COMPUTERS & SECURITY 77 (2018) 612–626

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

SADUS: Secure data deletion in user space for mobile devices

Li Yang[a,*], *Teng Wei*[a], *Fengwei Zhang*[b], *Jianfeng Ma*[a]

[a] *Xidian University, Xi'an 710071, China*
[b] *Wayne State University, Detroit, MI 48202, USA*

ARTICLE INFO

Article history:
Received 9 February 2018
Revised 27 April 2018
Accepted 22 May 2018
Available online 26 May 2018

Keywords:
Secure deletion
User space
Encrypted filesystem
Flash memory
Android

ABSTRACT

Conventional data deletion is implemented for reclaiming storage as a rapid operation. However, the content of the deleted file still persists on the storage medium. Secure data deletion is a task of deleting data irrecoverably from the physical medium. Mobile devices use flash memory as the internal storage. However, flash memory does not support the in-place update which is in direct opposition to efforts to securely delete sensitive data from storage. Previously practical secure deletion tools and techniques are rapidly becoming obsolete, and are rendered ineffective. Therefore, research on secure data deletion approaches for mobile devices has become a practical and urgent issue.

In this paper, we study the logic structure and operation characteristics of flash memory, and survey related work on secure data deletion. In addition, we define the adversary capability and threat model, putting forward the design goals that secure data deletion scheme needs to meet. Then an approach in user space that uses the user space file system to provide secure deletion guarantees at file granularity is proposed, independent of the characteristics of the underlying file system and storage medium. The approach encrypting every file on an insecure medium with a unique key that can later be discarded to cryptographically render the data irrecoverable. Moreover, we implement our secure data deletion approach on Android platform named SADUS. Finally, experiments are conducted, and the results indicate that SADUS prototype ensures the secure deletion of data in flash memory on mobile devices with comparable overhead and it can meet the requirements of the users in daily use.

1. Introduction

Technology trends have driven the commoditization of smart devices with more powerful processing capabilities and greater storage space for mobile computing. In contrast to traditional mobile devices, data-centric mobile smart terminals feature independent operating systems and the ability to install and run third-party software; they also provide mobile Internet services, data computing and storage services for users. Mobile smart terminals have become information processing centers and office assistants for people's daily work and life. As increasingly more personal sensitive information, such as address books, browsing histories, location information, and even companies' intellectual property, is being stored on mobile devices, data must be well protected. Due to the complexity of mobile environments, in the case where a mobile

* Corresponding author.
E-mail addresses: yangli@xidian.edu.cn (L. Yang), okweiteng@gmail.com (T. Wei), fengwei@wayne.edu (F. Zhang), jfma@mail.xidian.edu.cn (J. Ma).
https://doi.org/10.1016/j.cose.2018.05.013
0167-4048/

terminal is lost, stolen or sold, private personal information is faced with a huge security threat (Diesburg et al., 2016a).

With the increasing use of encryption systems, an attacker wanting to gain access to sensitive data is directed to weaker targets. One possible attack is the recovery of supposedly erased data from internal storage, possibly a flash memory card (Albano et al., 2011). After a file is deleted, its data remain stored on physical media until the actual data blocks are overwritten. Sometimes, this enables users to recover a file that they had mistakenly deleted. Unfortunately, a malicious user can also recover such a deleted file. Traditional mobile manufacturers provide a built-in factory reset method to initialize the device to the initial state. However, implementation flaws in factory resets lead to successful attacks through the recovery of private data directly from the flash image (Tang et al., 2012). Simon and Anderson (2015) showed that substantial amounts of private information from second-hand electronic storage devices can be recovered, proving that remnant flash memory data are very common, leading to serious data security issues. They concluded that there were approximately five million Android phones in the market that can still have their deleted data recovered even when restored to the original factory settings.

To ensure data security, sensitive data stored in flash memory must be deleted securely. This is important, as reserving obsolete data may not only endanger data owners' privacy but also violate the retention policies of relevant legislation (Bajaj and Sion, 2013; Lee et al., 2010). Secure deletion means that when a file is deleted, its content does not persist on the storage medium and cannot be recovered, and the device can be re-used (Zarras et al., 2016). Secure deletion is almost always ignored in file system design largely for performance reasons.

Researchers have proposed many techniques for secure data deletion from physical media. Conventional solutions for secure deletion rely on either overwriting (Bauer and Priyantha, 2001; Reardon et al., 2012b; Sun et al., 2008) or encryption (Lee et al., 2008; Reardon et al., 2013b). Most work (Bauer and Priyantha, 2001; Gutmann, 1996; Joukov and Zadok, 2005) has achieved secure deletion by relying on in-place updates, in which the data to be securely deleted are over-written by random numbers or new data. Due to the features of flash media, security methods based on overwriting for magnetic media are restricted (Wei et al., 2011). These special characteristics make it prohibitively expensive to perform an in-place update on flash. On the other hand, several works have claimed to be able to achieve secure deletion for flash memory by relying on encryption (Lee et al., 2008; Reardon et al., 2012a); however, these works are only suitable for certain file systems and have poor portability.

According to the requirements imposed by the mobile devices (Huang et al., 2015), we propose an encrypted file system in user space that can provide secure storage and deletion. In contrast to previous work (Diesburg, 2016), our scheme guarantees satisfactory irrecoverability of deleted files on different file systems or black box storage media with unknown characteristics. By extending the EncFS (2008) file system to ensure the confidentiality of sensitive data, we introduce a key management module that removes the keys of the sensitive files to make the deleted data irrecoverable. In addition to ensuring the security of sensitive data, we significantly improve the read and write performance to achieve an acceptable time complexity, thereby prolonging the flash memory life-time. Simultaneously, we consider the scalability of secure storage.

Considering current secure deletion limitations, in this work, we study a method for mobile terminal storage secure deletion from the viewpoint of storage architecture, and then, we design and implement a mobile terminal secure deletion file system in user space on an Android platform. The main contributions of this paper are summarized below.

- We conduct research on secure file deletion via an encrypted file system in user space on a commodity smart device given modern NAND technology as the storage media. To this end, we present a novel secure deletion model for flash memory located in user space.
- We design a generic secure data deletion framework named SADUS. In contrast to previous work, our framework guarantees satisfactory irrecoverability of deleted files on different file systems or black box storage media with unknown characteristics. In addition, we detail our analysis in terms of security versus performance tradeoffs.
- Finally, we analyze the security of SADUS and prove that our scheme can resist a strong attacker. In addition, we implement the secure data deletion file system on a commodity Android device and demonstrate that it can be successfully applied to smart mobile devices at a reasonable performance overhead.

The remainder of this paper is organized as follows: Section 2 offers related research on secure deletion. Section 3 introduces the background information and our threat model. The design of SADUS on an Android platform is discussed in Section 4. In Section 5, we analyze the security of our approach. In Section 6, we implement the SADUS and present the performance results under different file system operations, and Section 7 concludes the paper.

2. Related works

Various secure deletion approaches have been proposed at different layers (Reardon et al., 2013a; Sun et al., 2008). Researchers have proposed many solutions to the problem of securely deleting files to ensure that they remain unrecoverable (Jia et al., 2016). While the proposed approaches vary widely, they can be classified into two main groups: overwrite-based and encryption-based methods.

2.1. *Overwrite-based secure deletion*

Sanitizing data by overwriting them is the most intuitive approach to secure deletion solutions given its analogue in the analog world (Shin, 2016). By summarizing and comparing current methods for the secure storage and removal of data in personal computing environments, Diesburg and Wang (2010) propose that secure deletion and secure storage are equally important and that we must protect the data confidentiality its entire lifetime whenever we store or delete data. The most direct method is to rewrite the file content such as by

using the user space security methods shred (GNU, 2012) and scrub (Ben, 2009); however, the premise is that the file content must be updated in-place. Because flash memory does not support in-place updates, this solution is not suitable for data removal in flash memory.

Reardon et al. (2013b) state that deleting files in file systems is generally performed through the following three methods: unlinking files, truncating files and updating the location of content to be deleted. The file content remains present in the first two deletion method, and the last method ensures the secure deletion of file content. However, secure deletion is almost always ignored in file system design, largely due to performance reasons. Typically, deletion is implemented as a rapid operation whereby a file is unlinked, meaning its metadata state that it is no longer present, while the file's contents remain in the storage medium until overwritten by new data.

Trim (2008) and TrueErase (Diesburg et al., 2012) enable a file system to notify lower level device drivers to delete the file content so that the file can be deleted in place. For flash and solid-state storage, the trim command requires support by the operating system and device driver to achieve in-place flash content updating. Currently, most mobile terminals do not provide compatibility for this option.

