SecureFS: A Secure File System for Intel SGX

Sandeep Kumar
IIT Delhi
India
sandeep.kumar@cse.iitd.ac.in

Smruti R. Sarangi
IIT Delhi
India
srsarangi@cse.iitd.ac.in

Abstract

A trusted execution environment or a TEE facilitates the secure execution of a workload on a remote untrusted server. In a TEE, the confidentiality, integrity, and freshness properties for the code and data hold throughout the execution. In a TEE setting, specifically Intel SGX, even the operating system (OS) is not trusted. This results in certain limitations of a secure application's functionality, such as no access to the file system and network – as it requires OS support.

Prior works have focused on alleviating this problem by allowing an application to access the file system *securely*. However, we show that they are susceptible to *replay attacks*, where replaying an old encrypted version of a file may remain undetected. Furthermore, they do not consider the impact of Intel SGX operations on the design of the file system.

To this end, we present *SecureFS*, a secure, efficient, and scalable file system for Intel SGX that ensures confidentiality, integrity, and freshness of the data stored in it. SecureFS can work with unmodified binaries. SecureFS also considers the impact of Intel SGX to ensure optimal performance. We implement a prototype of SecureFS on a real Intel SGX machine. We incur a minimal overhead ($\approx$ 1.8%) over the current state-of-the-art techniques while adding freshness to the list of security guarantees.

1 Introduction

In a typical cloud computing setting, it is necessary for users to run jobs on remote machines that need not be trustworthy [10, 14, 28]. Providing security in this setting is unfortunately not possible to achieve with software-only solutions [9, 21] because the remote OS cannot be trusted. Hence, it is necessary to have architectural support to ensure that the remote machine cannot steal secrets from the client's program or tamper with its execution. To support this paradigm, different processor vendors have created hardware-based trusted execution environments (TEEs) [4, 12], where it is not possible to read the memory state of a *secure* program even if the OS or hypervisor are compromised. Some popular examples of such TEEs are Intel SGX [20] and ARM Trustzone [4]. They ensure three essential properties to different extents: confidentiality, integrity, and freshness. *Confidentiality* means that it is not possible for a rogue program on the remote machine to read data without proper authorization, *integrity* does not allow undetected malicious tampering of the program's state running on the TEE, and *freshness* means that stale data (valid in the past) is not returned by read operations.

Even though such TEEs have solved most of the problems with secure remote execution as far as the compute part of the program is concerned, we are still dependent on the OS for some key services, notably file system accesses [5, 8, 10, 14, 33]. Applications require access to the file system to read the data, save results, saving partial state, etc. Hence, it is necessary to also provide a *secure file system* to such remotely executing programs.

Prior works have proposed solutions for creating a secure file system that is immune to most of the known attacks on file systems. However, they have ignored an important type of attack known as the *replay attack* (freshness property). In this case, a malicious adversary can replace the current copy of some blocks of files with an older copy and remain undetected. Such replay attacks tamper with the integrity of the execution unbeknownst to the TEE; they have also been shown to be potent enough to change the execution of a secure program such that it reveals its sensitive secrets [7, 29]. We were able to successfully mount a replay attack on two state-of-the-art file systems: Graphene (protected file mode) [10] and Nexus [14]. This motivated us to design a file system called *SecureFS* that additionally provides the property of freshness, while simultaneously minimizing the performance overhead.

We first analyzed different applications that have frequently been used in papers [16, 22, 35, 40] in this area. Furthermore, we used the insights obtained to refine SecureFS's design. The key idea is that we need to modify the design of file system structures to efficiently store the metadata such that we can easily provide the three security guarantees: confidentiality, integrity, and freshness. Moreover, we found that it was not possible to trivially extend prior proposals to additionally guarantee freshness; it was necessary to design a bespoke file system such that freshness can be guaranteed with negligible overheads. We propose both a FAT table-based and an inode-based design, and thoroughly evaluate their performance in a TEE setting.

The list of contributions is as follows.

Anonymous Submission to RAID, 2021
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1. We show that the current state-of-the-art secure file systems are susceptible to replay attacks by mounting successful attacks on them.
2. We thoroughly characterize workloads that are used in Intel SGX related literature, in terms of their file-access behavior. Based on the results, we derive a set of insights.
3. We study the impact of Intel SGX constructs on the performance of different file system designs.
4. We propose a novel design of a file system to ensure all three security guarantees while at the same time incurring a minimal slowdown ($\approx 1.8\%$).

The rest of the paper is organized as follows. We start by providing the necessary background in Section 2. We discuss the related work and how we mounted replay attacks in Section 3. Subsequently, we characterize relevant workloads in Section 4 and obtain insights regarding their behavior. Using the insights obtained we present the design and implementation of SecureFS in Section 5. We evaluate the performance of SecureFS in Section 6, and finally, conclude in Section 7.

2 Background

In this section, we cover the background required for the rest of the paper.

2.1 A Primer on Intel SGX

Intel Secure Guard eXtensions (SGX) [20] is a *trusted execution environment*, or TEE, solution from Intel. It allows an application to execute securely on a remote machine. At boot time, a part of the system memory is reserved for SGX, called PRM or *processor reserved memory.* A part of the PRM is used to store SGX metadata, and the rest of it is used for user mode applications. The latter is called the EPC or *enclave page cache.* To execute an application within SGX, the application is allocated pages from the EPC along with other bookkeeping structures, collectively known as an *enclave* [12, 18, 19]. While executing an enclave, the CPU is said to be in a *secure mode.* In this mode, the traffic between the last level cache (LLC) and main memory is transparently encrypted by a dedicated hardware unit called the Memory Encryption Engine, or MEE [12]. The MEE is also responsible for maintaining the data's integrity and freshness.

Although Intel SGX ensures secure execution, it does not provide protections against side-channel attacks [7, 16, 25, 29]. In general, all schemes that are based on SGX also share its vulnerabilities.

2.1.1 Challenges with System Calls. A major limitation of SGX is that it does not allow system calls from an enclave due to security concerns. As a result, to make a system call, the application (running within the enclave) needs to exit the secure mode, make the system call, collect the results and return to the secure mode. Specifically, to enter the secure mode, applications need to call an `ECALL` function and to exit the secure mode, they need to call an `OCALL` function.

2.1.2 Enclave Entries and Exits . During execution, an enclave may have to switch from the secure to the unsecure mode many times due to context switches, exceptions within the secure code, or a call to an `ECALL` or `OCALL` function. While transitioning from the secure to the unsecure mode, it is necessary to flush the TLB entries that contain mappings for the secure pages . This is done to prevent data leaks from the caches [12].

When the enclave switches back to the secure mode, it starts with a cold TLB and encounters many TLB misses, resulting in many page table walks. While adding entries in the TLB, if the entry points to a page in the EPC then it is validated by SGX. For this purpose, SGX maintains an inverted page table in the EPC called the Enclave Page Cache Map or EPCM. It contains details about every EPC page's state, its owner, and the corresponding virtual address for which it was allocated. SGX ensures that the TLB entry is inserted only for the process that spawned the enclave, and contains the same virtual address for which it was initially allocated.

Finally, during an enclave exit, the context of the executing enclave is saved in the secure region along with the register values. These functions have overheads of roughly about 14,000 cycles (measured by Weisse et al. [37]).

2.1.3 Eviction from EPC. The size of the EPC is typically 92 MB in current SGX implementations [19]. However, Intel SGX allows an enclave to allocate more memory. It transparently handles the eviction of pages from the EPC to the untrusted memory. If an enclave's working set is much larger than the size of the EPC, it will trigger frequent page-faults (as the pages have to brought in from the untrusted region to the EPC). A single page-fault takes approximately 64,000 cycles, as measured by Liu et al. [23]. Note that security is not compromised in this process.

All of these factors collectively increase the overhead of making a system call in the secure mode. Hence, it is a wise design choice to limit the transitions from and to secure mode, and limit the amount of trusted memory used.

