2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2027-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPCA53966.2022.00043

Filesystem Encryption or Direct-Access for NVM Filesystems? Let's Have Both!

Kazi Abu Zubair
Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina
kabuzub@ncsu.edu

David Mohaisen
Computer Science
University of Central Florida
Orlando, Florida
mohaisen@ucf.edu

Amro Awad
Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina
ajawad@ncsu.edu

***Abstract*—Emerging Non-Volatile Memories (NVMs) are promising candidates to build ultra-low idle power memory and storage devices in future computing systems. Unlike DRAM, NVMs do not require frequent refresh operations, and they can retain data after crashes and power loss. With such features, NVM memory modules can be used partly as a conventional memory to host memory pages and partly as file storage to host filesystems and persistent data. Most importantly, and unlike current storage technologies, NVMs can be directly attached to the memory bus and accessed through conventional load/store operations.**

As NVMs feature ultra-low access latency, it is necessary to minimize software overheads for accessing files to enable the full potential. In legacy storage devices, e.g., Flash and Hard-disk drives, access latency dominates the software overheads. However, emerging NVMs' performance can be burdened by the software overheads since memory access latency is minimal. Modern Operating Systems (OSes) allow direct-access (DAX) for NVM-hosted files through direct load/store operations by eliminating intermediate software layers. Unfortunately, we observe that such a direction ignores filesystem encryption and renders most of the current filesystem encryption implementations inapplicable to future NVM systems. In this paper, we propose a novel hardware/software co-design architecture that enables transparent filesystem encryption without sacrificing the direct-access feature of files in emerging NVMs with minimal change in OS and memory controller. Our proposed model incurs a negligible overall slowdown of 3.8% for workloads representative of real-world applications, while software-based encryption can incur as high as 5x slowdown for some applications.

***Index Terms*—Non-Volatile Memory; Direct-Access (DAX); Security; Privacy; Filesystem Encryption**

I. Introduction

Emerging Non-Volatile Memories (NVMs), such as Micron's and Intel's 3D XPoint [18], provide an opportunity to merge storage and memory systems. Unlike DRAM, emerging NVMs can retain data for a long time after losing power. For instance, Phase-Change Memory (PCM) can retain data for years. Moreover, NVMs promise high density and the potential to scale much better than DRAM. NVMs also have access latencies that are comparable to DRAM. Finally, emerging NVMs will be offered in Dual-Inline Memory Module (DIMM) form factor. Hence, they can be accessed through conventional load/store operations compared to I/O block requests in current storage systems.

Since emerging NVMs offer ultra-low access latencies, the system software overhead can dominate the access latency. In current storage systems, access to the filesystem will result in copying file pages to a software-managed page cache buffer where applications can access them directly. While such copying latency is marginal compared to the access latency of current storage devices, it can easily dominate the access latency of emerging NVM devices. Moreover, there is no need to copy pages of a file to page cache buffer if pages can be accessed directly in the NVM device. Significant development and research efforts have been devoted to redesigning operating systems and filesystem implementations, allowing direct access [22], which can potentially unlock the full potential of emerging NVM devices. Direct-Access for Files (DAX) is the access method provided by Linux to directly access NVM devices through load/store operations, without the need to copy file pages to a page cache buffer. DAX minimizes the system software overhead by directly memory-mapping file pages to the address space and bypasses costly data flow within different software stacks.

Unfortunately, there is a *fundamental challenge* for using direct-access with NVM-based filesystems. Specifically, implementing filesystem encryption is no longer feasible when the DAX feature is enabled. Filesystem encryption relies on encrypting pages before evicting them from the page cache and decrypting them when copied to the page cache. However, DAX completely removes such interactions with the page cache. Thus, implementing filesystem encryption for NVM-based systems is now limited to two options: ① Interrupting the first access to each page, decrypting the page, and then creating another copy that will be mapped instead of the original page. Unfortunately, this defies the purpose of the DAX support and can create significant capacity overhead due to duplication. Moreover, it enlarges the attack surface by having unencrypted file pages in memory for a long time. Using a small buffer for decrypted pages would still cause many page faults when accessing uncached encrypted pages. ② Relying on applications to decrypt the file pages when reading them and encrypting them when writing them to memory. Unfortunately, this requires modifying applications and adding significant overheads to each read/write operation to a file, which renders NVM-hosted filesystems less attractive.

Ideally, ***we need a transparent encryption/decryption of filesystem data when written to and read from NVM-hosted filesystem, while preserving the file-based flexible security guarantees provided by filesystem encryption***. Like hardware-based full-disk encryption, NVM memory encryption is insufficient due to the lack of file-based protection, and thus filesystem encryption is needed regardless[1].

In this paper, we propose a practical hardware-assisted filesystem encryption implementation suitable for NVM-hosted filesystems. Specifically, we revisit the basic concepts of filesystem encryption. We argue that hardware support, with minimal system software changes, is the correct approach to enable filesystem encryption for NVM devices while still leveraging the ultra-low access latency. We adopt a layering-based security implementation philosophy like *defence-in-depth* [27], where we use filesystem-level encryption on top of existing memory encryption. To distinguish between file requests and general memory requests within the memory controller, we use a single bit (`DF-bit` or DAX File bit) in the physical address. During file creation and page faults, the kernel stores an identifier of the file and associated key in a memory-mapped I/O registers that the memory controller can access directly during the encryption and decryption process. Our solution is simple compared to the available stacked filesystem encryption methods and can provide efficient, transparent, and fine-grained file encryption within the memory controller.

To evaluate our design, we use Gem5 [11], a full-system architectural simulator, and use it to run a modern Linux kernel (4.14) with Ubuntu 16.04 as the operating system. We test the system with representative real-world persistent workloads and memory-intensive synthetic benchmarks. The persistent memory region has been mounted as an EXT4 filesystem with DAX enabled. The simulation results show that our transparent filesystem encryption (FsEncr) scheme for NVM incurs only **3.8% slowdown**[2] for several real-world benchmarks on an average. In contrast, a traditional software-based approach can incur a significant slowdown for different applications (e.g., $\approx$ **5x slowdown for YCSB**). Such minimal overhead in our method is an evidence that we can leverage the low latency of NVM devices without the need to sacrifice security and filesystem encryption. To the best of our knowledge, our paper is the first to discuss the challenges of filesystem encryption in NVM devices and the first to present a scheme to bridge the gap between direct-access of NVM filesystems and filesystem encryption.

The rest of the paper is organized as follows. In Section II, we give an overview of memory encryption and filesystem encryption and discuss the impact of the direct-access feature. In Section III, we discuss our proposed hardware-assisted filesystem encryption scheme along with the required hardware and software changes. Later, in Section IV, we discuss our evaluation methodology. Section V shows our evaluation of the performance overhead due to enabling our support of filesystem encryption. In Section VI, we discuss the impact of our scheme on different filesystem management aspects and security, and in Section VII we discuss the related work. Finally, we conclude our work in Section VIII.

[1]Filesystem encryption limits the exposure, in case of compromising a key, to the file uses that key only, whereas whole medium encryption provides a single layer of security. Typically, both file-based and whole encryption are deployed together to achieve *defence-in-depth* [27].

[2]The reported slowdown is only for the encrypted files in the DAX filesystem. We only evaluate the performance of applications that intensively access persistent files. For other operations not involving I/O for encrypted files, the system is not affected by our scheme.

II. Background

In this section, we discuss the main aspects of filesystem encryption, DAX support, and secure NVM systems.

A. Filesystem Encryption

Filesystem encryption has been explored heavily in the past. Several implementations of cryptographic filesystems have been proposed, including CFS [12], Cryptfs [39], eCryptfs [17] and TCFS [13]. In contrast to our work, all these implementations are software-based. While the software-based encryption/decryption overheads could be marginal when accessing SSDs or HDDs, the overhead can be dominant for DAX files in fast NVM-based systems.

One related research direction is Full-Disk Encryption (FDE) [8], [26] and its counterpart memory encryption [9], [14], [34], [38]. This research direction focuses on hardware encryption of the whole storage or memory regardless of its semantic. Typically, FDE is used transparently and executed inside the SSD or HDD; however, it provides a much coarser granularity than filesystem-level encryption. While both FDE and filesystem-level encryption try to protect the stored contents, each has its advantages and disadvantages. For instance, if FDE is compromised, all files in the disk are vulnerable. On the other hand, filesystem-encryption can encrypt different files with different keys; however, some file metadata are not encrypted in most cases [3], requiring another layer of security on top of the filesystem-level encryption. Therefore, it is generally advised to have both filesystem-level encryption and FDE to have a maximum security [1]. Although some protection levels are duplicated, this is typically recommended as a layering tactic, conceived by the U.S. National Security Agency (NSA), and called *defense-in-depth* [27].

