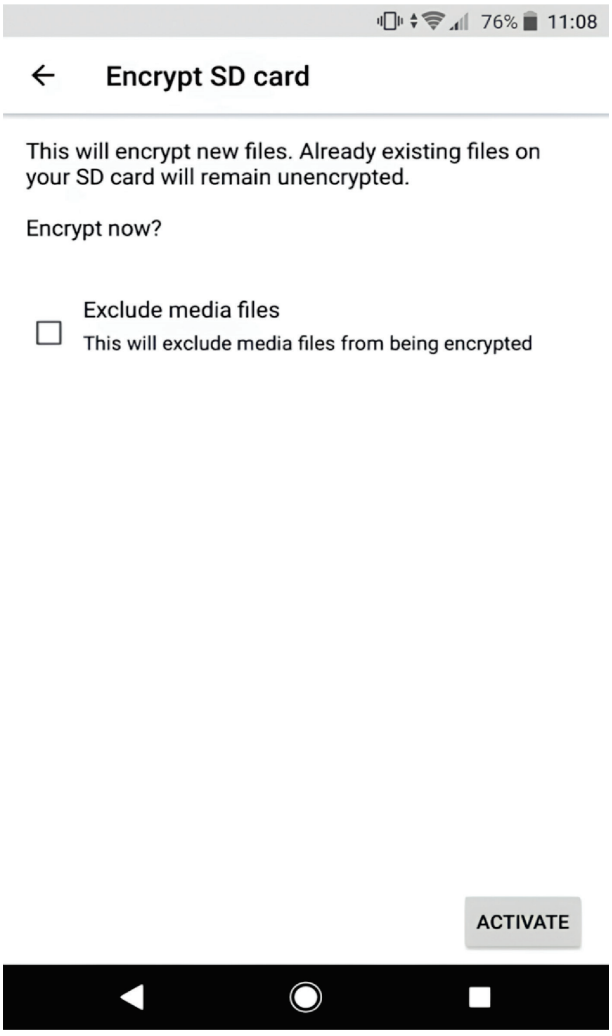

Figure 1. Enabling of the file encryption option

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 21, MARCH 2025

Analysis of the encrypted file structure using a hex editor has revealed a header that corresponds to the eCryptFS utility [12] file-based encryption format (Figure 2).

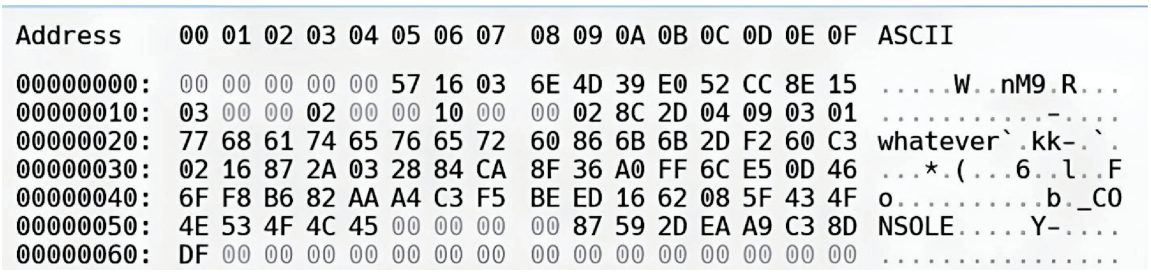

Figure 2. File header encrypted by the device

The file header consists of information about the original file size, format marker, flags, number of extents (continuous data segments, segmentation is necessary to ensure random access to data), their size (corresponds to the file system block size), encrypted file encryption key, and the extents themselves (starting from offset 0x2000) (Figure 3) [13]. For encrypting extents, the AES algorithm is presumably used in Cipher Block Chaining (CBC) mode with a 128-bit key length.

