UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

File System Security And Protection From Software-based Fault Injection Attacks

Diploma Thesis

John Karamporos

Supervisor: Aspasia Daskalopoulou

October 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ασφάλεια Συστήματος Αρχείων Και Προστασία Από Επιθέσεις Εισαγωγής Σφαλμάτων

Διπλωματική Εργασία

Ιωάννης Καράμπορος

Επιβλέπων: Ασπασία Δασκαλοπούλου

Οκτώβριος 2023

Εγκρίνεται από την Επιτροπή Εξέτασης:

Επιβλέπουσα **Ασπασία Δασκαλοπούλου**
Αναπληρώτρια Καθηγήτρια, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Μέλος **Γεώργιος Σταμούλης**
Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Μέλος **Χρήστος Αντωνόπουλος**
Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελούν αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλουν οποιασδήποτε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχουν έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Δηλώνω επίσης ότι τα αποτελέσματα της εργασίας δεν έχουν χρησιμοποιηθεί για την απόκτηση άλλου πτυχίου. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο Δηλών

Ιωάννης Καράμπορος

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

The Declarant

John Karamporos

Diploma Thesis

File System Security And Protection From Software Based Fault Injection Attacks

John Karamporos

Abstract

Traditional file systems always relied on the underlying Operating System, for confidentiality and integrity assurances. While designing a file system from scratch is rather difficult and time consuming, developers can utilise the functionality that is provided by user-space file system APIs or stacked file systems, to implement a file system with specific features. With the emerge of cloud services, file systems can be implemented on virtual machines residing on a server, utilising the accessibility, the computing power and storage capacity of data centres. However, a cloud file system assumes that the hypervisor and the underlying OS can be trusted to prevent unauthorised access or data tampering. With the introduction of Trusted Execution Environments, as Intel SGX, these assumptions can be lifted to some extent. All the sensitive data can be processed and stored safely inside a secure environment, which is completely isolated from the underlying malicious OS. Strong security primitives and mechanics ensures that the OS itself cannot directly access data stored inside a protected environment or modify the flow of a program that runs in protected mode. But even with TEEs assistance on designing a file system, there are several issues that can potentially be exploited to violate their integrity and confidentiality. Various methods and schemes have been proposed to mitigate security risks, including hardware and software approaches, but designing a flexible file system and shielding it from security threats, while satisfying the constraint of reasonable introduced overhead, are diametrically opposed objectives and a topic of extensive research.

Διπλωματική Εργασία

Ασφάλεια Συστήματος Αρχείων Και Προστασία Από Επιθέσεις Εισαγωγής Σφαλμάτων

Ιωάννης Καράμπορος

Περίληψη

Τα παραδοσιακά συστήματα αρχείων βασίζονταν πάντα στο Λειτουργικό Σύστημα για να εγγυήσεις όσον αφορά την εμπιστευτική λειτουργία και διασφάλιση της ακεραιότητάς τους. Ενώ ο σχεδιασμός ενός συστήματος αρχείων από το μηδέν είναι μία χρονοβόρα και δύσκολη διαδικασία, οι σχεδιαστές μπορούν να αξιοποιήσουν την λειτουργικότητα που παρέχεται από τις διεπαφές των user-space συστημάτων αρχείων ή των stacked συστημάτων αρχείων για να υλοποιήσουν ένα νέο σύστημα αρχείων με τις δικές τους προδιαγραφές. Με την άνοδο των υπηρεσιών cloud, τα συστήματα αρχείων μπορούν να υλοποιηθούν σε εικονικές μηχανές που βρίσκονται σε κάποιο ακοκεντρομένο διακομιστή, αξιοποιώντας έτσι την εύκολη πρόσβαση, την υπολογιστική ισχύ και τον αποθηκευτικό χώρο των data centers. Παρόλ' αυτά, ένα σύστημα αρχείων εντός του cloud, υποθέτει ότι ο επόπτης ή το λειτουργικό σύστημα είναι άξιο εμπιστοσύνης και μπορεί να αποτρέψει μη εξουσιοδοτημένες προσβάσεις ή αλλοιώσεις των δεδομένων. Με την εισαγωγή των Περιβάλλοντων Εμπιστευτικής Εκτέλεσης, όπως το Intel SGX, αυτές οι υποθέσεις μπορούν να μην ισχύουν σε κάποιο βαθμό. Όλα τα εμπιστευτικά δεδομένα μπορούν να επεξεργάζονται και να αποθηκεύονται με ασφάλεια μέσα σε ένα ασφαλές περιβάλλον, το οποίο είναι απομωνομένο από ένα κακόβουλο Λειτουργικό Σύστημα. Ισχυρές κρυπτογραφικές προδιαγραφές και μηχανισμοί μπορούν να εγγυηθούν ότι το Λειτουργικό Σύστημα δεν μπορεί να έχει άμεση πρόσβαση στα δεδομένα που βρίσκονται μέσα σε ένα ασφαλές περιβάλλον ή να αλλάξουν την ροή ενός προγράμματος που τρέχει μέσα σε ένα τέτοιο. Αλλά ακόμη και με την βοήθεια των TEEs, στην υλοποίηση ενός ΣΑ, υπάρχουν αρκετά προβλήματα που μπορούν πιθανώς να εκμεταλλευτούν κακόβουλοι χρήστες για να παραβιάσουν την εμπιστευτικότητα και την ακεραιότητα των δεδομένων. Ποικίλες μέθοδοι έχουν προταθεί για να μετριάσουν τα ρίσκα ασφαλείας, συμπεριλαβανομένων τόσο λογισμικών όσο και υλισμικών προσεγγίσεων, αλλά ο σχεδιασμός ενός συστήματος αρχείων και η προφύλαξη του από απειλές, κρατώντας παράλληλα τον επιπλέον εισαγώμενο φόρτο και τις δαπάνες σε λογικά επίπεδα, είναι αντιδιαμετρικοί στόχοι και κατηγορία προβλήματος ενδελεχής έρευνας.

Table Of Contents

1. Introduction

This paper consists of a collection of different papers and gathered information that provide the necessary knowledge for the broad area of virtual file system design and shifts it's focus on the examined subject, which is the security of the file systems with the assistance of Intel SGX and the threat models that affect their functionality and integrity and attempts to illustrate a basic concept of a user-space virtual file system that is assisted by SGX. There are many different topics and issues that we need to consider before implementing even a simple secure file system with a rather limited thread model, including techniques the can be used to develop new file systems, management of superblocks and potential keys, compromised kernel modules and how they can affect the control flow of even protected execution, methods of hardening the security of an already implemented fs utilising Trusted Execution Environments.

1.1 File System Definition And Cloud FS Era.

File systems are a vital component for the functionality of an operating system. At the same time, users expect file systems to provide them with the necessary api to be able to perform a variety of operations. These operations should always include creation, modification, retrieval, deletion and persistent storing of files. These tasks seem to be rather straightforward but in fact they are difficult and complex procedures. File systems should be equipped with policy routines to have the ability to determine which or if users should be granted access to certain data or modify them.

Following the rapid growth of widely available cloud services, file systems can now harness remote computing power from large pools of specialised hardware, to perform various task efficiently. Many large tasks can be broken into smaller and performed in parallel, taking advantage of the vast fleets of CPUs available. Furthermore, cloud storages can provide large amounts of storing space and high bandwidth for data transfers, while keeping latency low. One example of a cloud file system is the CNFS or Cloud-Native File System, which consists of two virtual volumes, a high performance one and a low-cost but slower one. This hierarchical approach, which is a general strategy for implementing cloud file system, incorporates certain protocols for storing, accessing and migrating files. Files that are frequently accessed, should be promoted the high bandwidth volume and perhaps the one that has the lowest latency among the others. [1]

Based on [2], there are 4 methods for implementing a custom file system. These methods are the following: The file system is developed in user space and runs as user process. The file system can be developed in user space as user daemon and run with the help of a kernel module like FUSE. The file system is developed in kernel mode and runs as a privileged process, which result in low introduced overhead but requires developer to deal with high complexity. Testing this kind of FS can be time demanding and does not offer debugging support because it directly runs in kernel mode and potential errors can cause kernel panic. Finally, a file system can be developed on top of an existing file system and the functionality it provides is stacked on top of it.