Current filesystem-level encryptions are software-based and rely on OS support for accessing filesystem semantics, managing access permissions, and key management. The file key is usually stored in kernel memory (e.g., in Linux keyring) or within the encrypted file's metadata after encrypting the key with a master key (e.g., in Windows EFS and Linux eCryptfs). While opening a file, the file key can be decrypted using the master key, which can be derived from the user prompted password [2], [16]. If an unauthorized user attempts to open a file, the file key is decrypted wrongly and cannot be used to decrypt the file correctly. However, most filesystem encryption deliberately makes no attempt to re-implement the existing permission mechanism provided by the OS and trusts the OS to maintain access permissions [32].

B. *Direct-Access for Files (DAX)*

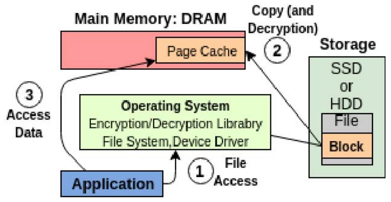
Emerging NVMs are expected to be offered in forms that can be attached directly to the memory bus, thus can be accessed with typical load/store operations. In comparison, accessing files in current systems is typically handled through the software of the filesystem layer and then converted into commands that are handled by the device driver of the storage device. Finally, the obtained data is copied to a memory-hosted software structure called *page cache*. The software performance overhead, including copying data to page cache, is relatively low when compared to disk access latency of legacy drives (e.g., SSD or HDD). However, this will change in the presence of very fast emerging NVM devices where the software layer overhead becomes a bottleneck. Accordingly, modern implementations of filesystems, e.g., ext4 in Linux 4.10+, are beginning to provide support to directly access filesystem's files through load/store operations without the need to copy files' pages to the software page cache. This direct access feature is referred to by Direct-Access for Files (DAX) [22].

To better understand the difference between conventional and DAX-based filesystems, Figure 1(a) depicts the steps typically required to access data in a file in conventional systems. In Step 1, an application accesses a file's page either through typical system calls or by accessing a range of addresses that are mapped to a file (e.g., through `mmap`). Later, in Step 2, a page fault occurs where the OS will handle the fault by going through multiple software layers. The process involves invoking functions from the filesystem layer and the device driver to get the actual data in the file. Finally, the OS copies the page (or block) from the device (SSD or HDD) to the page cache in memory. Note that if the file is encrypted, the page will be decrypted after being copied to the page cache. Finally, as in Step 3, the application can access the page in the page cache. Since the page cache size is limited, old pages will be evicted (and encrypted) then replaced by new pages.

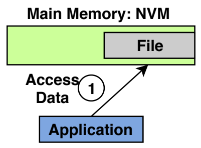
Unfortunately, invoking the OS and copying pages to the page cache is a very expensive process. While the overhead of invoking the OS and copying a page is low when compared to the access latency of HDDs or SSDs, it becomes a bottleneck when fast emerging NVMs are used. In contrast, in DAX-based systems, and after mapping the file into the application's address space, the application can directly access the NVM-hosted file through typical load/store operations, as depicted by 1(b).

C. *Memory Encryption*

Encryption in NVM-based main memories is achieved under one of two assumptions, modified processor and unmodified processor. In the latter approach, the internal NVM controller transparently encrypts the main memory content [14], and only encrypts the infrequent data in most cases. The hot data (frequently accessed data) is then proactively decrypted and stored as plaintext in memory without getting encrypted. Under the first assumption (modified processor-side memory controller), the processor chip is the trust base, and any data



(a) The conventional way of accessing files.



(b) Accessing DAX-based files.

Fig. 1. Accessing files in conventional vs DAX-based filesystems.

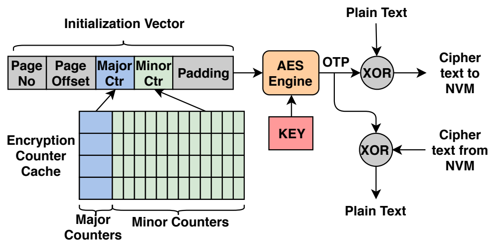


Fig. 2. State-of-the-art counter mode encryption (AES).

that is sent off-chip is insecure and needs to be encrypted [29], [35], [38].

The encryption engine AES is used to encrypt and decrypt cache blocks, which are either written to or read from memory. Different modes use the AES engine to encrypt each cache block effectively. Direct encryption, with ECB mode, directly inputs the cache block to be written to the AES engine to encrypt the data. However, direct encryption degrades the system performance by adding decryption latency. On the other hand, in the Counter Mode (CTR) encryption, the encryption algorithm is applied to an Initialization Vector (IV) to generate a one-time pad (OTP) while data is read in parallel, as depicted in Figure 2. The OTP is then XOR'ed with the data to encrypt and decrypt the data. With this approach, encryption/decryption latency is hidden underneath the memory access latency, and only XOR latency is added as an overhead. The Initialization Vector of the CTR mode encryption consists of various fields as shown in Figure 2: unique page ID and page offset to provide spatial uniqueness, a per-block *minor* counter to distinguish different versions of the data value of a block over time, and a per-page *major* counter to avoid overflow in counter values.

Memory-side encryption is vulnerable to bus snooping attacks, and encryption in ECB mode is not entirely secure (e.g., it can be vulnerable to dictionary-based attacks and replay attacks). Hence, we rely on counter mode processor-side encryption similar to [29], [35], [38], which is considered as a standard memory encryption mode in NVM security.

TABLE I
VULNARABILITY OF VARIOUS ENCRYPTED SYSTEMS

Memory Encryption Key Revealed	Single Filesystem Key Revealed	All File Keys Revealed	Vulnarability		
			System A	System B	System C
Y	X	X	Yes	No	No
Y	Y	X	Yes	Yes	No
Y	Y	Y	Yes	Yes	Yes

Encryption counters in counter mode encryption are stored in the main memory. To avoid accessing memory frequently for encryption counters, a counter cache is usually used. In the split-counter scheme, the major counter (64 bit) and all 64 minor counters (7 bit each) are stored together in a cache line, which covers a 4KB page [35]. For every write, a block's minor counter is incremented to construct a new Initialization Vector, which is used to generate an OTP using a secret encryption key (Figure 2). When the minor counter overflows, the major counter is incremented, and the entire page is re-encrypted. Since the counters are incremented for every write, the OTP generated to encrypt the data is unique, and hence the counter mode is secure. Counters are usually protected by Merkle Tree to detect any tampering to their values [29].

Why filesystem encryption is needed in encrypted and integrity protected NVM: General memory encryption and file system encryption have different goals and adversarial models. For instance, the purpose of general memory encryption is to protect the memory contents from active attackers who do not have access permission to the 'system'. Such protection guarantees confidentiality even if the NVM gets stolen and used in a different system to extract the contents. However, filesystem level encryption aims to protect one user's file from the other where all of the users have access to the system. For an example, if an inside employee who has physical access to the system (but not system admin), such as an IT technician can boot the system using a different Operating System. This will allow all memory contents to be decrypted correctly if only memory encryption is used.

Table I depicts different scenario of an encrypted NVM system that holds files in it. First, the system may only have memory encryption (System A). The files hosted in NVM are encrypted as typical memory blocks. Second, there can be an additional key that encrypts the entire filesystem (System B). Finally, in System C, each file is encrypted differently using identical keys. System A is vulnarable to any internal attacker that has physical access to the system. There are no separate protection for files. System B provides somewhat protection for the files with another key. Such keys should generally be set by the administrator using a passphrase. If this gets revealed, all other files can be decrypted easily. The best protection can be guaranteed if dedicated keys per file is used so that any unauthorized access to the files can be prevented. Note that such practice is similar in the context of storage drive protection where the best practice is to use Full Disk Encryption (FDE) and additionally use file encryption, as suggested by professional security solution vendors [1]. Double layers of protection guarantees that the files are protected even if the memory encryption is broken, and even if one or few other file keys are revealed.