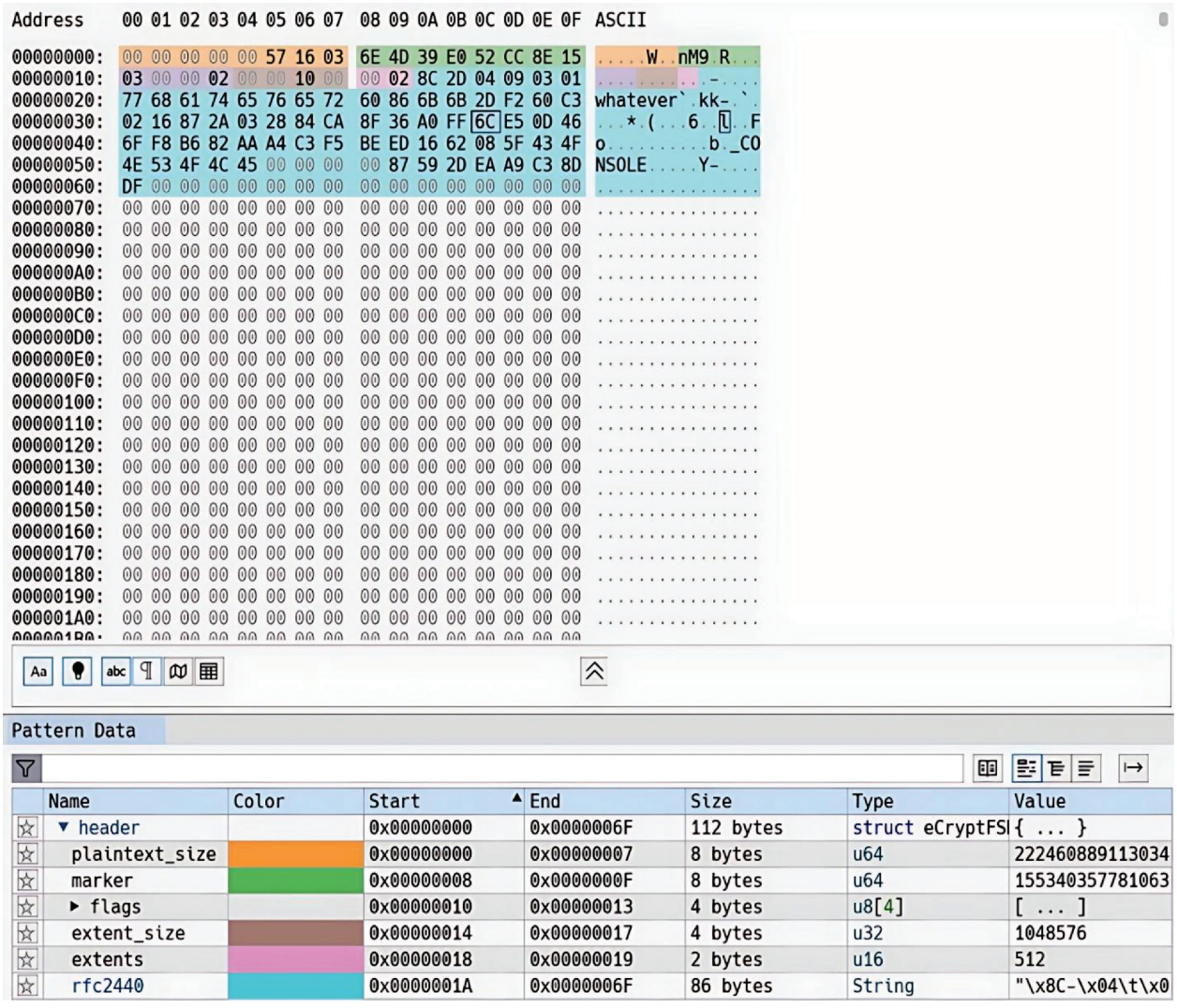

Figure 3. eCryptFS file header structure

DOI: 10.37943/21LMQF2486

Several main stages and components are involved in the process of encrypting and decrypting files using eCryptFS (Figure 4).