2.2 Metadata organization of a File System

Here, we discuss some of the metadata organizations for a file system.

2.2.1 FAT table-based organization. In this organization, the metadata is organized in a single table called the *FAT table* – a large array. Here, if a file consists of multiple blocks, then the multiple entries are stored as a linked list within the table, as shown in Figure 1. Each entry in the FAT table has two pointers: one pointer to a block in the disk corresponding to the entry, and the other points to the FAT table entry corresponding to the next block in the file.

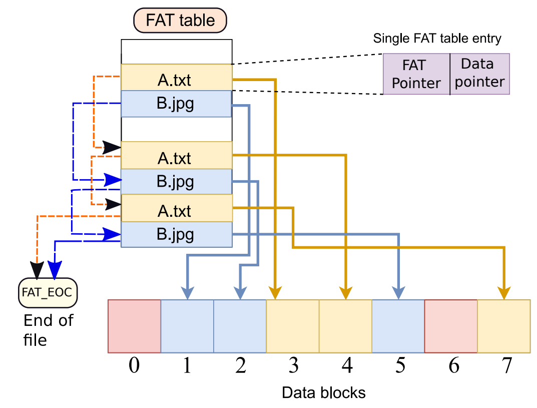


Figure 1. Design of a FAT table-based metadata organization.

2.2.2 Inode-based organization.

In this organization, we have an *inode* for every file in the file system. An inode is organized in a tree-like structure (see Figure 2). Here, the first few data blocks of a file are directly pointed to. As we increase the size of the file, the subsequent blocks are pointed to using indirect blocks. The level of indirection can be single, double, or even triple, depending upon the size of the file and the amount of metadata entries a single level can hold.

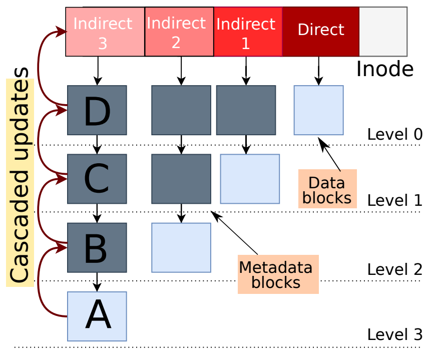


Figure 2. Design of an inode-based metadata organization.

Cascaded Operations: In the case of a secure file system, typically, the key of a block is stored in its parent [14]. The key is changed for every block update to ensure its freshness. As shown in Figure 2, an update to the data block A, which is at the leaf-level, will encrypt it with a new key. The key will be stored in A's parent, i.e., in metadata block B, essentially modifying B. This triggers an update operation till the root, which is in the inode– cascading of write operations. Note that this is required only when writing the blocks to the untrusted region. An update in metadata block C, can be cached till C is flushed to the untrusted region. A similar situation happens while reading the blocks. Again, reading block A requires access to block B since it contains the key required to decrypt block A. This dependency is present till the root level – cascading of read operations.

This issue has been pointed out by the authors of Nexus [14]. However, they do not take any steps to mitigate this. We show (Section 6) that *cascading* is expensive, and further exacerbated in an SGX setting.

2.3 Attacks on File Systems

The most popular attacks on secure file systems are as follows:

File System Snooping attack A malicious OS keeps track of all the calls made by the application to the file system, along with its arguments [11, 14, 30, 33]. This gives a clear picture of what the application is doing; this may further allow the OS to access the data that the application is reading or writing.

Replay attacks In this case the malicious OS replaces a file and its hash (used to maintain integrity) by a previously seen $\langle file, hash\rangle$ pair. Secure file systems that rely on simply verifying the hash are susceptible to this kind of attack [10, 14].

Page fault attacks Here, a malicious OS exploits its control of the application's page table. The OS invalidates all the mappings in the TLBs and page tables, which results in a page fault when the application tries to access a page. This provides the OS with a sequence of pages accessed by the application. This has been shown [7, 39] to leak some amount of secret information.

In this paper, we address the first two attacks. Tackling a page fault attack requires a change (in the OS and hardware) in how the page tables are managed by Intel SGX. This is beyond the scope of the paper.

3 Related Work and Replay attacks

A secure file system uses the untrusted host file system to store files in an encrypted format [10, 14]. To mount a replay attack, an attacker copies the encrypted blocks of a file, say F, to a different location on the system. The secure file system continues updating F, changing its state to $\hat{F}$. Now, the attacker replaces the encrypted blocks of $\hat{F}$ with that of F. If this remains undetected, i.e., the file system accepts the copied blocks as the latest blocks of the file, then the replay attack succeeds. We show that the current state-of-the-art secure file systems – Graphene protected files [10] and Nexus [14] – are susceptible to this attack.

3.1 Graphene Protected Files

Graphene [10] is a shim layer that can run unmodified binaries securely within an SGX enclave. This feature saves developers many hours of effort in porting their code to Intel SGX. As already mentioned, within an SGX enclave, an application cannot make system calls. Graphene solves this problem by providing system call support as a library service. This feature also allows transparent access to the file system on the host OS.

Graphene has a *protected files* mode – henceforth called *GraphenePF*– that protects the confidentiality and integrity of a file while reading or writing to it. All the protected files are encrypted using a single key that remains fixed [27]. The encrypted file contains the hash of the data and its path. Incorporating the path in the hash prevents it from a *file-swapping* attack.

3.1.1 Attack:

Since all the protected files are encrypted using a single key, we were able to replace an encrypted file with an old encrypted version of the same file, and successfully mount a replay attack on it. GraphenePF has no way of telling whether the current file is the latest or not, as both the conditions that will trigger an error, integrity failure or path check failure, still hold.

3.2 Nexus

Nexus [14] is a secure file system for Intel SGX that guarantees confidentiality and integrity of the files stored in it. Its design is predominantly targeted towards ease of sharing secure volumes among different trusted users, with an efficient access revocation mechanism [14]. Nexus has a much more complex organization of files as compared to GraphenePF. Nevertheless, it is also susceptible to a replay attack. As noted by the authors of Nexus, the design of Nexus is susceptible to a *forking attack*, where the updates from one client to the file system are kept hidden from another client – this definition is subsumed within our definition of a replay attack.

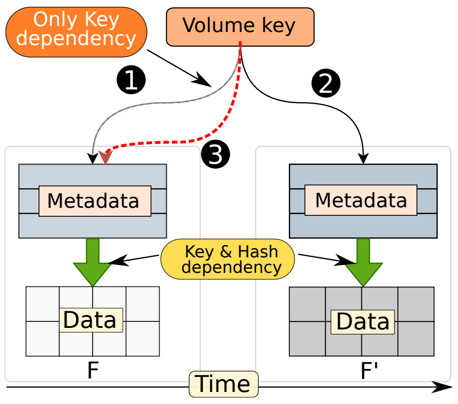


Figure 3. Mounting a replay attack in Nexus [14].

Nexus, prior to storing a file on the disk, splits the file into fixed-size blocks. It then encrypts each block with a different key, and stores the key within the metadata of that file. The key is changed every time the corresponding block is written to. The data blocks are also integrity checked. In this setting, as there is a *(key & hash)*-dependency, it is not possible to replay old encrypted data blocks as they will be decrypted with a different key.

However, while unmounting the volume, Nexus encrypts the metadata structure of all the files with a single key, called the *volume key* (see Figure 3). This is obtained from the user via remote attestation [12]. Even though the integrity of the metadata is preserved, the use of a single key (only *key*-dependency) allows an attacker to replay the old ⟨data, metadata⟩ pair, which is accepted as a valid (and latest) file by Nexus during the next mount.