D. Security Metadata Crash Consistency

Secure processors leverage caching to reduce accessing security metadata (encryption counters, Merkle-tree nodes) from memory. The volatility of cache memory technologies (e.g., SRAM and DRAM) causes inconsistency between the data and security metadata if a system crash happens. Strictly persisting the security metadata in the memory with every metadata update provides crash consistency at the cost of an extreme slowdown. Several state-of-the-art approaches look at solving such a problem [24], [28], [36]. Among them, one recent approach, *Osiris* [36], encodes counter information with the data block's ECC. With limiting consecutive NVM updates of counters in the cache and having ECC to ensure the correctness of the counter, Osiris can recover any counter value after a system crash. Note that, after recovering the encryption counter, the Merkle tree can be regenerated and verified through the root stored inside the processor.

E. Motivation

As mentioned earlier, DAX is designed to bypass costly data flow within various OS and filesystem stack layers. This feature can enable outstanding performance improvement in persistent file management and database applications if today's fast and byte-addressable NVMs are used. With the elimination of intermediate software stacks to achieve direct load/store, many critical software features (e.g., filesystem encryption) are difficult to implement. Therefore, files stored in NVM and accessed directly using DAX can no longer be protected using filesystem-level encryption and is only protected by the underlying memory encryption. If the software-based filesystem encryption is used in DAX, the additional software overheads will obliterate the benefits of DAX.

To obtain more insights on this matter, we have simulated eCryptfs over the emulated persistent memory region (using `memmap`) in the Gem5 full-system simulation. We then observe the performance implications[3] compared to plain ext4-dax. The overheads clearly show the slowdown caused by having a software-level encryption mechanism. As shown in Figure 3, on average, the software encryption incurs **2.7x slowdown** for the benchmarks we have tested to compare Direct Access with software encryption—More information about the workloads can be found in the Methodology Section IV Table II. This slowdown is primarily because of encryption/decryption granularity of 4KB (page size) for every access and intermediate software layers required to allow filesystem encryption (eCryptfs) to function. As a result, it leaves only two options for NVM operated in a direct access manner: no filesystem-level encryption, or no performance benefit for

[3]Due to the complexity and high simulation time for simulating applications enabling eCryptfs in full system mode, we only simulate three whisper benchmarks to form our hypothesis in Figure 3. In our final evaluation, we evaluate all applications with and without FsEncr.

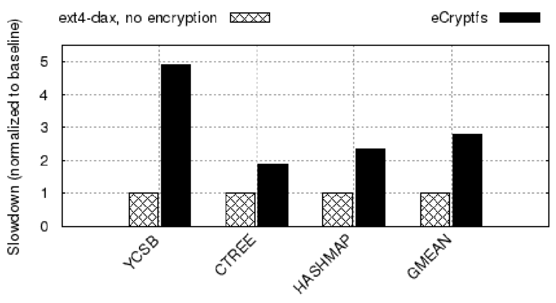


Fig. 3. Overheads of software encryption.

using DAX. Therefore, it is important to explore practical solutions to enable both byte-addressable, direct-access, and low-overhead filesystem encryption for NVM-hosted files. To the best of our knowledge, this is the first paper to discuss this challenge and propose a novel solution to address this problem.

III. Hardware Assisted Filesystem Encryption

In this section, we discuss the threat model, design requirements, and the proposed hardware and software changes.

A. Assumptions and Threat Model

Our threat model is similar to that assumed in state-of-the-art secure memory systems [9], [14], [34], [38] and traditional filesystem encryption [5], [16]. First, and as shown in Figure 4, an outside attacker (Attacker X) can potentially obtain the physical memory and scan through it for passwords and other confidential data. The attacker can also snoop the data communicated between the processor and the memory to learn confidential information. Moreover, external attackers can replay or tamper with memory contents.

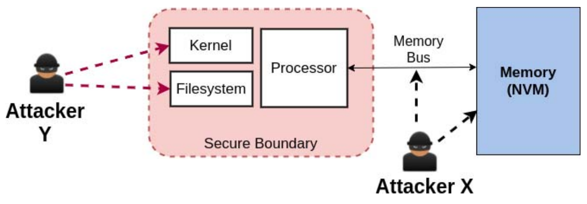


Fig. 4. Our assumed threat model.

Since the operating system handles the filesystem, relying entirely on hardware to enable filesystem encryption would be complicated. Most filesystem encryption frameworks rely on the kernel to maintain access permissions to avoid unnecessary re-implementation of the access control mechanism already present within the OS [32]. Similarly, we trust the OS kernel to provide correct and minimal information about the users and applications to the memory controller. However, we guarantee protection for files against an unauthorized user (Attacker Y) who tries to alter the filesystem or the OS kernel as long as a trusted OS is installed and operated by a trusted system admin. Even if the Attacker does not have proper access permission, files can become accessible to the attacker for many non-ideal practices of file management. One good example of this is improper use of `chmod` to change file permission. A user (or admin) may accidentally use `chmod 777` which will make the file accessible to the attacker if not encrypted using a dedicated key. Similar things can happen if the system admin is adversarial and tries to gain access to other user's file that is private to the user. While we protect files against such threats, our solution will not be able to protect the files under an untrusted Operating System. Protecting files under an untrusted OS requires a more aggressive threat model, and we leave such exploration for future research.

B. Key Challenges for Hardware-Assisted Filesystem Encryption

Usually, the processor cannot provide per-user and per-file encryption in the memory controller by itself—such fine-grained operation is possible only at the OS level. Therefore, to allow hardware-level encryption/decryption, the OS needs to communicate with the hardware and provide minimal information about files and access control semantics. This can be done by writing to specific memory-mapped I/O registers on particular OS events, which the memory controller can access directly. The design goal is to allow transparent and on-chip encryption for file requests on top of the existing memory encryption. For the memory controller to apply different confidentiality levels for the various memory requests (request for general memory access and DAX file access), the following challenges must be addressed.

1) The memory controller should be able to distinguish between regular memory requests and DAX file read/write requests.
2) The memory controller should be able to map a file I/O request to a unique file key.
3) The OS should provide the memory controller with the necessary information to make 1 and 2 possible at the memory controller level.

C. Recognizing Persistent Files

Traditionally, the OS allocates physical memory to the application's address space for all memory accesses needed by the application, including the allocation for memory-mapped files. While the OS and the application know which file is being accessed, such logical accesses are transparent to the memory controller, which only sees a physical address being accessed. The virtual to physical address translation happens at the MMU, and the OS creates the mapping in the Page Table during page faults. For the memory controller to add an additional security layer for memory-mapped files, the requests' physical addresses should be distinguishable at the memory controller. We use one bit (`DF-bit`, or DAX File bit) in the physical address to allow the memory controller to identify DAX file accesses. In our approach, we realize that the number of physical address bits used for most systems is much larger than needed. For instance, in Intel's *IA-32e*, the address translation mode in 64-bit OS maps a 48-bit virtual address to a physical address of 52-bit maximum [15], which is sufficient to support 4PB memory capacity. Using one bit of the physical address can still allow sufficient memory to be used. Note that, using physical address bits for additional security features is a well-adopted design choice in industry (e.g., AMD's `C-bit` [19], Intel's `KeyID` in MKTME [41]). Figure 5 shows how FsEnc incorporates `DF-bit` within the physical address.

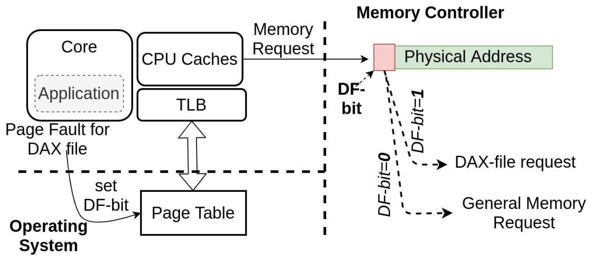
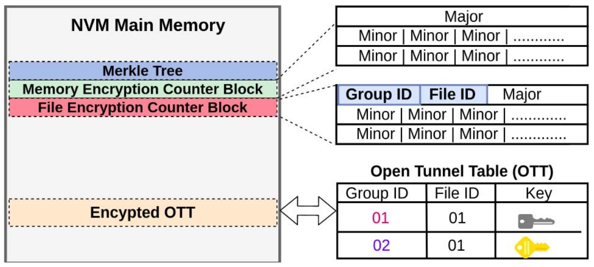


Fig. 5. DAX file recognition process.

