VACUUM: An Efficient and Assured Deletion Scheme for User Sensitive Data on Mobile Devices

Li Yang, Cheng Li, Teng Wei, Fengwei Zhang, Jianfeng Ma, and Naixue Xiong

***Abstract*—Embedded devices (eg, mobile phones, smart watches, etc.) store a large amount of sensitive information. However, Android-based devices may leak a lot of user information if unsafe data deletion. Therefore, research on secure data deletion for embedded devices has become a practical and urgent issue. In this paper, we study the logic structure, operation characteristics, and data management mechanisms of flash memory. Then, we propose a novel method VACUUM that uses a user-space file system and can provide fine-grained file deletion guarantees. Our approach encrypts files on an insecure medium with a unique key that can later be discarded to cryptographically render the data irrecoverable. Additionally, we use TrustZone as a secure key vault, and a garbage collection mechanism is introduced to purge the memory. Finally, we carried out experiments on the Android system, and the results showed that the solution is efficient and can meet the needs of real applications.**

***Index Terms*—Mobile Devices, Secure Deletion, Flash Memory, User Space, Encryption**

I. INTRODUCTION

WITH the widespread use of smart homes, smart healthcare, wearable devices, etc., a large number of embedded devices (eg, mobile phones, smart watches, smart TV, etc.) collect various privacy data of users, including health data, location information and even biological information [1], [2]. Currently, these embedded devices are mostly composed of single or multiple sensors devices. Which is impossible to use devices such as mechanical hard drives for data storage, so we can only choose a flash storage device with a small size and low power consumption [3]. When the local storage space is insufficient, part of the data will also be stored in the cloud. There are many researches on the secure storage and deletion of data in the cloud [4], [5]. That still many security issues with Flash storage [6]. At the same time, many embedded devices use Android system. Unfortunately, Android may face security issues such as information leakage. The main reason is due to the insecure data management method used by the Android system and applications. Previous studies [7]–[9] have discussed different aspects of failure in data security operations that have led to revealing users' private information. However, there is little discussion about the threat of residual data after deletion because Android does not explicitly state how to process data stored on devices. With the resale, loss, or theft of a mobile smart device, the insecure deletion of data may reveal a large amount of user private information. Previous work [10] has confirmed that the data stored in the Android device (including private personal information) can still be recovered after deletion, and the data will persist for a quite a long time.

Many approaches have been proposed to make data deletion more secure, such as methods based on file overwriting and encryption. Meanwhile, it is practically significant to apply theoretical research results to the protection of personal privacy data in real life. Previous studies [11]–[15] have analyzed the characteristics of user data stored in mobile devices and proposed their own data secure deletion mechanisms. However, these secure deletion mechanisms are not targeted and are short on savings and applicability; that cannot be met the user's requirements for secure deletion of private data, existing private data secure deletion (management) solutions have difficulty achieving full lifecycle data protection capabilities. Data are born from data generation and extended to data conversion, calculation, transmission, storage, and deletion. Failures at any stage can lead to an unexpected privacy leak. To avoid this issue, user data protection should implement protection mechanisms throughout the whole lifecycle.

A. Our Contributions

This paper will first explain the research problems such as user privacy data usage, secure deletion, and data unrecoverable in Android devices. And introduced the data storage method of Android devices. Then, we present VACUUM, a novel secure deletion scheme based on the user space file system FUSE [16] framework. Additionally, we design a secure user space encrypted file system.The problems of single application scenarios, complicated implementations, and difficulty in porting in the existing data security deletion research are solved. In addition, an adversary cannot recover and crack the residual ciphertext. At the same time, the system effectively reduces the loss of the storage device and improves the erasing efficiency. In this work, we study the method of mobile device storage security deletion from the viewpoint of storage architecture. We then design and implement a mobile

L. Yang and C. Li are with the School of Computer Science and Technology, Xidian University, Xi'an, 710071 China. E-mail: yangli@xidian.edu.cn, 159licheng@gmail.com.

F. Zhang is with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China. E-mail:zhangfw@sustech.edu.cn.

T. Wei and J. Ma are with the School of Cyber Engineering, Xidian University, Xi'an, 710071 China. E-mail:okweiteng@gmail.com, jfma@mail.xidian.edu.cn.

N. Xiong is with the Department of Mathematics and Computer Science, Northeastern State University, OK, 74464 USA. E-mail:xiongnaixue@gmail.com, xiong31@nsuok.edu.

Manuscript received May 26, 2020; revised March 15, 2021.

secure deletion file system in the user space on the Android platform. To demonstrate the effectiveness of our proposed approach, we implement a prototype of VACUUM on a real Android mobile device.

The main contributions of this paper are therefore fourfold as follows.

- We initially propose an approach named VACUUM that uses the user space file system to provide secure deletion that guarantees file-level granularity independent of the characteristics of the underlying file system and storage medium. VACUUM maintains file system semantics while having user-level development convenience.
- The approach encrypts every file on an insecure medium with a unique key that can later be discarded to cryptographically render the data irrecoverable. We use TrustZone as a secure key vault to ensure the security management of the keys. In terms of data storage, a data integrity verification is designed to improve system security.
- Efficient Memory Purging (EMP) : a garbage collection mechanism is introduced to purge the memory reclaiming the discarded flash pages. It mainly focuses on improving the security and efficiency of logical padding. At the same time, the EMP mechanism can minimize the loss of the Flash device and prolong its service life.
- We implement the prototype system, VACUUM, on an Android mobile device. Experiments are conducted, and the results indicate that VACUUM prototype ensures the secure deletion of data in flash memory on mobile devices with comparable overhead. In addition, the system meets the user requirements for daily use.

The highlights of this paper are as follows: while providing secure storage for files on mobile devices, VACUUM effectively improves the deletion efficiency and security without destroying the original data permissions of the system, thus achieving fine-grained deterministic deletion such that the deleted data cannot be recovered.

B. Organization

The rest of this paper is organized as follows: Section II introduces the background information and our threat model. The design of VACUUM on the Android platform is described in Section III. In Section IV, we implement VACUUM and present performance results under different file system operations. Section V offers related research on secure deletion, and Section VI concludes the paper.

II. PRELIMINARIES

In this section, we first give some background and then present our adversarial model.

A. Background

1) Flash Memory: Flash memory is an electronic (solid-state) nonvolatile computer storage medium that can be electrically erased and reprogrammed. It is widely used in embedded devices, especially mobile phones, smart bracelets and other devices. The most prominent characteristic of flash memory is that prewritten data can only be dynamically updated via a time-consuming erase operation. Furthermore, every block in flash memory has a limited program/erase cycle, which is typically 10^4 to 10^5 times. Since the size of the file read and written by a block file system is inconsistent with the size of the flash page, there are multiple copies of the modified contents such that, even if the file is deleted, the discarded flash page still has the remaining file information. The flash characteristic of out-of-place updates is the main reason for the data post-removal disclosure.

2) Flash Translation Layer: To conceal the characteristics of NAND flash, a special purpose firmware called Flash Translation Layer (FTL) is implemented inside flash-based devices. FTL allows external computing components (e.g., file systems) to access flash memory using a block-based interface. Most flash-based devices are equipped with FTL including USB sticks, SD cards, EMMC cards, and SSDs. In embedded devices, FTL is a hardware implementation and is software in the raw flash memory. In general, an FTL should at least provide the following functionalities: address translation, garbage collection, and wear leveling.

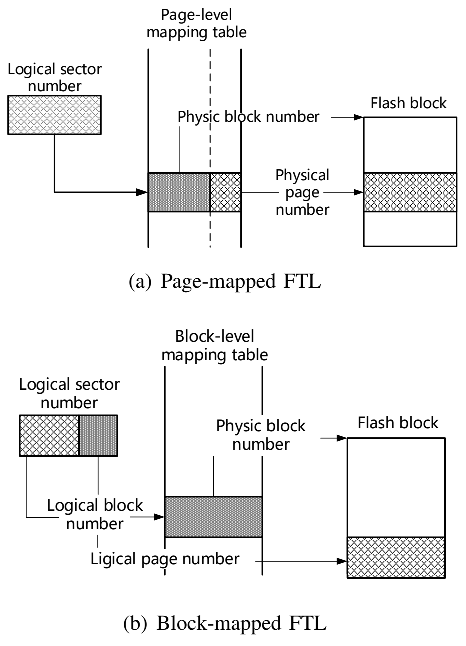

Fig. 1. The address translation in FTL

Address translation is the most basic function provided by FTL. As shown in figure1, the physical address corresponding to the logical address is determined by the FTL mapping algorithm. The main function of FTL is to convert the logical address of the file system to the physical address of flash. The management method of flash storage draws on the principles of the log-based file system. The FTL handles the read, write, and erase operations of the flash memory, and at the same time, receives the read and write requests and maps the logical address to the physical address of the memory. Since the number of erasures (Program/Eraser, P/E) of each data block is limited, until they wear out and become unusable. When an overwrite operation is issued, FTL redirects the physical address to the available space, thereby avoiding the use of P/E operations. After the data is updated, FTL changes the address mapping information. Outdated blocks can be deleted later. In addition to basic address translation, FTL also needs to consider performance and durability enhancement issues. Performance enhancement refers to the possibility of reducing the number of read, write, and erase operations. Among these three operations, reducing the number of erase operations is the most critical issue because the cost of erase operations is very high compared to read and write operations. Durability enhancement refers to erasing each physical block as uniformly as possible without degrading performance. If a block is higher than the program / erase cycle limit, the block may not work properly, resulting in data loss.

