Efficient Page-cache Encryption for Smart Devices with Non-volatile Main Memory

Kenichi Kourai
Kyushu Institute of Technology
kourai@ksl.ci.kyutech.ac.jp

Naoto Fukuda
Kyushu Institute of Technology
naoto@ksl.ci.kyutech.ac.jp

Tomohiro Kodama
Kyushu Institute of Technology
kodama@ksl.ci.kyutech.ac.jp

ABSTRACT

Since smart devices such as smart phones and tablets are at high risk of theft, they prevent information leakage from storage by full disk encryption (FDE). Recently, however, information leakage from memory is being critical as non-volatile main memory (NVMM) is emerging. In smart devices with NVMM, even if storage is encrypted, sensitive data in the page cache can be stolen by physical attacks. In this paper, we propose efficient page-cache encryption called *Cache-Crypt* to prevent part of the storage data from leaking via the page cache. Cache-Crypt is well integrated with FDE to avoid redundant cryptographic operations and take advantage of FDE. It bypasses encryption and decryption by FDE and stores encrypted data in the page cache. In response to application's requests, it executes encryption and decryption using FDE's cryptographic mechanism. To reduce the overhead, Cache-Crypt defers re-encryption of decrypted data temporarily. We have implemented Cache-Crypt in Linux and confirmed that the performance was comparable to that in FDE only.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; *Mobile platform security*; • **Software and its engineering → File systems management**;

KEYWORDS

Page cache, Full disk encryption, Non-volatile memory

ACM Reference format:
Kenichi Kourai, Naoto Fukuda, and Tomohiro Kodama. 2018. Efficient Page-cache Encryption for Smart Devices with Non-volatile Main Memory. In *Proceedings of SAC 2018: Symposium on Applied Computing, Pau, France, April 9–13, 2018 (SAC 2018),* 7 pages.
https://doi.org/10.1145/3167132.3167251

1 INTRODUCTION

Recently, smart devices such as smart phones and tablets are widely used. The shipments of Android devices are over one billion units in 2015 and are being increasing [10]. Since smart devices store a larger amount of more sensitive information than traditional cellular phones, users suffer from large damages on device theft. For example, personal information such as passwords and credit card numbers and customer information such as phone numbers and e-mail addresses can leak. In addition, smart devices are at higher risk of theft than laptop PCs because they are much smaller and more lightweight. It is reported that 3.1 million devices were stolen in the U.S. in 2013 [3]. In case of device theft, the operating systems for smart devices usually enable full disk encryption (FDE) to prevent information leakage from storage.

Recently, however, information leakage from memory is being a new threat as non-volatile main memory (NVMM) is emerging [2, 15, 18]. NVMM is next-generation non-volatile memory and the replacement of DRAM with high power consumption is expected. Since NVMM can keep data without a power supply, attackers can steal sensitive information from NVMM more easily than from current volatile main memory by physical attacks. Although NVMM is not widely used yet, the cold boot attack [1] has been proposed against DRAM. This attack cools DRAM modules to preserve data, resets the device, boots attacker's system, and steals data left in memory. A similar attack has been reported for Android devices [11]. In the main memory, most of the operating systems maintains the page cache to make file access faster. Therefore, attackers can steal part of the storage data in the page cache even if storage is fully encrypted.

This paper proposes *Cache-Crypt*, which efficiently encrypts the page cache in memory to prevent information leakage by physical attacks. Since FDE is already used in smart devices, Cache-Crypt is well integrated with FDE to avoid redundant encryption and take advantage of FDE. Upon file reads, it bypasses decryption performed by FDE and directly stores encrypted data in the page cache from storage. It decrypts data in the page cache using FDE's decryption mechanism only when applications read files. Upon file writes, Cache-Crypt encrypts data written by applications using FDE's mechanism and stores it in the page cache. The data is directly written back to storage without encryption by FDE. Using Cache-Crypt, the target of information leakage is limited only to the regions being accessed in the page cache.

To reduce cryptographic overhead, Cache-Crypt keeps the page cache decrypted temporarily and defers encryption. With this *delayed encryption*, encryption and decryption are eliminated in the critical paths of the read and write system calls, resulting in higher performance. The delay time is a trade-off between performance and security. We have implemented Cache-Crypt in the Linux kernel, whose variant is used in Android. In the current implementation, Cache-Crypt is integrated with dm-crypt in Linux as FDE. According to our experiment, it was shown that the performance of file access in Cache-Crypt was comparable to that when only dm-crypt was used.