Wei et al. (2011) propose scrubbing, which reprograms the original physical pages to delete data. They used the term scrub budget to refer to the number of times that the flash memory has experimentally allowed multiple overwrite operations without presenting a significant risk for data errors. When the scrub budget for a block is exceeded, secure deletion is instead performed by invoking garbage collection.

Reardon et al. (2012b) propose a user-level secure removal method. Because most embedded flash devices use the built-in FTL algorithm, which is a black box to the upper file system, the device driver cannot be manipulated to perform garbage collection for discarded flash pages. We can reduce the file system's available free space to encourage more frequent garbage collection, ensuring that no deleted data can remain on the storage medium.

SADUS deletes a file by filling the storage medium to its capacity, therein being divided into active and passive triggers. When encountering a strong adversary, garbage purging is passively triggered to purge the key storage area and perform a mandatory recall of the flash page.

2.2. *Encryption-based secure deletion*

For encryption-based secure deletion (Peters et al., 2015), decryption keys are usually stored in disks to achieve secure deletion. Those keys for the deleted data are removed (Leom et al., 2016). Unlike overwrite-based method, encryption-based solutions focus on which layers are to be encrypted and how to handle the keys (Lai et al., 2017; Reardon, 2016).

A solution of using encryption to remove the data was originally proposed by Boneh and Lipton (1996); they delete a small encryption key to achieve the goal of deleting the entire encrypted tape. Peterson et al. (2005) use this method to improve the efficiency of a multi-version backup file system for the secure data removal of the disk media. They reduce the addressing time to locate a single file using in-place updates to remove relationships between tags and keys. Diesburg et al. (2016b) also demonstrate TrueErase can serve as a building block by cryptographic systems that securely delete information by erasing encryption keys.

There are many designs and implementations at the kernel level for different popular block device file systems (Bitlocker overview, 2004; Czeskis et al., 2008; Teufl et al., 2014). 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. (2012a) propose the UBIFSec file system to implement data node encryption by extending the UBIFS file system. UBIFSec provides a key for each data node and establishes a mapping between the data node and the key stored in the key storage area. Due to the out-of-place update feature of the flash memory, by modifying the FTL algorithm, UBIFSec organizes the key store area, forcing the file key to be deleted, and the unused key is recovered whenever the files are deleted. In addition, UBIFSec establishes a time checkpoint for each operation to maintain the consistency of file operations.

Because the kernel-level approaches require re-compiling the file system, the key and encryption mode cannot be changed once it is determined. Thus, these approaches are very limited in terms of portability.

One user space encryption file system uses the FUSE (Filesystem in Userspace) (Szeredi, 2001) framework. Without modifying the kernel space code, it can use a user space application to achieve a built-in virtual file system. FUSE translates the VFS (Virtual File System) (Vnodes et al., 1986) and calls a redirect to the user space file system, which adds security features; then, it forwards calls to the underlying file system. EncFS (2008) and CryptoFs (Hohmann, 2003) both achieve a secure storage file system based on FUSE. The two systems are very similar: 1) They store the encrypted file and file name in an encrypted directory. 2) The user is required to use the correct key to mount the encrypted directory to a special mount point to decrypt the encrypted file name and file. 3) The user is prompted to enter a password to generate the encryption key. 4) Public encryption algorithms, such as AES, DES, Blowfish, and Twofish, are supported. 5) The files are encrypted by block. These user space encryption file systems use a standard encryption mechanism that provides robustness, therein supporting a flexible security policy and permitting the use of different encryption algorithms. However, the user space file system needs to switch between user space and kernel space multiple times to perform file operations, which increases the performance overhead compared with the kernel space file system.

Wang et al. (2012) propose an approach to optimize the EncFs system in an Android system, mainly by modifying the size of the encrypted file block and using a Direct-IO method to read the file based on the FUSE feature. Their experiments show that the read and write performance of the user space encrypted file system meets user requirements and that the performance overhead is acceptable for users. Because 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 file can be decrypted, and thus, all data are exposed to the adversary.

We present the SADUS user space file system. SADUS creates a unique key for each file and deletes the file by deleting the corresponding key. Because flash memory uses out-of-place updates, we encrypt all file keys simultaneously. Once a strong opponent enters an incorrect user password, all file keys in the encrypted file system will be purged.

3. Preliminaries

In this section, we first give some background; then, we present our adversarial model. We also present our assumptions and the formal definition of secure deletion.

3.1. *Background*

3.1.1. *Flash memory*

Flash memory is a electronic (solid-state) non-volatile computer storage medium that can be electrically erased and reprogrammed. The most prominent characteristic of flash memory is that rewritten data can only be dynamically updated via time-consuming erase operations. Furthermore, every block in flash memory is subject to a maximum number of program/erase cycles, typically 10^4 to 10^5 cycles. Because the size of the file read and written by the block file system is inconsistent with the size of the flash page, there are multiple copies of the modified content so that even if the file is deleted, the discarded flash page still possesses the remaining file information. The flash characteristics of out-of-place updates are the main cause of post-removal data disclosure.

3.1.2. *Flash translation layer*

To conceal the characteristics of NAND flash, a special-purpose firmware called the 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, including USB sticks, SD cards, EMMC cards, and SSDs, are equipped with the FTL. In embedded devices, the 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.

Address translation is the most basic functionality provided by an FTL. As shown in Fig. 1, the physical addresses corresponding to the logical addresses are determined by a mapping algorithm of the FTL. During address translation, the FTL looks up the address mapping table. The mapping table is stored in SRAM, a fast but high-priced memory, which is used for mapping logical addresses to physical addresses in units of sectors or blocks. When issuing overwrites, the FTL redirects the physical address to an empty location (free space), thus avoiding the erase operations. After managing an overwrite, the FTL changes the address mapping information in SRAM. The outdated block can be erased later. The FTL achieves performance and durability enhancement in addition to the basic address translation. The performance enhancement refers to the issues of reducing the number of read, write and erase operations. Among the three operations, reducing the number of erase operations is the most critical issue because the cost of an erase operation is very high compared to that of read and write operations. Durability enhancement refers to erasing every physical block as evenly as possible without causing performance degradation. If a block is above the program/erase cycle limit, the block may not function correctly, thus causing data loss.

3.1.3. *EncFs*

EncFs is a FUSE-based cryptographic file system. It transparently encrypts files, using an arbitrary directory as storage for the encrypted files. FUSE (Szeredi, 2006) provides a framework for implementing 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 file system: the source directory, and the mount point. Each file in the mount point has a specific file in the source directory to which it corresponds. The file in the mount point 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 utilized to decrypt this key.

3.2. *Threat model*

Smart phones have been a data-centric model, with the rich set of sensors integrated within these devices; the data gathered and generated present highly sensitive user privacy issues.Due to the high churn rate of new devices, it is compelling to create innovative security solutions that are hardware agnostic. The application sandbox approach protects application-specific data from other applications on a phone. However, improper placement, resale or disposal of mobile devices will result in the leakage of sensitive data. Mandatory evidence requests by law enforcement can also result in unauthorized access to data. Such data can also be intentionally exfiltrated by malicious programs via communication channels, and an attacker can compromise a smart phone and access the sensitive data by technical means. To ensure the secrecy of the data for its lifetime, we must provide robust techniques to store and delete data while ensuring confidentiality and integrity.

Adversarial model. We consider a scenario in which personal data are stored in a mobile device equipped with flash memory, where users hope that the data cannot be subject to unauthorized access after its deletion and that the deleted data cannot be recovered by technical means to ensure that the data are secure and controllable forever.

In this work, we model a novel type of attacker that we call the peek-coercion-recovery attacker. This attacker is more powerful than the strong coercive attacker considered in other secure deletion works. A coercive attacker (Chang et al., 2015) can, at any time, compromise both the storage medium containing the data along with any secret keys or pass phrases required to access them. The peek-coercion-recovery attacker extends the coercive attacker to also allow the attacker to

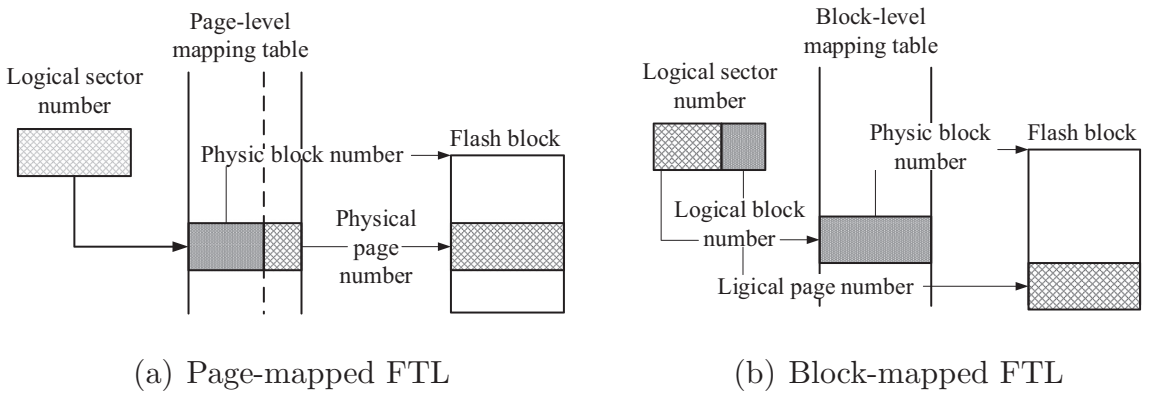


Fig. 1 – The address translation in FTL.