After an application opens a persistent file stored in NVM and accesses its contents on a page for the first time, a page fault occurs, and page table mapping is updated with the physical page number. While updating the page table after a page fault, FsEncr sets `DF-bit` to 1 for DAX files. Later access to any byte that maps to the same physical page will have a `DF-bit` set to 1, which will indicate that the request is for a DAX file. This is done with simple and straightforward modifications within the kernel. For instance, in the `dax_insert_mapping` function in Linux 4 or `dax_insert_entry` function in Linux 5, the physical address can be set within the page table entry as `((1UL<<51)|pfn)` assuming a physical address size of 52 bits.

D. Encryption Metadata

As the reader is now aware of how FsEncr distinguishes the file requests at the memory controller level, we now illustrate the security metadata used in FsEncr to allow multi-level encryption. We use counter mode encryption for encrypting both general memory cache lines and DAX-file cache lines. First, there is general memory encryption for all cache lines using a memory encryption key and memory encryption counters. On top of that, file cache lines are encrypted using separate file keys and file encryption counters. We term the general counter block as the Memory Encryption Counter Block (MECB) and the dedicated counters for file encryption as the File Encryption Counter Block (FECB). A file encryption counter block follows each memory encryption counter block in the metadata region. MECB organization in our scheme is similar to the traditional split-counter scheme (Figure 6). However, we modified the FECB organization to store Group ID (18 bits), File ID (14 bits), 32-bit major counter, and 64 7-bit minor counters [4]. Such organization is similar to the 64-way split-counter organization where each counter block covers 4kB page in the memory. The combination of a major and a minor counter is used to encrypt one data cache line (64B). Storing Group ID and File ID within the FECB aids in recognizing the file encryption key specific to a file. In Linux, the file-sharing and permission control is enforced by the concept of grouping, where multiple users who share a shared resource (directory or file) are within the same group. If the file is not shared with anyone, the owner is the only member of the group. The owner can set permissions for accessing files for different users within a group. The kernel can send the file ID (`mapping->host->i_ino`) and the group ID (`mapping->host->i_gid`) to the memory controller by writing to a memory-mapped I/O register during page fault, and the memory controller updates the corresponding FECB with the group ID and the file ID. Note that if the corresponding FECB block is already cached in the metadata cache, it is sufficient to update the cached FECB block with the File ID and Group ID and flag it as dirty. The integrity of the counters is protected using an 8-ary Bonsai Merkle tree. All security metadata (MECB, FECB, and Merkle-tree nodes) are cached in the Metadata Cache inside the processor chip. However, it is possible to partition the metadata cache for each metadata (FECB, MECB, and MT nodes) to equitably distribute the cache capacity.


NVM Main Memory
Merkle Tree
Memory Encryption Counter Block
File Encryption Counter Block
Encypted OTT
Major
Minor | Minor | Minor | ....
Minor | Minor | Minor | ....
Group ID
File ID
Major
Minor | Minor | Minor | ....
Minor | Minor | Minor | ....
Open Tunnel Table (OTT)

Group ID	File ID	Key
01	01	
02	01	

Fig. 6. Encryption Metadata and OTT Table.

E. Key Management

Most software-based filesystem encryption utilizes the keyring mechanism [23] of the Operating System to generate and store keys for the encrypted files. For instance, `fscrypt` [5] and `eCryptfs` [16] maintains hierarchical keys where a master key is used to decrypt the file keys. For instance, `eCryptfs` encrypts protected files using a File Encryption Key (FEK), which is encrypted using a File Encryption Key Encryption Key (FEKEK). FEK is randomly generated, and FEKEK is derived from a user prompted passphrase [16]. Similarly, in FsEncr we allow the OS to create encryption keys for a file using the owner's password. Once the file is created, the kernel sends the Group ID, File ID, and the file key to the memory controller. The keys are stored in a

[4]Unlike memory encryption counters, which are expected not to overflow during the lifetime of the device, the file encryption counters are only expected to last only during the files' lifetime. With the deletion of the file or the creation of a new file key, the FECB can be re-initialized.

hardware structure that we term as Open Tunnel Table (OTT). Each entry in OTT has one File ID (14 bit), one Group ID (18 bit), and a key (128 bit). The OTT is implemented as eight fully associative 128 entries that are searched in parallel, similar to TLB, but to avoid significant power loss, we allow the latency to take 20 cycles instead of a single cycle as in TLB. If any entry is evicted from the OTT, it is stored in a dedicated memory region after encrypting with a dedicated OTT key. The OTT key never leaves the processor similar to the memory encryption key. The encrypted OTT region can be implemented as a set-associative hash-table maintained by the memory controller. The File ID and Group ID bits are hashed to identify the memory location of the encrypted OTT of fetched from memory. The memory controller can fetch the file key anytime from that protected region if a specific key for a File ID and Group ID is not found in the OTT entry, but `DF-Bit` is set. When the file is removed, the corresponding entry from both OTT and encrypted OTT region in memory is removed. OTT only serves as a means to quickly extract the file key for a file request. Note that the OTT updates are rare and only happen when a new page is mapped to the application's address space. Therefore, OTT implementation has a very negligible impact on system performance.

F. Putting It Altogether

1) Hardware-Software Communication: FsEncr requires small OS changes to allow the memory controller in recognition and fine-grained encryption of files. First, the OS generates the file key during file creation and sends it to the memory controller along with the File ID and Group ID by writing to memory-mapped I/O registers. The memory controller stores the keys in the Open Tunnel Table structure. OS does not need to communicate anything during read, write system calls; however, it informs the memory controller to remove the key when the file is deleted. During page faults for DAX files, OS sets the `DF-bit` to 1 in the page table entry and signals the memory controller to update the File Encryption Counter Block with Group ID and File ID. Note that page fault happens only at the first access to the page, and later accesses do not require OS to send any information to the memory controller.

2) Read Operation: Figure 7 shows a simplified read operation in our scheme. The following are the overall steps in a read request.

1. When an application wants to access a DAX file, the requested virtual address will be translated to the corresponding physical address at the MMU. Note that if the DAX page for the file is accessed for the first time, a page fault will follow, and the corresponding page table entry would be created with `DF-bit` set in the most significant bit position. Later access to that page would have a physical address with `DF-bit` set to 1, indicating the request as a DAX file request.

2. If the request for the data cannot be satisfied from the processor cache, the memory controller will try to satisfy the request by accessing the main memory. At that point, the Memory Controller gets the `DF-bit` and decides whether the

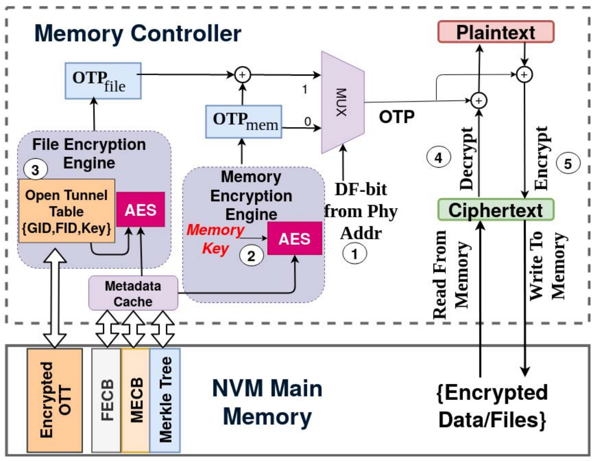


Fig. 7. FsEncr Read and Write Operation.

request is for a file or a general memory cache line (as shown in ① in Figure 7). If the request is a general memory request (`DF-bit` set to 0), the request is treated as a typical non-file memory request (Figure 5).

3. If the request is typical memory access (not for DAX file), the MECB counter block is fetched from the Metadata Cache or memory in order to prepare $\texttt{OTP}_{\texttt{mem}}$ in the Memory Encryption Engine (as shown in ②), which will be used to decrypt the block. Note that the decryption should be followed by Merkle-tree-based integrity verification of the counter before passing the decrypted data to the cache hierarchy.

4. If the physical address of the request has `DF-bit` set, the File Encryption Engine creates $\texttt{OTP}_{\texttt{file}}$ using the file key taken from the OTT and the file encryption counter taken from FECB. First, the Group ID and File ID are extracted from FECB, and the corresponding key is searched in the OTT. For such file requests, $\texttt{OTP}_{\texttt{mem}}$ is also created in parallel using memory encryption key and memory encryption counter. $\texttt{OTP}_{\texttt{file}}$ and $\texttt{OTP}_{\texttt{mem}}$ is XORed together which is used as the final One Time Pad (OTP) to decrypt the cipher in step ④. Finally, the decrypted data is passed to the processor cache.