3.2.1 Attack:

As shown in Figure 3, to mount a replay attack, ❶ we mounted the Nexus file system [13] and created a new file (F). As per the design of Nexus, this file is created in a memory mapped region and is flushed to disk (in an encrypted format) at the time of unmounting. Then, we unmount the file system and copy the encrypted blocks of F to a different location. The encrypted blocks can be easily identified as they were the newest blocks (even though Nexus obfuscates the file name with an 8-byte unique id called `UUID`). ❷ We then mount the file system again, modify the file (making it F') and unmount the volume. ❸ After the second unmount, we replace the encrypted data and metadata blocks of F' with that of F (saved in the first step). In the subsequent mount, Nexus recognized the replayed blocks of the file F as the latest blocks.

4 Characterization

In this section, we start out by choosing a set of workloads that have been used in prior work that uses Intel SGX as the baseline platform [16, 22, 35, 40], along with a few more workloads that are similar in nature [2, 3, 6, 31]. The workloads are summarized in Table 1.

4.1 Experimental Setup

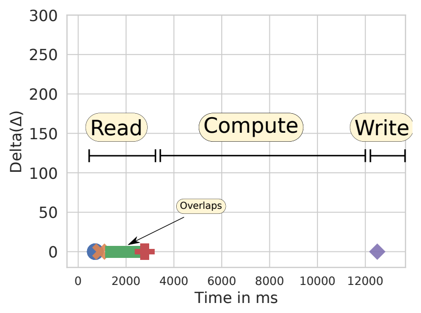
We used the *strace* tool [38] to profile the file system calls made by an application. The trace generated by *strace* contains the file system function calls, their arguments, and the return values. This gives a clear picture of the interaction of the application with the file system.

All the applications were executed on a machine with an Intel Core i7-10700 CPU (16 cores) running at a frequency of 2.90 GHz, 16 GB of DRAM, and 256 GB of SSD (details in Table 2).

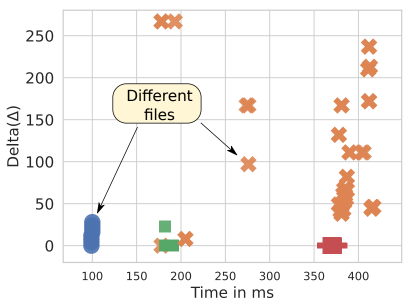
4.2 Access patterns

The access pattern of an application tells whether it accesses (read or write) data sequentially or randomly. To determine this, we track two metrics: *stride*, the length of a read or write access, and *Delta* Δ, the movement due to a *seek* call. A *read* (or *write*) call reading (or writing) 1000 bytes has a stride of 1000. A *seek* (say i^{th}) moving the file pointer by 100 bytes has a Δ_i of 100 . Total movement is defined as $\bar{\Delta} = \sum_{i=0}^{N} |\Delta_i|$, here, N, is the total number of *seek* calls.

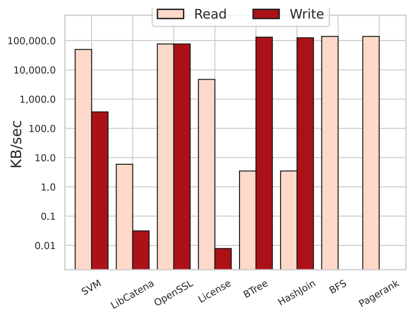
Sequential access patterns have $\bar{\Delta} = 0$. This is mainly due to $N = 0$, i.e. no seek calls. Whereas random-access



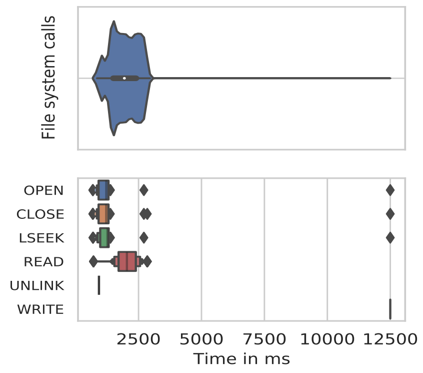
(a) Sequential access pattern for SVM, showing read, compute, and write phases.



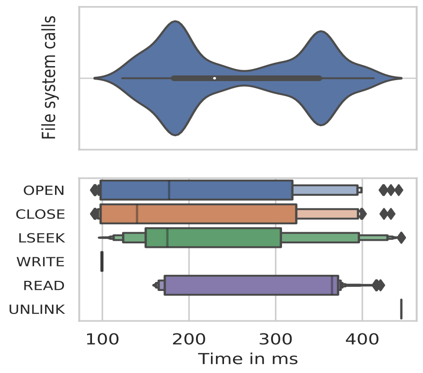
(b) Random access pattern for License3j (only the license check phase).



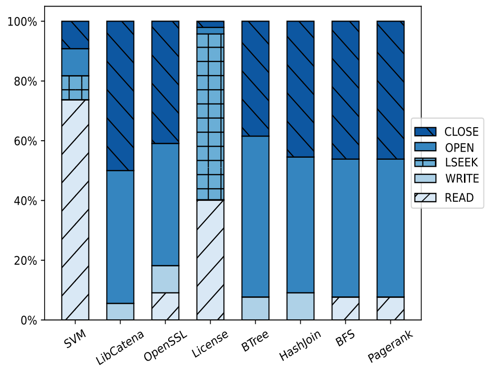
(c) Throughput of each workload



(d) Distribution of calls for SVM



(e) Distribution of calls for License3j (only the license check phase).



(f) Total file system related calls.

Figure 4. Characterization of workload in terms of their access patterns, file system calls, amount of data accessed, and total files accessed.

Benchmark	Description & Setting	Data accessed
SVM [17]	Classification of data using the support vector machine (machine learning). Classifies 100,000 samples.	50 MB
LibCatena [24]	Generating blocks in a blockchain. Reads the genesis block and adds blocks to the chain.	0.005 MB
OpenSSL [26]	Cryptographic algorithms library Reads a file, decrypts it, verifies it, processes it, and saves the result in an encrypted file.	150 MB
License3j [34]	License verification Check a license file before executing a compute-intensive task.	4 MB
BTree [2]	BTree Look-ups (used in databases) Table with 2M elements with 1M lookups.	128 MB
HashJoin [3]	Hash table probing Table with 1M elements	122 MB
BFS [32]	Graph traversal Processes a symmetric graph with 1M nodes and 10M edges.	135 MB
Pagerank [32]	Ranking of nodes in a graph Processes a symmetric graph with 1M nodes and 10M edges.	135 MB

Table 1. Description of the workloads used in the paper.

patterns have different values of Δ_i. In most (7 out of 8) of the workloads, the access pattern is sequential. We show a representative plot of the values of Δ_i across time in Figure 4a for the SVM workload. It remains equal to 0. Each data point represents a sample (many are not visible because they overlap). However, we saw a lot of random access for the license manager in Figure 4b (different values of Δ).

4.3 Distribution of File Access Calls over Time

Let us now understand the pattern of file system calls made by the applications. Figures 4d and Figures 4e show violin plots for the relative proportion of system calls in SVM and License3j, respectively (SVM is representative of 7/8 workloads, and License3j is an outlier). The top half of each plot shows the pdf (and its mirror image) for the distribution of file system calls over time. The bottom half shows a box plot (0, 12.5, 25, 50, 75, 87.5, 100 percentiles) of how file system-related system calls are invoked over time (normalized only with respect to themselves).

Figure 4d tells us that, for SVM, almost all the open calls happen in the first 1250 ms. Reads happen till 2500 ms at which all the data is loaded in the memory. The writes happen later (towards the end) to save the results. For License3j, as shown in Figure 4e, during the license check phase, the application opens different files and reads them in a random manner.

4.4 Throughput

We observe that the throughput ranges from roughly 1 KB/s to 100,000 KB/s (Figure 4c). To do the same in a secure file system, we need to finish all the computations, memory accesses, and system calls within that time. One more noticeable trend in this figure is that sometimes reads outnumber writes (except for BTree and HashJoin), as shown in Figure 4f. Thus, we need to optimize our file system for reads; moreover, reads are often on the critical path.