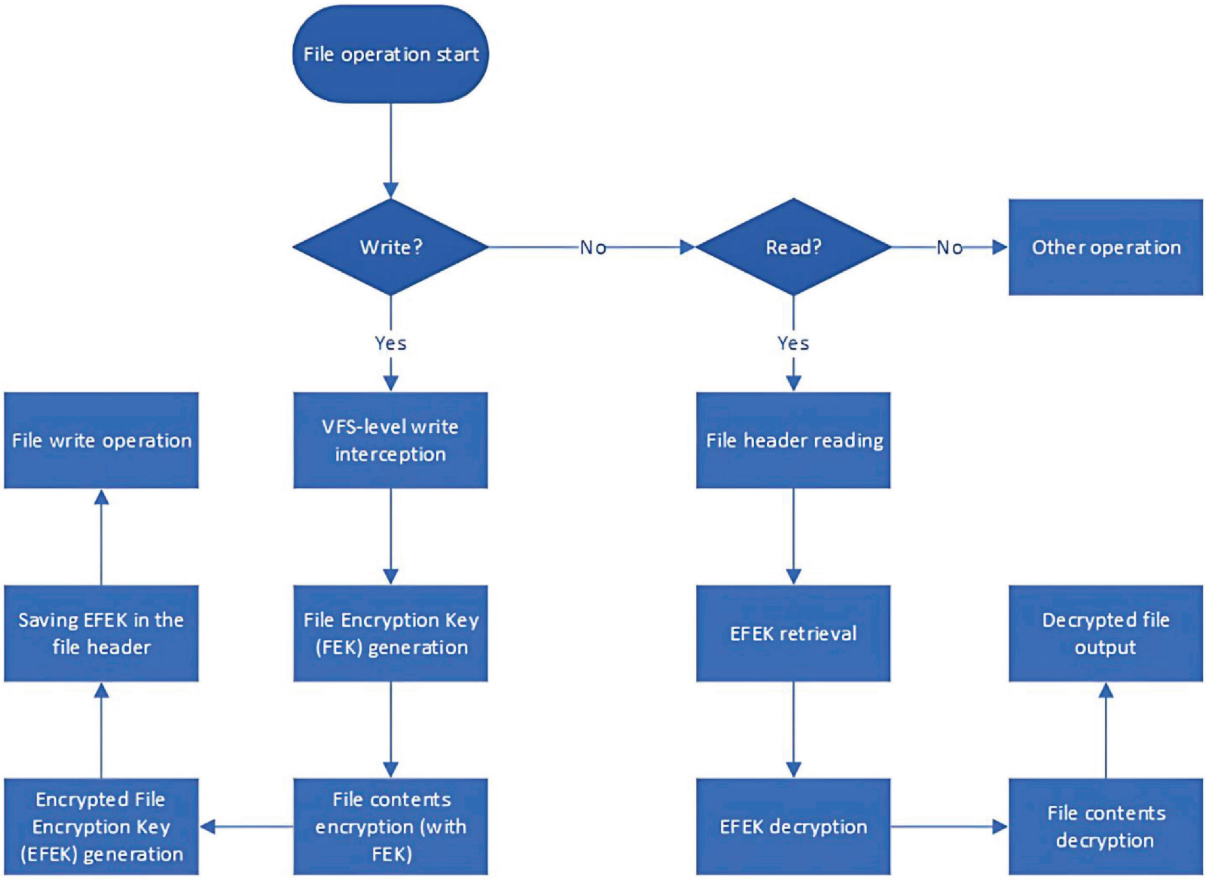

Figure 4. eCryptFS file operation process

File encryption

1. VFS-level interception

The Virtual File System (VFS) in Linux serves as an abstraction layer between user applications and specific filesystem implementations. This architecture allows Linux to support multiple filesystems while presenting a unified interface to applications. VFS-level interception works through a technique called «stacking,» where one filesystem mounts on top of another. When a filesystem such as eCryptFS is implemented as a stacking filesystem, it intercepts file operations before they reach the lower filesystem.

The following processes are conducted when an application reads from a file on an eCryptFS file system:

1. The application calls a standard read() system call.
2. The VFS layer routes this call to the eCryptFS read implementation.
3. eCryptFS retrieves the encrypted data from the lower filesystem.
4. eCryptFS decrypts the data using the appropriate key.
5. The decrypted data is returned to the application.

This architecture allows eCryptFS to provide transparent encryption and decryption without requiring modifications to applications or the underlying storage filesystem.

2. FEK Generation

File encryption key (FEK) is generated with the Linux system call (Linux/drivers/char/random.c) get_random_bytes(void *buf, size_t len). The Linux kernel has gathers entropy from

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 21, MARCH 2025

many system events, such as user input events, CPU clock jitter, I/O timings and such. Entropy data is gathered into pools, which are used to seed and reseed the deterministic RNG, based on ChaCha20 stream cipher, that is repeatedly running its block function, generates a stream of pseudo-random bytes. A "fast key erasure" mechanism is used to update the DRNG state and maintain forward security. The get_random_bytes() function in the drivers/char/random.c file is responsible for generating a specified number of random bytes and storing them in a buffer.

ChaCha20 is a symmetric stream cipher designed by Daniel J. Bernstein in 2008 as an evolution of his earlier Salsa20 algorithm. It provides high-performance encryption with strong security properties, making it suitable for various data encryption applications and random number generation. ChaCha20 operates by transforming a fixed-length input into a pseudorandom keystream, which is then combined with plaintext through XOR operations to produce ciphertext. The encryption process follows a defined sequence:

A. Initialize a 4×4 state matrix using constants, the encryption key, a counter, and the nonce.
B. Apply 20 rounds of mixing operations (10 column rounds alternating with 10 diagonal rounds).
C. Add the resulting matrix to the initial state matrix.
F. Generate the keystream by serializing the final state.
E. XOR the keystream with the plaintext to produce the ciphertext.

Typical FEK length is 128 bytes, but it may differ based on the encryption algorithm in use (AES-128 is the default). ECryptFS may use any encryption algorithm provided by the Linux kernel crypto API.

3. File contents encryption.

ECryptFS has utilizes Linux kernel Crypto API calls to perform data page encryption. The first step of the page encryption is its initialization vector generation. Page IV is calculated as md5(ascii(page_offset) | root_iv). Root IV is stored in the file header. If HMAC verification was enabled at the mounting stage, then the HMAC checksum will be generated for every page write operation and it will be stored in the lower file.

4. EFEK generation

The file encryption key should be converted to the EFEK using the user's passphrase or public key. Android systems typically use pubkey-based EFEK derivation. EFEK pubkey generation is performed by the userspace, which communicates with the kernel using an OpenPGP-like kernel-userspace communication protocol. Four message formats are determined: Tag 64 (Public Key Decryption Request), Tag 65 (Public Key Decryption Reply), Tag 66 (Public Key Encryption Request) and Tag 67 (Public Key Encryption Reply). Passphrase-based EFEK generation is implemented in kernelspace without transferring the key data into userspace.