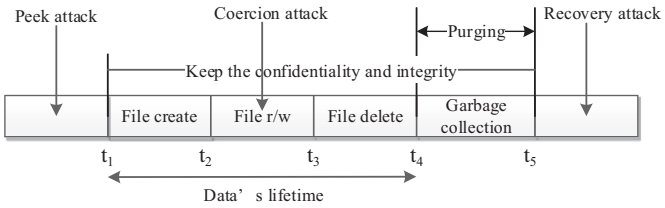


Fig. 2 – Peek-coercion-recovery attacker during file lifetime.

obtain the content of the storage medium at some point(s) in time prior to the storage medium being compromised.

Fig. 2 shows the timeline of data storage and an attack. Time is divided into discrete epochs. We assume that purging is an atomic operation. The lifetime of a piece of data is then defined as all the epochs from the epoch in which it was written to the epoch in which it was deleted. Data are written at time t_1 and deleted at time t_4, and the data's lifetime includes all times between these two points. Here, the peek attack (read access to the entire storage medium) occurs before t_1; then, the coercion attack (full compromise of the storage medium and secret keys) is performed during the file access. Finally, the recovery attack (attempt to recover the deleted data by technical means) occurs after time t_5.

The coercive attack model for legal subpoenas assumes that that users are required to forfeit devices and reveal passwords. Because the time of the attack is arbitrary and therefore unpredictable, no extraordinary sanitation procedure can be performed prior to the compromise time. Because the attacker is given the user's secret keys, it is insufficient to simply encrypt the storage media (Joukov et al., 2006). The peek-coercion-recovery attack enables an attacker who additionally obtains temporary read access to the medium (e.g., hidden malware that is forced to send suicide instructions upon being publicly exposed) to then subsequently perform a coercive attack. This is roughly analogous to forward secrecy in the sense that a secure deletion scheme is resilient to a peek-coercion-recovery attacker. This prevents recovery of deleted data even if a prior snapshot of the data on the storage medium is available to the attacker.

3.3. *Security definitions*

A secure data deletion scheme should be able to perform two security functions: a) Storing encrypted data such that the adversary cannot break the ciphertext within a limited time in a violent manner after performing a quick delete operation and b) removing data from the physical storage media permanently such that the adversary cannot recover the deleted data or have access to the data. In the following, we provide a formal definition of a secure deletion scheme.

Let Λ be a deletion scheme. A flash-based block device implements Λ. Let Ω be the encrypted file life cycle stored in the flash memory of this block device. $\Omega = \{C, A, D, G\}$. Ω is a quaternion, in which each element represents the file creation, file access, file deletion, and file garbage collection operations. Let $\mathcal{A}$ be the set of access operations (e.g., read and write) performed on the flash memory of this block device. $\mathcal{A} = \{A_{t_1}, A_{t_2}, \ldots, A_{t_n}\}$, in which A_{t_i} represents an access operation at time t_i, where $0 \le i \le n$. The flash-based block device is normally and safely unmounted at time t_a, and the adversary obtains a complete image of the flash at time t_b, where $t_n < t_a < t_b$.

We say that Λ is a secure deletion scheme if and only if the following two conditions can be satisfied simultaneously: 1) The adversary cannot have access to any data after D in Ω

that has been deleted before t_a. 2) Let p be the probability that the adversary can recover the deleted data after G in Ω; then, $p \to 0$.

3.4. Security and functionality requirements

The proposed scheme must keep the data confidential and integrated and delete files in a fine-grained manner. Moreover, it should be easily implemented and need minimal extra resources. Therefore, the proposed scheme needs to meet the following security and functional requirements.

3.4.1. Security requirements

Confidentiality. Confidentiality requires that all secure storage data are able to resist a competent attacker so that the file contents cannot be accessed by unauthorized users Tang et al. (2010).

Integrity. Integrity requires that all secure storage data are able to withstand the ability of a competent attacker to tamper with the file contents and keep the file complete.

3.4.2. Functionality requirements

Fine granularity. Fine granularity requires the secure storage and deletion of files regardless of how small the file is. The system must include capabilities for modifying, truncating files, or deleting content from the database.

Effectiveness. In resource-constrained mobile terminals, we require secure storage and deletion operations to be efficient because mobile resources are limited in terms of, e.g., battery resources, computing power, storage, and equipment lifetime.

Applicability. We demand that our scheme is simple, and the underlying file system must be easily achieved. To this end, we directly store and delete files securely on top of the given file system.

Flexibility. We wish to implement our approach by minimizing the given file system and isolating the main functions and data structures. In addition, our changes should easily be audited and analyzed by security experts. Moreover, we place no restrictions on the underlying file system features.

4. System design

In this section, we design a secure data storage and deletion system for mobile terminal, ensuring that the operations of storing and deleting files are secure. First, we describe our expected goals for the scheme; then, we detail the system design process.

4.1. Architecture

We now present our secure deletion solution and show how it fulfills the listed requirements.

The user stores the data to the physical storage medium generally through the following steps. Using a common interface of the Virtual File System (VFS) to find the given underlying file system, the underlying file system calls the device driver to transfer data to the physical media controller. Finally, the physical controller writes data to the physical medium. However, in the process of storing data, each level almost considers the efficiency of storage and does not fully consider secure storage and deletion. In addition, because the flash uses out-of-place updates, the difficulty of secure data deletion is significantly increased. By modifying the kernel-level file system or device driver to achieve secure storage and deletion, there would be numerous modifications. Most flash storage media controllers give priority to performance instead of security, and they are black boxes to the upper drive device. Therefore, we propose SADUS, a secure data storage and deletion file system in user space using the FUSE driver. The FUSE driver was added between the VFS and the underlying file system, therein using the user space file system to handle file operations.

The architecture of our SADUS method, shown in Fig. 3, consists of several components. The SADUS layer abstracts the implantation details of the underlying file system and makes it suitable for a variety of deployment scenarios. The actual SADUS functionality requires three core components: the Encryption module, the Key Manager module, including the corresponding Key Manager Storage submodule, and the Purging module. The encryption module encrypts individual files by translating all requests for the virtual SADUS file system into the equivalent encrypted operations on the raw file system to ensure the confidentiality of personal data. The key manager module manages the file key when, for example, performing the store, update, and delete operations. The purging module ensures that there are no obsolete data in the flash memory. Finally, SADUS provides an easy interface to store and delete files. We use different keys to encrypt different files and add the Key Manager module to manage the key distribution in a transparent manner to effectively remove the deleted key of the encrypted file, thus achieving file secure deletion. The file is encrypted before writing to the storage medium and is decrypted before reading. All operations are performed in memory. Each file key is stored in the Key Storage Area and managed by the Key Manager module.

4.1.1. Encryption

SADUS uses the encryption layer to ease integrating the encryption and decryption functions. This layer conceals the details of the underlying encryption scheme and MAC function implementation. The cryptography layer is mainly implemented by EncFs. EncFs is a FUSE-based encrypted file system that transparently encrypts files using any directory as the directory for all encrypted files. Three major components are required to make EncFs work on any platform: the kernel FUSE library support, user space libfuse, and EncFs binaries. To make an encrypted file system work on the Android operating system, a modified bootstrapping process and password login were integrated into the operating system framework. EncFs uses standard OpenSSL cryptographic libraries in user space. This gives us various advantages over using a kernel-based cryptographic library. Libfuse and libc can run stably on different platforms and are rarely affected by the device. Standard libcrypt and libssl libraries implement a variety of encryption algorithms and are reliable and compatible. EncFs supports two block cipher algorithms: AES and Blowfish. Both algorithms support key lengths of 128 to 256 bits and block sizes of 64 to 4096 bytes. We configure SADUS using the

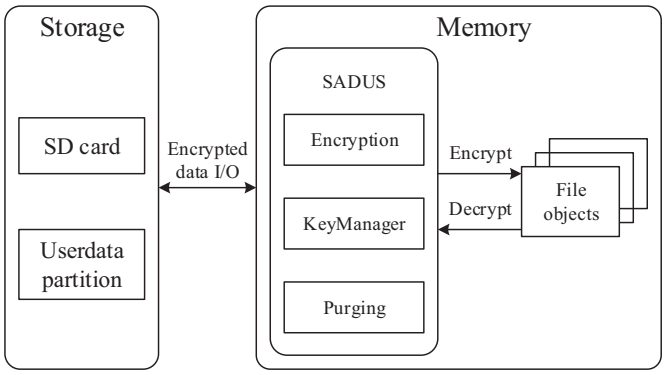


Fig. 3 – Abstraction of SADUS architecture.