4.5 Key Insights

Based on the characterization, we arrive at few key insights. The design of our secure file system is based on this. The insights are as follows (figures for insights 3, 4, and 5 were not added due to a lack of space):

1. Applications access data in a sequential manner.
2. Most of the accesses are read operations and the writes are generally done at the end to save the results of the execution.
3. The file system hierarchy has 2-6 levels. (moderately deep).
4. The total number of files accessed by an application can go up to 1000.
5. At a given time, an application has a maximum of 10 open files.

5 Design and Implementation

In this section, we discuss the design and implementation of SecureFS.

5.1 Design Goals

- Ensure confidentiality, integrity, and freshness of the data present in the file system.
- Use the insights from workload characterization to optimize the performance of SecureFS.

5.2 Threat Model

We assume the traditional threat model used by Intel SGX. The attacker has access to the physical machine and the code of the application and the file system. Any modification to the application's or file system's code will be detected by Intel SGX. We do not consider side-channel attacks and denial-of-service attacks, as they do not fall under the purview of Intel SGX [12].

5.3 Overview of the Design

A high-level design of SecureFS is shown in Figure 5. The file system related calls from a secure application are captured by SecureFS and processed within the SGX environment.

SecureFS can be configured in two modes: ❶ We can either have a per-process SecureFS volume, ❷ or a shared volume among multiple secure processes. In the former case, each secure application is linked with a shim layer that captures the file system related calls and processes it within the SGX environment. In the latter case, SecureFS is executing as a secure process in Intel SGX. Every other secure application is linked with a shim layer that captures the file system related calls from the application and forwards them to the SecureFS processes via a shared memory channel. The SecureFS process processes the request and sends the result back via the same channel. This design allows us to provide concurrency because a single SecureFS process is responsible for handling all the requests and maintaining the consistencies of the data structures. Given that they are designed in a similar manner, for the rest of the discussion we assume a per-process SecureFS volume.

5.4 Organization

We split a SecureFS volume into 4 KB blocks – henceforth called *chunks* (Figure 5). We also reserve 256 bits within a chunk to store its hash (0.78% of 4 KB). Notwithstanding internal fragmentation issues, this design choice helps us simplify our design. All the operations in SecureFS happen at the chunk level. Like Nexus [14], we use per-chunk keys to ensure the confidentiality of chunks. The key of a chunk is stored in the corresponding metadata entry. A metadata entry of a chunk contains the unique ID of the chunk (discussed next, Section 5.5), a 128-bit key for the chunk, and a *modified* bit. This can be augmented to store additional data based on the metadata organization used (e.g. a next pointer in the case of a FAT Table-based organization).

The design of SecureFS is independent of the metadata organization. As per the requirements, a FAT Table-based or an inode-based organization can be used. The basic abstraction is that the metadata should have a one-to-one mapping with the data chunks, i.e. the metadata contains one entry per data chunk in the SecureFS volume, and vice versa.

5.5 Storage on the Host File System

To store a chunk on the untrusted host file system, we first fetch its metadata entry (details in Section 5.9.2). A chunk is only written to disk if the modified bit is set (Insight 2). Prior to writing, we calculate the hash of the chunk using the SHA-256 algorithm, and store it in the reserved bits within the chunk. After hashing, the chunk is encrypted using the AES-128 algorithm [15] (cipher block chaining (CBC) mode). The key required for the encryption is randomly generated and is stored in the corresponding metadata entry.

We assume that in the host OS's file system we have a single directory that contains all the encrypted chunks (each stored as a 4 KB file). To anonymize the accesses we have hidden the directory structure. Each chunk is simply assigned a 8-byte unique id – hereinafter referred to as the *UID*. The next time when the same chunk is written back, we change its UID by generating a new 8-byte random number. Every time we write back a chunk we generate a new UID, and thus create a new file. The old file needs to be deleted – this need not be done immediately and can be done at a later point of time (to confuse the adversary).

To read a chunk, we get its metadata entry. We use the UID to read the content of the file from the untrusted host-OS. Using the key in the metadata entry, we decrypt the chunk and calculate its hash (ignoring the last 256 bits). We then match the calculated hash with the hash stored in the last 256 bits. If it matches, the read is successful otherwise an error is thrown, the volume is made read-only, and unmounted.

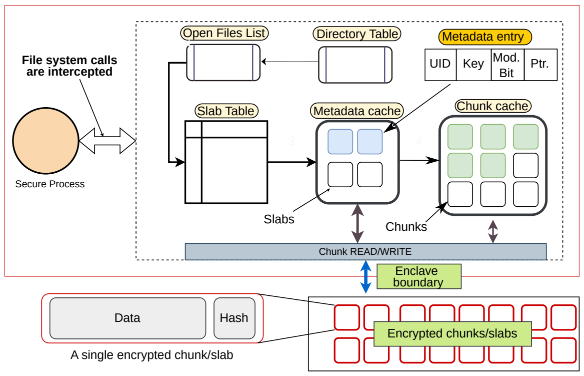


Figure 5. Overview of SecureFS showing the key data structures, enclave boundary, chunk organization, and metadata entries.

5.6 Key Structures of SecureFS

5.6.1 Chunk Cache and Metadata Cache. For a 1-GB file system, the amount of metadata required is ≈ 9 MB (9.7% of the EPC) for a FAT Table-based organization (see Section 5.8). It is not desirable to pin this in the trusted memory, since this might affect the performance of other secure applications. Also, reading or writing a chunk for every read or write operation from the untrusted medium is an expensive operation as it involves hashing and encryption.

To ensure optimal performance, we maintain a small cache of decrypted slabs and chunks – *metadata cache (**M**) and chunk cache (**C**)*, respectively – in the trusted memory and dynamically manage it. We keep the size of these caches small to ensure minimal pressure on the EPC and reduce the expensive EPC page faults [23]. All the data and the metadata accesses happen from **C** or **M**, respectively. For the rest of the discussion, we assume a 2 MB chunk cache and a 100 KB metadata cache. We present a sensitivity analysis in Section 6.6 to justify these values.

5.6.2 Slabs and Slab Table. As noted earlier, we divide the data region into 4 KB chunks. Similarly, to ease the maintainability of the metadata, we propose to divide the metadata region into a set of contiguous regions known as *slabs*. A slab is an analog of a chunk; we use different naming conventions to differentiate between data and metadata. A slab is written and read from the disk using the same mechanism as defined for the chunks in Section 5.5 (the hash is stored in the last 256 bits).

Consequently, we need a structure to hold the keys of the slabs in the trusted region – when a slab is flushed to the disk (similar to the keys of a chunk). For this purpose, we introduce a novel structure called the *slab table* that contains one slab entry for each slab. Each slab entry contains a slab UID, a 128-bit key, a pointer to the metadata cache (**M**), and a modified bit. In our implementation, the size of the slab table is ≈ 56 KB. We pin the slab table in the trusted memory, thus, providing a root-of-trust for the metadata entries. A slab table prevents replay attacks on the file system (details in Section 5.10).

5.6.3 Directory table. We maintain the directory table as a flat-hierarchy (Insight 3), in an array, since the total number of files accessed by a secure program is small (Insight 4). The directory table contains absolute path names mapped to their metadata entries. The directory table is pinned in the trusted memory (size is few KBs).

5.6.4 Open File List. We maintain a small list of the 100 most recently used file paths (Insight 5), and their corresponding metadata pointers (current location of the file pointer within the file). This is also pinned in the trusted memory.

5.7 Metadata Organization

The organization of metadata plays an important role in determining the performance, especially in the Intel SGX setting because of the high cost of enclave transitions and EPC faults [12, 23]. However, to the best of our knowledge, it has received no attention in the literature [8, 10, 14, 33].