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167251

This paper is organized as follows. Section 2 describes information leakage from the page cache. Section 3 proposes Cache-Crypt and Section 4 explains the implementation details. Section 5 shows experiments on Cache-Crypt. Section 6 describes related work and Section 7 concludes this paper.

2 INFORMATION LEAKAGE FROM THE PAGE CACHE

Smart devices usually enable FDE in case of device theft. FDE is a mechanism for encrypting the entire partitions of storage to protect sensitive data. For example, Android uses dm-crypt in Linux as FDE and encrypts the partition where applications' data is located. FDE decrypts data on disk reads and encrypts data on disk writes. To obtain the master key for encrypting and decrypting data, FDE requires a user to input a PIN or password at boot time. Using the input information, FDE generates a key and decrypts the master key from the generated key. Therefore, unless attackers know user's PIN or password, they cannot steal information in encrypted storage even if they remove storage from stolen devices and attaches it to their devices.

Recently, however, information leakage from memory is being critical as NVMM such as STT-MRAM and NVDIMM is emerging [2, 15, 18]. Unlike traditional volatile memory such as DRAM, NVMM can keep data without a power supply. From this feature, it is expected that NVMM replaces DRAM in smart devices to reduce power consumption. If data used by the operating system and applications is always preserved in NVMM, users could boot the system faster. On the other hand, in case of device theft, attackers could mount physical attacks and steal sensitive data from NVMM more easily than from DRAM. This is because any data in NVMM is not destroyed even if attackers power off the stolen device to prevent the system from erasing sensitive data.

Even for currently used DRAM, the cold boot attack is possible to physically steal information from memory [1]. This attack cools memory modules, forces to reset the device, boots the device using attacker's system, and steals data left in memory. Usually, data in volatile memory is gradually destroyed while a power is not supplied after a device reset, but it takes a certain time before all the data is destroyed. This destruction of memory data can be delayed by cooling memory modules. Unlike normal shutdown, the operating system does not have any time to erase sensitive data in memory because the cold boot attack resets a device suddenly.

For Android devices, the cold boot attack has been also reported [11]. The reported method cools the entire device and thereafter resets the device by removing its battery. Next, it installs attacker's recovery image from a PC via USB and boots the device using that image. When installing the recovery image, the entire data in storage is erased if the boot loader has to be unlocked, but data in memory is not erased. The operating system included in the recovery image can dump data in memory and extract sensitive information such as cryptographic keys.

If the main memory of smart devices is compromised by physical attacks, data in the page cache can be stolen. In most of the operating systems, the page cache is used for making file access faster. When an application reads a file, the operating system first reads file blocks in storage to the page cache. If an application accesses the same file blocks, the operating system can return the data in the page cache immediately. When an application writes data to storage, the operating system first writes the data to the page cache and writes it back to storage later. Therefore, the page cache contains data that is the same as or newer than that in storage. Consequently, part of the storage data leaks even if storage is fully encrypted by FDE. Recently, the amount of memory equipped with smart devices is increasing and most of free memory is used as the page cache. This means that the amount of stolen data tends to increase.

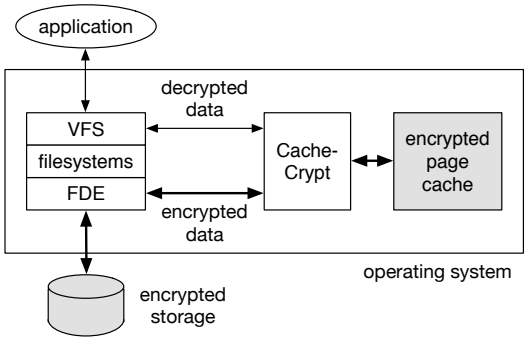


Figure 1: The system architecture of Cache-Crypt.

3 PAGE-CACHE ENCRYPTION

In this paper, we propose efficient page-cache encryption called Cache-Crypt.

3.1 Threat Model

We assume that attackers eavesdrop on the page cache using physical attacks. We do not consider information leakage from the other kernel memory or applications' memory. Data in applications' memory can be protected by several techniques [12, 13]. In addition, we do not consider that malware installed in smart devices accesses decrypted files and sends them to attackers. This type of attacks can be protected by malware detection. Also, we assume that the screen is locked on device theft and that attackers cannot access decrypted files by normal operations.