4.1. Passphrase-based key derivation

Passphrase-based key derivation mechanism is based on the Iterated and Salted S2K as being described in RFC 2440 "OpenPGP Message Format" (deprecated by RFC 4880), Section 3.6.

S2K (string to key) procedure has converts human-readable secrets to machine-readable keys, which can be used for cryptographic purposes. RFC 2440 (and RFC 4880) define 3 types of S2K procedures: "Simple S2K", "Salted S2K" and "Iterated and Salted S2K"; the latter one is used by the eCryptFS key derivation. Iterated and Salted S2K uses an 8-octet (64 bits) salt value, which is concatenated with the input and a 1-octet count value, which is calculated as count = ((Int32)16 + (c & 15)) << ((c >> 4) + EXPBIAS), where EXPBIAS is 6. Overall, the data block is iteratively MD5-hashed 65536 times and then used to encrypt the file encryption key.

MD5 (Message Digest Algorithm 5) is a widely recognized cryptographic hash function that produces a 128-bit (16-byte) hash value. Developed by Ronald Rivest in 1991 to replace the earlier MD4 algorithm. At its foundation, MD5 operates in the realm of modular arithmetic, specifically using operations in the 2^{32} integer space.

DOI: 10.37943/21LMQF2486

MD5 begins with four 32-bit initialization vectors (A, B, C, D):

- A = 0x67452301.
- B = 0xEFCDAB89.
- C = 0x98BADCFE.
- D = 0x10325476.

Four primary nonlinear functions are employed during the four rounds of processing:

- $F(B, C, D) = (B \wedge C) \vee (\neg B \wedge D)$.
- $G(B, C, D) = (B \wedge D) \vee (C \wedge \neg D)$.
- $H(B, C, D) = B \oplus C \oplus D$.
- $I(B, C, D) = C \oplus (B \vee \neg D)$.

For each 512-bit block of the padded message, the algorithm performs:

A. Buffer Copy: The current state (A, B, C, D) is copied to temporary variables (a, b, c, d).

B. Round Processing: For each of the 64 steps (16 steps in each of the 4 rounds), the algorithm performs:

$$a[i+1] = b[i] + ((a[i] + g(b[i], c[i], d[i]) + X[k[i]] + T[i]) <<< s[i]).$$

Where:

- g is one of the nonlinear functions (F, G, H, or I).
- X[k[i]] represents the k[i]-th 32-bit word from the current block.
- T[i] is the i-th element of a precomputed table.
- s[i] specifies the amount of circular left shift.

After each step, the variables are rotated: $(a, b, c, d) \rightarrow (d, a, b, c)$

***State Update**:* After 64 steps, the algorithm updates the state variables: $A = A + a$ $B = B + b$ $C = C + c$ $D = D + d$.

Researchers have identified mathematical techniques to find two different inputs that produce the same hash, which undermines its security. These collision attacks leverage differential cryptanalysis techniques to find specific patterns that, when modified in particular ways, do not affect the final hash value.

4.2. Pubkey-based key derivation

RSA encryption has used to perform FEK encryption with public keys. Pubkey authentication tokens store the public key and its hash as the token signature; the private key is used to decrypt the EFEK. Any number of Public Key Infrastructures may be used with plugins, the most common PKI plugins are OpenSSL and GnuPG. The most suitable PKI for hardware-backed encryption is TSPI (TPM Serial Peripheral Interface). Android devices may use hardware-tailored PKI plugins for TEE keybox access.

4.3. EFEK generation process

File Encryption Key (FEK) is then encrypted with generated FEFEK (FEK Encryption Key) using the AES algorithm in CBC mode. Key length is defined at the filesystem mount stage.

5. EFEK storage

Newly created EFEK (Encrypted FEK) is stored in the file header, along with the necessary cryptographic context to perform its decryption. Cryptographic context data are also known as "Authentication token" in eCryptFS terminology.

File decryption

1. EFEK retrieval

Header of the file is read and parsed. The authentication token identifier is compared to the set of tokens in keyring, then the type of the token is determined and, in case of a match between two instances of tokens, corresponding EFEK decryption process is initiated.

2. EFEK decryption

If the EFEK is encrypted with the passphrase, an FEFEK is being generated using the "Iterated and Salted S2K" procedure. If the EFEK is encrypted with a public key, it will be forwarded to

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 21, MARCH 2025

the userspace and decrypted using the determined PKI plugin. Decrypted FEK is then returned to the kernel module and is used for the data decryption and encryption.

3. File contents decryption

On a read call, eCryptFS will recover the page index, calculate the IV for a particular page and transfer the data to the crypto API to perform the decryption. Decrypted data will be transferred to the userspace application via VFS syscalls. This makes file encryption transparent for userspace applications and efficient, as only the needed blocks are decrypted and transferred.

When a user creates or saves a file in an encrypted directory, eCryptFS first intercepts the operation at the Virtual File System (VFS) level. At this stage, the system generates a unique File Encryption Key (FEK) [14], [15]. This key is randomly created for each individual file.