1.2 Sectors That Highlight The Importance Of FS Security

Next, we provide two examples that showcase the importance of shielding and securing file systems:

Healthcare Industry Cloud storing of patient folders is necessary for doctors to be able to rapidly apply certain algorithms on medical data, utilising the power of cloud computing. In addition doctors can access patients medical info, while not being at work in case of emergencies. Integrity and confidentiality of patient’s data should always be guaranteed. Integrity poses a vital requirement, while confidentiality should be maintained preventing third parties from eavesdropping.

Banking Industry: Banking sector can benefit from cloud technologies by leveraging it’s distributed characteristics for maintaining an accessible and flexible real time database. Banking environment, by the scope of cloud services, can be divided to two components as we have observed. The client side and the internal side which consists of branch offices, bank centres and third party bank portals. Clients need to access their banking details and proceed with their desired transactions or request certain financial records in real time by using the interface each bank provides. The internal sector needs to be able to facilitate the exchange of large quantities of financial records between branch offices and centres, while balancing connection congestions that can occur at high demand hours. This can only be achieved by hierarchically maintaining file on distributed cloud storages, which can also prevent single points of attacks, which can occur from maintaining large amounts of data locally on each branch office. Cloud provides a unique opportunity for banking sector to reduce the operational costs and delays. But cloud computing for banking purposes, by definition, should be able to guarantee the integrity, confidentiality and freshness of data, and the availability of services for internal and external sectors.

1.3 Motivation For Developing TEE-based FS To Protect From Malicious Fault Injection And Disclosure Attacks

Both of these examples facilitates the importance of shielding cloud file storages, not only for the purposes of preventing a potential leakage of sensitive data, but also ensuring that stored data cannot be tampered in any meaningful way and any attempt to do so should be at least detectable in short time, if not fully prevented. In case of malicious modification, replacement or deletion of stored blocks, file system implementation should be able to detect any such occurrences by verifying the integrity of all those stored blocks, not only after mounting and initialisation time but also through out run-time execution. It is also developer's responsibility to establish how these event should be dealt with. For example, in case of such events, there can be a read-only policy that forces all files and folder to be read-only to prevent further damage, that the file-system can't be recovered from. In addition,, the file system can also be flexible, by isolating the directory that these potential errors occurred and allowing interaction with files from every other directory and sub-directory that belongs to the same hierarchical level as the affected directory. However this functionality relies on heavy monitoring and integrity verification of the state of all the directory nodes and files that belong to them, because otherwise, there can be no guarantees, that allowing operations on non-affected same level other directory nodes, which effectively modifies the state of the current path that the directory or file resides on, can be built on top of a verified state.

Simultaneously, they highlight the necessity of a standardised method for gaining confidence that a service can be trusted with data storage and manipulation, apart from provider and vendor legal assurances. The provider can ensure users that it's password-storage application or their cloud storage service, satisfy all the security requirements. However, after various security researches and vulnerability discoveries, it has been proven that provider’s assurances, large existing user-base and even a provider’s history of no breaches, can not be the only way that services can be trusted with carrying user’s everyday tasks and most importantly, tasks that involves some level of confidentiality as we showcased with the previous examples. A technique or scheme that can verify the trustworthiness of a service, that not only operates on our private computer but also of a service that resides on a data centers, based only upon security primitives and not on the integrity and trustworthiness of the software that participates on this scheme, is essential. By establishing a standardised scheme that can detect any foreign interventions and verify that a channel that services are communicating over, is secure, developers can no longer have to rely on heuristics or implement their own security measures, which, most of the time, because they are software-based, have increased overhead, to build their functionality.

We showed that the motivation behind secure environments is the need for trustiness, secure execution, and tamper-proof data storage. To achieve this functionality, software can no longer assist us on creating such isolated spaces that prevent from hostile takeovers or violation of their memory, by itself. To achieve this level of security, hardware components should must be used to ensure that a secure computing base has been establish and software can operate on top of it. Both the software that operates on these specialised hardware together with this underlying hardware can provide a a secure base that do not rely on software heuristics and provider’s trustworthiness, but rather on security primitives that are provided by the hardware itself. The main target of designing such hardware supported environments, is that we should also not take for granted that each Operating System and it’s kernel, that our software runs on, will not be malicious. Without the Trusted Execution Environment, the integrity and confidentiality of our operations and data relied on an honest OS and it’s ability to prevent malicious activity. By considering the Kernel not to be trustworthy, we can no longer rely on it for security assurances. Kernels although are stress tested on daily basis for vulnerabilities, encapsulate a large code base which result in a wide attack surface. In addition monolithic kernels, like Linux, even though they are open-sourced and designed around the idea of security and isolating the roles by levels of privileges, in case they are compromised, there is no additional software-based mechanism to detect the breach or prevent any further activity by isolating any other component. If an entity escalates to higher privileges, as ring 0, linux kernel can not prevent any attempt of this malicious entity to tamper data or modify the execution flow or just observe the whole memory and each user's activity and dump any important information like private keys and files that should be kept confidential.

Taking into account the protected environments, we can define that software-based faults from the scope of our paper can now be injected not only by potential malicious user space applications that effectively found a vulnerability in our application or a vulnerability in a driver that interacts with our application, but also from the kernel itself. Kernel is considered to act maliciously or be able to inject byzantine faults. For example a high privileged software can now directly observe and modify the whole memory layout, tamper system cal return values, modify execution flow, highjack secure channels, feed insecure IO. Kernel has no ability to mitigate such high level attacks and one can only hope that will result in a kernel panic or potential system errors that can occur and notify the user for malicious activity or else there is no way without an underlying hardware to be able to detect such compromise.

1.4 Overview

The rest of the paper is structured as follows: Section 2 provides the necessary background for understanding the key elements of this paper, which are the user-space file system framework and trusted execution environment of SGX. We elaborate further on how sgx achieves an isolated environment, inside of which we can execute secure code and how a developer can utilise this framework and create applications by also taking into account the limitation that exist. Section 3 dives into several SGX based file system implementation and describes what is the target of each one and if there are any known issues or limitations that users need to know before using them. Following we discuss how we designed our own basic file system concept by using FUSE and SGX to create, modify, retrieve or delete simple and rather limited in size, files

2. Background

2.1 User-Space File System

In this section we are going to illustrate the architecture of the FUSE framework. By developing the file system as a user level process, the complexity of kernel level programming can be avoided. One of the most advantages of developing a file system as a user level process is that the file system can be installed by a user without the assistance of a system administrator. This provides the user with greater flexibility in how they use files. [3]

2.1.1 Architecture

FUSE consists of a kernel part and a user-level daemon as shown in **image 1.1**. The kernel part is implemented as a Linux kernel module that, when loaded, registers a fuse file-system driver with Linux's VFS. This Fuse driver acts as a proxy for various specific file systems implemented by different user-level daemons. The kernel module maintains several queues that are used for message forwarding. When an operations takes place at a FUSE mounted volume, the Virtual File System or VFS forwards the operation to the FUSE kernel module. The driver then either translates this operation to a specific request or slices this request to multiple others and adds them to the appropriate queue. The user-daemon receives the corresponding request by reading the queue and determining which procedure should follow. When the related request is fully processed, the daemon writes back the response to the FUSE queue.
The High-level FUSE API builds on top of the low-level API and allows developers to skip the implementation of the path-to-inode mapping.