3.2 Cache-Crypt

Cache-Crypt keeps the page cache encrypted while data in the page cache is not accessed. It decrypts minimal regions in the page cache only when applications access files. As such, it can protect most of the data in the page cache even if attackers can physically eavesdrop on the entire main memory. Specifically, it can limit the target of information leakage only to the regions of the page cache that applications are accessing. Figure 1 illustrates the system architecture of Cache-Crypt. The virtual filesystem (VFS) layer invokes Cache-Crypt and accesses the encrypted page cache. Although there are various filesystems underlying the VFS, Cache-Crypt is independent of individual filesystems.

Cache-Crypt is well integrated with FDE, which is usually enabled in smart devices. The aims are (1) to avoid redundant cryptographic operations by both FDE and Cache-Crypt and (2) to avoid the increase in user's management cost, e.g., key management for both FDE and Cache-Crypt. Cache-Crypt bypasses the encryption and decryption of FDE and executes them using FDE's mechanism with its cryptographic key. Without such integration, the performance of file access to encrypted storage would degrades. Data read from encrypted storage has to be decrypted by FDE and be encrypted by Cache-Crypt to store the data in the page cache. Then, Cache-Crypt has to decrypt the data when applications read it. When encrypted data in the page cache is written back to storage, it has to be decrypted by Cache-Crypt and be encrypted by FDE.

In Cache-Crypt, when data in encrypted storage is read, it is directly stored in the page cache. Decryption by FDE is bypassed at this time. Instead, FDE registers cryptographic information needed for later decryption to Cache-Crypt. That information includes the cipher and decryption key used in FDE and associated sector numbers used only in the block I/O layer. When an application reads the data by issuing the read system call, Cache-Crypt decrypts the data in the page cache using registered cryptographic information and stores the decrypted data in the application's buffer. The application can handle decrypted data stored in its buffer without being aware of encryption by Cache-Crypt. As in FDE only, the number of decryption is only once in Cache-Crypt.

When an application writes data to a file using the write system call, Cache-Crypt encrypts the data and stores it in the page cache. An application is not aware of encryption by Cache-Crypt. If the start and end of the written data are not aligned to the boundary of FDE blocks, Cache-Crypt decrypts only the first and last block because in-between blocks are completely overwritten. To execute such encryption and decryption using FDE's mechanism, Cache-Crypt retrieves cryptographic information from FDE when the page cache is allocated for the data. This is because FDE is not applied yet at this time. When the data in the page cache is written back to encrypted storage, the encryption by FDE is bypassed. As in FDE only, the number of encryption is only once in Cache-Crypt.

Cache-Crypt assumes to protect the cryptographic key of FDE using ARMORED [7]. If the cryptographic key is stored in memory, it can be stolen by physical attacks. As a result, attackers could decrypt data in the page cache using the key. Since ARMORED stores the cryptographic key in CPU debug registers, it can prevent the leakage of the key even under physical attacks. In addition, it performs cryptographic operations using CPU's registers provided for the SIMD extension instruction set instead of memory. Therefore, it can also prevent the leakage of intermediate results of encryption and decryption processing.

3.3 Delayed Encryption

The performance of the write system call is largely degraded by Cache-Crypt because the system call just stores data in the page cache without storage access. In FDE, the encryption of written data is executed asynchronously on writeback to storage. In Cache-Crypt, however, it has to be executed in the critical path of the system call. In addition, excessive encryption and decryption of the same FDE block can occur in Cache-Crypt. For example, when an application reads one FDE block (typically 512 bytes of sector size) little by little, the same block is decrypted many times because Cache-Crypt has to decrypt the entire block using FDE's decryption mechanism to obtain part of its data. When an application writes one FDE block little by little, decryption and encryption are repeated for the same block. Even if an application accesses the entire FDE block at a time, a similar situation occurs when the same block is repeatedly accessed.

To reduce the overhead of such wasteful encryption and decryption, Cache-Crypt defers encryption of the page cache. Once Cache-Crypt decrypts the page cache or stores unencrypted data, it keeps that page cache decrypted for a while. If the page cache is not accessed for a specified period, Cache-Crypt encrypts the page cache asynchronously. As such, the performance of Cache-Crypt is improved because Cache-Crypt does not need to always encrypt and decrypt data when applications read or write files. For example, written data is encrypted later outside the critical path of the write system call. While an application accesses one FDE block continuously, the page cache is not encrypted.