The system has uses the generated FEK to encrypt the file contents. Encryption has occurred on a per-page basis, typically in 4 KB blocks. This enables efficient access to individual parts of the file without needing to decrypt the entire file. The FEK itself is encrypted using the user's master key, which is derived from the user's password.

The encrypted FEK is stored in the file metadata along with additional information, such as the encryption algorithm used and other parameters. On the next stage, the encrypted data and metadata have passed to the lower level of the file system for physical storage on the disk [16].

When a user attempts to access an encrypted file, the process has occurred in reverse order. First, the system has read the file metadata and extracts the encrypted FEK. Using the user's master key (which is obtained from the password when mounting the file system), the system has decrypted the FEK.

After obtaining the decrypted FEK, the system uses it to decrypt the file contents. Decryption also occurs on a per-page basis, allowing efficient operation with large files. The decrypted data is passed to the program that requested file access.

eCryptFS has provided transparent encryption, meaning automatic encryption and decryption of files without requiring special actions from users or programs. The system also supports filename encryption, providing an additional level of confidentiality.

It's important to note that eCryptFS keeps the directory structure unencrypted, encrypting only file contents and, when necessary, their names. This allows the file system to work efficiently with the directory tree while maintaining data confidentiality.

To decrypt a file encrypted by eCryptFS, it is necessary to obtain the FEK. In full GNU/Linux family operating systems, the encrypted mount key file (master key) is in the user's home directory and is decrypted either using a passphrase/key (legitimate method) or through brute force (illegitimate method). Since Android OS doesn't have a user directory, the next step was to check the key storage directory /data/misc/keystore/user_0 (note that this directory is only accessible with root privileges). After scanning all key files in this directory using the ecryptfs2john utility from the standard Kali Linux distribution toolkit, no key was recognized as an eCryptFS wrapped passphrase (Figure 5).