3) Write Operation: Following are the steps in a memory write operation in our scheme (Figure 7).

1. From the physical address of the write request, the memory controller decides whether the request is a DAX file request or a typical memory request using the `DF-bit` (as in ①).

2. If the request is a typical memory write, the data block is encrypted using $\texttt{OTP}_{\texttt{mem}}$ and memory encryption key, which is written directly to memory without further encryption.

3. For DAX writes, FECB block is also fetched from the Metadata Cache or the main memory, and appropriate file key is read from the OTT (as shown in ④) by mapping the Group ID and File ID taken from the FECB. Next, the File Encryption Engine generates the $\texttt{OTP}_{\texttt{file}}$ which is XORed with the $\texttt{OTP}_{\texttt{mem}}$ to create the final One Time Pad. Finally, the plaintext data is encrypted by XORing it with the OTP and written to the NVM as shown in ⑤ in Figure 7.

G. Protecting File Keys and Preserving Integrity

Once there is no entry available in the Open Tunnel Table but more encrypted files are created, the least used OTT entries (Group ID, File ID and File Key) are written to a dedicated memory region after encrypting with an OTT key. This additional encryption helps to protect the file keys (if they were dumped to memory) even if the general memory encryption is compromised. The integrity of the security metadata (MECB, FECB, OTT) are protected through the Merkle tree. The root of the tree is always kept inside the processor and acts as a verification point of any metadata read from memory. Any tamper/replay of the protected metadata will be detected if a root mismatch occurs.

H. Crash Consistency

One major requirement for using secure memories with NVMs is the ability to recover after crashes. Major security and correctness issues can arise if we are unable to recover the most recent updates to encryption counters [36]. There are a few state-of-the-art approaches to address this issue of maintaining crash consistency. Some approach provides atomicity to a very small subset of counters [24], and with the help of careful programming to define which counters to persist achieves recoverability. A more recent scheme, Osiris [36], removes this limitation (persisting very few counters and compiler/programming dependency) by relying on ECC. We adopt this state-of-the-art scheme, Osiris [36], to recover updated encryption counters. Moreover, we can rely on internal persistent registers to ensure transaction-level atomicity in a way similar to REDO logging [10]. However, any counter recovery scheme and crash consistency mechanism would be sufficient to provide and maintain security and persistence. To enable crash consistency for OTT, two approaches can be adopted. First, OTT updates can be logged immediately to the protected memory region for OTT. Since this update happens rarely (only when a file is created), this logging will incur insignificant overhead. Recent studies like Anubis [6] shows that such hardware structure can be recovered easily. Second, due to the small size of the OTT table (2kB), it can be logically persisted during crashes using backup power. Modern persistent processors allow flushing of processor buffers using backup power when a system crash happens [30].

IV. Evaluation Methodology

In this section, we describe our evaluation methodology. Our work involves kernel modifications and requires support for persistent applications, thus we use Gem5 simulator [11] in its full-system mode. We use a modified version of Linux Kernel 4.14 as our kernel. The disk image we use is based on Ubuntu 16.04 distribution. The kernel was initialized by configuring the 4GB starting from 12GB as a persistent region, using `memmap=4G!12G`. The persistent space is formatted with DAX-enabled ext4 filesystem, then mounted for use with persistent applications and libraries. We incorporate additional dedicated cache for security metadata (MECB, FECB and Merkle-tree nodes): the Metadata Cache. Table III presents the architectural configurations of our simulated system and Table II shows the benchmarks.

TABLE II
BENCHMARK DESCRIPTIONS

Benchmark	**Description**
DAX Benchmarks (in-house micro-benchmarks)	
DAX-1	*Application accesses 1 byte after each 16 bytes from a persistent file in direct access manner*
DAX-2	*Application accesses 1 byte after each 128 bytes from a persistent file in direct access manner*
DAX-3	*Initializes two array of 16B size in two different location and swaps the contents.*
DAX-4	*Initializes two array of 128B size in two different location and swaps the contents.*
PMEMKV Benchmarks (Engine - BTree; Threads - 2)	
Fillrandom-S/L	*fillrandom benchmark; Value = 64B(S)/4KB(L) loads value in random key order*
Fillseq-S/L	*fillseq benchmark; Value = 64B(S)/4KB(L) loads value in sequential key order*
Overwrite-S/L	*overwrite benchmark; Value = 64B(S)/4KB(L) replaces value in random key order*
Readrandom-S/L	*readrandom benchmark; Value = 64B(S)/4KB(L) reads values in random key order*
Readseq-S/L	*readseq benchmark; Value = 64B(S)/4KB(L) reads values in sequential key order*
Whisper Benchmarks	
YCSB	*Yahoo Cloud Service Benchmark; R/W ratio = 0.5; Workers=2*
Hashmap	*data-size=128B; Threads=2*
CTree	*data-size=128B; Threads=2*

TABLE III
SIMULATION PARAMETERS

Processor	
CPU	8-core, 1GHz, out-of-order x86-64
L1 Cache	private, 2 cycles, 32KB, 8-way, 64B block
L2 Cache	private, 20 cycles, 512KB, 8-way, 64B block
L3 Cache	shared, 32 cycles, 4MB, 64-way, 64B block
DDR-based PCM Main Memory	
Capacity	16GB
PCM Latencies	60ns read, 150ns write [21]
Organization	2 ranks/channel, 8 banks/rank, 1KB row buffer, Open Adaptive page policy, RoRaBaChCo address map-ping
DDR Timing	tRCD 55ns, tXAW 50ns, tBURST 5ns, tWR 150ns, tRFC [21], [24] 5ns tCL 12.5ns, 64-bit bus width, 1200 MHz Clock
Encryption Parameters	
AES Latency	40ns [24]
Metadata Cache	512KB, 8-way, 64B block
Merkle-Tree	9 levels, 8-ary, 64B blocks on each level

We use a mix of in-house synthetic persistent micro-benchmarks and real world persistent benchmarks. The micro-benchmarks access the persistent memory in a sequential and random access manner after memory mapping a large dax file. Additionally, we also use Whisper [25] and PMEMKV [4] benchmarks to measure the performance of our scheme.

V. Results

To better understand the impact of hardware-assisted filesystem encryption on a system's performance, we run each application with and without hardware-assisted filesystem encryption support. We fast forward all applications to the post-file-creation point and simulate the file access behavior of the applications. We compare the following schemes in our evaluation.

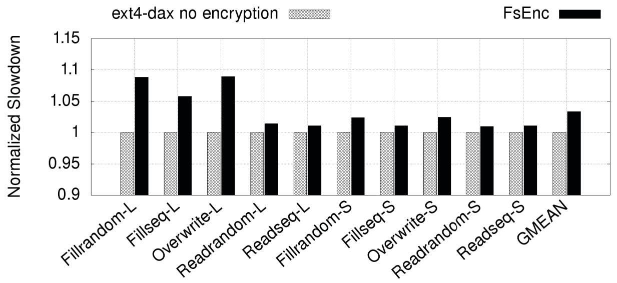


Fig. 8. Slowdown (Normalized to the baseline): PMEMKV Benchmarks

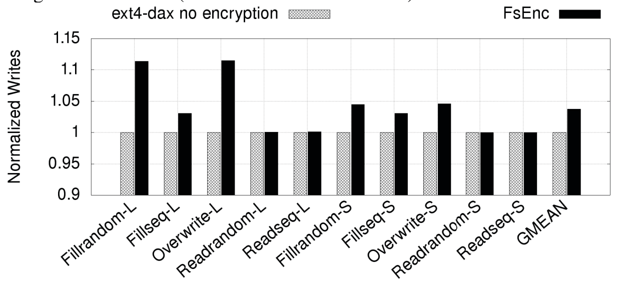


Fig. 9. Number of writes (Normalized to the baseline): PMEMKV Benchmarks

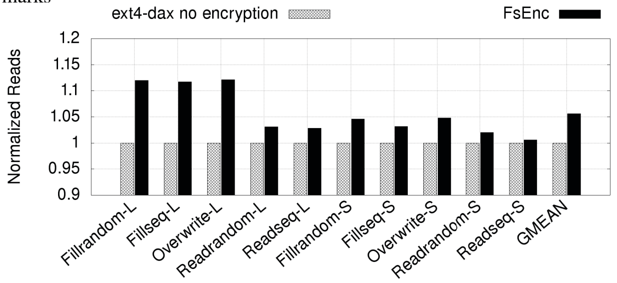


Fig. 10. Number of reads (Normalized to the baseline): PMEMKV Benchma

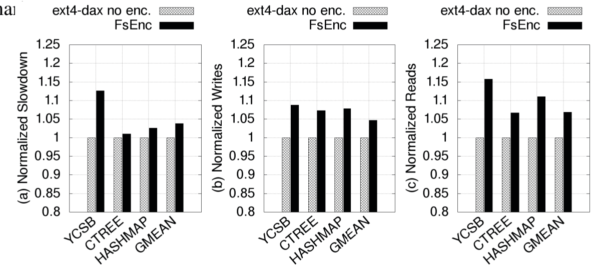


Fig. 11. (a) Normalized Slowdown, (b) Number Of Writes, (c) Number of Reads (Normalized to baseline): Whisper Benchmarks