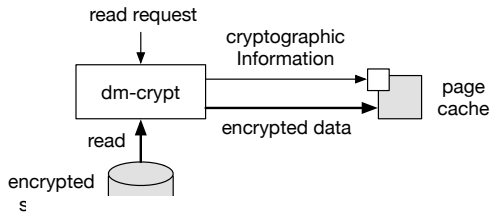
The period for this *delayed encryption* is a trade-off between performance and security. Longer delay can increase performance, but a larger amount of decrypted data in the page cache increases a risk of information leakage in case of device theft. Fortunately, it takes not a short time to mount physical attacks to stolen devices. In particular, the cold boot attack needs to cool device memory. Therefore, security is preserved if the page cache is encrypted before attackers succeed physical attack. To further reduce the risk, Cache-Crypt does not defer encryption while the screen in a smart device is locked, which means that the user does not access the device.

4 IMPLEMENTATION

We have implemented Cache-Crypt in Linux 3.5.7. In the current implementation, Cache-Crypt is integrated with dm-crypt, which is one of the most popular FDE and is also used in Android.

4.1 dm-crypt for Cache-Crypt

We have modified dm-crypt for enabling Cache-Crypt. When dm-crypt receives a read request, it registers cryptographic information to the specified page-cache pages, as illustrated in Fig. 2. The information includes dm-crypt config and sector numbers where the requested data is stored in storage. When encrypted data is read from storage, the callback function of dm-crypt is invoked. Unlike traditional dm-crypt, the modified dm-crypt does not decrypt the data at this time. When metadata of the filesystem is read, dm-crypt decrypts it as usual. Unlike file data, metadata is processed by the VFS and individual filesystems. If metadata were encrypted, Cache-Crypt would have to repeat decryption and encryption. This complicates the implementation and increases the performance overhead. In addition, metadata does usually not include so sensitive information. Cache-Crypt determines that a page-cache page contains metadata when the inode number associated with the page is zero. Each page-cache page contains either only metadata or file data.



Figı

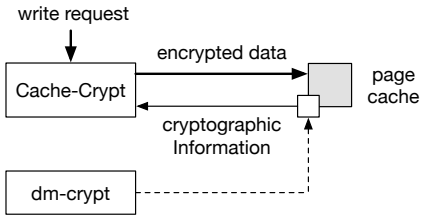


Figure 3: Data writes with Cache-Crypt.

When the modified dm-crypt receives a write request, it just copies each block in the specified page-cache page to the allocated buffer if the block is encrypted. If the block is not encrypted, dm-crypt first encrypts the block and then stores it in the buffer as usual. This encryption is executed only when the page contains metadata or when Cache-Crypt defers encryption of blocks in the page. Then, the buffer is written back to encrypted storage.

4.2 Encryption/Decryption in Cache-Crypt

Cache-Crypt decrypts a page-cache page using the decryption mechanism of dm-crypt. Originally, dm-crypt decrypts data read from storage asynchronously with read requests. In contrast, Cache-Crypt decrypts data synchronously to return decrypted data to the VFS as fast as possible. Since it is invoked to decrypt data that is already stored in the page cache, instead of data in storage, the decryption latency is more critical. To execute decryption, Cache-Crypt uses dm-crypt config registered to the target page-cache page. It also uses registered sector numbers to generate an initialization vector of the cipher algorithm, e.g., AES-XTS. The decryption is done by a sector size of 512 bytes. Cache-Crypt maintains an encryption bitmap for eight sectors in one page. After decryption, Cache-Crypt clears the corresponding bits in the bitmap.

Similarly, Cache-Crypt encrypts a page-cache page using the encryption mechanism of dm-crypt. Since the timing of encryption is controlled by the mechanism of delayed encryption, Cache-Crypt provides a function for synchronous encryption. After encryption, Cache-Crypt sets the corresponding bits in the encryption bitmap. To execute encryption, Cache-Crypt also uses dm-crypt config and sector numbers registered to a page-cache page, as illustrated in Fig. 3. Unlike decryption, however, dm-crypt cannot register necessary information to the page before encryption because the page can be encrypted by Cache-Crypt before dm-crypt is invoked for handling write requests.