DOI: 10.37943/21LMQF2486

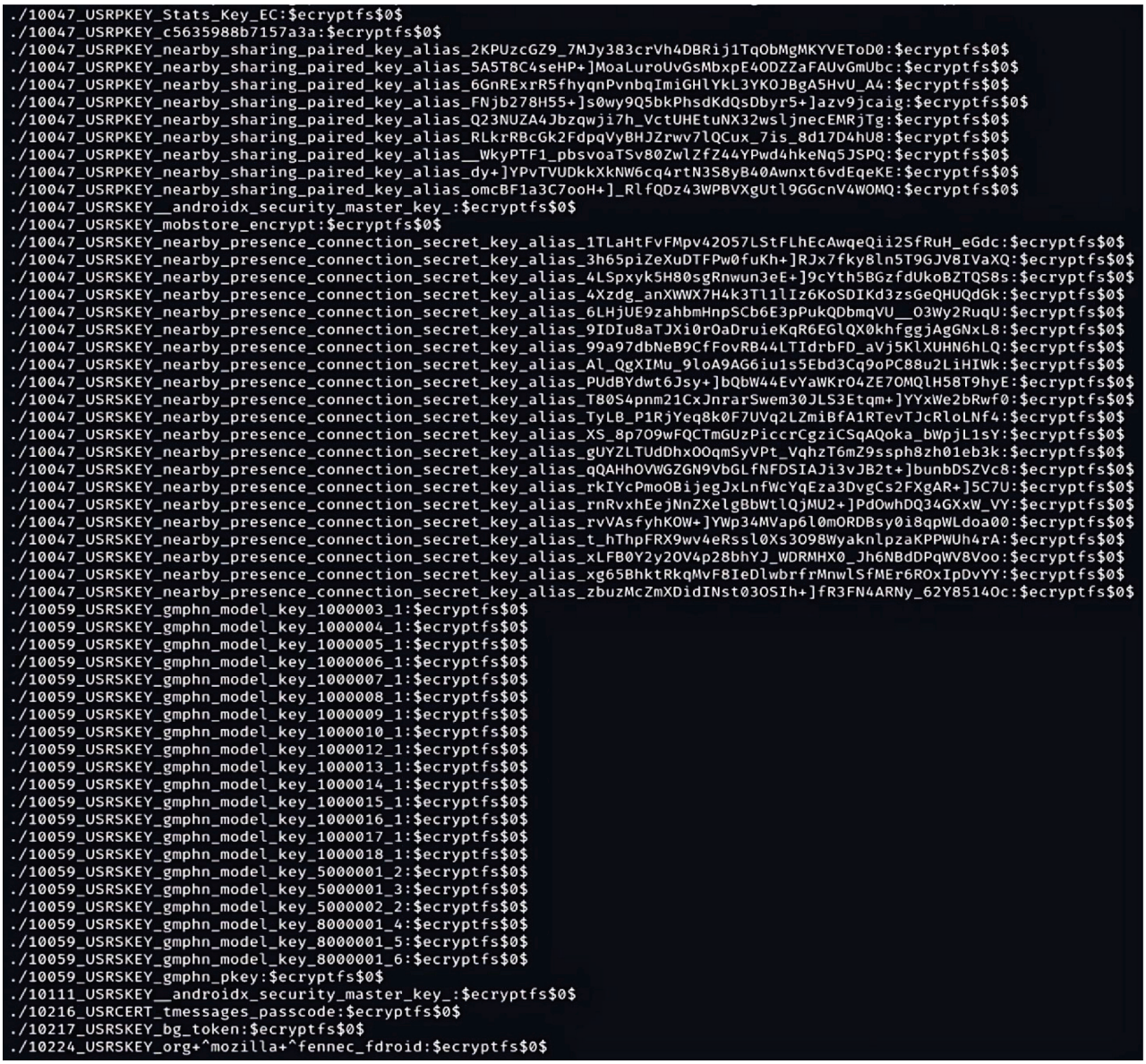

Figure 5. Checking for eCryptFS encryption keys in the Android system keystore

The use of eCryptFS is specific to several Android device manufacturers, with Sony and Samsung being the most notable among them. The utilities for working with eCryptFS in these manufacturers' firmware are not standard and are not accessible through the superuser console. Therefore, further steps involve file-by-file system inspection and analysis of firmware binary files to identify file decryption key storage locations.

Data Encryption Within Connecting a Card as Internal Storage Extension

To verify Adoptable Storage encryption, a Xiaomi Redmi 5 Plus device was used (Android 8.1.0, custom firmware, crDroid 4.7 version). The card was formatted for Adoptable Storage using standard firmware tools and connected with the transfer of 104 megabytes of data from internal memory. After data transferring, the SD card was removed after properly shutting down the device for further analysis.

The operating system's standard tools have created two partitions on the memory card (Figure 6) with an unknown file system.

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 21, MARCH 2025

```
gilah@magi-arch ~ % sudo fdisk -l /dev/mmcblk0
Disk /dev/mmcblk0: 29.12 GiB, 31266439168 bytes, 61067264 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: B7B39FFC-D913-4603-A07E-725D555A5F32

Device         Start      End  Sectors  Size Type
/dev/mmcblk0p1  2048    34815    32768   16M unknown
/dev/mmcblk0p2 34816 61067230 61032415 29.1G unknown
```

Figure 6. Memory card partition structure in Adoptable Storage mode

Verification using built-in cryptsetup [17] and dmsetup [18] has utilities (Figure 7) also revealed no use of LUKS, leading to the conclusion that the dm-crypt module's plain mode is being used, which does not contain partition headers and is essentially indistinguishable from random data.

```
gilah@magi-arch ~ % sudo blkid /dev/mmcblk0p1
/dev/mmcblk0p1: UUID="4936-1C14" BLOCK_SIZE="512" TYPE="vfat" PARTLABEL="android_meta" PARTUUID="c6fb2eca-
gilah@magi-arch ~ % sudo blkid /dev/mmcblk0p2
/dev/mmcblk0p2: PARTLABEL="android_expand" PARTUUID="419aa35f-5fcb-4ea7-8b6b-c968eeed90eb"
gilah@magi-arch ~ % sudo cryptsetup luksDump /dev/mmcblk0p1
Device /dev/mmcblk0p1 is not a valid LUKS device.
1 gilah@magi-arch ~ % sudo cryptsetup luksDump /dev/mmcblk0p2
Device /dev/mmcblk0p2 is not a valid LUKS device.
1 gilah@magi-arch ~ %
```

Figure 7. Results of LUKS technology usage verification

Dm-crypt supports encryption using both plain mode and LUKS (Linux Unified Key Setup) [19], [20]. Plain mode is the basic encryption mode that provides only core encryption functionality without additional key management capabilities. When using plain mode, the password is directly converted into an encryption key through a hash function. This mode doesn't store any metadata on the disk, making it more resistant to cryptanalysis but significantly less convenient to use.

LUKS is a more modern and functional solution. It has created a special header on the disk that contains all necessary information about encryption parameters and keys. LUKS has supported up to eight different keys for a single partition, allowing access for different users or maintaining backup keys. An important feature is the use of a master key, which encrypts the data, while user passwords encrypt this master key.

Regarding practical applications, LUKS is the recommended option for most use cases as it provides:

- standardized disk header format.
- ability to change passwords without re-encrypting data.
- enhanced protection against brute force attacks using salt values.
- support for managing multiple keys.

Plain mode can be useful in specific scenarios, such as:

- when maximum concealment of the encryption fact itself is required.
- when creating custom key management systems.
- in cases where implementation simplicity is critically important.

DOI: 10.37943/21LMQF2486

It's worth noting that LUKS requires additional disk space for header storage (usually several megabytes), while plain mode requires no additional space. However, this difference is insignificant for modern storage systems.