3) EncFs: EncFs (Encrypted Filesystem) is a FUSE-based cryptographic filesystem that transparently encrypts files using an arbitrary directory as storage for the encrypted files. FUSE provides a framework for implementing the user space file system. EncFs uses the libfuse dynamic library and the fuse kernel module to implement a full-featured user space file system without any kernel privileges. Two directories are involved in mounting an EncFs filesystem: the source directory and the mountpoint. Each file in the mountpoint has a corresponding specific file in the source directory. The file in the mountpoint provides the unencrypted view of the file in the source directory. Filenames are encrypted in the source directory. Files are encrypted using a volume key, which is stored encrypted in the source directory. A password is used to decrypt this key.

B. Threat Model

Attack Time. Attack time refers to the time the adversary gains access to the device. There are two cases: (1) the attack time is controlled by the user, and the user can delete as much data as possible before providing the device to the adversary; (2) The adversary obtains permission to use the device without the user's knowledge. In case (2), the safe deletion operation is not dependent on the user to complete it, but is completed by the previously performed clear operation. This paper assumes that the time when the adversary deleted the data is unknown.

Number of Accesses. Access can be divided into single access and multiple access. Single access scenario, such as device loan. Multiple access scenarios, such as the target device being deployed by malware multiple times. This article assumes that the adversary has multiple access capabilities.

Certificate disclosure. This scenario is defined as the adversary can decrypt the data without getting the key.Certificate disclosure is divided into non-mandatory and mandatory. Non-mandatory adversaries do not obtain user passwords and credentials used to protect data on physical media, but mandatory adversaries can obtain this information. The adversary can also obtain the user's password by guessing the user's weak password, and the device is not locked [17]. This article assumes that the adversary has the ability to force an attack.

Computational Constraints. Many secure deletion methods rely on encrypting data and storing only their ciphertext. By deleting the decryption key [18]–[20], data is guaranteed to be unrecoverable. Therefore, encryption is used as a compression technique to reduce the cost of secure deletion, because only small keys need to be deleted. In this article, the ability to impersonate an opponent is limited, and the encrypted data cannot be violently broken, thereby ensuring that the encryption system is not destroyed.

To prevent adversary from obtaining information in the mobile device, this paper develops a secure deletion method for mobile devices by maximizing the attack capability of the adversary.

For the adversary's ability to assess safe deletion methods, the adversary's goal is to recover deleted data objects. This paper will analyze data deletion threats and characterize the types of hostile capabilities in these situations to develop confrontation models. The threat model defined in this paper is as follows.

After the sensitive data are calculated, the adversary can perform an offline attack and read the physical storage medium. This assumption covers many real scenarios: people with bad intentions break into a hotel room to obtain snapshots of the target device (e.g., a laptop's SSD, USB, smart phone SD card), mobile devices might be used in certain oppressed occasions and be confiscated resulting in device intrusion, and so on.

III. SYSTEM DESIGN

In this section, we design a secure data storage and deletion system for Android devices ensuring that the file storage and deletion operations are secure. We first describe our expected goals for the scheme and then detail the system design process.

A. Goals

The proposed scheme should meet the following requirements and have achieved data confidentiality.

Confidentiality. Prevent unauthorized users from accessing, protect user data only for authorized users, and resist illegal access by competent adversaries.

Integrity. Data cannot be changed without authorization, requiring securely stored data to resist rival spoofing, splicing, and reply attacks to maintain the integrity of the device data.

Fine-grainedness. The files are required to be securely stored and deleted regardless of file size. Ensure that the data of the mobile device can be completely deleted from the device in a fine-grained way instead of resetting, which is the method that came with the device.

Efficiency. The data deletion operation must be efficient and take into account the two efficiency indicators of the loss of the flash device and the deletion delay.

Non-root authority. The implementation of the safe deletion scheme cannot obtain the device root authority and prevent the destruction of Android's own sandbox data protection mechanism.

Flexibility. The application of the solution does not depend on the specific file system and storage media characteristics.

Stability. The safe deletion scheme should be easy to develop without modifying the kernel code of the system, and it cannot cause instability in the device's native system.

B. Architecture

In this paper, we designed a Secure Data Deletion in User Space system, called VACUUM, based on the user space file system.

The system uses the FUSE framework to run in the user space of the Android system while providing file system semantics and security deletion guarantees at the file granularity level to ensure that deleted data are not recoverable. VACUUM does not require modifications to the Android file system. It can be developed and deployed as a standalone module that can run on any Android distribution and any file system.

VACUUM provides users with an enhanced user-level encrypted file system for secure deletion. Users can create encrypted file systems without root permissions. The file system can only be accessed by the user who created it. Other users can only see the ciphertext encrypted in the directory and file, which effectively protects data security. In performing a secure deletion operation, we only need to discard the key and the logically deleted encrypted file, so we achieve high execution efficiency. Now, we present our secure deletion solution and show how it fulfills the listed requirements.

Figure 2 shows the general system data storage architecture, which is divided into application layer, file system layer, page caching layer, and device driver layer.

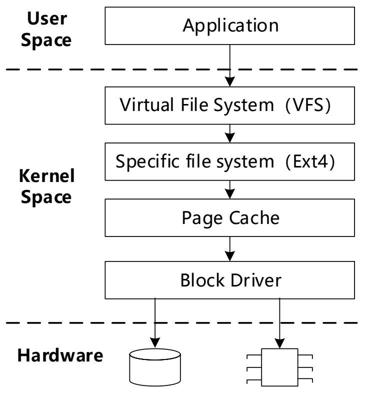

Fig. 2. The process of data storage

Among these layers, the file system layer is the most suitable location for the safe deletion of files. By definition, the file system already knows the data blocks for any file and also has full control over the file metadata, all of which significantly simplify the development process. However, modifying and maintaining each specific file system will be error-prone, which is likely to lead to instability in the original system. Moreover, the scheme for the property file system is not scalable, and the scheme is more adaptable. Despite the abovementioned problems, the file system layer is still the most suitable location for file deletion. Based on the file system under the FUSE framework, user-level development can be well achieved. This not only ensures that the underlying architecture of the file system is not destroyed, but also benefits the application system development. Furthermore, the file-safe deletion scheme developed based on the FUSE framework can run under any file system and is compatible with any physical block device.

Generate a random encryption key for each file, this method can ensure the security of the data, and it is relatively easy to implement the system. When data is written to a storage device, it should first be encrypted; before reading the file, it should also be decrypted. Then, the safe deletion of the file is converted into the safe deletion of the corresponding key file. Compared with the data file, the key file is smaller and the operation is simpler. For adversaries with limited computing power, data cannot be recovered without a key.

This solution uses the FUSE framework of the user space file system, proposes a secure deletion method based on encryption technology, designs a secure deletion and enhanced user space encryption file system, and solves problems such as a single application scenario, complex implementation, and difficulty in porting the existing data security deletion. The corresponding key file is safely deleted when the data are deleted so that the adversary cannot decrypt the residual ciphertext information. At the same time, by deleting the key method, the deletion space is reduced, the security deletion time is shortened, and the loss of the flash device is reduced.

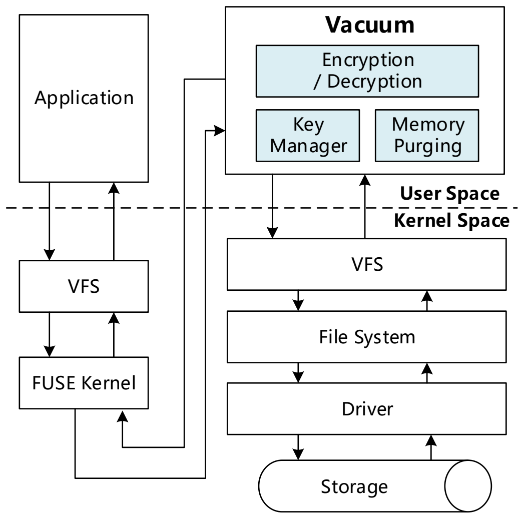

Fig. 3. Abstraction of VACUUM architecture

Figure 3 shows the file system architecture of VACUUM. VACUUM uses the FUSE framework to implement a virtual layer that is stacked on the file system and runs in the user space. VACUUM can be encrypted for a directory in a specific file system without having to encrypt the entire actual file system. The system includes the following modules: user space file system, encryption and decryption module, key management module, data integrity verification module, and memory cleaning module.

Next we will introduce the three main modules of VACUUM: 1) FUSE-based data encryption and storage module. The encryption scheme is implemented based on the FUSE framework. It masks the specific file system and storage media characteristics and is implemented in the user space; thus,

2327-4662 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Southern University of Science and Technology. Downloaded on January 06,2022 at 06:14:07 UTC from IEEE Xplore. Restrictions apply.

it meets the design requirements in terms of flexibility and stability. Data are based on the granularity of the file to perform encryption and decryption operations to meet the fine-grained requirements. Encrypted data can prevent unauthorized users from accessing it, which meets the confidentiality requirements. 2) Based on the TrustZone key management module, the key management can meet the data confidentiality requirements. The safe deletion of execution file granularities meets the fine-grained requirements. Moreover, the deletion of keys meets the efficiency requirements. 3) The efficient memory purging module based on file overwriting can thoroughly clean the discarded ciphertext data and discarded keys in the flash memory of the mobile device, which can make the data irretrievable.

For non-root privilege requirements in the design goals, it is required that VACUUM development does not require root privilege to implement all modules and system functions of VACUUM. Users do not need additional root authority to use VACUUM. The FUSE-based VACUUM allows each user to create an encrypted file system for himself or herself regardless of whether he or she is a root user. VACUUM uses any directory as the mount point of the file system and can be hung in the root directory of the Android file system, thus achieving the effect of encryption like full-disk data. The root directory of the Android file system is designed to be read-only by the native system. If VACUUM is mounted in the root directory, the device must obtain root privileges. However, after obtaining root authority, the sandbox data protection mechanism of the Android system is destroyed, and all system resources are exposed to the adversary, which can lead to other more dangerous consequences. Therefore, users are allowed to select the VACUUM mount point independently to meet the non-root authority requirements in the design goal along the two dimensions of development and usability.