① **Baseline Security:** This is the baseline ext4-dax with necessary memory encryption only. We use counter mode encryption with Bonsai Merkle Tree for integrity verification. There is no additional encryption for files in this scheme.

② **Filesystem Encryption (FsEncr):** An additional counter mode encryption, OTT management, and integrity check for dax files are added on top of the baseline system.

For both schemes, we add crash consistency support similar to Osiris [36]. However, any crash consistency support can be added.

A. Performance Analysis

1) PMEMKV and Whisper Benchmarks: The Persistent Memory Key-Value (PMEMKV [4] provided by Intel is a key-value database engine. It provides several sequential and random access applications. Both PMEMKV and Whisper benchmarks use Intel's PMDK library for persistent applications. We run each application in a multi-threaded setting and with two different value sizes for PMEMKV. We use 64B value as small value size (denoted as `benchmark-S`) and 4096B value as large value size (denoted as `benchmark-L`). Figure 8 shows slowdown, Figure 9 shows number of writes, and Figure 10 shows number of reads for the PMEMKV workloads, and Figure 11 shows the slowdown, write and read characteristics for Whisper benchmarks. All results are normalized to the baseline security scheme. We observe that write-intensive persistent benchmarks have higher overheads compared to read-intensive applications as every writes are required to be persisted (e.g., cache flush). PMEMKV-S benchmarks access 64B data for each iteration.

Accessing small data size renders higher hit in the metadata cache than accessing larger data. For instance, a counter cache block covers 4KB data and hence can cover 64 values of 64B each. On the other hand, a 4094B data size will incur higher miss in the metadata cache. Consequently, applications that access larger data (4096 B) have poor utilization of metadata. For Whisper applications, We have tested three benchmarks (YCSB, Hashmap and Ctree). For YCSB, we keep the read/write ratio to 50%. YCSB shows a higher percentage of access to the file, and hence slightly higher overheads. On average, all persistent benchmarks incur approximately 3.8% slowdown. We observe that FsEncr can provide a 98.33% reduction in a slowdown for filesystem encryption in the case of whisper applications.

B. Sensitivity Analysis

In the following, we analyze the sensitivity of our scheme with the help of some highly memory-intensive in-house synthetic micro-benchmarks. We also vary the Metadata Cache size to see the impact of caching.

1) In-house Micro-Benchmarks: In addition to real-world persistent benchmarks, we also develop in-house synthetic benchmarks to stress memory access. The DAX-1 and DAX-2 benchmarks access a byte after each 16-byte and 128-byte, respectively. Since each counter block only covers 4KB data, DAX-2 incurs a higher miss for security metadata in the metadata cache than DAX-1 and incurs higher performance overheads, writes and reads (Figure 12,13 and 14). The DAX-3 and DAX-4 benchmarks initialize two arrays of size 16B and 128B, respectively, in two different locations and swaps their value. These two benchmarks have both sequential behavior (within the array) and random behavior (for initializing and accessing array in random location). As a result, when reading the array from a random location, a miss occurs in the metadata cache, causing a higher read in FsEncr. However, during swap operation, bytes are swapped in sequential order allowing single MECB and FECB block to be utilized better

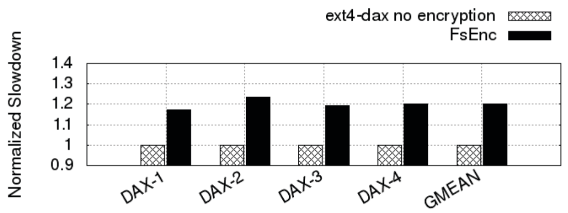


Fig. 12. Slowdown (Normalized to the baseline): Synthetic Microbenchmarks

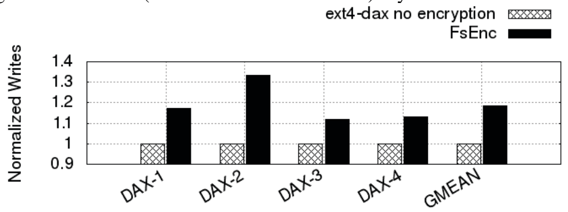


Fig. 13. Number of writes (Normalized to the baseline): Synthetic Microbenchmarks

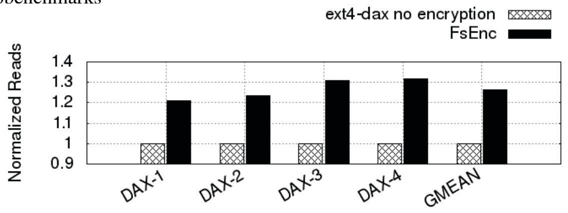


Fig. 14. Number of reads (Normalized to the baseline): Synthetic Microbenchmarks

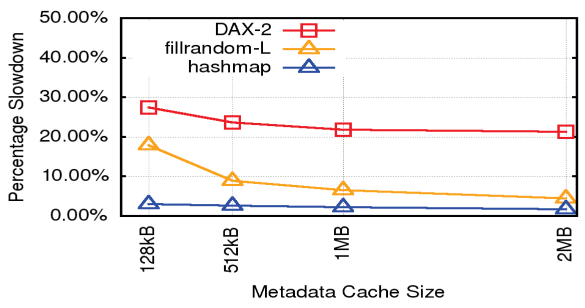


Fig. 15. Sensitivity to Metadata Cache Size (Baseline Similar to Performance Analysis).

(most of the minor counters within a block are updated), causing less metadata to become dirty per swap operation and hence less writes for metadata evictions. On average, all four benchmarks have a 20.03% slowdown over the baseline scheme.

2) Sensitivity to Cache Size: Apart from the encryption-decryption delays, the performance of our scheme mostly depends on the availability of the security metadata in the metadata cache. A cache with a larger size and higher hit rate can effectively reduce the additional overheads significantly. We, therefore, perform a sensitivity analysis by varying the size of the cache and observing the performance impact. We choose one application from each set of benchmarks (pmemkv, whisper, and in-house microbenchmark). Figure 15 shows the analysis of how much slowdown each application incurs over the baseline in percentage. For real-world benchmark, we test Fillrandom-L and Hashmap for the sensitivity study, and for synthetic benchmark, we study DAX-2. It is noticeable that the real persistent benchmarks perform significantly better with larger cache due to natural utilization in real workloads. On the other hand, the synthetic benchmarks have a higher frequency of persistent memory access and have low metadata cache utilization compared to real benchmarks. As a result, the performance only improves slightly with the increase of the metadata cache size for the synthetic benchmark.

VI. Discussion

In this section, filesystems' operational aspects related to our work are discussed.

Resetting Filesystem Encryption Counters: As mentioned earlier, our filesystem encryption uses counter-mode encryption. For counter-mode, the encryption counters should never be reused to do encryption with the same key. However, if the file key changes, there is no risk for resetting the filesystem encryption counters; the OTPs will be different from those that could have been potentially observed, e.g., using known-plaintext attacks, with the old key. While resetting the counters can potentially reduce the overflow rate of filesystem encryption counters, none of the promising NVM technologies can sustain a number of writes that can cause an overflow. However, in a pathological case of overflow of a file encryption counter, the system is still protected if the memory encryption key is not revealed. However, in such a case, a new file key can also be issued to rule out the possibility that the attacker may obtain the memory encryption key, as well as mount a replay attack using the overflow of file encryption counter. Instead of re-encrypting the entire file at once, the memory controller can keep both keys and silently decrypt with the old key (when reading pages with saturated counter) and encrypt with the new key during access to pages.