In the Android operating system, the Adoptable Storage partition encryption key is stored at /data/misc/vold. The partition can be decrypted if the encryption key is obtained using root privileges or other methods [21]. The key file was successfully transferred to a computer, after which the memory card partition /dev/mmcblk0p2 was successfully decrypted. Through trial and error, it was determined that the AES-128 encryption algorithm is used in CBC mode [22], [23] with ESSIV (Encrypted Salt-Sector Initialization Vector) [24], [25] initialization vector computation algorithm (Figure 8).

```
gilah@magi-arch ~ % sudo cryptsetup open --type plain --cipher aes-cbc-essiv:sha256 \
\`> --key-file expand_419aa35f5fcb4ea78b6bc968eeed90eb.key --key-size=128 /dev/mmcblk0p2 adoptable

gilah@magi-arch ~ % sudo mount /dev/mapper/adoptable /mnt
gilah@magi-arch ~ % ls -lha /mnt
total 30K
drwxr-xr-x  7 root  root  4.0K Dec  2 22:51 .
drwxr-xr-x 19 root  root  4.0K Nov 29 16:18 ..
drwxrwx--x  2 gilah gilah 3.5K Dec  2 22:51 app
drwxr-x--x  3 root  root  3.5K Dec  2 22:51 local
drwxrwx---  3  1023  1023 3.5K Dec  2 22:51 media
drwx--x--x  3 gilah gilah 3.5K Dec  2 22:51 user
drwx--x--x  3 gilah gilah 3.5K Dec  2 22:51 user_de
gilah@magi-arch ~ % sudo blkid /dev/mapper/adoptable
/dev/mapper/adoptable: UUID="7c888bdc-c501-4f58-8b9b-6c07734eff87" BLOCK_SIZE="4096" TYPE="f2fs"
gilah@magi-arch ~ %
```

Figure 8. Successful decryption of the SD card partition and its contents

Since the memory card was just formatted, it contains no files, only the directory structure created by the system (Figure 9). The purpose of the directories has corresponded to their names.

```
gilah@magi-arch ~ % sudo tree /mnt
/mnt
|-- app
|-- local
|   `-- tmp
|-- media
|   `-- 0
|-- user
|   `-- 0
`-- user_de
    `-- 0

10 directories, 0 files
gilah@magi-arch ~ %
```

Figure 9. Directory structure of an empty memory card in Adoptable Storage mode

An analysis of the decrypted data has shown that the system creates a standard directory structure on the memory card, which has corresponded to the main categories of data that can be stored on the device. All files written to the memory card are automatically encrypted by the system using the same encryption key.

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 21, MARCH 2025

It is important to note that by removing the memory card from the device, access to the data becomes impossible without the corresponding encryption key. This provides reliable protection of information in case of loss or theft of the memory card. However, as the research has shown, having root access to the device allows obtaining the encryption key and, consequently, access to the data on the memory card.

Comparing both studied data encryption methods, it is noted that Adoptable Storage mode provides more comprehensive data protection through full-disk encryption, while Portable Storage with file-based encryption offers greater flexibility in use but may be less secure due to the possibility of analyzing the file system structure and file metadata.

The research has also revealed that the implementation of encryption mechanisms can vary depending on the device manufacturer and Android operating system version, which creates additional complexities when analyzing data security on removable media. This emphasizes the need for further research in this direction, especially in the context of new Android versions and various implementations from device manufacturers.

Conclusion

This research has examined the main data encryption methods for removable media used in the Android operating system: file-based encryption and full-disk encryption. File-based encryption can be used for connecting a memory card as external storage, while full-disk encryption is mandatory by connecting a memory card as an extension of the device's internal memory.

It was determined that full-disk encryption uses the dm-crypt kernel module in plain mode with AES-256-CBC-ESSIV:SHA256 cipher, with the partition encryption key stored in the device's internal memory at /data/misc/vold/expand_<PARTUUID>.key; file-based encryption in the studied device uses the eCryptFS kernel module, though the location of its master key could not be determined.

Acknowledgment

This study was carried out with the financial support of the Committee of Science of the Ministry of Science and Higher Education of the Republic of Kazakhstan under Contract №388/PTF-24-26 dated 01.10.2024 under the scientific project IRN BR24993232 "Development of innovative technologies for conducting digital forensic investigations using intelligent software-hardware complexes".

References

[1] Bhat, P., & Dutta, K. (2019). A survey on various threats and current state of security in Android platform. ACM Computing Surveys (CSUR), 52(1), 1-35. https://doi.org/10.1145/3301285

[2] Wei, F., Roy, S., Ou, X., & Robby. (2018). Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Transactions on Privacy and Security (TOPS), 21(3), 1-32. https://doi.org/10.1145/3183575

[3] Nurse, J.R., Creese, S., & De Roure, D. (2017). Security risk assessment in Internet of Things systems. IT professional, 19(5), 20-26. https://doi.org/10.1109/MITP.2017.3680959

[4] Scrivens, N., & Lin, X. (2017, May). Android digital forensics: data, extraction and analysis. In Proceedings of the ACM Turing 50th Celebration Conference-China (pp. 1-10). https://doi.org/10.1145/3063955.3063981

[5] Tam, K., Feizollah, A., Anuar, N. B., Salleh, R., & Cavallaro, L. (2017). The evolution of android malware and android analysis techniques. ACM Computing Surveys (CSUR), 49(4), 1-41.