1) FUSE-based data encryption and storage: The VACUUM file system is a user-space application with file system semantics. It provides document granularity and file encryption functions that are transparent to users. Users can choose which files to encrypt. Data confidentiality is ensured through data access rights and data encryption. Data encryption is mainly achieved through encryption algorithms and encryption modules. Access control is mainly implemented through passwords for user authentication so that it can prevent other users from accessing sensitive data or other overriding operations.

Data encryption process. VACUUM uses AES encryption algorithm, CBC mode. As a symmetric encryption technology, AES has fast encryption speed and high algorithm security. Its CBC mode is suitable for text transmission. VACUUM encrypts file contents and file names by absolute file paths. Figure 4 is the data encryption process. FEK_i is a file encryption key, generated by the key management module, IV is an initialization vector which will take the message block authentication code (MAC) value, m is the data plaintext and c is the data ciphertext. In the encrypted file, the data is first divided into n blocks of equal size, namely $data = \{m_1, m_2, \cdots, m_n\}$, and then the selected plaintext is encrypted by FEK_i.

The file access process of VACUUM. Figure 5 shows the VACUUM file system access process. The specific process of

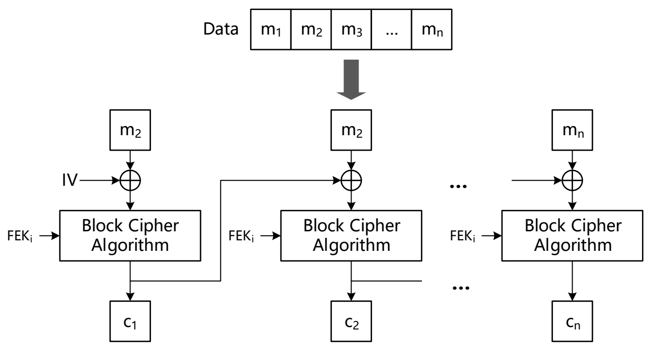

Fig. 4. Data encryption process

the program file write operation is as follows: When the file write instruction reaches the file system layer, the VACUUM system key management module generates a file ID according to the absolute path of the file, i.e., ID = Generate (FileAbsolutePath), and generates a corresponding ID according to the ID. A FEK (file encryption key) and IV (initialization vector), i.e., (FEK, IV) = Generate (ID), are generated such that each file has a unique IV stored with the keys in the FEK since encryption is performed page by page.

In addition, the pages of the file can be read or written in any order so that the page IV is derived from the file's unique IV and the file offset of the processed page. The MK (MasterKey) is used to encrypt the FEK and IV, and the encrypted result is stored in the MasterKeyStore to ensure the security of the file key, that is, FEK' = E_{MK}(FEK,IV); then, the encryption process shown in Figure 6 is followed.

The encryption process uses FEK to encrypt the contents of the file, namely, c = EFE_K (m), and stores it on the device storage medium.

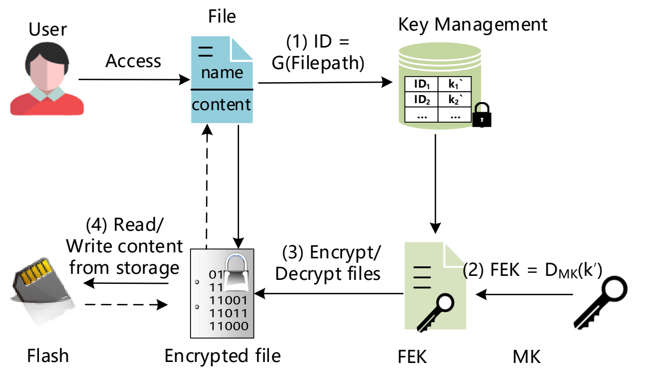

Fig. 5. The access to file process of VACUUM

The specific process of implementing the file read operation of the solution is as follows: When the file read command reaches the file system layer, the VACUUM system key management module generates a file ID according to the absolute path of the file, that is, ID = Generate (FileAbsolutePath), and the key management module according to the ID in the KeyStore finds the encrypted FEK', that is, FEK' = Select

(ID); then, MK is used to decrypt FEK' and retrieve FEK, that is, FEK = D_{MK} (FEK'), and the encryption management module decrypts the file ciphertext using FEK, that is, m = D_{FEK}(c). VACUUM then passes the decrypted plaintext content m back to the application.

2) TrustZone-based key management: Key management consists of the generation, storage and deletion of keys. To encrypt and decrypt the file data on mobile devices, the VACUUM's key management module needs to generate encryption keys for encrypting data. The secure storage of keys can resist illegal access from adversaries. A unique encryption key for each file is generated by VACUUM, which implements encryption at the file granularity level, stores multiple keys in the encrypted files and equates the security of the original file to the protection of the key. File fine-grained encryption is the foundation of file fine-grained security deletion. A unique key is generated when the file is being stored. After the deletion of files, the encryption keys will be deleted securely, which ensures that the file cannot be restored.

Key generation. The limitations of the traditional solution are as follows: A straightforward method of using cryptographically based erasure to guarantee file deletion is to simply generate a random encryption key for each file, after which the secure deletion of the file is translated into safe deletion of the corresponding file key. As shown in Figure 6, a single FSEK (file system encryption key) is used to encrypt all files, and the encrypted EFs (encrypted files) are stored in the storage medium, e.g., FDE (full disk encryption) in Android systems.

However, this method has a fatal flaw: There is no way to ensure that the file key is safely deleted. Every time a single file is deleted, the main key needs to be changed. Then, the whole file system is re-encrypted, which is inefficient, expensive, and inflexible. The file key must be saved to the physical storage medium when the system is rebooted or the system is down, which makes it possible for adversaries to recover the file key and decrypt the data.

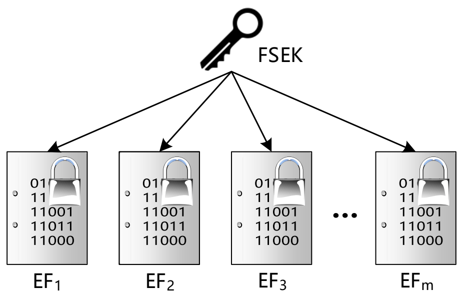

Fig. 6. Single key data encryption method

Therefore, we must use another main key to encrypt the other keys. This is a recursive procedure and leads to the second problem: the simple two-tier model mentioned above means that deleting a single file requires re-encrypting all the file keys. As modern storage devices suffer from phenomena such as flash wear, data may be permanently stored in multiple physical locations, and such processes are completely out of the control of the operating system. Therefore, to ensure that the file data are not recoverable, the main key must be replaced, and the old key must be safely deleted from the KeyStore, which ensures that it is impossible for an adversary to recover the cipher text from the physical storage medium by brute force using computational methods. Since the MasterKey must be replaced, all file keys must be re-encrypted before they are saved to disk, which is called encryption amplification. This is a time-consuming operation that should be avoided for any available system.

Improved key generation strategy: To solve the above problem, the key generation process in the VACUUM encryption mechanism is as follows, as shown in Figure 7. (a) Derive a MK using PBE (password based encrypion). (b) Derive a FSEK from the MK using the KDF (key derivation function). (c) Use an ordinary key derivation function (such as PRNG) to generate a FEK from the absolute path of the file. (d) Use the FEK to encrypt the file content and file names, and use the FSEK and random initialization vector IV to encrypt the FEK. (e) Then, use the mapping mechanism and find the encrypted file through the FEK and IV $(FEK||IV \rightarrow ID)$.

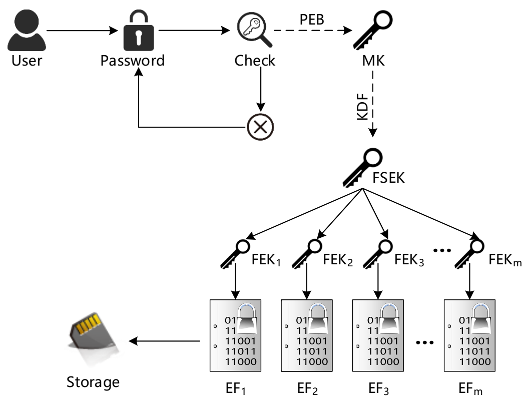

Fig. 7. Key generation process of VACUUM

Each time VACUUM is mounted, the user enters the password (Password), and the PBE generates MK, that is, MK = PBE (Password). The MK is stored in the MasterKeyStore and is protected by TrustZone and cannot be accessed. Then, FSEK is derived from MK. Each time FSEK is calculated by MK, that is, FSEK = KDF (MK), the FEK corresponding to the FSEK encrypted file, that is, FEK' =E_{FSEK} (FEK), and FEK is stored in encrypted form. In the device, the FEK encrypts the file and stores the ciphertext in the device.

Key storage. The VACUUM key management mechanism must rely on trusted components as the secure storage area for the MasterKey. This MasterKey secure storage area, called the MasterKeyStore, must satisfy the following three conditions: The key vault must (i) be large enough to store the MasterKey; (ii) allow the system to use the stored MasterKey and perform encryption and decryption operations; and (iii) allow the system to perform update operations replacing old keys with new ones.

The VACUUM stores the MasterKey in the NVRAM of the TrustZone chip. When the MasterKey needs to be regenerated after the file reading process or when the user changes the password, the VACUUM can reliably discard the discarded key and provide a strong defense against unauthorized retrieval of the MasterKey.