Therefore, instead of dm-crypt, Cache-Crypt registers such information when a page-cache page is allocated on file writes. First, Cache-Crypt finds the block device associated to the page and obtains the request queue for it. If the request queue is created for handling requests of the device mapper, Cache-Crypt determines that the page is used for the device mapper. The device mapper is a framework for mapping physical block devices onto higher-level virtual block devices and is used by dm-crypt. Next, Cache-Crypt obtains queue data from the request queue and finds the device-mapper table from the data. Then, it finds a device-mapper target from the table and determines whether the page is used for dm-crypt. If so, Cache-Crypt can obtain dm-crypt config from the device-mapper target. The sector numbers are calculated from the numbers of the blocks contained in the page.

4.3 Delayed Encryption in Cache-Crypt

When Cache-Crypt defers encryption of a page-cache page, it adds the page to the tail of the *delay list*. In the list, the page is linked in the order in which the remaining time to encryption is shorter. If the page is already linked to the list, Cache-Crypt moves it to the tail of the list. After adding the page, it resets the encryption timer by the remaining time of the page at the head of the list. When the timer is expired and the timeout function is invoked, Cache-Crypt starts asynchronous encryption. First, it removes the timed-out page from the delay list and adds it to the *ready list*. Then, it wakes up the encryption thread to execute actual encryption. This is because the timeout function should not execute processing that needs a long time. The encryption thread removes the page from the ready list and finally encrypts it.

4.4 Supporting Memory-mapped File

Cache-Crypt decrypts a page-cache page allocated for a memory-mapped file when a fault for the page occurs at the first access. The page is kept decrypted because an application directly access the page and the operating system cannot mediate later accesses to the page. It should be noted that page-cache pages for a memory-mapped file are not decrypted immediately after the mmap system call is issued. When data in the page is written back to storage, dm-crypt encrypts the data as usual. The page is re-encrypted when it is unmapped by the munmap system call. To reduce the risk of information leakage, Cache-Crypt can periodically remove access permission from a page decrypted for a memory-mapped file and encrypt it. As an initial state, the page is kept encrypted until it is accessed again. However, performance of accessing memory-mapped files degrades.

4.5 Supporting tmpfs

Although Cache-Crypt is independent of filesystems, it supports tmpfs specially. tmpfs is a memory-based filesystem, which is also known as initramfs used at boot time. To enable file data to be preserved in the page cache, tmpfs performs its own management of the page cache. In addition, Cache-Crypt cannot use cryptographic information registered by dm-crypt because files in tmpfs are not read from storage via dm-crypt. Therefore, when a new page is added to the page cache for tmpfs, Cache-Crypt marks the page and applies its own cryptographic operations to the marked page.

5 EXPERIMENTS

We conducted experiments to examine the performance of Cache-Crypt. We measured the performance of file access in Cache-Crypt with or without delayed encryption, independent page-cache encryption with the existing dm-crypt, and dm-crypt only. For the experiments, we have developed page-cache encryption that was not integrated with dm-crypt. Since we could not obtain NVMM, we used traditional SDRAM, assuming NVDIMM-N or STT-MRAM, which will have almost the same performance as DRAM. For ease of experiment, instead of a smart device, we used a PC with Intel Xeon E3-1270v3, 8 GB of DDR3-1600 SDRAM, 1 TB of HDD, and 64 GB of Intel X25-E SSD. We encrypted the SSD with dm-crypt and used the encrypted SSD for measuring performance. The used cipher was AES-XTS and the key length was 256 bits. For the page-cache encryption independent of dm-crypt, we used AES-ECB for the cipher because of an implementation issue. We ran modified Linux 3.5.7.

To measure the performance of file access, we used the fio benchmark. To exclude the impact of asynchronous writeback of dirty pages, we disabled writeback. For Cache-Crypt, we configured sufficiently long delay time for delayed encryption to avoid being encrypted during the experiment. In this experiment, we first invalidated the page cache, created a file, and wrote 2-GB and 1-GB data to the file sequentially and randomly, respectively. Then, we rewrote data to the same file without invalidation of the page cache. In addition, we measured the throughput of the write and rewrite including the issue of explicit fsync. Next, we invalidated the page cache and read the file. Finally, we reread the file, whose data was kept in the page cache. The block size for I/O units was 4 KB and the number of thread was one.