We implement SecureFS using two different metadata organizations: a FAT Table-based and an inode-based organization. They are called SecureFS-FAT and SecureFS-inode, respectively, in the rest of the paper. We present a detailed performance comparison of the two organizations in Section 6.2. We discuss the working of SecureFS using a FAT Table-based organization. We point out some differences for an inode-based organization in Section 5.12.

5.8 SecureFS-FAT

As mentioned before, for each chunk we have a metadata entry – a FAT table entry in this case. Theoretically, if the size of the file system is K KB, then we have $K/4$ FAT table entries arranged as a linear array. This makes lookup of a FAT entry for a chunk easy. The entry for chunk i is the location at the ith index of the FAT table. This design allows for easy metadata lookups for sequential operations (Insight 1).

Each entry of a FAT table entry contains three pointers: UID of the chunk it is pointing to, a 128-bit chunk key, one pointer to the *chunk cache*, and a pointer to the FAT table entry of the next chunk in the file. If the size of our file system is 1 GB, then we shall have 256K chunks, and will require the same number of FAT table entries. The size of the FAT table in our implementation is $\approx$ 9 MB.

5.8.1 Mounting a Volume. To mount a volume, we fetch the volume key and the hash of the slab table from a trusted third party via remote attestation. The key is used to decrypt the superblock stored on the host file system. The total number of chunks required for a super block is stored in the first chunk. The superblock contains the slab table and the key and hash for the directory-table. We calculate the hash of the slab table and ensures that it matches with the value provided by the trusted third party. This ensures that a *volume replay attack*, where an attacker instead of replaying chunks of a file replays the old state of an entire volume, cannot be mounted. We use the directory key to populate the directory table (after verifying the hash). We start with an empty metadata cache (**M**) and chunk cache (**C**), and manage them dynamically based on the file system operations.

5.9 File System Operations

We discuss the following file system operations, *open*, *read*, *write*, and *close*, using a running example, as shown in Figure 6.

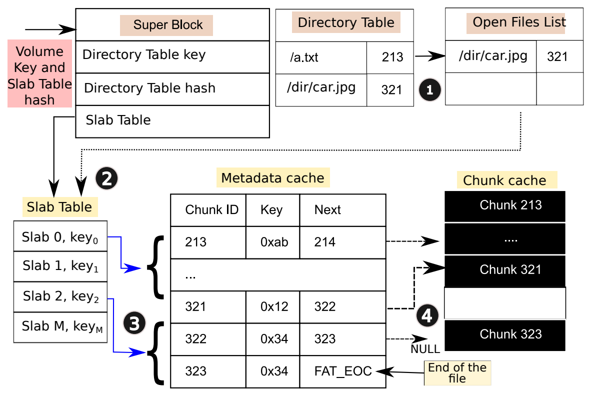


Figure 6. Steps in opening and reading a file **/dir/car.jpg** in SecureFS-FAT.

5.9.1 Opening a File. To open a file with a given name, we linearly scan the directory table to get its first FAT table index. As the total number of files in a volume is small (Insight 4), a linear scan is not expensive, and is only done while opening a file. If the file is not present (and the `O_CREAT` flag is set), we create a new entry in the directory table, and assign an empty FAT table index as its first entry. We then add an entry into the open files list: $\langle$absolute file path, FAT table index$\rangle$ (step ❶). The open file lists tracks the location of the file pointer within the file.

5.9.2 Reading a File. To access a file, assuming that the file is already open, we fetch the current location of the file pointer (the FAT table index) from the open files list. We use this index to calculate the slab id (step ❷). If the FAT table index is 321, and each slab contains 120 slab entries, then the slab id will be $\lfloor 321/120 \rfloor = 2$, and the offset will be $321 \: mod \: 120 = 81$, i.e. the 81^{th} entry of the 3^{rd} slab (slab index starts from 0).

We fetch the corresponding slab entry and analyze the metadata cache pointer. If the pointer is not `NULL` (metadata-hit), we read the corresponding metadata entry (step ❸), else if the pointer is `NULL` (metadata-miss), we fetch the slab (using the UID) and then read the metadata entry.

The metadata entry points to the next chunk of the file. We fetch the metadata entry of the next chunk (might be in a different slab) and then check its chunk cache pointer. If the pointer is not `NULL`(chunk cache-hit), we read the chunk (step ❹). If the pointer is `NULL` (chunk cache-miss), we read the chunk into the chunk cache.

Note that reading a chunk into either of the caches might require an eviction. We write the contents of a chunk (or a slab) being evicted using the steps described in Section 5.5. If the chunk (or slab) was modified, then a new key is generated and is stored in the corresponding metadata entry (or slab entry), with the pointer to the cache updated to `NULL`.

5.9.3 Writing a File. For writing, we ensure that the required chunk is in the chunk cache using the same method as for a read. Once we have the chunk cache index, we update the chunk with the required data, and set the modified bit in the metadata entry.

5.9.4 Closing a File. While closing a file, we write back all the modified chunks (encrypted) to the SecureFS volume and update all the FAT table entries.

5.9.5 Unmounting a Volume. While unmounting a volume, we close all the open files, writeback all the data and metadata, and update all the slab entries. We then calculate the hash of the slab table and send it to the trusted third party. If the hash is successfully sent, then we update the hash of the directory table in the superblock, and write encrypted chunks of the superblock to the disk (using the volume key). Optionally, we can also change the *volume key*, and send it to the remote third party along with the hash.

5.10 Preventing Replay Attacks

The slab table enforces a (*key* and *hash*)-dependency between the metadata and the superblock of the file system (not present in Nexus). An attacker can either replay the entire volume, a superblock chunk, a slab, or a chunk. The key+hash from the trusted third party prevents replaying a superblock chunk or the entire volume. Replaying a slab or a chunk will also cause integrity checks to fail as they will be decrypted using a key that is not the same as the one used to encrypt it. An attacker can try to replay a pair: ⟨a slab and a chunk ⟩ (the metadata entry of the chunk is in the slab). This will cause an integrity check failure while reading the slab as it will be decrypted with a key stored in the slab table. Given that the key changes every time, the integrity check will fail.

5.11 Concurrency and Consistency

As already mentioned, upon detecting an attack, SecureFS makes the volume read-only and unmounts it. At the next mount, it might be possible that the offending chunk is read again, triggering another unmount. This will continue till the attacker replaces the offending block with the original one.

A tricky scenario is as follows: the host OS decides to maliciously stop the machine at an arbitrary point in time. The next time we mount the volume, we will read the slab table and use it to verify the slabs. Note that if the volume was not closed, some of the slab entries might not be synchronized with the slab table entries. In this case, if we wish to partially recover, we can verify the hashes of the slabs, and then delete all those metadata entries whose slab hashes do not match.

5.12 SecureFS-inode

For an inode-based metadata organization, each file has a corresponding inode that points to a data chunk, either directly or indirectly. Here, the directory table for each file points to an inode. The inodes are kept in the trusted memory due to the small number of total files (Insight 4). The open files list maps a file to an inode and contains the file pointer (chunk id and offset within the chunk). The metadata entry corresponding to a chunk may require access to different slabs based on the depth at which the data chunk is located (cascade operations). The algorithm to read slabs and chunks remains the same as that of SecureFS-FAT.

6 Evaluation

In this section, we evaluate the performance of SecureFS in two settings: ❶ with a FAT Table-based metadata structure and ❷ with an inode-based metadata structure (implemented on top of state-of-the-art platforms). Table 2 shows the setting of the test-bed used for the evaluation and Table 3 lists the notations used in the rest of the paper for discussion. We report the mean values of 10 runs (seen to be enough), unless otherwise stated.