[6] Yang, L., Wei, T., Zhang, F., & Ma, J. (2018). SADUS: Secure data deletion in user space for mobile devices. *computers & security*, *77*, 612-626. https://doi.org/10.1016/j.cose.2018.05.013

[7] Android Open-Source Project. Traditional storage | Android Open-Source Project. Android Open-Source Project. URL: https://source.android.com/docs/core/storage/traditional
[8] Android Open-Source Project. Adoptable storage | Android Open-Source Project. Android Open-Source Project. URL: https://source.android.com/docs/core/storage/adoptable
[9] What happened to Android's adopted storage option that allowed you to mount the SD card as internal storage space? My S10 Plus had an upgrade to Android 12 and I can't find it anywhere. Quora. URL: https://www.quora.com/What-happened-to-Android-s-adopted-storage-option-that-allowed-you-to-mount-the-SD-card-as-internal-storage-space-My-S10-Plus-had-an-upgrade-to-Android-12-and-I-cant-find-it-anywhere
[10] Linux Kernel Organization, Inc. (n.d.-c). WHAT IS Flash-Friendly File System (F2FS)? The Linux Kernel Documentation. URL: https://docs.kernel.org/filesystems/f2fs.html
[11] Linux Kernel Organization, Inc. (n.d.-b). ext4 data structures and algorithms. The Linux Kernel Documentation. URL: https://docs.kernel.org/filesystems/ext4/index.html
[12] Kirkland D. eCryptfs. eCryptfs. URL: https://www.ecryptfs.org/
[13] Euresys s.a. (n.d.). eCryptfs header. Euresys Documentation. URL: https://documentation.euresys.com/Products/PICOLO_NET_HD1/PICOLO_NET_HD1/en-us/Content/encrypted-media-storage/ecryptfs-header.htm
[14] Halcrow, M.A. (2005, July). eCryptfs: An enterprise-class encrypted filesystem for Linux. In Proceedings of the 2005 Linux Symposium (Vol. 1, pp. 201-218). URL: https://www.kernel.org/doc/mirror/ols2005v1.pdf#page=209
[15] Linux Kernel Organization, Inc. (n.d.-a). Encrypted keys for the eCryptfs filesystem. The Linux Kernel Archives. URL: https://www.kernel.org/doc/html/v4.17/security/keys/ecryptfs.html
[16] Kaaniche, N., Laurent, M., & Belguith, S. (2020). Privacy enhancing technologies for solving the privacy-personalization paradox: Taxonomy and survey. Journal of Network and Computer Applications, 171, Article 102807. https://doi.org/10.1016/j.jnca.2020.102807
[17] Cryptsetup / cryptsetup · GitLab. GitLab. URL: https://gitlab.com/cryptsetup/cryptsetup
[18] Kerrisk M. dmsetup(8) – Linux manual page. URL: https://man7.org/linux/man-pages/man8/dmsetup.8.html
[19] Demir, L., Thiery, M., Roca, V., Tenkes, J., & Roch, J. (2020). Optimizing dm-crypt for XTS-AES: Getting the best of Atmel cryptographic co-processors. In Proceedings of the 17th International Joint Conference on e-Business and Telecommunications (ICETE 2020) – SECRYPT (Vol. 1, pp. 263–270). SCITEPRESS – Science and Technology Publications. https://doi.org/10.5220/0009767802630270
[20] Anton Dănuț, S., & Simion, E. (2019). Linux Unified Key Setup (LUKS) – The good, the bad, the ugly. In 2018 10th International Conference on Electronics, Computers and Artificial Intelligence (ECAI). https://doi.org/10.1109/ECAI.2018.8678978
[21] PO David. How to decrypt and split adopted storage?. XDA Developers. URL: https://xdaforums.com/t/how-to-decrypt-and-split-adopted-storage.3383666/
[22] Chang, K.-C., Teng, Y.-T., & Chin, W.-L. (2023). High-throughput CBC mode crypto circuit. Electrical Science & Engineering, 5, 20–30. https://doi.org/10.30564/ese.v5i1.5636
[23] Alimzhanova, Z., Skublewska-Paszkowska, M., & Nazarbayev, D. (2023). The periodicity detection of the substitution box in the CBC mode: Experiment and study. IEEE Access. https://doi.org/10.1109/ACCESS.2023.3295909
[24] Severo, V., Ferreira, F., Spencer, R., Nascimento, A., & Madeiro, F. (2024). On the initialization of swarm intelligence algorithms for vector quantization codebook design. Sensors, 24(8), Article 2606. https://doi.org/10.3390/s24082606
[25] Fesenko, A. (2024). Cryptanalysis of Strumok cipher initialization. Tatra Mountains Mathematical Publications. https://doi.org/10.2478/tmmp-2024-0009