5.1 Throughput

Figure 4 shows the throughput of sequential file access. When page-cache encryption was not integrated with dm-crypt, the performance was degraded by 28-95%, compared with that in dm-crypt only. For read, the performance degradation was 28%. This overhead is relatively small because this file read involved storage access. For write and rewrite with fsync, the degradation was about 50% because encryption was executed in the critical path of the write system call. For write, rewrite, and reread, however, the throughput was degraded by 95%. The reason is that these accesses need only memory access in addition that encryption and decryption were executed in the critical paths.

Using Cache-Crypt, the performance was improved even if delayed encryption was not enabled. For read, there was no performance degradation thanks to the elimination of wasteful encryption and decryption. For write and rewrite with fsync, the throughput was also improved by 20% point. However, the performance was still 30% lower than that in dm-crypt only. For write, rewrite, and reread, the integration with dm-crypt did almost not improve the performance because these accesses were performed only to the page cache and dm-crypt was not invoked.

The performance of Cache-Crypt with delayed encryption was comparable to that in dm-crypt only. For write and rewrite with fsync, the throughput became almost the same as that in dm-crypt only. This is because encryption was eliminated in the critical path.

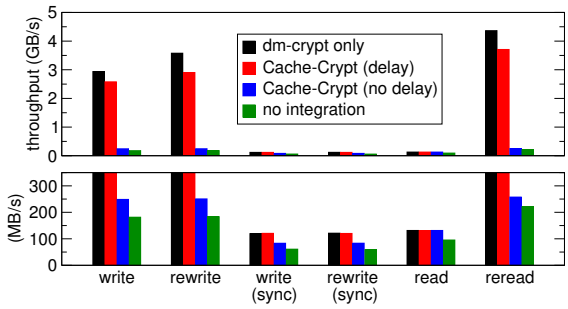


Figure 4: The throughput of sequential file access. The bottom chart magnifies lower part of the top one.

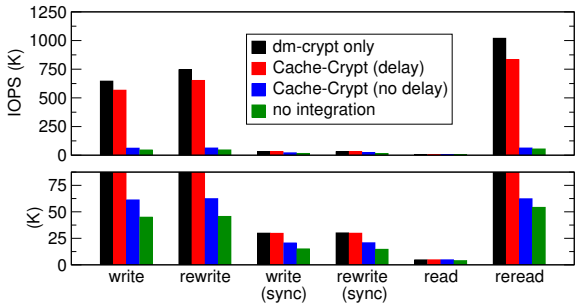


Figure 5: The throughput of random file access.

Even for write, rewrite, and reread, the performance degradation was reduced to 12-19%. Upon write and rewrite, unencrypted data was stored in the page cache without encryption. Upon reread, data in the page cache was already decrypted by the preceding read. The remaining overhead is checking the encryption status of accessed page-cache pages, locking the pages for that check, and resetting the timeout of delayed encryption.

For random access, the result was very similar to that of sequential access except for read access, as shown in Figure 5. The performance of read access was much lower in all cases. Even when page-cache encryption was not integrated with dm-crypt, the performance degradation of read access was only 16% due to low performance of random storage access.

5.2 Latency

Figure 6 and Figure 7 show the latency of sequential and random access. For write and rewrite, the latency was increased by 20 μs, compared with dm-crypt only, when page-cache encryption was not integrated with dm-crypt. This overhead comes from the encryption of one page-cache page. For reread, the latency was increased by 17 μs due to the decryption. Using Cache-Crypt, the increase in latency was still 14 μs. The reason of this improvement is that the encryption and decryption of dm-crypt are more efficient. Delayed encryption in Cache-Crypt suppressed the increase to 0.15-0.24 μs. The remaining overhead is caused by the management of delayed encryption.

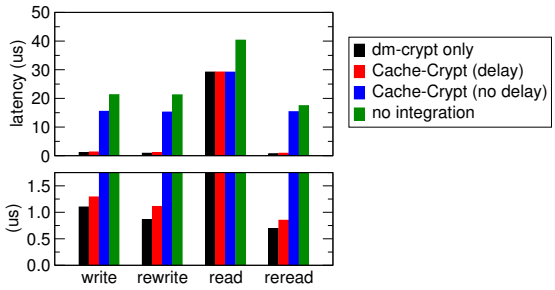


Figure 6: The latency of sequential file access.