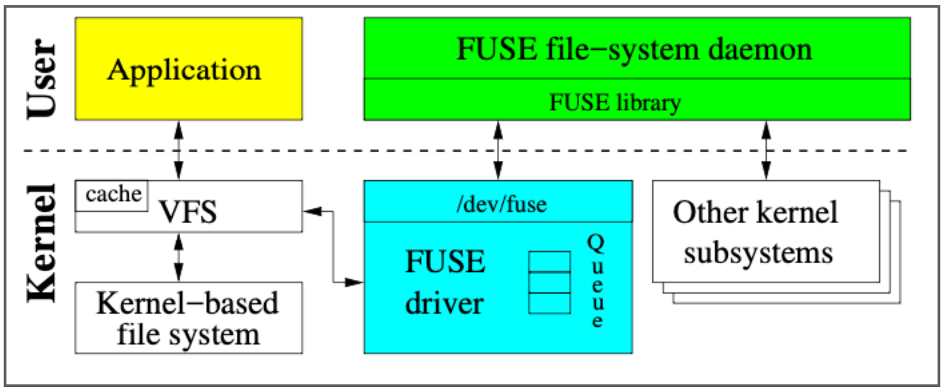


Image 1.1: FUSE architecture [3]

Although that the queues lengths are not explicitly limited, a threshold value exists. If the umber of asynchronous requests reaches the value of congestion threshold parameter, FUSE informs the Linux VFS that is congested and VFS throttles the user processes that write to this file system.

FUSE's by default does not poses a custom permission logic for file access and manipulation. When the kernel evaluates if a user process has per-missions to access a file, it generates an ACCESS request. By handling this request, the FUSE daemon can implement custom permission logic. However, typically users mount FUSE with the default permissions option that allows kernel to grant or deny access to a file based on its standard Unix attributes

Throughout the developing and maintenance of FUSE framework, various optimisations have been made to minimise overhead. For example multithreading support was added as parallelism got more popular. Thread invocation happens dynamically, in case two of more request are available in the pending queue. Although there is not an explicit upper limit on the number of threads invoked by FUSE framework, there is a threshold for requests residing on processing queue and therefore a implicit max number of concurrent threads exists.

2.1.2 Programming Model

FUSE api uses wrapper functions that are related to each file system common operation. Developers build their custom functionality for each operation inside these wrapper functions and they can also define more functions that can be used to add extra functionality.

To be able to recognise and invoke the functions that correspond to each file system operation that takes place on the virtual file system, FUSE requires developers to initialise a struct's function pointer fields with the appropriate custom wrapper functions. The main function of the custom file system file then invokes the FUSE initiator function with the initiated struct as a parameter.

Depending on the workload and hardware, FUSE can perform as well as Ext4, but in the worst cases can be 3× slower. [3]

2.2 Intel SGX

SGX design and implementation literature is rather limited not only because it is an established Intel patent, but also for the purpose of maintaining SGX underlying architecture as a black box. If the entire architecture was revealed, it would defeat Intel's purpose to provide a successful protected environment for trusted computing, by enabling different third party intelligences to gather vital information that could potentially be proved to accelerate the process of finding zero-day vulnerabilities. On top of this, we should highlight that SGX security is hardware based and undefined behaviour of hardware or hardware vulnerabilities cannot be fixed entirely with firmware patches, but rather with replacement of the underlying hardware. However, there are plenty conducted studies that aim on shedding light on the internal management and security primitives for research purposes. A deep knowledge of the components of the SGX interface can provide developers with enough information to be able to determine what code segments should reconsidered, the corner cases that can potentially occur because of the limited available memory and the way SGX enclaves communicate with the kernel. Intel also provides an extensive documentation for developing on different platforms, without revealing any important underlying components that should remain hidden and a community forum for further discussions. We frequently consulted intel community threads for issues regarding of SGX API and installation of Intel's SDK.

A basic illustration of how the memory region of DRAM looks like if SGX is enabled and properly initialised is given in image 2.1. Reserved memory by SGX contains all the control structures for the secure initialisation and execution of enclaves and is divided into several pages that each belong to a particular enclave. The collection of these pages consist the EPC region of the enclaves.

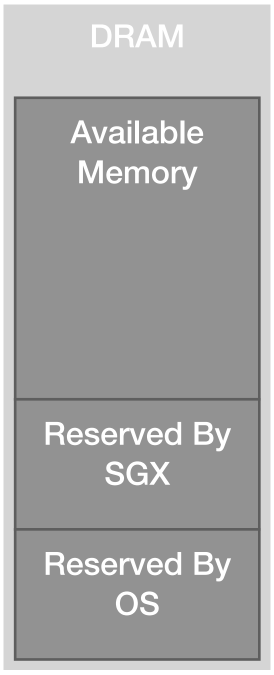


Image 2.1: DRAM layout with SGX support enabled.

2.2.1 Enclave Signatures

SGX enclaves can be signed with two methods. The signature can be created by measuring the code or by author's identity, which is his private key, together with other measurement of data that Intel requires as inputs and should not be disclosed.

The same enclave, regardless of who signed it, will produce the same MRENCLAVE. MRENCLAVE signature, can allow different authors and users, to run inside the particular enclave and unseal data, that where previously sealed by another author, but only under the same system. Different underlying hardware, would result on different MRENCLAVE signatures, because the measurement of the SGX will be different although the code of the application is the same.

The second method that can be used to sign an enclave is signed on the author's identity. This allows only the author to execute inside the enclave or anyone else that possesses the private key of the author. By signing on author's identity, the application can be upgraded without worrying about not being able to access data that were encrypted by the previous

version and also enables the application to run and decrypt any provided data not only on the host machine that enclave the first time, by in every SGX supported CPU system.
The sign tool is provided by intel SDK, and has embedded a private key creation or one can also use the openssl command to create a private key with desired properties at any time before executing the signed command. Fo example, after testing other private key creation, we concluded that only openssl satisfy the requirement for enclave signing as RSA exponential equal to 3 and RSA bit length equal to 3072.

2.2.2 App-Enclave Communication

Syscalls are prohibited inside the enclave. Developers have to use an OCALL with a wrapper function to execute the desired syscall, outside of the enclave, in kernel mode. If a syscall command is allowed inside the enclave, it means that the Intel SGX SDK has replaced the syscall function, with a function of the already implemented ones from it's library and developer has to take no further action.
It is important to note that ECALLs and OCALLs do not serve as returns or resumes of trusted or untrusted codes. An ECALL is an invoke of a trusted function and OCALL is an invoke of an untrusted function, which most likely will be a wrapper function that consists of functions and code that need to be executed in kernel or user space mode. Nested ECALLs (and OCALLs) can occur throughout the execution of a SGX application. If an enclave calls an untrusted function, for example for some I/O, and the untrusted function calls a trusted function (which can be the same first trusted function), for some additional sensitive data operations, the second invocation of the trusted function has to return to the untrusted function with an OCALL before the first one can return or resume the execution, assuming the untrusted one does not further invokes any ECALL. This simple scenario is indicative of what nested ECALLs (two level in this case) are. Although the depth of nested ECALLs is not restricted, nested ECALLs are limited by the amount of stack reserved inside the enclaves.
It is worth noting that you can limit which ECALLs you are allowed to make from within the context of an OCALL, via entries in the EDL file. For example only certain ECALLs can be initiated from specific OCALLs. In general developers are instructed to avoid

nested ECALLs, because they increase the complexity of the application and further complicates the security analysis.

For the purposes of our paper, we can showcase a simple example, in which the file system after entering the enclave to verify and determine that the operation should be allowed, needs additional data to proceed further. Such an example is accessing a block of data of a particular and already opened file, that is not present inside the enclave. The enclave verifies that the file exists, searches the file system cache that reside inside the enclave for the block and because the block is not present, it will issue an OCALL to retrieve the block from the untrusted memory, decrypt it and then complete the request by returning it to the user.