The MasterKey is stored and managed by the KeyMaster in the secure world. In key management, access to the MasterKey is required to provide security assurance. When the VACUUM acquires the MasterKey, it needs to request the KeyChain Service from the Android Framework layer. Finally, the request is converted into a TrustZone request through the KeyStore and then communicated with the KeyMaster TA process of the secure world. Masterkey reads are done in a secure world space.

VACUUM chooses to use the TrustZone chip as the MasterKeyStore. At present, the processor for Android mobile devices provides TrustZone technology. The rapid application of security technology has further accelerated the deployment of TrustZone applications, such as digital rights management and electronic payments. When modifying the hardware configuration is not feasible, one can use a medium with similar secure storage attributes, such as the use of cloud storage features, to achieve the key and cipher text isolation storage, increase the difficulty of acquiring the enemy to achieve the purpose of dispersing the risk. Therefore, VACUUM's MasterKeyStore requirements can be easily implemented in many cases.

Key deletion. When the delete operation is invoked, the VFS forwards the file deletion function to the FUSE kernel driver, the FUSE kernel routes the file deletion operation calls to VACUUM, and the VACUUM calls the key management module to delete the FEK corresponding to the file.

The specific flow is as follows: 1) First, VACUUM calls the encryption module and generates a file ID based on the absolute path of the file, i.e., ID = Generate (File Absolute Path). 2) The key management module queries the FEK in the key storage area according to the file ID, i.e., FEK = select (ID). 3) The key management module first overwrites the file key FEK with the preset mode (0x00) and then deletes the FEK. 4) The key management module disconnects the key access area where the FEK is located and copies the remaining keys to the new one. The abovementioned FEK is encrypted and stored by the MK. The security of the MK is protected by the TrustZone security mechanism. In the above steps, the content of the key storage area is cut off as ciphertext, and an adversary with limited computing capability cannot crack the file key without a key. Then, it is impossible to crack the discarded ciphertext, and safe deletion of the file is achieved.

VACUUM can implement the encryption and decryption operations of the file granularity such that the safe deletion of a file requires only one erase operation. In addition to the erase operation, the VACUUM will overwrite the deleted block. Figure 8 shows the file deletion process. To safely delete the encryption key, the VACUUM method follows the following steps: Search for FEK_i with deleted file i. Use 0x00 as a mode to overwrite pages stored by the deleted FEK_i. The "1" bit stored in a specific valid page is converted to a "0" bit. Check if there is a valid page in the block. If there is a valid page containing FEK, the block will not be deleted, and the deletion process will end. If there is no valid page, the flash is forced to trigger the GC to erase the block.

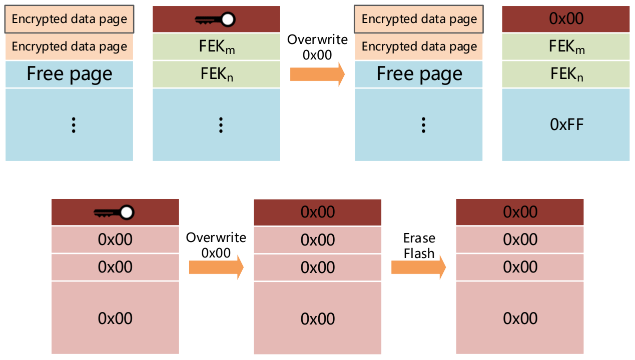

Fig. 8. The file key deletion process of the VACUUM

3) Data integrity verification based on Merkle-Tree: The integrity of the data is very important, so this article will add an integrity check module to the VACUUM scheme. Merkle-Tree [21] can verify the integrity of the spatiotemporal relationship of memory divided into separate blocks. Each leaf node of Merkle-Tree represents a complete identifier, such as a hash or an identity authentication identifier, which is used to index the specified memory block. Each internal node of Merkle-Tree corresponds to the hash of its child nodes. Therefore, the root hash of the Merkle-tree can represent the integrity of the memory.

Figure 9 shows the process of encrypting a plaintext block (PlainBlock) into a ciphertext block (CipherBlock). In the data encryption process, first use the HMAC function to calculate the message authentication code MAC for each file data block, then each file data block corresponds to a MAC value, and finally manage the generated MAC value through Merkle-Tree.

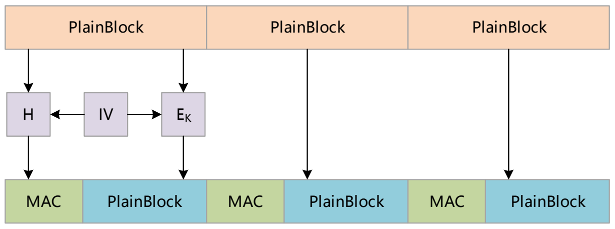

Fig. 9. Encrypted file memory layout

Merkle-Tree is a complete binary tree, and each node corresponds to an identity authentication data. The purpose of our Merkle-Tree is to prevent replay attacks. Each node in the binary tree has a corresponding hash value. The authentication path of the leaf node refers to all sibling nodes from the leaf node to the root node. Using Merkle-Tree directly to protect the authentication label of each data block will cause a lot of memory consumption. To reduce the need for memory, we store them in management blocks. After that, the management block will be stored in permanent storage, not in the cache. The hash value of the root node of Merkle-Tree can represent data integrity. VACUUM calculates its MAC value after reading all management blocks, and reconstructs Merkle-Tree. Then the hash value of the Merkle-tree root node is compared with the reference value specified by the user. If they are equal, the VACUUM operation was successful, otherwise the data integrity has been destroyed. Figure 10 shows the process of building a Merkle-Tree.

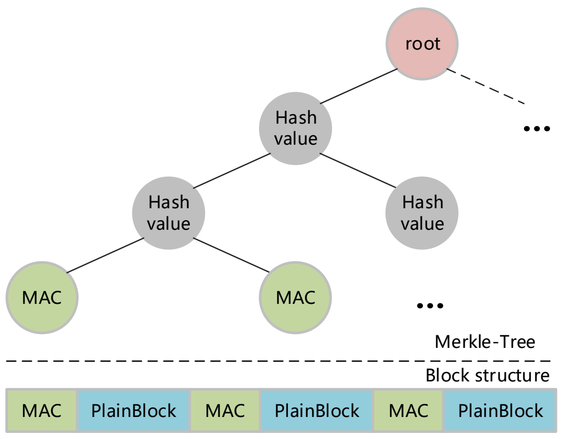

Fig. 10. The build process of Merkle-Tree

Every time a leaf node is inserted, the tree is updated, where the root value represents the integrity of the entire memory in the space-time relationship. The last data block is written to memory.

4) Efficient memory purging algorithm based on file overwrite: The VACUUM uses any catalogs as the file system's mounting point and can be mounted on the Android file system's root catalog to accomplish similar overall data encryption. The Android file system's root directory is a native system designed for read-only operations. If mounting VACUUM on the root directory, the device must obtain root permissions. Subsequently, all system resources are exposed to the enemy. Therefore, the user selects the VACUUM mount point independently, thus reaching the design goal along the two dimensions of development and usability. According to the storage location of the deleted files, VACUUM deals with two situations: (I) Files to be deleted are stored in the VACUUM file system; and (ii) Files are stored outside VACUUM and deleted by the Android file system logically.

For the files stored in the file system of VACUUM, performing safe delete operations is relatively simple. Invoking the FEK corresponding to the file discarded by the VACUUM key management module can ensure that the deleted data cannot be recovered. For files stored outside the VACUUM, the operating system logically deletes many files but leaves many sensitive data in memory. The conventional solution for clearing memory units occupied by these files is invalid, and from the user's point of view, it is impossible to know where the data to be erased is written. This situation happens, for example, when an application creates a temporary file and deletes it manually. The temporary file may be a decrypted copy of the encrypted file saved by the encrypted file system to allow other applications to process specific file types such as images, documents, and spreadsheets. Whether one imports temporary files or does not import encrypted file systems, they must all be safely removed.

The general safe deletion solution of traditional file overwrite has the following limitations: First, it takes time proportional to the free space to be cleaned and the speed of the memory writes. Second, if one uses calls frequently, it is possible to shorten the life of the flash memory. Therefore, to reduce the negative impact on flash, it should minimize the implementation of MP operations for a single file to be safely deleted.

In view of the efficiency of the above method in terms of implementation, this paper proposes an Efficient Memory Purging (EMP) algorithm based on file overwriting. It focuses on improving the security and efficiency of logical padding. The specific implementation is as follows:

Algorithm 1 Efficient memory purging algorithm based on file overwriting (EMP)

```
Require:
    Files to be safely deleted (TargetFiles).
    File overwrite times (threshold).
Ensure:
    Remove the invalid physical page/block making the obsolete data unrecoverable;
    Boolean IsSuccess = EMP(TargetFiles, threshold).
 1: Initialize OverwriteTimes = 0. Record the current number of overwrites.
 2: Compute the available size of flash space (FreeSize).
 3: Accelerate the creation of temporary file (TempFile) through fallocate, truncate and other functions. The file size is FreeSize, and the file content is empty.
 4: while TargetFiles is exist do
 5:     Remove information related to TargetFiles.
 6:     Remove TargetFiles.
 7: end while
 8: while OverwriteTimes ¡ threshold do
 9:     Create a garbage fill file (GarbageFile). The content is empty and the size is 0.
10:     Recalculate all free space in flash memory.
11:     while GarbageFile.size() ¡ FreeSize do
12:         Write(GarbageFile, random data), Fill random data in GarbageFile.
13:     end while
14:     Ensure that internal buffer data are synchronized to disk files.
15:     Remove GarbageFile.
16:     Add one to OverwriteTime.
17: end while
18: Remove TempFile.
19: System.gc() forces flash controller to electrically erase discarded pages.
20: Return(IsSuccess).
```

Variable TempFile represents a temporary file that occupies all available space, TargetFiles are files that the user needs to safely delete, and GarbageFile is a file that is used to overwrite the remaining free space after the target file is deleted. EMP improves the efficiency of safe deletion by reducing the available space.