Copying or Moving Files Within Same Device: Within the same device, i.e., NVM memory, copying a file would require reading it and writing it elsewhere through the processor. When the kernel does the copying, and for each minor page fault while copying a page into a new physical page of the new file, the kernel sends information to the memory controller about the group ID and the file ID. However, since the Initialization Vector (IV) established from filesystem encryption counter also uses the page physical address, the spatial uniqueness is preserved. Hence, no potential for replaying old OTPs when using the filesystem encryption counter values for the new file. Note that there is also no correctness issues as any data in the copied file will be written to the new file using the new location's filesystem encryption counters, and thus reading it later would result in correct decryption.

Moving Entire Filesystem To New Machine: The current implementation for the hardware supported memory protection (e.g., Intel's SGX) preserves the memory encryption keys and root hash withing the processor chip, making it difficult to port the entire memory module (or the entire filesystem) to a new system. However, this can be achieved by allowing the system to securely transport the keys along with the memory module through an authorized user interface. In the case of our hardware supported filesystem encryption, all OTT entries can be flashed to the encrypted OTT region, along with the memory encryption key, OTT key, and the integrity tree root securely transferred through the user interaction as mentioned above. When plugged-in to the new system, an authentication procedure can be performed between the memory module and the processor to make sure that the new system is authentic and trusted.

Distributed Filesystems: Our hardware-assisted scheme mostly focuses on a single processor system. However, extending this work to distributed filesystems requires no further changes except that each node in such a system should support our proposed hardware changes. Additionally, each OS runs on each node should communicate the filesystem encryption information, which we have discussed earlier, e.g., file ID and encryption mode. Distributed filesystems are out of the scope of this work, and we leave them for future work.

Memory Encryption Key Revealed: If the memory encryption key is revealed, then encrypted files would still be secure. However, one exception is when the filesystem keys are naively stored by the OS in a general memory region. In such a case, all filesystem keys will be easily obtained right after decrypting the memory with the memory encryption key, and simply decrypting the memory again using the obtained filesystem key will expose the encrypted files. FsEncr stores the keys (if OTT is overflowed) in a dedicated memory region after encrypting with the OTT key, which never leaves the processor. With such, the attacker needs to break both, the memory key and OTT key to discover the file keys.

Protecting Files from Internal Attacks: A practical attack that is plausible in this scenario is when an user who has physical access to the system tries to scan the memory by booting up the system using another Operating System to bypass the admin login. Such an attacker can then scan the entire memory contents and try to decrypt the files. Note that the OTT region is not accessible from the kernel space or user space and is additionally encrypted using a dedicated OTT key. Every boot-up sequence of the system will need to go through admin's login credentials that is communicated with the memory controller. An unauthorized admin credential will lock the FsEnc decryption, and only memory encryption will be functional. Hence, the attacker will see all files and memory data decrypted only by the memory encryption key.

Protecting Files from Accidental Permission Changes Even though the system is being operated under a trusted operating system that the system admin powered up using his credentials, accidental changes in permission can happen, which needs to be guarded against. For instance, mistakenly using `chmod 777`, whether by the user or even by misconfigured scripts (e.g., buggy Makefiles), would enable other users to access the file. However, the trusted OS will always ask for the passphrase of the file when opening the file that will be communicated to the memory controller to verify whether the key generated from the passphrase matches what was previously stored in the OTT. A wrong passphrase will deny the opening of the file, and hence other 'curious' users will not be able to access the file even though they may have access permission.

Integrity of Filesystem Encryption Counters and OTT: Generally, the Merkle Tree is employed alongside the Counter-Mode encryption in order to protect the integrity of the counters. To protect the integrity of memory encryption counters as well as the filesystem encryption counters, we adapt the Merkle Tree implementation to additionally protect the filesystem encryption counter blocks. The Merkle-tree also covers the encrypted OTT region in the memory. However, it is also important to note that the integrity of the newly introduced fields (e.g., File ID), in addition to the counters in memory encryption, blocks must be protected. Note that this would not incur significant changes to the Merkle Tree since it calculates the MAC value for the whole 64B counter block regardless of its semantic.

Secure File Deletion: One major aspect is how to securely delete an encrypted file. A naive approach would just delete the file from software perspective, i.e., delete the `inode`, and declare that physical space free. Such an approach is typically not acceptable in critical infrastructure standards. For instance, the U.S. Department of Defense (DoD) mandates a standard (DoD 5220.22-M) that requires overwriting the file data multiple times. Unfortunately, given the limited write-endurance and gigantic file sizes can be hosted in NVM, it is preferred to use a more efficient mechanism to do that. Silent Shredder [9] is an NVM-friendly mechanism which can be used to enable fast shredding by repurposing the initialization vectors of encryption counters. However, when a file is shredded/deleted, even if a process provides the same key used for encrypting the blocks, it should return unintelligible results. For instance, if user X now has the key for a file, once the file is deleted, if user X manages to use the previous physical pages used for the deleted file, and provide the correct key, it should not be able to read the previous data from NVM.

Untrusted Operating Systems (OSes): In this paper, we assume a trusted OS that can help directly communicating keys, file IDs, Group IDs to the memory controller when a file is memory-mapped or during a minor page fault. However, recent security models advocate for operating in environments where the OS is untrusted, e.g., cloud systems. An example of such models is Intel's Software-Guard Extension (SGX), where applications need to only trust the processor chip. Each application in SGX can declare its enclave memory that is protected by the processor, and it is guaranteed that no other processes, including OS, can reveal information about the enclave or change its data without detection. We believe that applying the idea of hardware enclaves to our filesystem encryption scheme would require applications to directly communicate their key, file ID, and encryption mode to the hardware, which otherwise should have been done by the OS. We leave hardware-support for filesystem encryption in untrusted OSes as future work.

VII. Related Work

Several implementations of cryptographic filesystems have been proposed, examples are CFS [12], Cryptfs [39], eCryptfs [17] and TCFS [13]. NV-eCryptfs [40] provides software stack on top of eCryptfs in order to reduce the performance overhead of eCryptfs' software-based encryption mechanism. However, all of the above-mentioned approaches are software-based, and their overhead can be significant if compared with the fast direct accessible Non-Volatile memory reads and writes. While software-based decryption overhead could be marginal when accessing SSDs or HDDs, the overhead can be dominant for directly accessible files in fast NVM-based systems. We note that software-level encryption and decryption can be implemented at different layers in the system, although explicit software-based encryption/decryption would be required.

An orthogonal research direction is Full-Disk Encryption (FDE) [8], [26] and its counterpart memory encryption [9], [14], [34], [38]. This research direction focus is on hardware encryption of the whole storage or memory regardless of its semantic. Typically FDE is used transparently and executed inside the SSD or HDD; however, it provides a much coarser granularity than filesystem-level encryption. Furthermore, filesystem-level encryption provides protection from internal users trying to access other users' files. Note that many systems deploy both FDE and filesystem-level encryption to protect against different types of attacks. Such layering tactics are meant to provide stronger security, conceived by the U.S. National Security Agency (NSA), known as *defense-in-depth* [27].

Recent works on secure NVMs addressed the problem of persistence and security [7], [20], [24], [36], [37], [42]. In persistently-secure systems, it is required to also persist the accompanying security metadata, e.g., Merkle Tree nodes and encryption counters, along with the data. Failing to do so would leave the system either unrecoverable or insecure (due to the possibility of replay attacks). Liu et al. [24] observed that many parts of NVM memory do not require such strict persistence requirements and can be relaxed. Such a selective persistence mechanism is orthogonal to our scheme as it can be directly applied to relax counter persistence requirements to non-file regions, i.e., the memory range not formatted as PMEM space. Most recently, Ye et al. [36] leveraged encrypted ECC bits along with data in addition to a stop-loss mechanism to significantly reduce the overhead of persisting encryption counters. Although any suitable recovery scheme can work alongside our work, we adopt a counter persistency model based on Osiris and use it along with the filesystem encryption scheme. Apart from the counter recovery scheme, there are several other works that aim to provide faster recovery of the Merkle tree after a crash. Triad-NVM [10] adopted isolating the memory region as persistent and non-persistent regions in order to avoid depending on the Compiler/Programming level works and persisted intermediate level nodes to facilitate faster recovery. Anubis [6], on the other hand, maintains a shadow table that tracks the most recently updated counters and Merkle tree for faster recovery. Taassori et al. [33] propose a per-application integrity tree to prevent side channels and allow for higher cache utilization for security metadata. These works, although not directly related to the problem described in this paper, can work alongside FsEncr for better overall performance.