2.2.3 Exchange of Data Between Enclaves

SGX does not support a shared memory between enclaves in EPC. That means that two or more enclaves cannot communicate from inside the protected environment and they have to rely on OCALLs. Because two ECALLs cannot happen either, different enclaves should employ local or remote attestation to communicate with each other. This means that for two enclaves to exchange data, there has to be a way to authenticate and verify each side. The exchange of data happen by a secure channel that has been established through the untrusted kernel. First Enclave issues OCALL to switch to untrusted space and then the untrusted space issues ECALL to transfer data to the second enclave. The results from that ECALL would then be returned to the first enclave as the return result of the OCALL of the second enclave via the untrusted domain.

2.2.4 Local/Remote Attestation

SGX enclaves possesses a mechanism to authenticate and verify if a genuine enclave is running on a platform.

Local Attestation relies on a one or more enclaves that are running on the same platform as the enclave that initiates the attestation process. The enclave that needs to be verified, receives a challenge from the verifier enclave and produces a report based on the measurement of the verifier enclave that was included in the challenge. He then sends the produced report to the verifier enclave and the verifier enclave derives a report key, which is unique for each platform and attempts to verify the received report. If the report is verified, then both enclaves are assumed to be running on the same platform. The enclave that needs to be verified can now initiate the same process, to verify if the respective counterpart is also running on the same platform. At the end of this procedure, both enclaves can be confident that the other enclave is running on the same platform and that a protected channel can be established between them that guarantees confidentiality integrity and replay protection. Developers can build security layers on top of the Local Attestation scheme to be able to verify and authenticate that client enclaves are granted permission to operate certain functionality.

Remote Attestation includes a remote Intel server to be able to verify that two remotes enclaves are genuine. This mechanism is required if an entity wants to gain confidence that a remote service provider can be trusted. Both client and server side collaborate on the procedure to verify that each enclave is running on a genuine platform and to establish a secure channel over the network. Remote attestation also includes the previous local attestation in it's procedure.

Remote attestation can verify three things: the application's identity, its intactness (that it has not been tampered with), and that it is running securely within an enclave on an Intel SGX enabled platform.

For the purposes of our paper, Remote Attestation should be employed in case a user needs to establish a secure channel with a cloud storage service that also run on top of a SGX supported platform. Before safely exchanging data and operating remotely on them, local platform can gain the trust of the remote server and vice versa and be sure that no malicious kernel be the end point of this channel. Even if the malicious kernel tries to intercept several packets that include blocks of a file, without the corresponding key, it will not be able to decrypt them. In addition, to prevent roll back attacks, remote attestation can be used to retrieve a new encryption key or a random nonce from a trusted third party remote server

2.2.5 Seal/Unseal

Because enclave's data inside the EPC are lost and possibly overwritten for security purposes with the destruction of it, enclaves are capable of encrypting data, without users involvement in the process, except for calling the appropriate functions. The encrypted data are then returned to the untrusted domain, so they can be saved on a persistent storage. Enclaves does not provide any assurances regarding the storing of these data, and is it user’s responsibility to safely store them and be able to retrieve them for future use. This feature is called sealing and it can be used to preserve the integrity and confidentiality of data or a required component for creating stateful enclaves, that can also determine if the current state that they retrieved was corrupted. Following the sealing of data, enclaves can be provided with these sealed data at a later time and decrypt them. This is the opposite of sealing and is called unseal. However, to be able to unseal data, the same sign policy of the enclave that sealed the data must be followed from the enclave that will attempt to unseal them. As we already reviewed, there are two methods for signing an enclave, signing based on enclave’s measurement and signing based on a user’s private key. Sealing data to code identity requires enclaves with the same code measurements to be able to unseal. A different version of an enclave, even with slight modification, will result in an unsealing error. At the same time, if data are sealed based on signer-assigned identity, can be used to unseal them.

However, SGX does not natively provide assurances for the freshness of data. If provided with two different set of sealed data (or even the same data that were sealed at different times), there is no mechanism to determine which data were sealed after the other. If this is a required feature by developers, it is their responsibility to implement a freshness policy or use external assistance, but taking into account either case will involve the untrusted domain as a communication channel. For reference, we identified a few methods that can be used to provide freshness assurances for sealed data. **Table 2.1** provides these methods and their respective base and scope. User-input is the simplest and most straightforward one, that requires user to supply the enclave with a separate secret each time he desires to seal data or at each start of cycle of the enclave. For example, we can supply the enclave with a secret input and this input secret can be used as salt for sealing data at a particular cycle of life of the enclave with the risk of having a single key and in case of a potential leakage, the attacker can retrieve all data that were sealed, supposing there is no mechanism to count unsuccessful attempt to unseal data. Supplying a different secret also each time an enclave seals data in a particular instance, can be a complex task because users need to account and manage each of these secret inputs that correspond to unique sealed files. In case of large amounts and frequent sealing of data, this can be proved to be not feasible for average usage.

On the opposite, both monotonic counters and TPM, are hardware based solutions. TPM is a separate chip that can be used to save current state in a cryptographic way that is impossible to be tampered without detection. Monotonic counters, which can also be provided by Intel CPUs, are hardware counters that can be initiated and increased by secure function calls and keep their current state between system power states. Both have to be reached by OCALLs from enclaves to establish a secure channel firstly. After that enclaves can request either a monotonic counter for their functionality or use the TPM to store a key or a certificate. For example each time, the file system needs to perform a file system unlink, the enclave can increase the monotonic counter and seal the superblock together with the counter. If the file system needs to be mounted again alter, the enclave will unseal the provided superblock and check if the resulted counter residing in the decrypted superblock, equals the monotonic counter. However, monotonic counters have limitations and cannot be invoked frequently or else the developer will be supplied the SGX_ERROR_BUSY error and will have to wait even 10 seconds before trying again. In addition each instance can have a max number of counter available. SGX provides functions that creates monotonic counters, but as stated before, this functionality is not native on SGX enclaves, but rather already implemented OCALLs by Intel. The frequent access limitation can be partially lifted by using a TPM for storing random generated keys that are used for encrypting files, although, because they are external third party devices that can be added and removed from motherboards, the manufacturer and vendor should be trusted. For our purposes, a random generated key from the enclave can be stored and retrieved each time the file system should be unlinked or mounted.

Method	Software/Hardware Based	Scope
User Input	Software	Local
Monotonic Counters	Hardware	Local
TPM	Hardware	Local
Trusted Server	Software,Hardware	Remote

Table 2.1: Methods for maintaining freshness of stored data on untrusted volumes

Enclaves can also rely on remote exchange of secrets to be able to provide freshness assurances. For example, each time the protected file system need to be mounted or unmounted, the enclave can request from the remote server the corresponding key or counter for the current superblock. However this method does not only require several OCALLs to succeed, but also increases the attack surface by relying on trustworthiness of the remote server.

Be advised, that a hybrid method can also be implemented that involves two or more methods for guaranteeing freshness, but at the cost of higher complexity and introduced overhead. A method that requires both local monotonic counters and remote server counters to be equal can be employed and in case that they do not agree, developer should decide if the process must wait for confirmation, end or seek another confirmation from a different source.

At the current date, Intel removed monotonic counter and trusted time services and the corresponding calls that where part of the sgx_tae_service library are no longer present in current SDK implementation and the library itself is completely removed.

2.2.6 Local/Remote Use

Even though it is not explicitly stated, it is important to highlight what is the general purpose of SGX technology and how can be utilised for local or remote usage. SGX technology mainly focuses on hardening cloud services security and lifting obstacles that prevent remote users from having a standardised method to establish a secure channel over an untrusted network. However to approach remote usage, we need first to understand what can be achieved on local environments and what are the limitation of this technology regarding the security assurances and especially the file system operations.