Hardware Setting		
Model: Intel Core i7-10700 CPU, 2.90 GHz	DRAM: 16 GB	Disk: 256 GB (SSD)
CPUs: 1 Socket, 8 Cores, 2 HT	L1: 256 KB, L2: 2 MB, L3: 16 MB	
AES hardware support: YES	SHA hardware support: **NO**	
System Settings		
Linux kernel: 5.9	DVFS: fixed frequency (performance)	ASLR: Off
Python version: 3.6	Java version: 1.8	GCC: 9.3.0
SGX Settings		
PRM: 128 MB	Driver version: 2.11	SDK version: 2.13

Table 2. System configuration

Notations	
Application + file system data is protected	
G-PF	Graphene with the protected files feature ON.
G-SF-F	Graphene with SecureFS-FAT.
G-SF-I	Graphene with SecureFS-inode.
Only file system data is protected	
Nexus	Current state of the art providing only encryption and integrity.
SF-F	SecureFS-FAT
SF-I	SecureFS-inode

Table 3. Notations used in the paper for discussion

6.1 Evaluation strategy

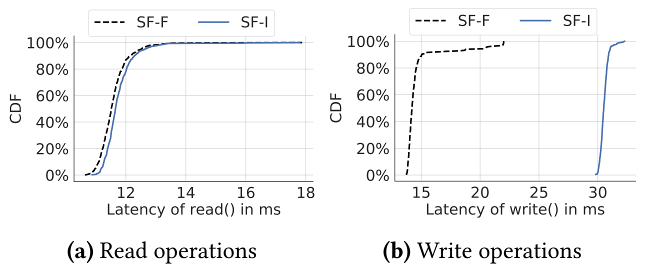
- **SecureFS-FAT vs SecureFS-inode**: We evaluate the performance of the two metadata organizations and point out certain limitations of the inode-based structure.
- **I/O-intensive workloads:** We compare the performance of SecureFS with GraphenePF for the popular file system benchmark Iozone.
- **Comparison with GraphenePF and Nexus**: We evaluate the performance of SecureFS with GraphenePF and Nexus using a set of synthetic and real world workloads (Table 1).
- **Sensitivity analysis**: We perform a sensitivity analysis of SecureFS for different chunk sizes and cache sizes.

The overhead of SecureFS is expected to be positive because we are providing more security guarantees. However, due to our optimizations, it is negligible ($\approx 1.8\%$).

6.2 SecureFS-FAT vs SecureFS-inode

Here, we compare the performance of SecureFS-FAT (SF-F) and SecureFS-inode (SF-I). As already noted, an inode-based implementation suffers from cascaded updates due to a tree-based organization. Figure 7 shows the latency of read and write operations. The latency of the read operations remains the same in both the cases (Figure 7a), as we do not update the keys if there are no modifications to a data chunk. However, for the write operations, the latency for SecureFS-inode is significantly higher (2×). This is due to the cascaded updates.

6.2.1 Discussion: Since the size of the chunk cache (2 MB) is significantly smaller than the amount of data being written (1 GB), it causes many chunk cache evictions – resulting



(a) Read operations **(b)** Write operations

Figure 7. CDF [1] of read and write latencies for Iozone: sequentially reading and writing 1 GB of data.

Setting	Metadata hit-rate	Chunk hit-rate
SF-F	99.66%	0.10%
SF-I	91.89%	0.10%

Table 4. Cache statistics for SecureFS-FAT and SecureFS-inode with Iozone while reading and writing 1 GB of data. used.

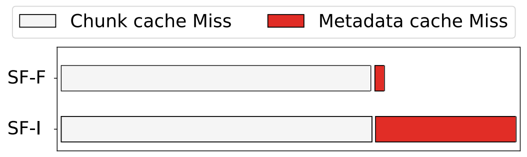
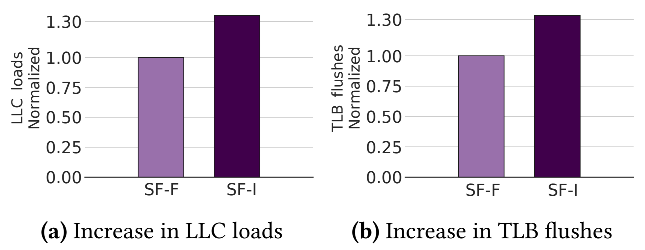


Figure 8. Total time taken to handle chunk cache misses and metadata cache misses. Iozone writing 1 GB of data. The x-axis shows the time.

in frequent metadata updates. Due to cascaded updates in SF-I, the time taken to handle metadata misses in SF-I is significantly higher (15×) than that of SF-F (see Figure 8).

Table 4 shows the hit rate of the metadata cache(**M**) and the chunk cache(**C**). For SF-I, out of all the **M** accesses, 23% were due to cascaded updates. This results in cache pollution bringing its hit rate down to 91.89% (as compared to 99.66% for SF-F).



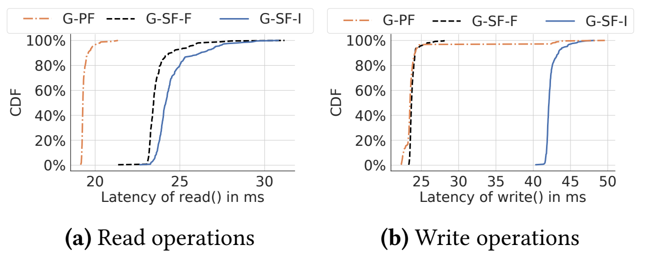
(a) Increase in LLC loads **(b)** Increase in TLB flushes

Figure 9. SecureFS-FAT optimizes the metadata structure resulting in performance benefits.

Furthermore, these additional cascaded updates result in increasing the LLC (last level cache) loads due to reading the parent slabs by ≈ 30% and also the number of TLB flushes (due to enclave entry and exits) by the same amount (see Figure 9). This results in a slowdown of 59% for the write operations done in SF-I as compared to SF-F (see Figure 11) when running in non-Graphene mode. For G-SF-F vs G-SF-I (Graphene mode), the slowdown in the latter is 44%.

6.3 Throughput of the File System: Iozone

Here, we compare the throughput of SecureFS (G-SF-F and G-SF-I) with GraphenePF (G-PF) for the stress testing workload, Iozone. We do not compare with Nexus because after a successful mount Nexus keeps the data in the main memory in a plaintext format, and only encrypts while unmounting the volume. Hence, it is not a fair comparison. Iozone differs from the other workloads used in the paper (Table 1) in the sense that it does not have a compute phase and is a purely I/O workload, as intended.



(a) Read operations **(b)** Write operations

Figure 10. CDF [1] of read and write latencies for Iozone while sequentially reading and writing 1 GB of data within Graphene.

Figure 10 shows the latency of read and write operations for G-PF, G-SF-F, and G-SF-I. The latency of SecureFS variants follow a similar trend (as discussed in the previous section). GraphenePF's read latency outperforms SecureFS. However, for write operations it has many long-latency operations (heavy tailed distribution).

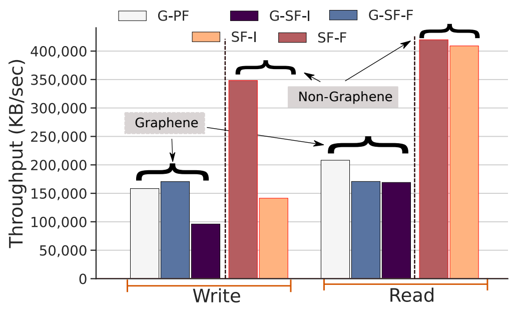
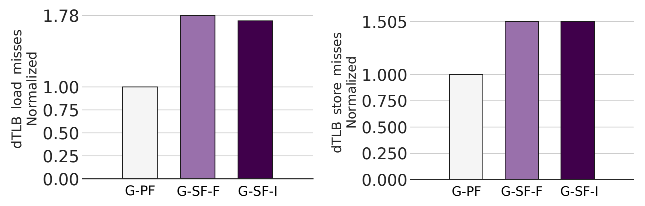


Figure 11. Performance comparison for Iozone read and write operations.