For the filesystem encryption in the Non-Volatile environment, especially in DAX based filesystem, we believe that there is much to explore. To the best of our knowledge, our paper is the first to address the security impact of using directly-accessible NVM-hosted filesystems. A recent related article discusses the reliability issues when using DAX-based systems [31]. In contrast, our work addresses security issues.

VIII. Conclusion

Directly-accessing NVM-hosted filesystems emerge as the most suitable access mechanism; however, it can challenge current security implementation. In particular, filesystem encryption for directly-accessible (DAX) filesystems is nearly-impossible to implement in software without giving up on the direct-access feature. However, the overheads of invoking software on each memory access would render emerging NVMs less attractive and, in fact, do not amortize its costs compared to other technologies. We have shown that filesystem encryption can be enabled along with direct-access with minimal OS changes and hardware support. We described the required changes and prototyped our implementation on the full-system cycle-accurate simulator with our software changes implemented on a modern Linux kernel. Overall, and with an extra performance overhead of only 3.8% for workloads representative of real-world application, we could support filesystem encryption along with direct-access.

Acknowledgement

This research was partially supported by National Science Foundation (NSF) through grant number CNS-1814417 and the National Research Foundation of South Korea through grant number NRF-2016K1A1A2912757.

References

[1] "Best practice for encryption," "https://media.kaspersky.com/pdf/b2b/Encryption_Best_Practice _Guide_2015.pdf", [Online; accessed 05-February-2020].

[2] "Encrypted keys for the eCryptfs filesystem," "https://www.kernel.org/doc/Documentation/security/keys-ecryptfs.txt", [Online; accessed 05-February-2020].

[3] "fscrypt," "www.kernel.org/doc/html/latest/filesystems/fscrypt.html", [Online; accessed 05-February-2020].

[4] "pmemkv," "https://github.com/pmem/pmemkv", [Online; accessed 05-February-2020].

[5] "The Linux Kernel," "https://www.kernel.org/doc/html/v4.18/filesystems/fscrypt.html", 2019, [Online; accessed 22-June-2020].

[6] K. Abu Zubair and A. Awad, "Anubis: Low-overhead and practical recovery time for secure non-volatile memories," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[7] M. Alwadi, V. R. Kommareddy, C. Hughes, S. D. Hammond, and A. Awad, "Stealth-persist: Architectural support for persistent applications in hybrid memory systems," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 139–152.

[8] Apple, "FileVault 2," "https://support.apple.com/en-us/HT204837", 2019, [Online; accessed 22-July-2019].

[9] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, “Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’16. New York, NY, USA: ACM, 2016, pp. 263–276. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872377

[10] A. Awad, Y. Solihin, L. Njilla, M. Ye, and K. Abu Zubair, “Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories,” in *International Symposium on Computer Architecture (ISCA)*, 2019.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[12] M. Blaze, “A cryptographic file system for unix,” in *Proceedings of the 1st ACM conference on Computer and communications security*. ACM, 1993, pp. 9–16.

[13] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano, “The design and implementation of a transparent cryptographic file system for unix,” in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 199–212. [Online]. Available: http://dl.acm.org/citation.cfm?id=647054.715628

[14] S. Chhabra and Y. Solihin, “i-nvmm: A secure non-volatile main memory system with incremental encryption,” in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA ’11. New York, NY, USA: ACM, 2011, pp. 177–188. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000086

[15] V. Costan and S. Devadas, “Intel sgx explained.” *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[16] M. A. Halcrow, “ecryptfs v0. 1 design document,” 2006.

[17] M. A. Halcrow, “ecryptfs: An enterprise-class encrypted filesystem for linux,” in *Proceedings of the 2005 Linux Symposium*, vol. 1, 2005, pp. 201–218.

[18] Intel, “Intel 3DXpoint,” ”http://newsroom.intel.com/docs/DOC-6713”, 2019, [Online; accessed 22-July-2019].

[19] D. Kaplan, J. Powell, and T. Woller, “Amd memory encryption,” *White paper*, 2016.

[20] V. R. Kommareddy, B. Zhang, F. Yao, R. Ewetz, and A. Awad, “Are crossbar memories secure? new security vulnerabilities in crossbar memories,” *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 174–177, 2019.

[21] B. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable dram alternative,” in *International Symposium on Computer Architecture (ISCA)*, 2009.

[22] Linux, “Linux Direct Access of Files (DAX),” ”https://www.kernel.org/doc/Documentation/filesystems/dax.txt”, 2019, [Online; accessed 22-July-2019].

[23] Linux, “Linux Programmer’s Manual: Linux Keyring,” ”http://man7.org/linux/man-pages/man7/keyrings.7.html”, 2019, [Online; accessed 22-July-2019].

[24] S. Liu, A. Kolli, J. Ren, and S. Khan, “Crash consistency in encrypted non-volatile main memory systems,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 310–323.

[25] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, “An analysis of persistent memory use with whisper,” in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1. ACM, 2017, pp. 135–148.

[26] NetApp, “NetApp Storage Encryption,” ”http://www.netapp.com/us/products/storage-security-systems/netapp-storage-encryption.aspx”, 2019, [Online; accessed 22-July-2019].

[27] NSA, “defense in depth. a practical strategy for achieving information assurance in today’s highly networked environments.”

[28] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, “Thynvm: Enabling software-transparent crash consistency in persistent memory systems,” in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 672–685.

[29] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, “Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly,” in *the proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40), 2007*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 183–196. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.44

[30] A. Rudoff, “Persistent memory programming,” *Login: The Usenix Magazine*, vol. 42, no. 2, pp. 34–40, 2017.

[31] S. Swanson, “Engineering Reliable Persistence,” ”https://www.sigarch.org/engineering-reliable-persistence/”, 2019, [Online; accessed 22-July-2019].

[32] systutorials.com, “eCryptfs,” ”https://www.systutorials.com/docs/linux/packages/ecryptfs-utils-111/ecryptfs-faq.html#no-ecryptfsac”, [Online; accessed 05-February-2020].

[33] M. Taassori, R. Balasubramonian, S. Chhabra, A. R. Alameldeen, M. Peddireddy, R. Agarwal, and R. Stutsman, “Compact leakage-free support for integrity and reliability,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 735–748.

[34] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, “Improving cost, performance, and security of memory encryption and authentication,” in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 179–190.

[35] C. Yan, B. Rogers, D. Englender, D. Solihin, and M. Prvulovic, “Improving cost, performance, and security of memory encryption and authentication,” in *the proceedings of the 33rd International Symposium on Computer Architecture (ISCA-33), 2006*, 2006, pp. 179–190.

[36] M. Ye, C. Hughes, and A. Awad, “Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories,” in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2018)*, no. 1. IEEE/ACM, 2018, pp. 135–148.

[37] M. Ye, K. Zubair, A. Mohaisen, and A. Awad, “Towards low-cost mechanisms to enable restoration of encrypted non-volatile memories,” *IEEE Transactions on Dependable and Secure Computing*, 2019.

[38] V. Young, P. J. Nair, and M. K. Qureshi, “Deuce: Write-efficient encryption for non-volatile memories,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’15. New York, NY, USA: ACM, 2015, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694387

[39] E. Zadok, I. Badulescu, and A. Shender, “Cryptfs: A stackable vnode level encryption file system,” Tech. Rep.

[40] L. Zhang, W. Liu, L. Cheng, P. Li, Y. Pan, N. Bergmann *et al.*, “Nv-ecryptfs: Accelerating enterprise-level cryptographic file system with non-volatile memory,” *IEEE Transactions on Computers*, 2018.

[41] K. Zmudzinski, S. Chhabra, R. Lal, A. N. Trivedi, L. S. Kida, P. M. Pappachan, A. Basak, and A. Trikalinou, “Technologies for controlling memory access transactions received from one or more i/o devices,” Jul. 25 2019, uS Patent App. 16/369,295.

[42] K. A. Zubair, S. Gurumurthi, V. Sridharan, and A. Awad, “Soteria: Towards resilient integrity-protected and encrypted non-volatile memories,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1214–1226.