In the presence of a malicious kernel, SGX enclaves by themselves cannot make a significant difference from the scope of local usage. In case the OS is compromised, the kernel has full control over the memory and therefore any IO and key that is stored in plaintext are visible. The already encrypted files can maintain their confidentiality as long as not access to them is performed throughout the existence of the malicious kernel. From this point of view, enclaves do not differ from just having your file encrypted by an honest kernel before the time it becomes compromised, because even if try to encrypt a file of retrieve a decrypted one via an ECALL to an enclave, the kernel can in both scenarios intercept these ECALL and read the parameters that are plaintexts. It is worth noting that the only way to shield from local scope, is to use secure IO, for example for storing private keys and establishing a secure local channel with a volume that it's firmware is also SGX assisted.

The main motivation behind SGX, as we already stated, was the secure cloud computing and communication. SGX enclaves are used to establish a secure remote channel between servers or clients and servers and exchange data or perform tasks. In the presence of a malicious OS or hypervisor in the server side, the confidentiality and integrity of the communication and the stored files, cannot be violated without the client being able to detect it.

2.2.7 SGX Threat Model

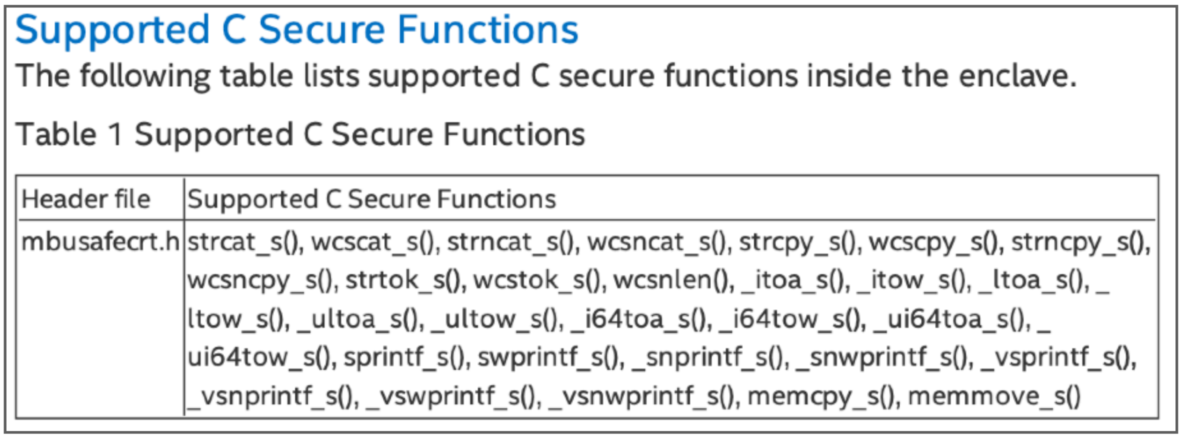
SGX threat model involves high privileged software that is able to view the whole address space. This includes software that runs on ring 0 or even a compromised kernel that can potentially act maliciously or introduce byzantine faults.

However, Intel SGX does not provide any assurances about DOS attacks. Even though a malicious kernel cannot violate the confidentiality of a secure data exchange, it can prevent the secure channel from functioning by intercepting messages and preventing them from reaching the honest end point.

In addition, SGX does not posses the ability to persistently store data in a secure volume or database. It always relies on external IO to retrieve any sealed data and therefore cannot guarantee that these data are present in each system and were not mistakenly or purposely deleted or modified. It can only trie to retrieve them and throw an error if they are not present or they were tampered, because the verification will fail.

2.2.8 SGX Programming Model

There are limited C functions that can be executed inside the enclaves. Even though the function names are identical to the known and well documented C core functions, they are in fact implement by Intel and have been submitted to exhaustive security evaluation. As images 2.2 and image 2.3 show, only functions that requires parameter for the size of the input buffer are selected from the function family that corresponds to a specific operation (i.e. snprintf_s() instead of supporting also sprintf()) or functions that already posses this feature by default (memcpy_s() requires size of the buffer by default, as memcpy()) or modified functions that require both input and output buffer sizes to be explicitly given as parameters.



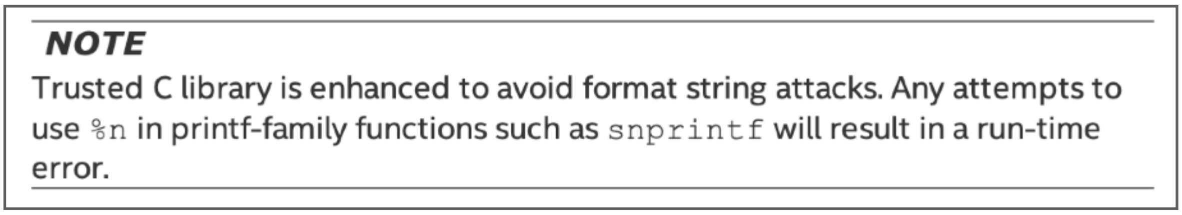
Supported C Secure Functions

The following table lists supported C secure functions inside the enclave.

Table 1 Supported C Secure Functions

Header file	Supported C Secure Functions
mbusafecrt.h	strcat_s(), wcscat_s(), strncat_s(), wcsncat_s(), strcpy_s(), wcscpy_s(), strncpy_s(), wcsncpy_s(), strtok_s(), wcstok_s(), wcsnlen(), _itoa_s(), _itow_s(), _ltoa_s(), _ltow_s(), _ultoa_s(), _ultow_s(), _i64toa_s(), _i64tow_s(), _ui64toa_s(), _ui64tow_s(), sprintf_s(), swprintf_s(), _snprintf_s(), _snwprintf_s(), _vsprintf_s(), _vsnprintf_s(), _vswprintf_s(), _vsnwprintf_s(), memcpy_s(), memmove_s()

Image 2.2: Supported C functions that can be used inside the enclave. [Intel® Software Guard Extensions (Intel® SGX) SDK for Linux* OS]



NOTE

Trusted C library is enhanced to avoid format string attacks. Any attempts to use `%n` in printf-family functions such as `snprintf` will result in a run-time error.

Image 2.3: sgx print-family functions have %n detection, to prevent format string attacks [Intel® Software Guard Extensions (Intel® SGX) SDK for Linux* OS]

Developers are required to have two separate code files that one of them contains the untrusted side of the app and the other the trusted, which will be executed inside a secure enclave. Functions that reside on enclave code can be ECALL functions that are used by the untrusted side of the software to communicate or transfer data to the secure side and OCALL functions that are used by the secure side to communicate or request additional data from the untrusted side. This edl file is used to define which functions are trusted and which are untrusted, because otherwise there is no way for enclave to know beforehand. If such a file would not be present, the developers will have to provide tags inside code file, which compilers would not understand and therefore throw errors. Instead of modifying the whole C++ or C compiler is it obviously much more flexible to define trusted functions in a separate file. EDL also defines input and output parameter directions, data types and sizes. One parameter of a function can be defined as input, output, both or user-defined, in which case user is responsible for performing sanitisation.

It is a good programming rule to avoid nested ECALLs or keep them limited at best.

Even though a specific enclave entity can only belong to one specific process, multiple threads of this process can enter the enclave concurrently after it's creation and use the reserved, available and visible to every thread, EPC address space, like in a regular running environment. SGX SDK provides developers with spin lock functions and mutex for synchronising enclave threads. However, only spin locks are implemented exclusively inside the secure environment and don't require any OCALLs to function, but can be cpu demanding as regular spin locks implemented for kernel purposes. In contrary, mutex support require OCALLs for it's functionality and therefore rely on the untrusted side for successful synchronisation of the threads. Each thread that enters the enclave has it's own thread control structure or TCS that needs to be initialised in the untrusted domain and verified in the secure environment, before executing any code.

2.2.9 Overhead

It should be already obvious that enclave calls do not come without a cost. There is high introduced overhead for various operation that need to be executed before and after entering the enclave. Apart from the complexity of the code that will be executed inside the enclave, that developers should be keep relative minimal, protected mode handler must prepare the input parameters of each ECALL, verify the memory region that the enclave correspond to is not affected in a malicious way, prepare the CPU caches with the enclaves data and many more non-disclosed operation that are required to switch the logical core from kernel mode to protected mode.