The EMP method solves this problem by using a new function for the system, $fallocate()$, which speeds up the creation of garbage files (TempFile). When this function is called, it can preallocate free space to any file. Many file systems support this function such as ext4, btrfs, and xfs. Because there are no data to write to this file, it is a very fast method. EMP can quickly use all available space by calling this function.

After all available free space is occupied by TempFile, the system will remove the information related to the target file, such as the file name and the file creation time. Then, the system will remove the target file that needs to be deleted and release the logical memory space that it occupies. For security reasons, the EMP algorithm will circularly remove the target file and its associated information until it is completely removed. Next, by creating a garbage file (GarbageFile), the EMP algorithm will overwrite the free space released by the target file with a random number and then continue to write random numbers to the file until the free space size is zero. The EMP method also calls the $Sync()$ system command to ensure that all random numbers are written to the physical disk instead of being written to the internal buffer. Overwriting will force the flash controller to reclaim the space occupied by all deleted target files, and the data retention of all possible logical and physical target files will be permanently erased. The number of file overwrites (Threshold) is specified by the user. After the free space overwrite operation is completed, the EMP algorithm will remove GarbageFile and TempFile, release the memory, and end the safe delete process.

IV. IMPLEMENTATION AND ANALYSIS

This section mainly proves the research goals proposed in this paper. First, conduct a security analysis on the attack hypothesis proposed in this paper; then, discuss the deployment and performance of the scheme, and draw relevant conclusions through the analysis of the result data.

A. Security Analysis

1) Online attacks: **Spoofing attack.** We assume that the adversary can pass any data as a valid data block. The VACUUM generates a verification tag and verifies it after performing decryption. If the adversary modifies the data block or label, the authentication operation will fail.

Splicing attack. We assume that the adversary can replace the current valid block with a valid block different from the same storage medium. That is, the space of the data block is replaced. This article adds the block index number as the associated data to the VACUUM encryption operation to prevent splicing attacks.

Reply attack. We assume that the adversary can successfully replace a previously recorded copy of a particular data block and compare it with the latest version of the data block. That is, the time shift of the data block. VACUUM uses Merkle-Tree to prevent replay attacks. Merkle-Tree effectively maps the integrity of each individual block and its spatiotemporal relationship to the Merkle-tree root node hash value. If an adversary has an old data block in the read operation, root hash verification will fail.

2) Offline attacks: We assume that the adversary can read the physical storage medium and obtain its mirror copy. Next, we separately analyze a series of encryption operations and deletion operations to show that VACUUM can resist offline attacks.

In VACUUM, when an application issues a write instruction, it generates a corresponding encryption key FEK according to the absolute path of the file, and uses FEK to encrypt the file. Then use MK to encrypt FEK and generate FEK'. Finally, FEK'and ciphertext are permanently stored. Since MK is stored in TrustZone, the security of MK can be guaranteed through the TrustZone mechanism. Even if the adversary obtains a copy of the ciphertext and the file key ciphertext FEK', it cannot obtain any information.

After the VACUUM performs the delete operation, it will disconnect the relationship between the file and the corresponding encryption key FEK, and call the file system garbage collection module to overwrite the FEK memory page. Even if the adversary successfully recovers the deleted data, only the ciphertext can be obtained. Even if the user password is disclosed, the adversary cannot successfully decrypt the ciphertext. In addition, the EMP module will periodically clean up the memory. After the flash controller is triggered, the system will recover invalid data blocks, and then completely delete the expired key and other expired files in the flash memory.

3) Security goals: Next we will discuss how the system achieves six design goals of ”Confidentiality”, ”Integrity”, ”Fine-granularity”, ”Non-ROOT authority”, ”Flexibility” and ”Stability”.

- ”Confidentiality”: In the design of the scheme, an encryption-based deletion scheme is used. Each file is encrypted with the corresponding FEK before being stored, and decrypted when it is read. When the file is deleted, the easy-to-operate FEK is erased. For an adversary without a key, it is impossible to recover data in a limited time. The key is stored in the trusted module, which ensures the security of the key. Thereby achieving the confidentiality of the entire system and data.
- ”Integrity” and ”Fine granularity”: In this paragraph, we show that VACUUM can delete the sensitive data and remove it from the medium permanently, which ensures that the data cannot be recovered and therefore are securely deleted.

  We suppose that an adversary can read the physical storage medium and obtain its mirrored copy. In the following, we analyze a series of encryption and deletion operations to show that VACUUM can resist offline attacks.

  In VACUUM, when an application program issues a write instruction, it generates a corresponding encryption key FEK based on the absolute path of the file and encrypts the file with the FEK. Then, it generates FEK' by encrypting the FEK with MK. Finally, the FEK' and the ciphertext are persistently stored in the mobile device's storage medium. Because MK is stored in the TrustZone, the security of MK is guaranteed by the TrustZone mechanism. Even if an adversary obtains a ciphertext copy and the FEK', it cannot obtain any message. After VACUUM performs a delete operation, it disconnects the link between the file and the corresponding encryption key FEK and calls the file system garbage collection module to overwrite the FEK memory page. Even if an adversary successfully recovers deleted data, only the ciphertext is obtained. Even if the user's password is disclosed, an adversary cannot successfully decrypt the ciphertext. Moreover, the EMP module periodically cleans the memory, and the system reclaims invalid erase blocks after triggering the flash controller and then completely removes the expired keys and other expired files in the flash.

- ”Flexibility”: First, the system is designed at the application layer to ensure system portability. At the same time, key information such as keys can be deployed not only in the TrustZone Module, but also in Intel SGX, Trusted Network and Trusted Cloud modules.
- ”Non-ROOT authority” and ”Stability”:The system does not use root privileges, and the system is developed at the application layer. The system kernel and system files are not operated to ensure the security and stability of the original operating system.

4) Attacks on identity authentication: Identity authentication systems exist in all mobile devices. Once an attacker destroys the identity authentication system, it will pose a great threat to the privacy of the device. Therefore, the MK of this scheme is stored in TrustZone, which ensures the safety of the MK. In the user's file encryption and decryption system (Vacuum), the system is deployed in the user space to provide users with fine-grained file encryption services. Users can choose the files they want to encrypt. First, the user needs to set a password for encrypting or decrypting files. When a user needs to perform an operation, he must enter a password for verification, and the subsequent encryption and decryption operations can only be performed after system verification. In this process, the Vacuum and the device identity authentication system are independent of each other. When using Vacuum, only after passing the password verification, the user can see the complete user space encrypted file system in this system, and the file and file name are also decrypted and displayed in real time. After illegal users enter the system (regardless of whether they use their identity credentials to enter the system), they cannot view encrypted files in the system file directory. The file encryption key is an encrypted file, which is stored in the system. The attacker cannot decrypt the file encryption key file without the key. So as to ensure the security of the data when the device receives an illegal intrusion.

However, in the process of using the system, identity authentication is an essential part. In order to reduce the complexity of user management passwords, the support of the continuous identity authentication system is needed to better protect data privacy and security.

B. Implementation and Experimental Analysis

This paper implements the file system VACUUM based on EncFS . EncFS is a user-level encrypted file system [22]. Without privileged users, it can create its own encrypted file system. This system also encrypts and stores all files under the user directory. EncFS is a program that uses FUSE to provide a virtual file encryption system for Linux. Wang et al. [23] successfully ported EncFS to an Android system. As a transparent encrypted file system, EncFS exports file system interfaces to user mode and runs entirely in user mode. The file encryption system is actually a program in user space and is run and ended by the user. However, the existing EncFS encryption process is incompatible for the design goals of this paper. We modify EncFS's encryption module and key management process and add data validation and garbage collection modules.

The software development environment used is CPU: Intel core i7-6700, RAM: 16GB, operating system: Windows 7, development tools: Android Studio, and the development language is Java. The system test environment of the paper is Android 6.0 version, CPU Snapdragon 810, RAM 3GB, ROM 32GB, using OpenSSL encryption library to ensure the backward compatibility of the system. In this encryption system, the information related to the file is processed, and only the encrypted files and folders (file types, etc. are hidden) can be seen outside the system.

1) Performance Testing: The design and implementation choices of this paper are all designed to build a practically available system on a mobile device. The paper can transplant VACUUM to an Android phone and mount VACUUM to any directory. The test runs directly on the hardware device without virtualization. Next, we will discuss the efficiency of the system. First, we test the read/write performance of the VACUUM system and compare it with the Android system FDE [24]. Second, we test the VACUUM file encryption and decryption overhead. Finally, we verify the EMP performance of user space garbage collection.

Read/write performance. To test how VACUUM affects the I/O performance of the underlying storage device, we use the dd command to test the sequential file read/write performance of VACUUM. For all operations, we repeat the test on FDE. We also test benchmark results that do not run VACUUM and FDE. The results are shown in Table I. These results are the average of five runs. The maximum relative standard deviation of the test results is less than 2% in all cases. The operating system cache is disabled during the test to ensure that the measurement results are not affected by previous operations.