For write operations, G-PF shows a slow down of 7% compared to G-SF-F while outperforming G-SF-I by 39%. For reads, G-PF outperforms both the SecureFS variants by 19%.

6.3.1 Discussion:

In GraphenePF, every file operation results into an OCALL. It does not maintain a cache of decrypted blocks within the trusted memory region. On the other hand, SecureFS maintains a cache of frequently accessed chunks and slabs within the trusted region.



(a) Increase in dTLB load misses. **(b)** Increase in dTLB store misses.

Figure 12. dTLB events for G-PF, G-SF-F, and G-SF-I.

By default, SecureFS memory maps the secure volume into the untrusted memory as compared to GraphenePF that uses the native OS's page cache. However, when we integrate SecureFS with Graphene, the *mmap* happens from the trusted region. This results in an increase in the total number of asynchronous exits because of the EPC faults. These additional faults results in a total increase of 78% and 50% for dTLB load misses and dTLB store misses, respectively (due to TLB flushing while exiting the enclaves), as shown in Figure 12. Due to the security guarantees of Graphene, it is not apparent how to share a pointer from the host to an application running with Graphene.

In SF-F (without Graphene), where the SecureFS volume is mapped in the untrusted region, these expensive page-faults are avoided, and the performance of writes improves by 120% (G-SF-F vs SF-F) (see Figure 11).

6.3.2 Hash operations.

In the case of SecureFS, the majority of overheads are due to the hash operations. The CPU we are using (see Table 2) contains dedicated instructions for AES. However, it does not have dedicated support for SHA, and hence, all the hash operations are done in software that causes significant performance overheads.

We compared the performance of a non-SGX version of SecureFS on an Intel CPU that lacks SHA instructions with a Ryzen CPU (AMD Ryzen 7 3700X) that has dedicated SHA instructions. Since Ryzen does not have SGX, we compared the performance with a non-SGX version of SecureFS. We conjecture, that the benefits should translate to SGX if executed with a CPU that has SHA and SGX support. We observe a performance improvement of 108% and 97% in SecureFS for read and write operations, respectively (using a FAT table-based metadata organization).

6.4 Comparison with GraphenePF: Regular Workloads

We compare the performance of GraphenePF (G-PF) with that of SecureFS-FAT (G-SF-F) and SecureFS-inode (G-SF-I) for our set of synthetic and real world workloads (see Table 1). Note that Graphene does not support Java applications, hence, we skip License3j in this setting.

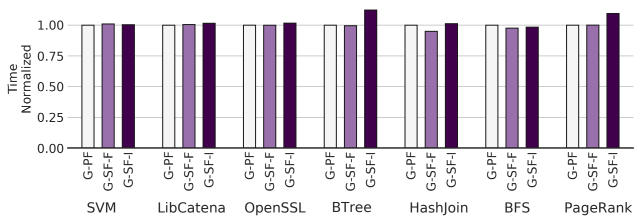


Figure 13. Comparison of the total time taken by G-SF-F and G-SF-I, w.r.t G-PF.

As shown in Figure 13, G-SF-F and G-SF-I outperform Graphene (G-PF) by up to 5% and 2%, respectively.

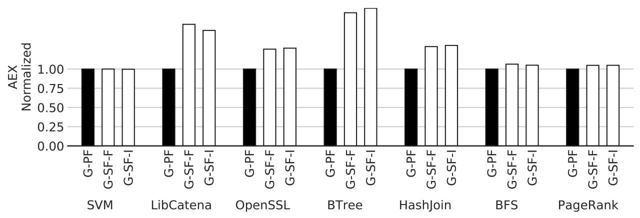


Figure 14. Total #asynchronous enclave exits (AEX).

6.4.1 Discussion.

As already discussed, GraphenePF performs an OCALL for every file system access. SecureFS optimized this by a combination of caching and memory-mapped volumes. This results in a complete elimination of ECALLs and OCALLs (apart from a few mandatory ones made by Graphene [10]). However, as the volume is mapped inside the trusted region, this results in an increase in page faults. Hence, total asynchronous exits (AEX) increase by 18% on an average (see Figure 14).

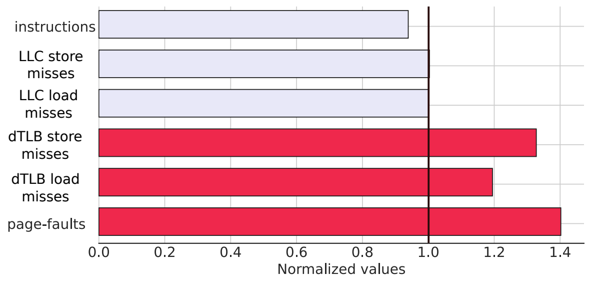


Figure 15. Different *perf* [36] events for G-SF-F, normalized to G-PF.

Figure 15 shows the impact of these additional asynchronous exits on HashJoin (representative workload). The total number of page faults, dTLB load misses, and dTLB store misses increase by 40%, 20%, and 30%, respectively. However, the number of instructions executed, LLC store misses and LLC load misses do not see any increase because during an enclave exit, only the TLBs are flushed, however, the caches are kept intact [12]. Additionally, the workloads (Table 1) are predominantly compute-intensive. The usual trend, as shown in Figure 4d, is that they read some amount of data, then processes it, and in the end write some data to the file system. Here, the compute phase dominates.

Due to all these reasons, the performance of the workloads remains roughly the same in all the three settings, unlike in the case of Iozone. The **key point** to note is that the performance is roughly the same and in fact we have occasional speedups with our file system; moreover, we provide an additional security guarantee – **immunity from replay attacks**.

6.5 Comparison with Nexus

Here, we compare the performance of Nexus and SecureFS for the workloads listed in Table 1. As noted above, as per the Nexus design, once the secure volume is mounted, it is accessible to any application in the system. The guarantee of confidentiality and integrity is provided before the mounting of the volume. Once mounted, all the data remains in a plaintext format in the main memory. It is encrypted again at the time of unmounting the volume.

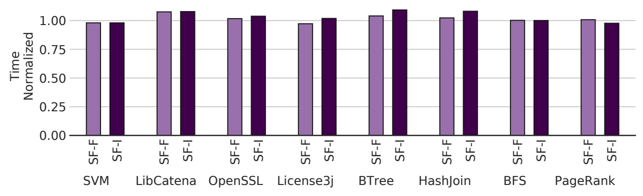


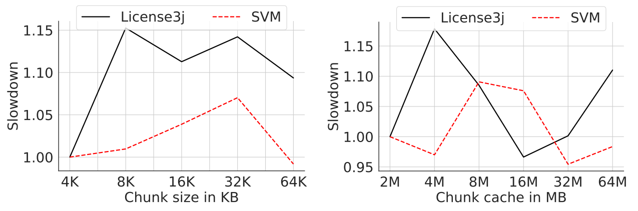
Figure 16. Comparison of total time taken by SF-F and SF-I, with respect to Nexus as the baseline (not shown).

We configure SecureFS such that only the file system is protected. The OS never sees the data in the SecureFS volume. However, it can read the address space of the application. The performance comparison of Nexus with SecureFS-FAT and SecureFS-inode is shown in Figure 16. SecureFS provides freshness (no replay attack) guarantees along with confidentiality and integrity with a negligible slowdown of 1.8% on an average.

6.5.1 Discussion. In spite of Nexus reading and writing data in plaintext in the memory, SecureFS incurs a negligible overhead. The reasons are similar to the case with GraphenePF. For an I/O intensive workload like Iozone, Nexus outperforms SecureFS by 5×, as all the read and write happen in the untrusted main memory, in plaintext. Much of this overhead is SGX's default overhead.

6.6 Sensitivity Analyses

Here, we analyze the impact of key implementation parameters: the chunk size and cache sizes on the performance of SecureFS. We show the variation in the time taken by two representative workloads: SVM and License3j. SVM reads and writes the data in a sequential manner. License3j reads the data randomly.