An example of introduced overhead that cannot be avoided, can be understood by illustrating the basic architecture of a CPU (in our case an Intel CPU) in **image 2.4**.

Each core has it's own L1 and L2 cache, but all of the available cores share the same L3 cache. This should be a problem for the secure operation of a protected environment that might have data residing on L3 cache. Because preventing any cores from accessing from accessing L3 cache or portion of L3 cache not only requires a vastly hardware architecture

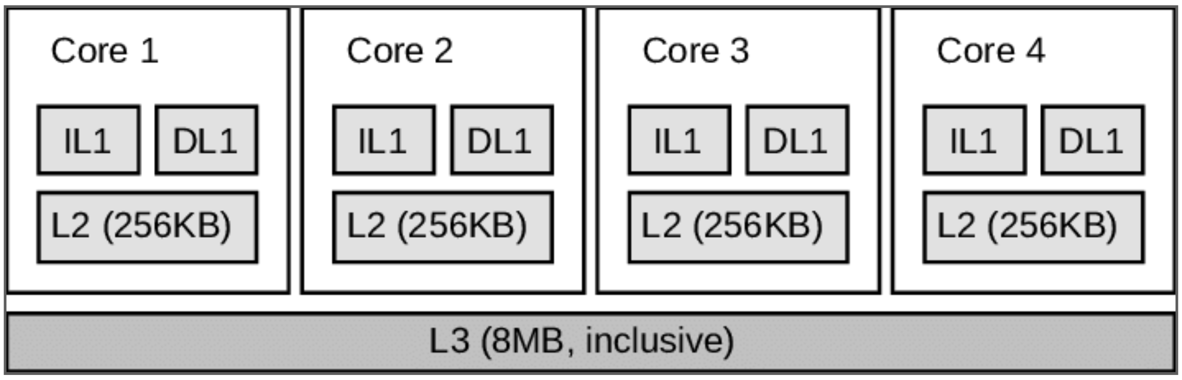


Image 2.4: Intel Core i7 cache architecture

change, but it actually defeats the purpose of having a cache based CPU for maximising performance. Therefore SGX should make sure that data that reside on L3 cache and belong to a particular enclave will be encrypted. This obviously introduces the overhead of having to encrypt and decrypt data each time data are demoted to L3 cache or each time data that were not found in L1 and L2 cache should be searched for in L3 cache.

We should point out also, that the whole region of reserved RAM memory, as shown by image[], is properly initialised and any memory that belong to a running enclave is encrypted, so that it can prevent any data leakage or execution flow disclosure. Each enclave has a unique key that was derived from the base fused CPU key, so that i can only access it's own data and cannot mistakenly or purposely access data that belong to other enclaves and read them. This also introduces overhead, because if data cannot be found on caches, they must be searched in RAM and more specifically on the EPC region, which in case of EPC-hit, must be decrypted before processed further. An EPC-miss would result also in a much higher overhead, because in this scenario, the enclave will issue an OCALL to search the region of memory outside of the EPC and therefore access the untrusted domain of RAM. This will not only have encryption and decryption overhead, taking into account that different keys are used for encrypting data that belong to an enclave but reside on untrusted domain than data that belong to the same enclave but reside on EPC, because potential key leakage should not provide knowledge for EPC encryption keys, but also require switching between protected mode and kernel mode to search DRAM untrusted domain for the desired data.

3. SGX File System Examples And Our Design Choices For Our Own File System

3.1. SGX-based FS Examples

Examples of already implemented file systems or file system architecture concepts that utilise Intel SGX enclaves are presented in this section.

3.1.1 Intel Protected File System Library

Architecture: Intel filled the missing component of their secure library, regarding a secure file system implementation, by releasing Intel Protected File System Library [5]. Although it is not a file-system nor a virtual one, these library provides an API to developers, so they can build their own file system or integrate the security assurances that this library provides, to an already implemented one. The functions that this API consists of, have partially the same function signature as the POSIX programming API, with the difference that each one has a protected side, that is executed inside a secure enclave and determines whether an operation should allowed or denied in case that the file system integrity has been violated. To be able to verify the correctness of the directory nodes and files that each file system consists of, every operation is tied to the private key that was used to sign the execution file that a user runs. This basic functionality of any SGX-based application, is also a vital component for providing a security layer over the file system's accesses and modification, because it ensures that no other party or entity that does not possess the private key that was used to sign the enclave, can potentially read or modify file system's data, without the author being able to detect or verify that any such operations were taken place. Again, as every other SGX software, based on the sign method, a particular secure file system, powered by protected API, can be used either by an entity that posses the private key that the author used to operate on his data or if the measurements of the code was used to provide the enclave's signature, every party that has the same code but with the limitation, that the code must run on the top of the same system or else the signature would differ. The latter means that provided with the full sealed superblock, the individual blocks that represent file system nodes, as the directories and the files and the code files that one must compile, sign and run to operate on these nodes, the system's hardware cannot differ from the system's hardware that the data were previously processed, which effectively means that the system should be the same.

Intel SGX protected file library, uses a modified merkle hash tree, which represents the super block, together with leaf nodes that represent blocks of data and a LRU-cache policy enforcement for determining which blocks of data should be maintained in the limited EPC and which should be sealed and evicted on untrusted storage.

A simplified merkle hash tree version that intel protected file system library uses to store the corresponding data of each file, is shown in image 3.1. Instead of the storing data nodes only on leaf nodes, every hash node can have many data nodes and at the same the combined hash of the child hash nodes. In the real version of this tree, hash nodes can have more than 2 data nodes and child hash nodes. Again, as the main benefit of the classic merkle tree and the idea behind it's development, the integrity of each file can be checked in log(N) time. A hash node takes as inputs the hashes of the corresponding data that is responsible for and the output of the hash of the each child node.

First advantage of this variant, is that adding content to the end of files, or appending data, can be trivial. Instead of having to move potential data nodes to a new leaf node, if there is no empty position for insertion, in this implementation, a new block of data that was appended to the end of file can be either stored on an existing node or a new node can be created without having to collapse any already stored data nodes. For example if we wanted to add a new content at the end of a file that can be represented ad hoc by the tree in image 3.1, we can create a new H4 hash node and store the block as a D8 node, without having to create a node H7 and H8 under H3 and then moving the existing D0 to H3 and inserting a new D4 node under H8, as we had to do instead for an ordinary merkle hash tree, as image 3.2 shows.

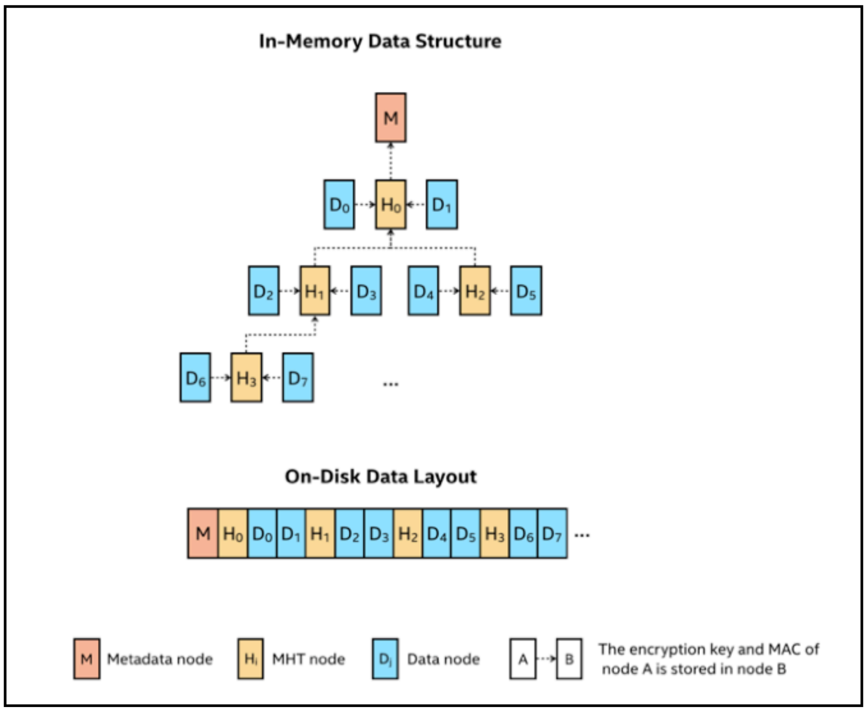


Image 3.1: A modified version of a merkle tree that can store data nodes on every hash node instead of the leafs only [6]