The results show that the throughput of the encrypted file systems is significantly lower than that of the unencrypted file systems. The read throughput of VACUUM and FDE files is almost similar, with approximately 5.84% of the overhead, which is within the user's acceptance range. However, write throughput of VACUUM has decreased slightly compared to FDE, and the overhead remains at 16.27%. The result of the overhead is expected because the VACUUM runs in user space, and frequent switching between the user state and the kernel state will introduce a large overhead. Once the

TABLE I
DISK I/O AND FILE SYSTEM PERFORMANCE OF VACUUM COMPARED TO ENCFS. BENCHMARK RESULTS ON AN UNENCRYPTED DEVICE ARE ALSO PRESENTED AS A BASELINE.

	No Encryption	FDE		VACUUM		
			Overhead vs.		Overhead vs.	
Operation	Performance	Performance	No Enc.	Performance	No Enc.	FDE
Read	110720 KB/s	93536 KB/s	15.52%	88074 KB/s	20.45%	**5.84%**
Write	43026 KB/s	26594 KB/s	38.19%	22267 KB/s	48.25%	**16.27%**

VACUUM has obtained the encryption key to process the file, the remaining task of encrypting and decrypting the file block is not much different from the FDE execution encryption. Table I shows that the VACUUM file system read and write performance can meet the daily operations of users. As equipment hardware is upgraded, computing power is increased, and performance overhead is negligible. Some devices with limited computing power can be optimized by adopting a caching strategy, especially when the write operation rate is slow. When the computing and storage resources of the device are idle, Vacuum starts the user's encryption operation, which will not affect the user's use of the device, but also make full use of system resources.

Encryption and decryption overhead. Due to the different file encoding formats, VACUUM tests its encryption and decryption overhead using five common file sizes (100 KB, 500 KB, 1 MB, 5 MB, 10 MB) and eight main file types (docx, mp3, jpg, gif, zip, txt, mp4, pdf). The experimental result is the average of 20 tests.

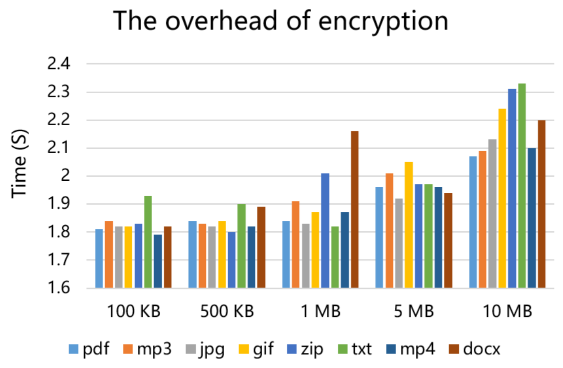

Fig. 11. The encryption overhead

Figure 11 shows the file encryption performance of VACUUM on an Android device. For 100 KB and 500 KB files, the decryption time is approximately 1.84 seconds. For a 10 MB file, the decryption time is approximately 2.18 seconds. VACUUM cryptographic file overhead consists of two parts: key generation and ciphertext storage.

Figure 12 shows the file decryption performance of VACUUM on an Android device. For 100 KB and 500 KB files, the decryption time is approximately 0.15 seconds. For a 10 MB file, the decryption time is approximately 0.69 seconds. When the file is small enough, VACUUM decryption overhead is mainly due to the reading of the file encryption key, FEK. As the size of the file increases, the decryption overhead is

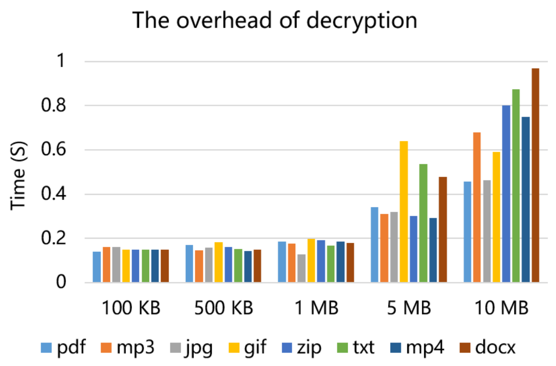

Fig. 12. The decryption overhead

mainly influenced by the data decryption processing, and the decryption overhead is within the acceptable range of the user.

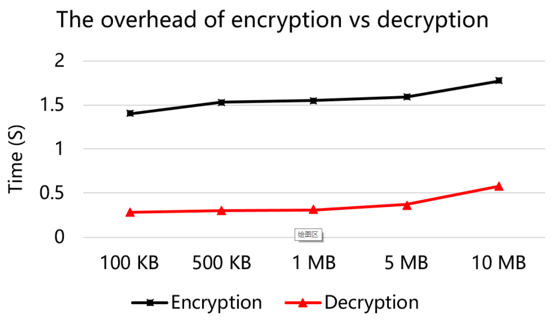

Fig. 13. Comparison of encryption and decryption overhead

Figure 13 compares the VACUUM's overhead for performing cryptographic operations on different sizes of the same format file. The decryption time overhead is less than the encryption time overhead because the encryption time cost involves the storage operation of the file encryption key (FEK) generation, storage, and the file's own data, and the decryption time overhead does not involve the write operation of the file data. For a small file, the overhead can be ignored by the user, even if the encryption and decryption overhead of a large file is within the acceptable range of the user.

EMP efficiency. To verify the user space garbage collection EMP performance, we measure the time overhead of EMP and Purging [25] based on storage usage in the following three scenarios: (i) Storage is mostly empty, i.e., usage is between 0 and 30%; (ii) Storage is moderately used, i.e., storage usage is 30% to 60%; and (iii) Storage has almost no free space, i.e., utilization rate is 60% 90%. In each case, we test the time it takes for EMP and Purging to delete different file sizes. Then, we compare the time overheads of EMP and Purging with different storage usages when deleting files of the same size.

Figure 14 shows the time overhead of erasing different file sizes for Purging and EMP in different scenarios. In the experiment, the storage size used is 8 GB, in which the result of graph (a) corresponds to scenario (i). The actual test has a storage rate of 15%, and the available space is 85%. Figure (b) corresponds to scenario (ii). The storage rate is 50%, and the available space is 50%. Figure (c) corresponds to scenario (iii). In the actual test, the storage rate is 75%, and the available space is 25%. The result shows that the time cost of EMP is lower than for Purging. The cost of the Purging operation is directly proportional to the size of the available storage space, regardless of the size of the deleted file. The EMP time cost is independent of the amount of free space available and is proportional to the size of the deleted file.

Figure 15 shows the time overhead of Purging and EMP for different storage usage scenarios when 1 GB files are deleted at the same time. The results show that the time overhead of the Purging operation is directly proportional to the storage available space because the Purging operation first uses a normal method to logically delete the target file and then uses a temporary file that is filled with random data to occupy the free space in the storage. Under different amounts of storage free space, the time cost of deleting the same size file is basically the same.

2) Performance Comparison: It is not difficult to find from the TABLE I,the program combines most factors, and the performance is relatively perfect.In the paper [23], Wang et al. successfully port EncFS to an Android device. Although it does not provide secure deletion, this document provides technical feasibility for implementing VACUUM. In the paper [25], Reardon et al. propose a method for deleting files. This method fills the available space in the literature system with garbage files and forces the performance of GC by flash to implement a memory cleaning operation. However, this method does not guarantee data confidentiality and integrity while removing inefficiency. Compared to the paper [26], Reardon et al. design a secure file deletion method based on the B-Tree method. However, this method is aimed at specific file systems and does not have good portability. The authors use this technology for their experiment and provide a proof of concept implementation. In contrast, one of the main goals of VACUUM is to provide the flexibility and easy development features of the solution that can be easily used daily in mobile devices. Compared with SADUS [27], the proposed scheme is optimized in the key management and garbage collection modules. It improves the system efficiency and ensures that the system key and ciphertext are unrecoverable, which makes the system more secure and efficient. The specifics will be discussed in the next section.

This paper focuses on analyzing the performance of VACUUM and [24]. The performance test indicates that, with an extra encryptographic and decryptographic operation, VACUUM and FDE have a larger performance overhead compared to the native file system read/write performance when the system performs read/write operations. This is a balance between security and performance. For write operations, VACUUM's overhead is 16.27% more than FDE, and these overhead results are expected because FDE encryption is running on the driver layer. As we can see in the previous analysis, the closer to the physical storage medium, the higher the encryption and decryption efficiency. VACUUM operates on the user level and maintains file system-related information through the FUSE framework. It frequently switches between user space and kernel space, causing performance degradation. However, VACUUM provides a fine-grained security delete function that is within the user's acceptable range. Despite this drawback, the actual number of VACUUM processes per second is still high. However, VACUUM produces a more significant performance impact during file creation. This effect may be because the VACUUM needs to perform a large number of extra I/O operations to repeatedly access its KeyStore and decrypt the corresponding 0 nodes to obtain the key corresponding to each newly created file.

Based on the above comparative analysis, it can be seen that the scheme of this paper has high applicability and safety. This is true especially in the read and write performance, the encryption and decryption overhead, and the efficiency of the garbage collection module. At the same time, many of the novelties of this work are related to solving the challenges that exist in different designs.

V. RELATED WORKS

Currently, data is mainly stored locally on the device or in the cloud. Therefore, research on data deletion is also carried out in two parts. There are many researches on the secure storage and deletion of data in the cloud [5], [28]–[30]. For example: Yu et al. proposed a new framework [4] to deal with the problem of embedded data deletion by integrating cloud, fog and devices. This is a reliable data deletion protocol that implements verifiable data deletion and flexible access control to sensitive data. The data deletion scheme for the device has also received extensive attention and research. While the existing secure deletion approaches vary widely, they can be categorized into two main groups: overwrite-based methods and encryption-based methods.

A. Overwrite-based secure deletion

Sanitizing data by overwriting is the most intuitive approach to secure deletion given its analog in the analog world [31].

Trim [32] and TrueErase [33] enable the file system to notify the lower level device drivers to delete the file content so that the file can be deleted in place. For flash or solid-state storage, the trim command requires the support of the operating system and device driver to achieve the flash content updating in place. At present, most mobile devices are not supported.