AES-256 algorithm to encrypt file name paths and file contents, and the packet encryption mode uses CBC and CFB. To save user space, the file path uses the 8-bit CFB mode. The file header includes a hash of the file path to prevent the file path from being tampered with. Subdirectory encryption is achieved with parental caching encryption. Encrypting the file directory prevents file directory information from revealing file names.

SADUS ensures the integrity of the contents as well as file confidentiality. Each encrypted block header contains the file message authentication code for this block, and the initialization vector for each encrypted file block is associated with the file index number.

4.1.2. *Key manager*

We use an encrypted database to store all SADUS file keys, the module managing all keys is called the Key Manager (KM), and the encrypted file storing all file keys and file system profiles is called Key Manager Storage (KMS). The KMS is managed by the KM and is isolated from the SADUS file system, belonging to the underlying file system. When a file is created, the KMS creates a ciphertext of the corresponding file key, which is encrypted by the KM master key. When writing a file to the storage medium, it encrypts the file using the corresponding key provided by the KM. When the file is deleted, we only need to mark the encrypted file key as deleted.

Each new file in the KMS database has a corresponding file key cipher, therein creating a mapping by the cipher text of the file's full path name and the cipher text of the file key. We encrypt the entire KMS database file to ensure the confidentiality of the file. Every time we delete a file, we need to modify the KMS database file to delete the file key and the mapped record. Because we use the KM master key to encrypt the KMS file with a strong encryption algorithm, ensuring that the deleted key is unable to recover the file contents given limited resources and limited time, we achieve the objective of secure deletion. When a file is deleted, the ciphertext and the key remain scattered in the flash memory, and we provide manually forced purging to ensure the ciphertext content and key are erased. Fig. 4 shows the complete SADUS access process.

When a file is created, the KM generates a key for the file, stores it in the KMS using the KM master key, and creates a mapping between the name and the file key in the KMS database. The operation of the KM is transparent to the upper application. Fig. 4 shows the secure write and read process in SADUS. The solid-line arrow represents the safe write process, and the dotted-line arrow represents the read operation. For the file write operation, SADUS first generates the ID stored in the KM according to the file name; then, it decrypts the file's key K' and the initial vector IV' according to the ID, which are used to encrypt the file. Finally, it writes the encrypted data to the storage medium. For the file read operation, as the dotted-line arrow shows, the first two steps are the same as in the write method: It first generates the ID stored in the KM according to the file name. Then, it decrypts the file's key K' and the initial vector IV' according to the ID and reads the encrypted data to memory. Finally, it decrypts the data using K and IV.

4.1.3. *Key manager storage*

The Key Manager Storage is the submodule of the Key Manager. The KMS stores the keys of all the files of the user space file system SADUS in the form of ciphertexts. The file encryption key is generated by the KM and stored by the KMS. We use the encryption method to store the SADUS configuration file and the file keys to ensure that the file ciphertext and the key ciphertext cannot be recovered without knowing the KMS password. The keys of the files are encrypted and stored by the master key of the KM. When the file is deleted, the file and its corresponding encryption key are unlinked.

Fig. 5 shows the change when deleting the cipher file key in the KMS. When SADUS receives a command to delete a file, for example, $File_1$, it first finds K_1 based on the ID_1 of $File_1$; then, it overwrites the K_1 cipher, forcing the Garbage Collection process to copy the original block contents to the newly erased block. Finally, it deletes K_1; thus, the cipher of K_1 cannot be indexed from the KMS.

When the file system is mounted, the KMS file is opened, and the password for the KMS database file needs to be provided. The storage layout is in the form of a B-tree for ease of operation. When modified, the contents of the original leaf node are copied to the new leaf node; then, the original leaf node is discarded. We need to define the correct storage layout and have the following three properties: (1) All file key cipher text is mapped through the file name corresponding to a unique ID. (2) The ciphertext of the file key stored in the KMS can be mapped to the ID generated by the file name and is unique. (3) We overwrite the key ciphertext before deleting it. When the key ciphertext of the deleted file is overwritten, the

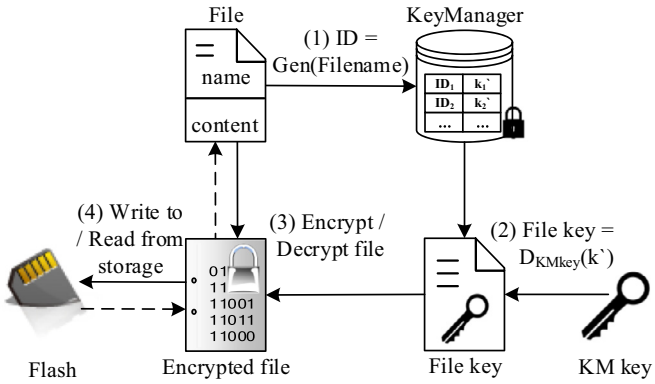


Fig. 4 – SADUS secure write and read process.

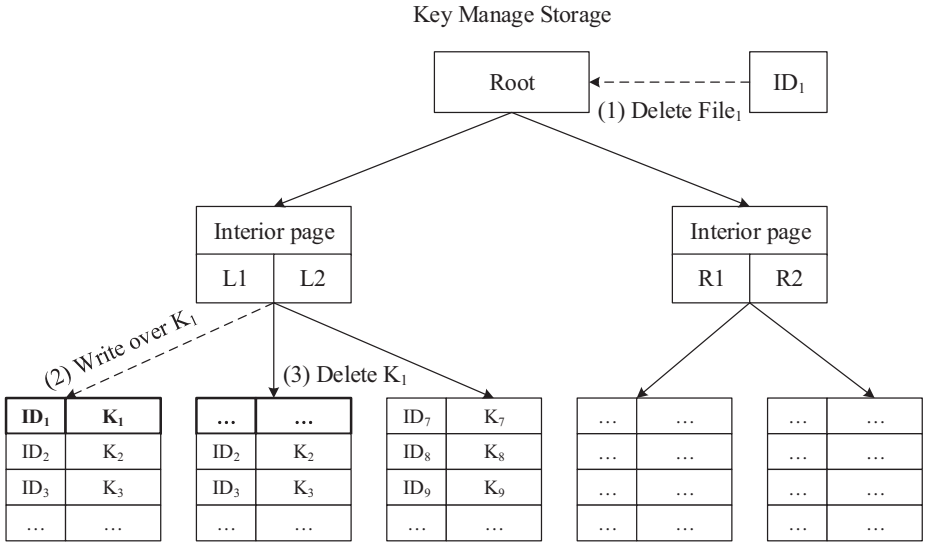


Fig. 5 – Delete the file key from the KMS.

block that originally stored the key ciphertext will no longer be used.

4.1.4. *Purging*

Purging is a program running in user space to clean garbage. The operation is designed to ensure that 1) it recycles all obsolete flash pages. 2) Whenever a strong adversary forces the user to hand over the SADUS password, the purging module deletes all file keys and recycles the memory space occupied by the KMS, making all file irreversible for the attacker.

The SADUS file system is a user space file system such that users do not need to reinstall a new system. Therefore, purging is designed without relying on the underlying hardware to provide the secure deletion in user space. User-level solutions can only create, modify, and delete the user's individual local files. However, such approaches allow the users themselves to achieve secure deletion without voiding warranties or relying on the hardware manufacturer to provide secure deletion. The purging operation complies with the following principle: It reduces the file system's available free space to encourage more-frequent garbage collection. Purging consists of filling the storage medium to capacity, thereby erasing the deleted keys and cipher text. As described in Algorithm 1, the purging operation can be triggered by typing an incorrect KMS password or by the file interface provided by SADUS such as when a strong opponent obtain an incorrect password or when an application with a high security level exits, triggering the cache being cleared.

The user can completely fill the storage medium and is not limited by the size of the storage medium; then, most of the erasable blocks of the storage medium are reclaimed. The storage medium is able to fully fill the storage medium based on the underlying file system pre-allocation policy of the erase block. Before the storage medium is completely filled, there remain some areas that contain data that can be obtained, and the data may be the ciphertext of the deleted key. Once the master key is exposed, these data may be cracked; however, the deleted content's location must also be known. We should be concerned with the fact that purging securely deleted files relies on the underlying file system implementation, in particular, we require the following: If the file system informs the algorithm that the storage space has run out, the previously

Algorithm 1 Data purge operation in different modes.