(a) Sensitivity to chunk size. **(b)** Sensitivity to chunk cache size (**C**).

Figure 17. Sensitivity of SecureFS to different chunk sizes and chunk cache sizes.

6.6.1 Chunk size. As mentioned earlier, we use a chunk size of 4 KB for our evaluation. Here, we measure the impact of increasing the chunk size to 8 KB, 16 KB, 32 KB, and 64 KB.

Figure 17a shows the variation in the performance using SecureFS-FAT. SecureFS-inode shows a similar trend (not shown here for readability). The performance of License3j worsens as we increase the chunk size. This is mainly because of internal fragmentation and the additional cost of decryption and hashing. Hence, our choice of 4 KB is justified.

6.6.2 Cache sizes. As can be seen in Table 4, the hit rate for the metadata cache is above 99% for SecureFS-FAT. Recall that the chunk cache hit rate was less than 1% while reading and writing 1 GB of data using Iozone. Hence, we evaluate the performance of SecureFS-FAT by increasing the chunk cache size from 2 MB to 4 MB, 8 MB, and 16 MB, respectively. They represent 2.1%, 4.3%, 8.6%, and 17.3% of the EPC, respectively.

The performance of SVM and License3j remains somewhat the same. License3j does not read a large amount of data; hence, its performance is not very sensitive to the chunk cache size (beyond our default size of 2 MB). SVM on the other hand accesses a large amount of data (50 MB). Increasing the chunk cache size increases the chunk cache hit rate from 32% to 66%. However, due to the additional memory requirement (for the chunk cache), the page fault rate increases by 12%, causing the dTLB load and store misses to increase by 61% and 15%, respectively. Given these conflicting requirements, 2 MB is a reasonable choice.

7 Conclusion

We showed in this paper that the needs of secure file systems in the context of secure remote execution are very different as compared to normal file systems. To leverage their unique characteristics, it is necessary to design a bespoke file system that can benefit from caching. Using our novel file system design, we showed that we could provide additional security guarantees namely immunity from replay attacks as compared to state-of-the-art work with a minimal ($\approx 1.8\%$) overhead.

References

[1] 2021. Cumulative distribution function plot > Frequency distribution > Continuous distributions > Distribution > Statistical Reference Guide | Analyse-it® 5.65 documentation. https://analyse-it.com/docs/user-guide/distribution/continuous/cdf-plot. (Accessed on 04/02/2021).

[2] Reto Achermann. 2020. mitosis-project/mitosis-workload-btree: The BTree workload used for evaluation. https://github.com/mitosis-project/mitosis-workload-btree. (Accessed on 10/03/2020).

[3] Reto Achermann. 2020. mitosis-project/mitosis-workload-hashjoin: The HashJoin workload used for evaluation. https://github.com/mitosis-project/mitosis-workload-hashjoin. (Accessed on 10/03/2020).

[4] ARM. 2019. TrustZone Arm Developer. https://developer.arm.com/ip-products/security-ip/trustzone. (Accessed on 12/14/2019).

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Mark L Stillwell, David Goltzsche, David Eyers, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. *Osdi*, 689–704.

[6] S. Beamer, K. Asanovic, and D. Patterson. 2015. The GAP Benchmark Suite. *ArXiv* abs/1508.03619 (2015).

[7] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1041–1056. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck

[8] Dorian Burihabwa, Pascal Felber, Hugues Mercier, and Valerio Schiavoni. 2018. *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (2018), 67–72.

[9] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. *Security* (2015), 161–176.

[10] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference*.

[11] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: why the system call API is a bad untrusted RPC interface. In *ASPLOS*.

[12] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[13] Briand Djoko. 2021. sporgj/nexus-code: Secure cloud access/usage control using client-side SGX. https://github.com/sporgj/nexus-code. (Accessed on 03/17/2021).

[14] Judicael B. Djoko, Jack Lange, and Adam J. Lee. 2019. NeXUS: Practical and Secure Access Control on Untrusted Storage Platforms using Client-Side SGX. *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2019), 401–413.

[15] R. Doomun, J. Doma, and S. Tengur. 2008. AES-CBC software execution optimization. In *2008 International Symposium on Information Technology*, Vol. 1. 1–8.

[16] Johannes Gtzfried, Moritz Eckert, Sebastian Schinzel, Tilo Mller, Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on intel SGX. *Proceedings of the Proceedings of the 10th European Workshop on Systems Security, EuroSec 2017, co-located with European Conference on Computer Systems, EuroSys 2017* (2017), 1–6.

[17] Marti A. Hearst. 1998. Support Vector Machines. *IEEE Intelligent Systems* 13, 4 (July 1998), 18–28. https://doi.org/10.1109/5254.708428

[18] Intel.). Intel Software Guard Extensions. https://software.intel.com/en-us/sgx/sdk. (Accessed on 10/25/2019).

[19] Intel. 2019. Intel SGX for Linux*. https://github.com/intel/linux-sgx. (Accessed on 09/23/2019).

[20] Intel. 2019. Intel Software Guard Extensions | Intel Software. https://software.intel.com/en-us/sgx. (Accessed on 12/14/2019).

[21] Sandeep Kumar, Diksha Moolchandani, Takatsugu Ono, and Smruti R. Sarangi. 2019. F-LaaS: A Control-Flow-Attack Immune License-as-a-Service Model .

[22] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. 2018. SeCloak: ARM Trustzone-based Mobile Peripheral Control. *MobiSys* 18 (2018), 13.

[23] Ximing Liu, Wenwen Wang, Lizhi Wang, Xiaoli Gong, Ziyi Zhao, and Pen-Chung Yew. 2020. *Regaining Lost Seconds: Efficient Page Preloading for SGX Enclaves*. Association for Computing Machinery, New York, NY, USA, 326–340. https://doi.org/10.1145/3423211.3425673

[24] Saurav Mohapatra. 2019. mohaps/libcatena: a simple toy blockchain written in C++ for learning purposes. https://github.com/mohaps/libcatena. (Accessed on 09/23/2019).

[25] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 227–240. https://www.usenix.org/conference/atc18/presentation/oleksenko

[26] OpenSSL. 2019. OpenSSL. https://www.openssl.org/. (Accessed on 12/07/2019).

[27] Paweł Marczewski,Dmitrii Kuvaiskii,Michał Kowalczyk . 2021. Performance tuning and analysis — Graphene documentation. https://graphene.readthedocs.io/en/latest/devel/performance.html. (Accessed on 03/27/2021).

[28] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. *Proceedings - IEEE Symposium on Security and Privacy* 2015-July, 38–54.

[29] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (Xi'an, China) *(ASIA CCS '16)*. ACM, New York, NY, USA, 317–328.

[30] Shweta Shinde, Shengyi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. 2019.

[31] J. Shun and G. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP '13*.

[32] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* 48, 8 (Feb. 2013), 135–146. https://doi.org/10.1145/2517327.2442530

[33] Shruti Tople, Ayush Jain, and Prateek Saxena. 2015.

[34] Peter Verhas. 2019. License3j: Free Licence Management Library. https://github.com/verhas/License3j. (Accessed on 11/18/2019).

[35] Nico Weichbrodt, Pierre Louis Aublin, and Rüdiger Kapitza. 2018. SGX-Perf: A performance analysis tool for intel SGX enclaves. *Proceedings of the 19th International Middleware Conference, Middleware 2018* (2018), 201–213.

[36] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2018. sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Middleware*.

[37] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 81–93.

[38] Wikipedia contributors. 2019. Strace — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Strace&oldid=922720825. [Online; accessed 17-November-2019].

[39] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings - IEEE Symposium on Security and Privacy*, Vol. 2015-July.

[40] Ning Zhang, Kun Sun, Wenjing Lou, and Y. Thomas Hou. 2016. CaSE: Cache-Assisted Secure Execution on ARM Processors. *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016* (2016).