Joel et al. [34] use the interface to delete data securely, but the whole scheme has certain limitations. Then, they [35] proposed a method based on padding and overlay to delete data safely; however, this is not a good solution. At the same time, in [36], the author also studied a data deletion scheme

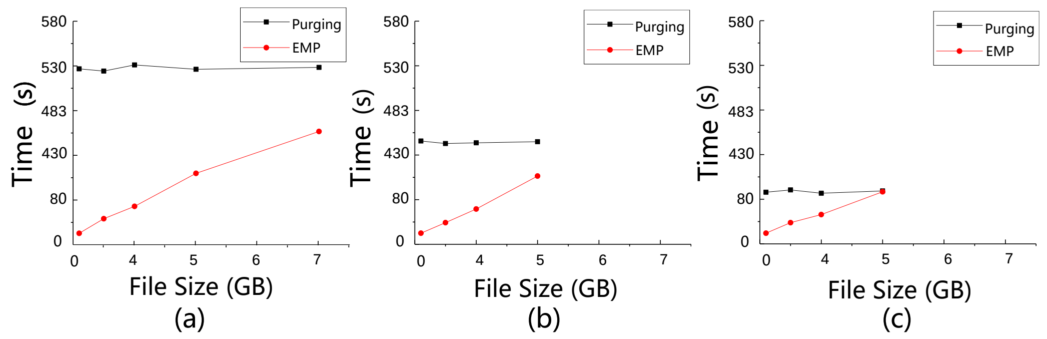

Fig. 14. The time overhead of Purging and EMP to delete different file sizes

TABLE II
COMPARISON OF DIFFERENT FACTORS OF EACH SCHEME.

	Secure Deletion	Confidentiality	Integrity	Fine-grained	Efficiency	Flexibility
Google [24]	×	√	√	×	n/a	×
Reardon *et al.* [26]	√	√	√	√	√	×
Reardon *et al.* [25]	√	×	×	√	×	√
Wang *et al.* [23]	×	√	×	√	n/a	√
Yang *et al.* [27]	√	√	√	√	×	√
VACUUM	√	√	√	√	√	√

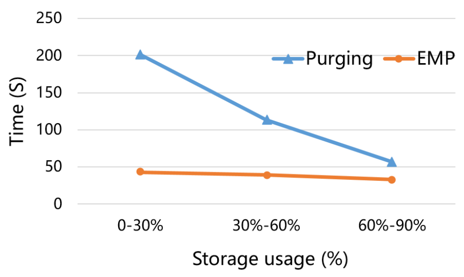

Fig. 15. The time overhead of different memory usages under Purging and EMP when deleting files with the same size

for fast file storage systems, which was constructed based on graph theory and shown to be effective through experiments.

Wang et al. [37] have devised a solution for efficient data deletion, which comes at the expense of limited security. Later, in [38], a secure data deletion scheme was designed to solve system overhead and inefficiency.

Chen et al. [39] propose a method for secure data deletion in fast file storage devices to solve security problems such as the data leakage caused by traditional data cleanup. At the same time, this method does not generate excessive system overhead.

Reardon et al. [25] propose a user-level secure removal method. Since most embedded flash devices use the built-in FTL algorithm, which is a black box for the upper file system, cannot manipulate the device driver to perform garbage collection for the discarded flash page. We can reduce the file system available free space to encourage more frequent garbage collection ensuring that no deleted data can remain on the storage medium.

VACUUM deletes the file by filling the storage medium to its capacity, which is divided into active and passive triggers. When one encounters a strong adversary, the garbage purge is passively triggered to purge the key storage area and perform a forced recall flash page.

B. Encryption-based secure deletion

For encryption-based secure deletion [40], decryption keys are usually stored in disks, and to achieve secure deletion, these keys for the deleted data are removed [41]. Unlike overwrite-based methods, encryption-based solutions focus on which layers should be encrypted and how to handle the keys [19].

The solution of using encryption to remove the data was originally proposed by Boneh et al. [18]; they delete a small encryption key to achieve the purpose of deleting the entire encrypted tape.

There are many designs and implementations at the kernel level for different popular block device file systems [42]–[44]. These schemes encrypt the file data nodes in the kernel space by extending the underlying file system, which supports a variety of encryption algorithms. Users can distinguish between encrypted files and ordinary files by mounting different directories.

Reardon et al. [26] propose the UBIFSec file system to implement data node encryption by extending the UBIFS file system. The UBIFSec provides a key for each data node and establishes a mapping between the data node and the key stored in the KeyStore area.

Wang et al. [45] design a data security removal scheme. First, establish a key tree and generate key encryption data. Then, the data deletion operation is controlled by setting a threshold. When the number of invalid data blocks is greater than the threshold, the data block is erased. When the number of invalid data blocks is less than the threshold, the key is deleted.

Yang et al. [27] propose the SADUS file system to assign a single key to each file only disconnecting the file from the corresponding key in a secure delete operation. However, this scheme cannot adequately protect the security of the key. Similarly, Xiong et al. [46] also proposed a secure data deletion scheme (SDDK) for building IoT devices, which can protect data privacy and delete invalid data blocks.

Wang et al. [23] propose an approach to optimize the EncFs system in the Android system mainly by modifying the size of the encrypted file block and using a Direct-IO way to read the file according to the FUSE feature. Since EncFs encrypts the file name and content using the key generated by the user-entered password, once the user password is compromised, the entire encrypted file system can be decrypted, and therefore, all data are exposed to an adversary.

We present the VACUUM user space file system. VACUUM creates a unique key for each file and deletes the file by deleting the corresponding key. Since flash utilizes out-of-place update, we encrypt each file key at the same time. Once a strong opponent enters the user's incorrect password, all the file keys in the encrypted file system will be purged. At the same time, due to the use of the EMP mechanism, which greatly improves the efficiency of garbage removal. The garbage collection mechanism is introduced to purge the memory periodically, reclaiming the discarded file encryption key and ciphertext.

VI. CONCLUSIONS

On the basis of previous related work which aiming at the urgent problem of safely deleting user data on embedded devices (such as mobile phones, smart TVs, etc.), this paper studied the logical structure, operation characteristics and data management mechanism of flash. We analyzed the process of data deletion and studied the reasons for the failure of data deletion in terms of physical storage, file system characteristics, and implementation of Android data deletion functions. Based on the data storage process, we have studied the data security deletion scheme including the physical layer, the driver layer, the file system layer, and the user layer and their advantages and disadvantages. Further, with the FUSE framework of the user space file system, we proposed a secure deletion method based on encryption technology, designed a user-space encrypted file system with enhanced deletion, and solved the problems of a single application scenario, complex implementation, difficulty in the existing data security deletion, and difficulty in migration. In addition, we implemented the prototype system, VACUUM, on an Android mobile device. We proved that the file system can resist specific online and offline attacks to ensure that the file is safely stored and that the deleted data are irrecoverable. In future work, we will also focus on sensitive data security backup issues. At the same time, further improvement of system performance is also a problem worthy of attention.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their careful reading and useful comments. This work was supported by the National Natural Science Foundation of China (62072359, 62072352, 61671360).

REFERENCES

[1] J. J. Rodrigues Barata, R. Munoz, R. D. De Carvalho Silva, J. J. P. C. Rodrigues, and V. H. C. De Albuquerque, "Internet of things based on electronic and mobile health systems for blood glucose continuous monitoring and management," *IEEE Access*, vol. 7, pp. 175 116–175 125, 2019.

[2] H. Ozkan, O. Ozhan, Y. Karadana, M. Gulcu, S. Macit, and F. Husain, "A portable wearable tele-ecg monitoring system," *IEEE Transactions on Instrumentation and Measurement*, vol. 69, no. 1, pp. 173–182, 2020.

[3] S. Sahay, M. Klachko, and D. Strukov, "Hardware security primitive exploiting intrinsic variability in analog behavior of 3-d nand flash memory array," *IEEE Transactions on Electron Devices*, vol. 66, no. 5, pp. 2158–2164, 2019.

[4] Y. Yu, L. Xue, Y. Li, X. Du, M. Guizani, and B.Yang, "Assured data deletion with fine-grained access control for fog-based industrial applications," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4538–4547, 2018.

[5] D. Zheng, L. Xue, C. Yu, Y. Li, and Y. Yu, "Toward assured data deletion in cloud storage," *IEEE NETWORK*, vol. 34, no. 3, pp. 101–107, 2020.

[6] K. C and B. Ponsy, "Detecting and confronting flash attacks from iot botnets," *The Journal of Supercomputing*, vol. 75, no. 12, pp. 8312–8338, 2019.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[8] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "Asm: A programmable interface for extending android security." in *USENIX Security Symposium*, 2014, pp. 1005–1019.

[9] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on android for diverse security and privacy policies." in *USENIX Security Symposium*, 2013, pp. 131–146.

[10] J. SHU, Y. ZHANG, J. LI, B. LI, and D. GU, "Why data deletion fails? a study on deletion flaws and data remanence in android systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 2, pp. 61(1)–61(22), 2017.

[11] S. M. Diesburg and A.-I. A. Wang, "A survey of confidential data storage and deletion methods," *ACM Computing Surveys (CSUR)*, vol. 43, no. 1, p. 2, 2010.

[12] S. Diesburg, C. Meyers, M. Stanovich, A.-I. A. Wang, and G. Kuenning, "Trueerase: Leveraging an auxiliary data path for per-file secure deletion," *ACM Transactions on Storage (TOS)*, vol. 12, no. 4, p. 18, 2016.