Require:
The master password (P_m) and the checked password (P_c) entered by the user;
The true password (P_t) and the false password (P_f) stored in the KMS.

Ensure:
Remove the invalid physical page/block, making the obsolete data unrecoverable.

```
Initialize the file system and input P_m and P_c;
Obtain the P_t and P_f from the KMS;
if P_c == P_t then
    User space file system mounts normally;
    Obtain the SADUS mount times (T_m) and the unmount times (T_u) from the key storage area;
    if T_m > T_u then
        Call the garbage collection module and recycle wasted pages;
    end if
else
    if P_c == P_f then
        The checked password is equal to the pre-set false password, performing the purging operation;
        Return(SUCCESS);
    else
        Re-enter the P_c;
        Return(FAILURE);
    end if
end if
Recycle all the obsolete pages;
Return(SUCCESS);
```

deleted ciphertext is no longer stored on the storage medium. This situation holds for the Yaffs file system and the Linux FTL, but there are differences on other underlying file systems and hardware FTL algorithms.

Purging is strongly affected by multi-threading; the purging operation needs to continuously fill the storage medium until the storage device reports a lack of free storage space. When the storage medium is full, this ensures that all previously deleted files are not recoverable. Another concern is that when the storage medium is full, other applications writing application data will report that the storage medium has no space available. However, the purging operation is triggered because the password is leaked; therefore, there is no space available for other applications and thereby can be ignored. In addition, the user actively triggering purging has an explicit understanding of the purging feature.

According to the user space method proposed by Reardon et al. (2012b), it is possible to continue the garbage filling process by maintaining a certain storage capacity. We will describe a purging experiment in the following sections to show that the contents of the previously deleted files are not recoverable when purging is running.

We defined the threat model in Section 3.2; from file creation to file deletion, the file is encrypted by the corresponding unique key, and the key is encrypted by the KMS master key, thereby ensuring the confidentiality of the file. When deleting a file, overwriting encrypted keys ensures that the deleted data cannot link to their own key, ensuring that the deleted file cannot be recovered. The purging operation initiates triggering of the GC process to recycle the discarded flash pages, therein thoroughly sanitizing the flash medium. As a result, an attacker may possess stored content at any time but only ciphertexts and not keys. A strong opponent cannot decrypt and recover stored content that has been deleted, thus enabling secure storage and deletion. Overall, SADUS provides secure storage and deletion to confront an attacker with limited computational resources.

5. Security analysis

In this section, we show that SADUS can delete sensitive data and remove them from the medium permanently, thereby ensuring that the data cannot be recovered and is therefore securely deleted. Let Ψ be the sensitive message, we say that Ψ is securely deleted if the adversary cannot recover the Ψ when performing peek-coercion-recovery attacks in the data's lifetime.

Peak attack. In Fig. 2, the peek attacker allows the attacker to peek into the content of the storage medium at some point(s) in time prior to compromise the storage medium. the peak attack occurs before Ψ is created, revealing neither the encrypted data nor the encryption key. The peek attack wants to take other sensitive information storing on the medium. However, all data are encrypted before persisting to storage, what the adversary peaks are the ciphertext just like meaningless random number.

Coercion attack. The coercion attack occurs during the operations of Ψ; when the attacker obtains a snapshot of the flash memory between t_1 and t_4, Ψ is encrypted and saved to the flash memory. Moreover, the encryption key is encrypted and stored in the KMS. When performing the delete operation on Ψ, its encryption key is securely deleted; the Ψ that the key was used to encrypt is then inaccessible, even to the user. For a computationally limited attacker, (s)he cannot decrypt the KMS and link Ψ with its corresponding key, and the adversary cannot obtain more knowledge except for some randomness. Additionally, if the adversary can coerce the device's owner to hand over keys, the user can deliver an incorrect password to the attacker, which will trigger the purging operation, resulting in all the file keys in the KMS being deleted and the encrypted content being removed.

Recovery attack. A recovery attack occurs after the file is completely deleted. Note that the purging operation has been performed before the attack, namely, the key as well the obsolete Ψ are removed; therefore, the adversary obtains nothing about Ψ. Although both the device and the password have been compromised, we present a fake password in SADUS. When the adversary enters the fake password, this will trigger the purge module and clear all the encryption keys and the encrypted data, leaving only garbage content.

SADUS provides guaranteed secure data deletion for mobile devices against a computationally limited peek-coercion-recovery attacker. When an encryption key is securely deleted, the data that it encrypts become inaccessible, even to the user. Then, all invalid data and obsolete encryption keys are

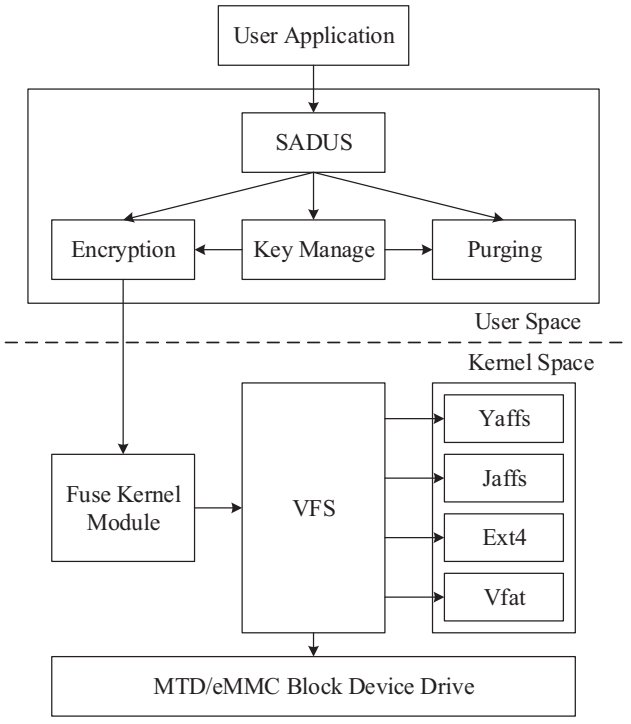


Fig. 6 – SADUS modules.

securely deleted during the purging operation. Thus, the adversary observes nothing but some randomness.

We can conclude that the adversary cannot have access to any data after D in Ω that has been deleted before t_a, and by observing the flash memory at time t_b, the adversary cannot recover a deleted file after G in Ω. Thus, $p \to 0$ holds.

6. Implementation and evaluation

6.1. *Implementation*

We describe the implementation details of SADUS. SADUS is an extended version of EncFs used to achieve secure storage and deletion for flash devices for mobile terminals. SADUS allocates a 256-bit key for each file, encrypts file blocks with a file size of 512 to 4096 bytes in CBC mode, and generates MAC values for each file block. The MAC value is stored in the block of each file and is used to verify the integrity of the file. The initial encryption vector for each block is generated by the file name and file block index. Our method requires an extra 96 bytes of storage for each file, although there is a compromise option, using the same directory encryption for files in the same directory, as discussed in Section 4.1. SADUS, shown in Fig. 6, consists of several components, and their detailed explanations are as follows.

6.1.1. *Encryption*

We implement encryption by EncFs, which provides an encrypted file system in user space. The algorithm runs in user space, therein using the FUSE library for the file system interface. EncFs encrypts individual files by translating all requests for the virtual EncFs file system into the equivalent encrypted operations on the raw file system. As described in Section III, EncFs uses robust algorithms to encrypt file names and contents to ensure file confidentiality and integrity.

6.1.2. *Key manager*

The KM provides a series of management functions such as generating a random key, creating a file and key and mapping them, encrypting and decrypting the key of each file, and triggering a purge operation. When the SADUS file system is created, the Key Manger creates a database file that stores the encrypted key and the configuration for SADUS. When the user actively triggers purging or for a password input error, the purge operation will clear all files, garbage filling is performed, and the pages occupied by deleted files and keys are recovered.

The Key Manager Storage is a submodule of the Key Manager. The KMS is a data file that stores the ciphertexts of the SADUS profile and the mapping relationship between the file name and the corresponding key. Consistency between the key and file mappings is essential to file confidentiality and secure deletion in the SADUS file system. When the purging operation is actively triggered, the deleted files and keys are cleared, and the mapping of other files is not covered.

We now describe how to create the mapping between the file and its keys. When the file system is mounted, the EncFs password and the KMS password are needed. The EncFs password is used to encrypt the master key of the EncFs, which is used to encrypt the file path and configuration file. The KMS password is used to manage the keys of the files. When the file is created, the key corresponding to the file is created by the KM, and the mapping of the file name is established. The key of the file is encrypted and stored in the KMS. For access to the file, the KM decrypts the file key from the KMS and then applies it to EncFs. When the file is deleted, the KM will overwrite the file's key and ciphertext, forcing the original block to become the new block. The original key ciphertext is used when the erased block is not being indexed by the KMS; then, the file key and file are deleted. Therefore, even when the password is known, the file key cannot be restored because the ciphertext cannot be indexed.