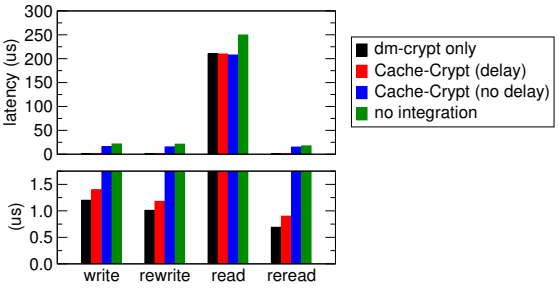
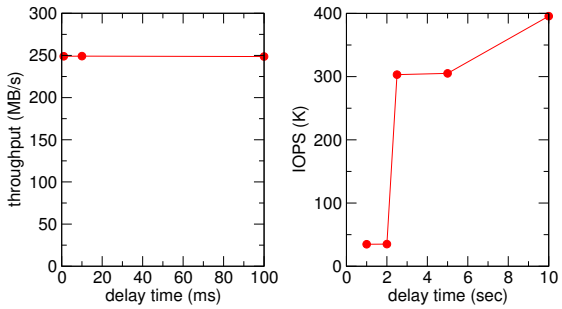


Figure 7: The latency of random file access.



(a) Sequential write **(b) Random read/write**

Figure 8: The performance with delayed encryption.

5.3 Impact of Delayed Encryption

We examined the impact of delayed encryption in Cache-Crypt. First, we measured the performance of sequential write access with asynchronous writeback enabled. As shown in Fig. 8(a), the throughput did not depend on the delay time and was approximately 250 MB/s constantly. This is because the encryption of page-cache pages was performed in parallel with fio. Second, we used a mixed workload of read and write and repeated random access to a 1-GB file 20 times. We reduced the delay time from 10 seconds to 1 second. Figure 8(b) shows the result. When we configured the delay time to 2 seconds, the performance suddenly degraded due to the encryption of page-cache pages to be read. Consequently, the impact of delayed encryption depended on workloads.

6 RELATED WORK

Instead of FDE, encrypted filesystems also decrypt files on disk read and encrypts them on disk write. eCryptfs [9] is an encrypted filesystem overlaid on the existing filesystem and stores encrypted data in the page cache used by the underlying filesystem. However, the page cache used by eCryptfs itself is not encrypted. EncFS [8] allocates its own page cache only while files are opened, but the page cache is not encrypted. On the other hand, TransCrypt [16] provides an optional mechanism for encrypting and decrypting the page cache when it copies data between the page cache and process's buffers. However, it neither consider the overhead nor evaluate it. These approaches require using specific filesystems, but Cache-Crypt allows using any filesystems.

ZIA [4] is a filesystem that decrypts files only when hardware tokens exist near devices. It first decrypts a key used for encrypting files by communicating with user's token and then decrypts files using the decrypted key. If the device is stolen and there is no token near the device, ZIA destroys the decrypted key and starts to encrypt the page cache. However, ZIA does not allow background processing using the filesystem such as mail receipts while users with tokens are far from their devices. Moreover, ZIA cannot protect files when both a device and a token are stolen.

In Keypad [5], users can notice files accessed after devices are stolen and can prevent file access after that. To achieve this, Keypad encrypts each file with a different key and stores keys in the server. Therefore, it cannot decrypt or securely encrypt files when devices are not connected to the network. To overcome this drawback, Keypad requires one more device. Keypad is designed so that the page cache is encrypted, but it is unclear how to handle the page cache because Keypad is implemented using FUSE.

Several mechanisms have been proposed to protect data in application's memory. Swap encryption [14] encrypts pages swapped out to storage and decrypts pages swapped in. Since it encrypts pages with a cryptographic key that is valid for only a short period, data in the swap space cannot be decrypted after process termination. Cryptkeeper [13] is a virtual memory manager that encrypts process' memory. It divides the memory into a small working set and the other and encrypts the latter. It traps access to encrypted pages and decrypts these pages on page faults. If the number of decrypted pages exceeds the upper limit, Cryptkeeper encrypts the page that has been decrypted the earliest. Software-based main memory encryption [12] encrypts data in the store instruction and decrypts data in the load instruction by instrumenting application code. The instrumentation is achieved statically and dynamically without any architectural support. This system provides not only full but also selective memory encryption.

Vanish [6] and CleanOS [17] encrypt sensitive data after a predefined time and store cryptographic keys in the Internet. CleanOS extends Android and stores sensitive data in Java objects called SDO. If SDO is not frequently used, CleanOS encrypts it and stores the used cryptographic key in a cloud. Therefore, when a device is not connected to the network, CleanOS has to keep SDO unencrypted or store cryptographic keys in the same device. These increases a risk of information leakage.