[13] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "Cleanos: Limiting mobile data exposure with idle eviction." in *OSDI*, vol. 12, 2012, pp. 77–91.

[14] S. Jia, L. Xia, B. Chen, and P. Liu, "Nfps: Adding undetectable secure deletion to flash translation layer," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 305–315.

[15] I. Shin, "Implementing secure file deletion in nandbased block devices with internal buffers," *IEEE Transactions on Consumer Electronics*, vol. 58, no. 4, pp. 1219–1224, 2012.

[16] M. Szeredi, *FUSE: File system in user space*, April 2006, http://fuse.sourceforge.net.

[17] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[18] D. Boneh and R. J. Lipton, "A revocable backup system." in *USENIX Security Symposium*, 1996, pp. 91–96.

[19] J. Reardon, "Robust key management for secure data deletion," in *Secure Data Deletion*. Springer, 2016, pp. 143–174.

[20] C. Yang, X. Chen, and Y. Xiang, "Blockchain-based publicly verifiable data deletion scheme for cloud storage," *Journal of Network and Computer Applications*, vol. 103, pp. 185–193, 2018.

[21] R. C. Merkle, "Protocols for public key cryptosystems," in *Security and Privacy, 1980 IEEE Symposium on*. IEEE, 1980, pp. 122–122.

[22] *EncFS: an Encrypted Filesystem*, Vgough, January 2008, https://vgough.github.io/encfs.

[23] Z. Wang, R. Murmuria, and A. Stavrou, "Implementing and optimizing an encryption filesystem on android," in *Mobile Data Management (MDM), 2012 IEEE 13th International Conference on*. IEEE, 2012, pp. 52–62.

[24] Google, *Android FDE: Full Disk Encryption*, April 2018, https://source.android.com/security/encryption/full-disk.

[25] J. Reardon, C. Marforio, S. Capkun, and D. Basin, "User-level secure deletion on log-structured file systems," in *Proceedings of the 7th ACM symposium on information, computer and communications security*. ACM, 2012, pp. 63–64.

[26] J. Reardon, S. Capkun, and D. Basin, "Data node encrypted file system: Efficient secure deletion for flash memory," in *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association, 2012, pp. 17–17.

[27] L. Yang, T. Wei, F. Zhang, and J. Ma, "Sadus: Secure data deletion in user space for mobile devices," *Computers & Security*, vol. 77, pp. 612 – 626, 2018.

[28] L. Xue, Y. Yu, Y. Li, M. H. Au, X. Du, and B. Yang, "Efficient attribute-based encryption with attribute revocation for assured data deletion," *Information Sciences*, vol. 479, pp. 640–650, 2019.

[29] Y. Tang, P. P. C. Lee, J. Lui, and R. Perlman, "Secure overlay cloud storage with access control and assured deletion," *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, vol. 9, no. 6, pp. 903–916, 2012.

[30] Y. Qu and N. Xiong, "Rfh: A resilient, fault-tolerant and high-efficient replication algorithm for distributed cloud storage," in *2012 41st International Conference on Parallel Processing*, 2012, pp. 520–529.

[31] I. Shin, "Supporting reliable data deletion for nand-based gadgets with limited memory," *International Journal of Applied Engineering Research*, vol. 11, no. 9, pp. 6381–6386, 2016.

[32] Wikipedia, *Trim*, April 2018, https://en.wikipedia.org/wiki/Trim_(computing).

[33] S. Diesburg, C. Meyers, M. Stanovich, M. Mitchell, J. Marshall, J. Gould, A.-I. A. Wang, and G. Kuenning, "Trueerase: Per-file secure deletion for the storage data path," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 439–448.

[34] J. Reardon, D. Basin, and S. Capkun, "Sok: Secure data deletion," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 301–315.

[35] ——, "On secure data deletion," *IEEE Security and Privacy*, vol. 12, no. 3, pp. 37–44, 2014.

[36] J. Reardon, H. Ritzdorf, D. Basin, and S. Capkun, "Secure data deletion from persistent media," in *CCS13 Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security*. ACM, 2013, pp. 271–284.

[37] W. Wang, Y. Chang, P. Huang, C. Tu, H. Wei, and W. Shih, "Relay-based key management to support secure deletion for resource-constrained flash-memory storage devices," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 444–449.

[38] W. Wang, C. Ho, Y. C. nd TeiWei Kuo, and P. Lin, "Scrubbing-aware secure deletion for 3-d nand flash," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2790–2801, 2018.

[39] B. Chen, S. Jia, L. Xia, and P. Liu, "Sanitizing data is not enough!: towards sanitizing structural artifacts in flash media," in *ACSAC 16 Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 496–507.

[40] T. M. Peters, M. A. Gondree, and Z. N. Peterson, "Defy: A deniable, encrypted file system for log-structured storage," 2015.

[41] M. D. Leom, K.-K. R. Choo, and R. Hunt, "Remote wiping and secure deletion on mobile devices: A review," *Journal of Forensic Sciences*, vol. 61, no. 6, pp. 1473–1492, 2016.

[42] Wikipedia, *BitLocker Overview*, Encryption, April 2018, https://en.wikipedia.org/wiki/Trim_(computing).

[43] P. Teufl, A. Fitzek, D. Hein, A. Marsalek, A. Oprisnik, and T. Zefferer, "Android encryption systems," in *Privacy and Security in Mobile Systems (PRISMS), 2014 International Conference on*. IEEE, 2014, pp. 1–8.

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

[45] M. Wang, J. Xiong, R. Ma, Q. Li, and B. Jin, "A novel data secure deletion scheme for mobile devices," in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2018, pp. 1–8.

[46] J. Xiong, L. Chen, M. Bhuiyan, C. Cao, M. Wang, E. Luo, and X. Liu, "A secure data deletion scheme for iot devices through key derivation encryption and data analysis," *Future Generation Computer Systems*, vol. 111, pp. 741–753, 2020.

Li Yang received the B.S. degree in Instructional Technology from Shaanxi Normal University in 1999, and M.S. degree in computer science from Xidian University in 2005, and Ph.D. degree in cryptography from Xidian University, Xi'an, China in 2010. He was a visiting post-doctoral researcher in the Complex Networks & Security Research (CNSR) Lab at Virginia Tech from 2013 to 2014. Now he is an associate professor in School of Computer Science, Xidian University. His research interests include applied cryptography, wireless network security, cloud computing security, and trusted computing.

Cheng Li is currently pursuing the Ph.D. degree in Computer Architecture from the School of Computer Science and Technology, Xidian University. His research interests focus on information security and cyberspace security.

Teng Wei received the Bachelors degree in computer science from Xidian University in 2011. He is Pursuing the masters degree in architecture of computer system in Xidian University. He has been committed to the mobile cloud computing and mobile data deletion research.

Fengwei Zhang earned his Ph.D. in Computer Science from Angelos Stavrous group at George Mason University in April 2015. His primary research interests are in the areas of systems security, with a focus on trustworthy execution, mobile malware analysis, debugging transparency, transportation security, and plausible deniability encryption.

Jianfeng Ma received his B.S. degree in mathematics from Shaanxi Normal University in 1985, and obtained his M.E. and Ph.D. degrees in computer software and communications engineering from Xidian University in 1988 and 1995 respectively. Since 1995 he has been with Xidian University as a professor. His research interests include information security, coding theory and network management.

Naixue Xiong (S'05–M'08–SM'12) is current an Associate Professor (6rd year) at Department of Mathematics and Computer Science, Northeastern State University, OK, USA. He received his both PhD degrees in Wuhan University (2007, about sensor system engineering), and Japan Advanced Institute of Science and Technology (2008, about dependable communication networks), respectively. Before he attended Northeastern State University, he worked in Georgia State University, Wentworth Technology Institution, and Colorado Technical University (full professor about 5 years) about 10 years. His research interests include Cloud Computing, Security and Dependability, Parallel and Distributed Computing, Networks, and Optimization Theory.

Dr. Xiong published over 200 international journal papers and over 100 international conference papers. Some of his works were published in IEEE JSAC, IEEE or ACM transactions, ACM Sigcomm workshop, IEEE INFOCOM, ICDCS, and IPDPS. He has been a General Chair, Program Chair, Publicity Chair, Program Committee member and Organizing Committee member of over 100 international conferences, and as a reviewer of about 100 international journals, including IEEE JSAC, IEEE SMC (Park: A/B/C), IEEE Transactions on Communications, IEEE Transactions on Mobile Computing, IEEE Trans. on Parallel and Distributed Systems. He is serving as an Editor-in-Chief, Associate editor or Editor member for over 10 international journals (including Associate Editor for IEEE Tran. on Systems, Man & Cybernetics: Systems, Associate Editor for IEEE Tran. on Network Science and Engineering, Associate Editor for Information Science, Editor-in-Chief for Journal of Internet Technology (JIT), and Editor-in-Chief for Journal of Parallel & Cloud Computing (PCC)), and a guest editor for over 10 international journals, including Sensor Journal, WINET and MONET. He has received the Best Paper Award in the 10th IEEE International Conference on High Performance Computing and Communications (HPCC-08) and the Best student Paper Award in the 28th North American Fuzzy Information Processing Society Annual Conference (NAFIPS2009).

Dr. Xiong is the Chair of "Trusted Cloud Computing" Task Force, IEEE Computational Intelligence Society (CIS), HYPERLINK "http://www.cs.gsu.edu/ cscnxx/index-TF.html" http://www.cs.gsu.edu/ cscnxx/ index-TF.html, and the Industry System Applications Technical Committee, HYPERLINK "http://ieee-cis.org/technical/isatc/" http://ieee-cis.org/technical/isatc/; He is a Senior member of IEEE Computer Society from 2012, E-mail: xiongnaixue@gmail.com.