6.1.3. *Purging*

The purpose of purging is to trigger the garbage collection process and recycle the pages of the deleted files and their corresponding ciphertext, ensuring that the files are completely deleted and cannot be recovered. Purging has two triggering modes. One mode is initially triggered by users, and the other mode is passively triggered by the strong adversary. When removing a more sensitive file from the storage device, the user can actively trigger purging to force the deleted file to be reclaimed. Because SADUS uses the password to decrypt the EncFs and the file key in the KMS, it is possible to hand over the password when encountering a strong adversary. In this case, when the user enters an incorrect password, the KMS performs mandatory removal after purging; therefore, all the keys are completely deleted under the SADUS file system, ensuring that the files cannot be recovered.

SADUS effectively ensures file integrity and confidentiality. SADUS allocates a unique key for each file to provide

Table 1 – Disk I/O and file system performance of SADUS compared to EncFs. Benchmark results on an unencrypted device are also presented as a baseline.

	No encryption	EncFs		SADUS		
			Overhead vs.		Overhead vs.	
Operation	Performance	Performance	No enc.	Performance	No enc.	EncFs
Read	30720.00 KB/s	11673.60 KB/s	60.01%	11264.00 KB/s	63.33%	**3.51%**
Write	2252.80 KB/s	1945.60 KB/s	13.64%	1925.12 KB/s	14.55%	**1.05%**

fine-grained secure storage and deletion. The storage space is used effectively, and the amount of additional storage required to store keys is much lower than the file. The two triggering modes of purging ensure that files are deleted in a completely irreversible manner.

6.2. *Experimental evaluation*

We have ported SADUS to a rooted Android phone and created a directory for SADUS as the system directory. In this section, we conduct experiments with the Android phone and a simulator.

Because SADUS encrypts not only the files but also the keys, it can easily cause serious resource consumption issues and result in intolerable delays. We therefore experimentally evaluate the throughput performance, power consumption, and computing time on an Android phone and compare with the EncFs and Yaffs2 file systems. EncFs is an encrypted file system running in user space, whereas Yaffs2 is the initial file system of the phone, which is specifically designed to be fast, robust and suitable for embedded use with NAND and NOR flash storage. Our experiment measured the overhead of our system: the extra power consumption, the throughput performance, and the resource consumption of the purging module. We used a simulator to simulate the file lifecycle. For the basic file operations, we used the Android phone to measure the performance. We further measured the time consumption of both read and write operations caused by the EncFs and Yaffs2 file systems.

6.2.1. *Throughput performance*

To observe how SADUS impacts the I/O performance of the underlying storage device, we first put our system under stress using the *dd* command; we configure the *dd* command to repeat the writing and reading of a 100 Mb file. All tests are also repeated using EncFs. Although our experiment focuses on comparing SADUS with EncFs, we also provide benchmark results obtained without running either as a baseline. The results are shown in Table 1.

The results reveal that when performing reads and writes on a small number of large files, SADUS achieves similar performance to EncFs, with the overhead remaining as 3.51% and 1.05%, respectively, as expected. Once SADUS obtains the encryption key for the processed file, the remaining task of encrypting and decrypting the file blocks on the fly is nearly identical to how EncFs performs files encryption.

We also present the throughput performance results for sequential read and write operations with different file sizes. We also used the *dd* file operation command to calculate the throughput of sequential read and write operations under the Yaffs2, EncFs, and SADUS file systems.

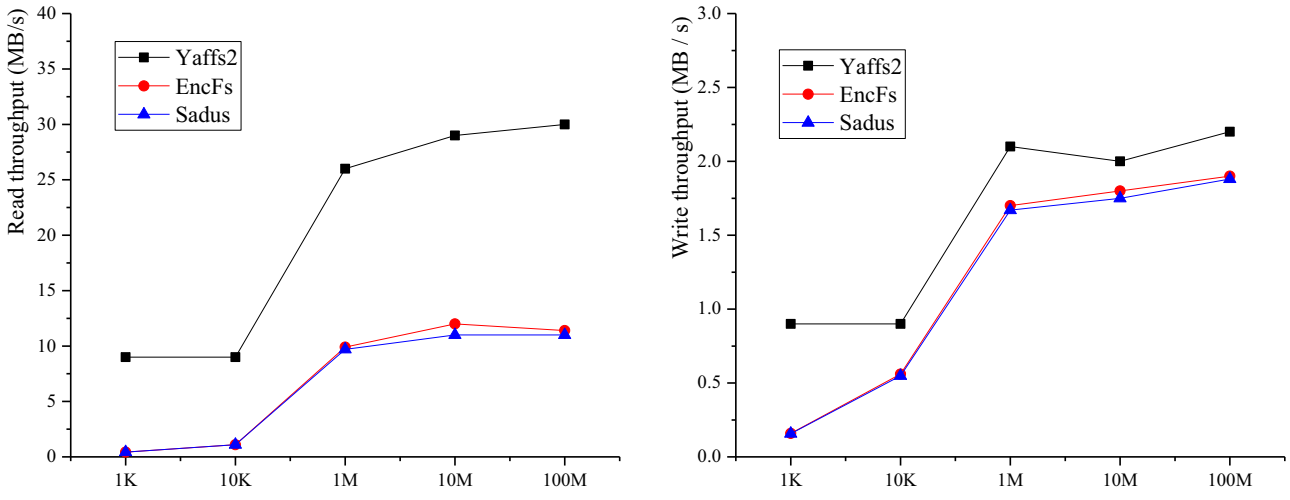
Fig. 7 compares the throughput for two typical file I/O operations, namely, sequential read and write. The experiments are run on the original Yaffs2, EncFs and SADUS file systems. Fig. 7(a) illustrates that the encrypted file system throughput is significantly lower than the non-encrypted file system throughput. Because the file size is too small, the initialization time of the *dd* test program affects the test results. When the file size is larger than 1 Mb, because EncFs and SADUS use the same configuration decryption algorithm, the file read throughputs are similar. Fig. 7(b) illustrates that the throughput rate of the encrypted file system for the sequential write operation presents an obvious decrease compared to the sequential read operation, typically reduced by 15%. Under the same configuration of the encryption algorithm, the write rate of SADUS is slightly lower than that of EncFs because the former uses different keys for each file, and the keys need to be initialized when performing a write operation. Our analysis shows that encryption/decryption contributes an overhead and presents the expected trade-off between security and performance, which is acceptable for users.

6.2.2. *Power consumption*

To test if our system has acceptable power consumption, we analyzed the power consumption of write and read operations. We disabled other applications, but some power consumption is required to maintain the basic functionality of the system. We record the current phone power, denoted as m_1, start the timing, and keep the phone idle for t minutes. We then denote the later phone power as m_2; therefore, the average idle power consumption per minute for mobile phones is $(m_1 - m_2)/t$.

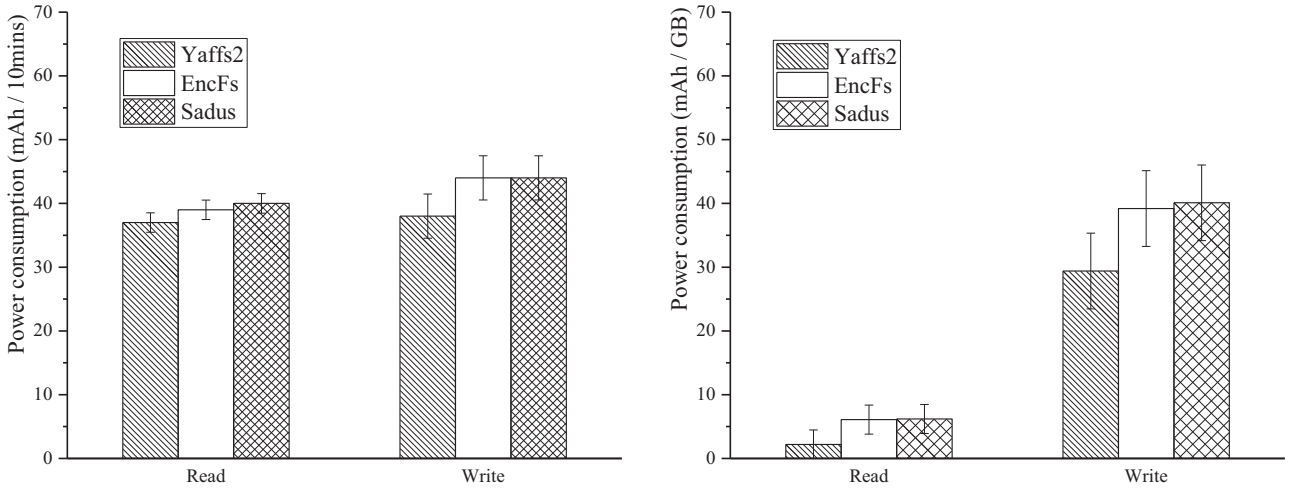
To measure the power consumption, we repeatedly read a large file and mount the file system in read-only mode. Then, we re-mount the file system to be read again. The purpose of this is to clear the cache of the underlying file system. We used the dd file operation command to read random characters from the device file */dev/urandom* and then write to a file. When the file is full, we record the time and write the same file again. Each read and write operation lasts 10 minutes. The statistical results are summarized in Fig. 8.