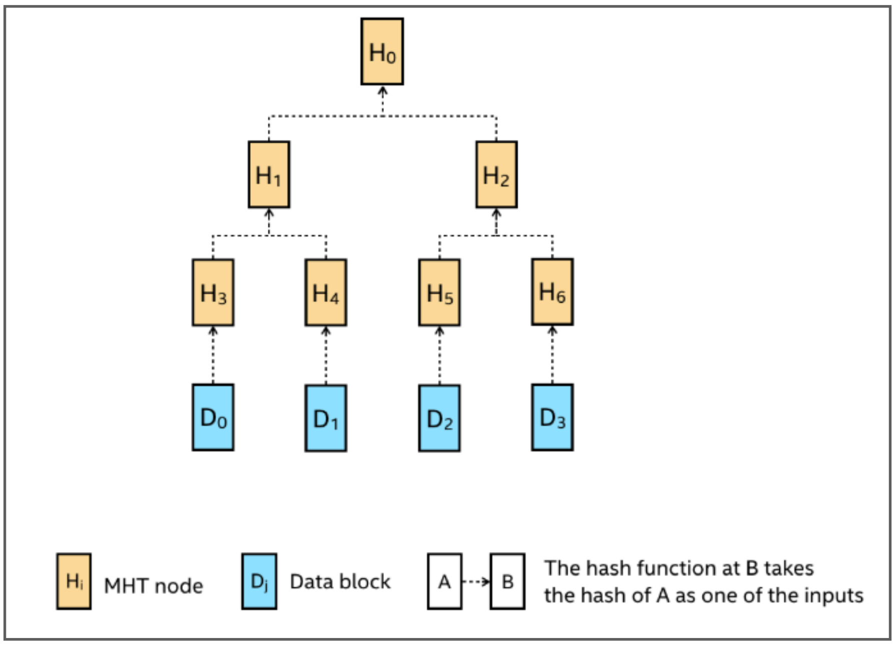

H0
H1
H2
H3
H4
H5
H6
D0
D1
D2
D3
Hi MHT node
Dj Data block
A B The hash function at B takes the hash of A as one of the inputs

Image 3.2: An ordinary merkle hash tree [6]

This integration of multiple hashes inside a single hash node also can simplify the on-disk layout representation of each file. As the image 3.1 shows, we can store each file as an array of nodes and the relative offset can be easily calculated without having to traverse the hole path of a merkle tree till the leaf nodes. At the same time, one can safely assume that in an ordinary merkle tree array representation, data nodes will most likely be stored at the end of the array, which poses a vulnerable position, as attackers can just overwrite or damage the last memory location of this array and effectively tampering data, where as in Intel implementation, any attempt to overwrite only the end of this array, can potentially destroy only data that are at the end of the file.

Non-Objectives: However Intel also states that their file system has limitation regarding the assurances that it can provide[5]. We further discuss some of these limitation and attempt to find out why Intel could not overcome. Keeping time stamps for operation at a certain file or directory, cannot be provided by SGX enclaves, because Intel no longer supports a secure time service in recent SGX SDK implementations. Therefore enclaves have to rely on an external source for timestamps that can either be an OCALL that requests a timestamp from the kernel or request a timestamp from a secure server, but again opening a socket via an OCALL that involves running code in kernel space. To overcome this developers should be able to establish secure communication channel with underlying hardware or remote server to request timestamps, which is the reason that Intel cannot provide an out-of-the-box solution with their SDK. Usage patterns and read write offsets can also be view by a malicious OS. In case of a cloud storage, a compromised OS can observe the existing connection and attempt to associate each connection with usage of files. SGX library itself cannot protect against such monitor and applied statistical analysis and therefore if a developers want to provide such a protection, needs to create scheme with dummy accesses and files, that can be periodically requested via OCALLs or can be issues each time a true and necessary OCALL is issued. However this will result in high introduced overhead.

3.1.2 sgx-fs / ram-fs

Two implementation [7], one that sends data for encryption and decrypion inside the enclave from untrusted memory and back and a second that keeps the whole record inside the EPC and seal portion of them when no space is available. The second approach results in lower overhead because the stored inside the enclave blocks of files can result in block-hits and OCALL is needed to provide the necessary data.

3.1.3 SecureFS

Two kind of file system approaches are proposed in this paper [8], one that allows different processes to access the same mounted file system, by intercepting system call and redirecting them to a process that can execute the desired operation inside the secure enclave and return the result back to the calling process and one that a volume can only belong to a specific process. Although both of these concepts are rather vague, the reviewed paper attempts to describe in detail the implementation and assurances that this file system approach provides. It should be noted, that this file system concept was designed to provide the security assurances that both Nexus FS and GrapheneFS lack.
splits data into fixed length chunks and assigns a unique random number to each chunk
uses remote attestation to verify and validate the super block each time a new mount is attempted, network is required for each mount and unmount thus is pseudo local file system.

3.1.4 BesFS

Written in a mathematical language (coq) that can provide proof of each state but because there is no direct translation into C, developers converted the LoC to C themselves. Their objectives was to provide security assurances for a FS under a byzantine OS. The OS can tamper not only system call return value but also ECALL and OCALL ones. Enclaves first should keep a record of the state of important variables before an OCALL and check after the OCALL finished. Iago attacks poses a threat to many already reviewed FS, because there is no proper return value checking. The developer has provided a table analysing how can file system transition from one state to another and guarantee that the later state is a safe state. [9]

3.1.5 Nexus fs

This file system enables file sharing between different users by leveraging remote attestation to prove valid enclave presence on the target system and uses versioning counter to tackle the individual file roll back attacks but is it subscript-able to overall file system state roll back attacks. [8]

3.1.6 GrapheneFS

It is the native file system of Graphene OS and uses a contant key to store files which is not only vulnerable to file system roll back attacks but also individual file swapping [8]

3.1.7 SGXIO

Although SGXIO is not a file system itself, it's reference is significant because it is directly related to the reliability of SGX-based file systems from both local and remote scope. SGXIO uses a SGX-based hypervisor to secure IO operations with disk and volumes and establish a secure channel between them and an application's enclave. [9]

3.2 Installing SGX

Our system setup is presented in figure 3.1. We used Ubuntu Linux Distro to install SGX and run our tests. We highlight the OS version, because later we will indicate the importance of OS's version in building and installing SGX and testing Samples codes.

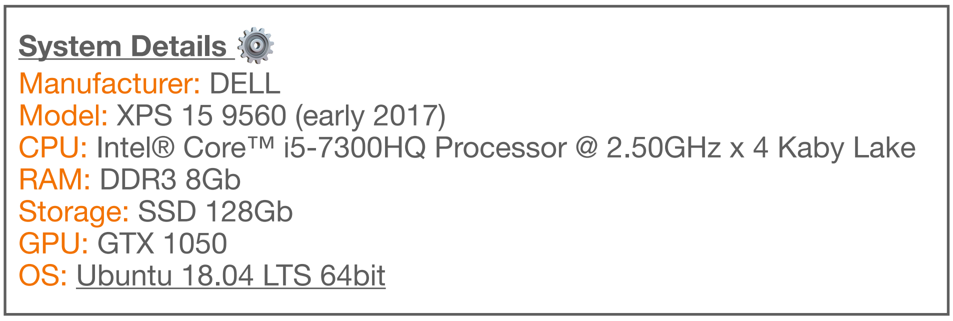

System Details
Manufacturer: DELL
Model: XPS 15 9560 (early 2017)
CPU: Intel® Core™ i5-7300HQ Processor @ 2.50GHz x 4 Kaby Lake
RAM: DDR3 8Gb
Storage: SSD 128Gb
GPU: GTX 1050
OS: Ubuntu 18.04 LTS 64bit

Figure 3.1: System details. All our codes were compiled and tested under this setup.