There are many studies on hardware for transparently protecting NVMM by encryption. MECU [15] encrypts all memory transfers between CPUs and NVMM with low overhead. i-NVMM [2] provides incremental encryption, which encrypts infrequently accessed pages but does not encrypt frequently accessed pages. DEUCE [18] reduces encryption overhead on writeback to NVMM by re-encrypting only the words that have been changed. These approaches need special hardware, but Cache-Crypt is a software-only solution.

7 CONCLUSION

In this paper, we proposed Cache-Crypt, which efficiently encrypts the page cache in memory to prevent part of the storage data from being stolen. Cache-Crypt is well integrated with FDE to avoid redundant cryptographic operations and take advantage of FDE. Upon file reads, it bypasses decryption by FDE and directly stores encrypted data in the page cache. Upon writeback to storage, it bypasses encryption by FDE and directly writes data back to storage. To decrypt and encrypt data in the page cache, Cache-Crypt uses FDE's mechanism. To reduce cryptographic overhead, Cache-Crypt performs delayed encryption. We have implemented Cache-Crypt in Linux and confirmed that the performance degradation was up to 19%, compared with dm-crypt only.

One of our future work is to apply Cache-Crypt to the Android operating system. We would like to investigate the impact on the performance and energy consumption in smart devices. Also, we are planning to evaluate Cache-Crypt using real applications. Another direction is to measure the performance of Cache-Crypt with ARMORED [7], which protects cryptographic keys but causes extra overhead.

REFERENCES

[1] J. Halderman and. 2008. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proc. USENIX Security Symp.*

[2] S. Chhabra and Y. Solihin. 2011. i-NVMM: A Secure Non-volatile Main Memory System with Incremental Encryption. In *Proc. Annual Int. Symp. Computer Architecture*. 177–188.

[3] Consumer Reports. 2014. Annual State of the Net Survey. (2014).

[4] M. D. Corner and B. D. Noble. 2002. Zero-interaction Authentication. In *Proc. Int. Conf. Mobile Computing and Networking*. 1–11.

[5] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. 2011. Keypad: An Auditing File System for Theft-prone Devices. In *Proc. European Conf. Computer Systems*. 1–16.

[6] R. Geambasu, T. Kohno, A. levy, and H. M. Levy. 2009. Vanish: Increasing Data Privacy with Self-destructing Data. In *Proc. USENIX Security Symp.* 299–316.

[7] J. Götzfried and T. Müller. 2013. ARMORED: CPU-bound Encryption for Android-driven ARM Devices. In *Proc. Conf. Availability, Reliability and Security*. 161–168.

[8] V. Gough. 2003. EncFS: an Encrypted Filesystem for FUSE. https://vgough.github.io/encfs/. (2003).

[9] M. A. Halcrow. 2005. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. In *Proc. Linux Symp.* 2495–2503.

[10] IDC Reseach, Inc. 2015. IDC Worldwide Quarterly Mobile Phone Tracker. (2015).

[11] T. Muller, M. Spreitzenbarth, and F. C. Freiling. 2013. FROST – Forensic Recovery of Scrambled Telephones. In *Proc. Int. Conf. Applied Cryptography and Network Security*. 373–388.

[12] P. Papadopoulos, G. Vasiliadis, G. Christou, E. Markatos, and S. Ioannidis. 2017. No Sugar but All the Taste! Memory Encryption without Architectural Support. In *Proc. European Symp. Research in Computer Security, Part II*. 362–380.

[13] P. A. H. Peterson. 2010. Cryptkeeper: Improving Security with Encrypted RAM. In *Proc. Int. Conf. Technologies for Homeland Security*. 120–126.

[14] N. Provos. 2000. Encrypting Virtual Memory. In *Proc. USENIX Security Symp.*

[15] T. Richardson, K. Butler, A. Smith, P. McDaniel, and W. Enck. 2008. Defending against Attacks on Main Memory Persistence. In *Proc. Asia-Pacific Computer Systems Architecture Conf.* 65–74.

[16] S. Sharma. 2006. *TransCrypt: Design of a Secure and Transparent Encrypting File System.* Master's thesis. Indian Institute of Technology Kanpur.

[17] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. 2012. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *Proc. Symp. Operating Systems Design and Implementation*. 77–91.

[18] V. Young, P. Nair, and M. Qureshi. 2015. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*. 33–44.