From Fig. 8, the battery consumption is relatively similar on the three different file systems when read and write operations last for 10 min. However, EncFs and SADUS involved encryption and decryption operations; when running an application in user space, they need to consume more power per GB. Compared with EncFs, SADUS included an additional



(a) Read operation throughput (b) Write operation throughput

Fig. 7 – SADUS throughput performance.



(a) Per 10 minutes (b) Per GB

Fig. 8 – Power consumption in three different file systems.

initialization operation for accessing the keys; therefore, they have almost the same power consumption.

6.2.3. *Time analysis*

SADUS provides the same encryption/decryption algorithm and key length, including the purging and Key Manager modules. We modify the encrypted file systems EncFs and SADUS to record the time needed to initialize the encryption/decryption algorithm and the key as well as perform the encrypt and decrypt operations. We record the time required to generate the file key, read the key, and perform encryption and decryption on the same flash page.

Table 2 shows presents no differences in performance, but SADUS prepares a different key for each file, and the key needs to be initialized when the file is read. This can also explain the similarities in the sequential read and write throughputs of the two encrypted file systems in Fig. 7.

Table 2 – Time consumption for four different operations.

Operation	EncFs	SADUS
Algorithm initialization	0.05 ms	0.05 ms
Key initialization	/	0.07 ms
Encryption	0.91 ms	0.91 ms
Decryption	0.92 ms	0.92 ms

6.2.4. *Purging analysis*

We first analyze the effectiveness of the purging operation. We take a snapshot of the storage device before the experiment,

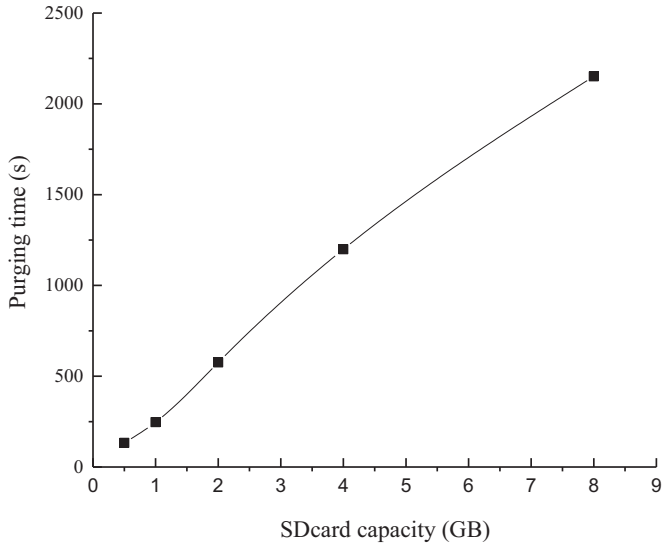


Fig. 9 – SADUS purging time.

and then, we copy the raw snapshot to an external memory card. We write a random string to the storage medium that is unique on the storage device; then, we take a snapshot of the storage device again to ensure that the random string exists on the storage device. We then delete the string and take the snapshot of the storage device again to confirm that the character remains in the storage device. Next, we perform persistent garbage filling on the storage device until its capacity is reached.

We test flash storage devices with different capacities. Assuming that the devices are almost empty, where the total purging time is shown in Fig. 9; this figure demonstrates that the purging time is proportional to the flash storage capacity. Because the purging time is equal to the flash write speed multiplied by the size of the file being deleted. When the flash write speed is Almost constant, the purging time is proportional to the flash capacity. Unless a large number of file write operations are needed, deleting the content of the deleted file will take a long time. Purging in user space forces the garbage collection operation to be performed.

The main drawback of purging is in increasing the extra wear suffered by the storage medium because the user initiates purging operations, thereby increasing the number of block erasures. However, we conservatively estimate that each erase block can be erased approximately 10^4 times, and a typical flash block erase number is between 10^4 and 10^5. We estimate the flash media lifetime as being up to almost 10 years, which is far longer than the average time to replace a phone. Therefore, the purging operation is certainly acceptable to users.

6.3. *Optimization*

To achieve better performance, we optimize SADUS. The first parameter that we tuned is the file system block size. EncFs supports block sizes of 64–4096 bytes, while 1024 bytes is the default. We create SADUS and use 4096 bytes as the block size; we also use the Director-IO mechanism to gain an increase in performance of 15% for sequential read. In addition, we employ a caching strategy for the KM so that the keys for frequently accessed files are available in memory. A dedicated kernel thread periodically synchronizes dirty cache entries to their disk blocks and evicts old cache entries. Having a cache, as opposed to always reading the keys from the disk, results in a significant performance gain. Moreover, we improve our scheme in the following two aspects.

6.3.1. *File key distribution policy*

In our scheme, we assign a random key to each file, which is encrypted and stored in the Key Manager Storage area. We can distribute the key according to the file sensitivity. By providing an interface, users can customize the encrypted file password, which would be utilized to generate the file key. Unlike the KSA policy, we can reduce the number of key assignments through insensitive files using the same key or choose to not encrypt the file to improve throughput of the read and write operations.

6.3.2. *Purging policy*

The timing of a purge is entirely controlled by the user, and it is a continuous operation. Under a strong opponent, the garbage fill purge time is limited. We use a hybrid approach to enhance the situation. After the file system is mounted, when the system is idle, garbage filling is performed, ensuring that the available space of the storage medium is constant. When triggering the purging operation, garbage collection is performed at the highest speed possible so that users are more confident that the deleted files cannot be recovered.

6.4. *Discussion*

6.4.1. *Prior purging-based secure deletion work*

As mentioned in Section 4.1, Reardon et al. (2012b) implemented a user-level secure deletion on log-structured file systems that utilized file overwriting called purging and ballooning. We also include a purging module in SADUS. However, a

significant difference between their prototype and SADUS lies in our focus on encrypting the files and managing the file keys, and the purging module is an enhancement module for the SADUS of secure data deletion.

First, although the adversary in Reardon's scheme can acquire a full copy of the flash image, the deleted data cannot be recovered. The attacker can access the data on the flash image because these data are not encrypted. In our scheme, SADUS can resist a peek-coercion-recovery attacker. Even when the attacker has acquired a full copy of the flash image, what (s)he obtains is simply some randomness for a computationally limited attacker because the files have been encrypted. Compared with Reardon et al. (2012b), SADUS not only ensures that the deleted data cannot be restored but also guarantees the safe storage of the data.

Next, Reardon's work leverages purging, which ensures that all data are deleted, and ballooning, which reduces the expected deletion latency for the deleted data. However, those two operations are explicitly executed across the entire flash memory. In SADUS, because the data are encrypted, the secure deletion operation is directed to the KMS area. Because the KMS area is relatively small, there is no need for a full-scale purging operation. When the file is deleted, SADUS unlinks the index between the file and the corresponding key, thereby achieving the goal of secure deletion for rivals with less computing power. The purging module is an enhancement module for SADUS against an attacker without a limited computing capacity for removing obsolete data permanently. The module runs when the phone is idle, clearing all discarded pages, including obsolete files and discarded keys.

6.4.2. *File deletion granularity*

In SADUS, we implement the secure data deletion process with file-level granularity. However, we note that file-level secure deletion is not necessarily the optimal granularity in all cases. For instance, Android applications may store and manipulate all their data in a single, file-backed database; in such cases, encryption and secure deletion at the database-entry granularity may be a more appropriate approach (Spahn et al., 2014). We leave exploration of this venue as future work.

7. Conclusion

In this paper, we presented SADUS, a flexible and novel encrypted file system running in user space. SADUS is able to provide effective security storage and guaranteed data removed for mobile flash devices by (1) using different keys to encrypt individual files against computationally limited peek-coercion-recovery attackers and (2) purging all available space to guarantee that the keys and ciphertext cannot be recovered when encountering strong adversaries. SADUS protects the files with fine granularity, where every file has a unique key. After deleting a file, SADUS unlinks the file and its content, deletes the corresponding key and performs a purge operation to guarantee that the file is deleted securely.

Experiments with our prototype implementation show that SADUS has an acceptable computing time and that the wear on the flash device is minimized. Our security analysis demonstrates that our method can defend against passive and active adversaries. SADUS is easy to install and configure across all Android platforms, including mobile phones, tablets, and small notebooks, without any user-perceivable delay for most typical Android applications. Indeed, we expect that SADUS will improve the efficiency, flexibility, and security of state-of-the-art alternatives for data storage and deletion.

Acknowledgments

We would like to thanks the reviewers for their careful reading and useful comments. This work was supported by the National Natural Science Foundation of China (61671360,61672415), the Key Program of NSFC-Tongyong Union Foundation under Grant (U1636209),the Key Program of NSFC Grant (U1405255), the Natural Science Basic Research Plan in Shaanxi Province of China (2017JM6082) and the China 111 Project (B16037).

REFERENCES

Albano P, Castiglione A, Cattaneo G, De Santis A. A novel anti-forensics technique for the android OS. Proceedings of the international conference on broadband and wireless computing, communication and applications (BWCCA). IEEE; 2011. p. 380–5.

Bajaj S, Sion R. Ficklebase: Looking into the future to erase the past. Proceedings of the IEEE 29th international conference on data engineering (ICDE),. IEEE; 2013. p. 86–97.

Bauer S, Priyantha NB. Secure data deletion for Linux file systems.. Proceedings of the USENIX security symposium, 2001.

Ben, G, (2009). Diskscrub. https://sourceforge.net/projects/diskscrub.

Bitlocker, (2004). Encryption, Bitlocker overview. https://technet.microsoft.com/en-us/library/cc732774.aspx.

Boneh D, Lipton RJ. A revocable backup system.. Proceedings of the USENIX security; 1996. p. 91–6.

Chang B, Wang Z, Chen B, Zhang F. Mobipluto: File system friendly deniable storage for mobile devices. Proceedings of the 31st annual computer security applications conference. ACM; 2015. p. 381–90.

Czeskis A, Hilaire DJS, Koscher K, Gribble SD, Kohno T, Schneier B. Defeating encrypted and deniable file systems: Truecrypt v5. 1a and the case of the tattling OS and applications. Proceedings of the HOTSEC, 2008.

Diesburg S, Feldhaus CA, Fardan MA, Schlicht J, Ploof N. Is your data gone? Measuring user perceptions of deletion. Proceedings of the 6th workshop on socio-technical aspects in security and trust. ACM; 2016a. p. 47–59.

Diesburg S, Meyers C, Stanovich M, Mitchell M, Marshall J, Gould J, Wang A-IA, Kuenning G. Trueerase: per-file secure deletion for the storage data path. Proceedings of the 28th annual computer security applications conference. ACM; 2012. p. 439–48.

Diesburg S, Meyers C, Stanovich M, Wang A-IA, Kuenning G. Trueerase: leveraging an auxiliary data path for per-file secure deletion. ACM Trans. Storage 2016b;12(4):18.

Diesburg, S. M. (2016). Ghosts of deletions past: new secure deletion challenges and solutions. arXiv:1611.04216.

Diesburg SM, Wang A-IA. A survey of confidential data storage and deletion methods. ACM Comput. Surv 2010;43(1):2.

EncFS, (2008). Vgough, EncFS: an encrypted filesystem. https://vgough.github.io/encfs.

GNU, (2012). Shred: remove files more securely. http://www.gnu.org/software/coreutils/manual/html_node/shred-invocation.html.
Gutmann P. Secure deletion of data from magnetic and solid-state memory. Proceedings of the sixth USENIX security symposium, san jose, CA, vol. 14; 1996. p. 77–89.
Hohmann, B.C., (2003). CryptoFS. http://http://reboot.github.io/cryptofs.
Huang N, He J, Zhao B. Secure data sanitization for android device users. Int. J. Secur. Appl 2015;9(5):61–8.
Jia S, Xia L, Chen B, Liu P. NFPS: Adding undetectable secure deletion to flash translation layer. Proceedings of the 11th ACM on asia conference on computer and communications security. ACM; 2016. p. 305–15.
Joukov N, Papaxenopoulos H, Zadok E. Secure deletion myths, issues, and solutions. Proceedings of the second ACM workshop on storage security and survivability. ACM; 2006. p. 61–6.
Joukov N, Zadok E. Adding secure deletion to your favorite file system. Proceedings of the third IEEE international security in storage workshop SISW. IEEE, 2005. 8–pp
Lai J, Xiong J, Wang C, Wu G, Li Y. A secure cloud backup system with deduplication and assured deletion. Proceedings of the International conference on provable security. Springer; 2017. p. 74–83.
Lee J, Heo J, Cho Y, Hong J, Shin SY. Secure deletion for nand flash file system. Proceedings of the ACM symposium on applied computing. ACM; 2008. p. 1710–14.
Lee J, Yi S, Heo J, Park H, Cho Y. An efficient secure deletion scheme for flash file systems. J. Inf. Sci. Eng 2010;26(1):27–38.
Leom MD, Choo K-KR, Hunt R. Remote wiping and secure deletion on mobile devices: a review. J. Forensic Sci 2016;61(6):1473–92.
Peters, T. M., Gondree, M. A., & Peterson, Z. N. J. (2015). Defy: a deniable, encrypted file system for log-structured storage. Network and Distributed System Security Symposium.
Peterson ZNJ, Burns RC, Herring J, Stubblefield A, Rubin AD, vol. 5; 2005.
Reardon J. Robust key management for secure data deletion. Proceedings of the secure data deletion. Springer; 2016. p. 143–74.
Reardon J, Basin D, Capkun S. Sok: Secure data deletion. Proceedings of the IEEE symposium on security and privacy (SP). IEEE; 2013a. p. 301–15.
Reardon J, Capkun S, Basin D. Data node encrypted file system: Efficient secure deletion for flash memory. Proceedings of the 21st USENIX conference on security symposium. USENIX Association, 2012a. p. 17–17.
Reardon J, Marforio C, Capkun S, Basin D. User-level secure deletion on log-structured file systems. Proceedings of the 7th ACM symposium on information, computer and communications security. ACM; 2012b. p. 63–4.
Reardon J, Ritzdorf H, Basin D, Capkun S. Secure data deletion from persistent media. Proceedings of the ACM SIGSAC conference on computer & communications security. ACM; 2013b. p. 271–84.
Shin I. Supporting reliable data deletion for NAND-based gadgets with limited memory. Int. J. Appl. Eng. Res 2016;11(9):6381–6.
Simon L, Anderson R. Security analysis of android factory resets. Proceedings of the 4th mobile security technologies workshop (MoST), 2015.
Spahn R, Bell J, Lee M, Bhamidipati S, Geambasu R, Kaiser GE. Pebbles: fine-grained data management abstractions for modern operating systems. Proceedings of the OSDI; 2014. p. 113–29.
Sun K, Choi J, Lee D, Noh SH. Models and design of an adaptive hybrid scheme for secure deletion of data in consumer electronics. IEEE Trans. Consumer Electron 2008;54(1):100–4.
Szeredi, M. (2001). Avfs: a virtual file system. https://sourceforge.net/projects/avf.
Szeredi, M. (2006). Fuse: file system in user space. http://fuse.sourceforge.net.
Tang Y, Ames P, Bhamidipati S, Bijlani A, Geambasu R, Sarda N. CleanOS: limiting mobile data exposure with idle eviction. Proceedings of the OSDI; 2012. p. 77–91.
Tang Y, Lee PPC, Lui JCS, Perlman R. Fade: Secure overlay cloud storage with file assured deletion. Secur. Privacy Commun. Netw 2010:380–97.
Teufl P, Fitzek A, Hein D, Marsalek A, Oprisnik A, Zefferer T. Android encryption systems. Proceedings of the international conference on privacy and security in mobile systems (PRISMS). IEEE; 2014. p. 1–8.
Trim. (2008). https://en.wikipedia.org/wiki/Trim_(computing).
Vnodes SRK, et al. An architecture for multiple file system types in sun UNIX. Proceedings of the USENIX summer conference proceedings; 1986. p. 238–47.
Wang Z, Murmuria R, Stavrou A. Implementing and optimizing an encryption filesystem on android. Proceedings of the IEEE 13th international conference on mobile data management (MDM). IEEE; 2012. p. 52–62.
Wei MYC, Grupp LM, Spada FE, Swanson S. Reliably erasing data from flash-based solid state drives. Proceedings of the FAST, vol. 11; 2011. p. 105–17.
Zarras A, Kohls K, Dürmuth M, Pöpper C. Neuralyzer: flexible expiration times for the revocation of online data. Proceedings of the sixth ACM conference on data and application security and privacy. ACM; 2016. p. 14–25.

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, Xin, 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.

Teng Wei received the Bachelor's degree in computer science from Xidian University in 2011. He is Pursuing the master's 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 Stavrou's 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.