First, we needed to activate SGX option at the BIOS menu, under the Intel Software Guard Extension tab, because it was disabled by default, especially after an urgent BIOS firmware upgrade to counter the Plunder-volt Vulnerability. The decision to make SGX disabled by default does not serve as a straightforward counter-measure, rather than a way to let users decide if they want to benefit from SGX technology, but with the potential of being vulnerable. This temporary solution is not exclusive to DELL laptops, but various laptop manufacturers followed at the lack of Intel response. After an urgent announcement from a US department that is responsible for such cases, which was uploaded to their website, that highlighted that the only meaningful technique to mitigate this vulnerability was in fact the change of the underlying hardware and that no software or firmware update or patch can serve as a permanent fix, the decision to disable SGX option by default, rather than wait for Intel's response or updates, was the obvious and least expensive solution.

Following the hardware activation, we need to download and install the SGX SDK [10] from Github official intel SGX repository. Throughout the following process we encountered various software issues, that need to addressed in detail, because there is no exclusive guide from manufacturers and vendors for their products, rather than only the official Github repository of intel SGX, intel community, intel development documentation and intel official articles that serve quick solutions to SGX installation issues. In case users want to install SGX in their own system, we can provide some important steps that, if followed, we can claim with benefit that the process will be less painful and more straightforward. The below flowcharts, tables and provided information serve as collection and analysis of personal and third party encountered issues, intel provided solutions and a great deal of trial and error.

As we stated at the beginning of this chapter, the OS version of Ubuntu is not only important, but changes the flow of building the installers and, for the matter of our system configuration, displayed that there exists a certain combination of CPU Generation, Operating System Version and Kernel Version for each system, that consists an acceptable environment for accommodating the SGX driver.

In the below simplified flow chart in form of stacks, figure 3.2, we present the, at least, two paths that exist, based on the Generation of each system's CPU. CPUs that belong on 7th Gen, does not support DCAP and therefore Kernel Versions over 5.11, that come with build-in DCAP driver support for SGX, are not recommended in our case. Moreover a manual installation of the legacy driver over the already build-in driver will result in conflicts and might cause invalid SGX device error encounter, so Intel highly recommend against it. We should be advised also that Ubuntu might be able to recognise that DCAP in a specific CPU is not supported and therefore not attempt DCAP driver installation. We can overcome this issue by downgrading Kernel Version or by using Ubuntu 18.04 which accommodates Kernel Versions lower that 5.11 with SGX driver not present.

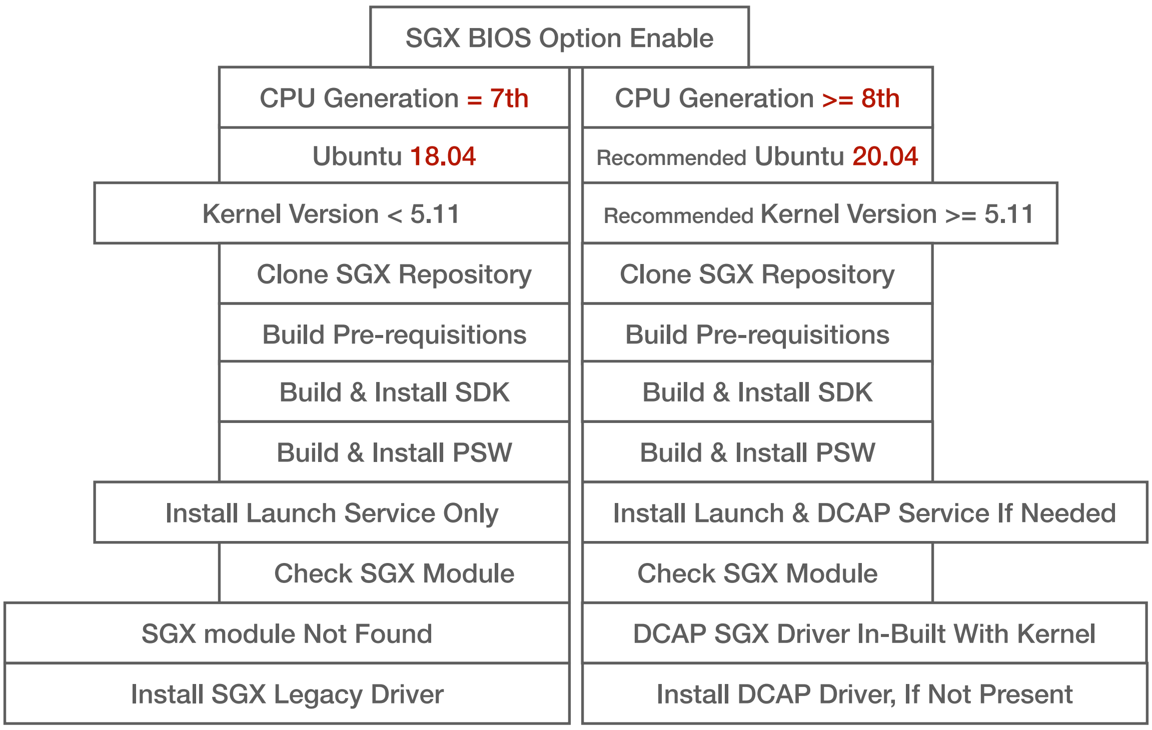


Figure 3.2: Installation paths based on Intel CPU generation and operating system kernels. For the purposes of our paper, we use Ubuntu as our Linux Distro and we have two paths, that are based on Intel CPU gen and with version of Ubuntu kernel we employ.

After we installed the legacy driver by running the appropriate .bin file, we can proceed to check if SGX module and device files are present. We also need to check if aesmd service of PSW is running. Both, the SGX device driver and aesmd service should be running to be able to load enclaves in hardware mode, as figure 3.3 shows.

Even though the SGX SDK has no specific path of installation, it is recommended to install it under /opt/intel/. Both PSW and SGX driver folder are by default installed to /opt/intel/

```
1.  >lsmod | grep sgx
    isgx

2.  >ls dev | grep sgx
    isgx

3.  >ps aux | grep -i aesm
    aesmd … /opt/instal/sgx-aesm-service/aesm/aesm_service …

4.  >service aesmd status
    should return green with no failed logs
```

Figure 3.3: Commands that can be used to check SGX module and the status of aesmd service.

and we encountered errors while linking the SDK libraries when executing already provided makefiles, that can be solved by manually changing the installation path variable of SGX libraries inside makefiles. In the end, /opt/intel/ should include 3 folders like in figure 3.4.

```
1. sgx-aesm-service
2. sgxdriver
3. sgxsdk
```

Figure 3.4: Folders that need to be present under /opt/intel/, in order for SGX to work properly.

If everything was successful, we can proceed to compiling and running a couple of Sample codes to verify that we can link the SGX libraries and load the enclaves without errors. Be advised that because /opt/intel/ is privileged directory, every command must be run with sudo or instead we can copy the SampleCodes directory to a directory that our user has permission to execute (i.e. Desktop).

3.3 SGX-Based FUSE FS Concept

In this section we will present our choices for designing a FUSE file system with the assistance of SGX. The proposed file system architecture, as the previous examined ones, is divided in a trusted and untrusted part, with ongoing and outgoing CALLs to and from the protected enclaves.

Lets update the image 1.1 that represents the architecture of a FUSE based file system, to include the SGX service and driver. The resulting of integrating SGX enclaves inside our

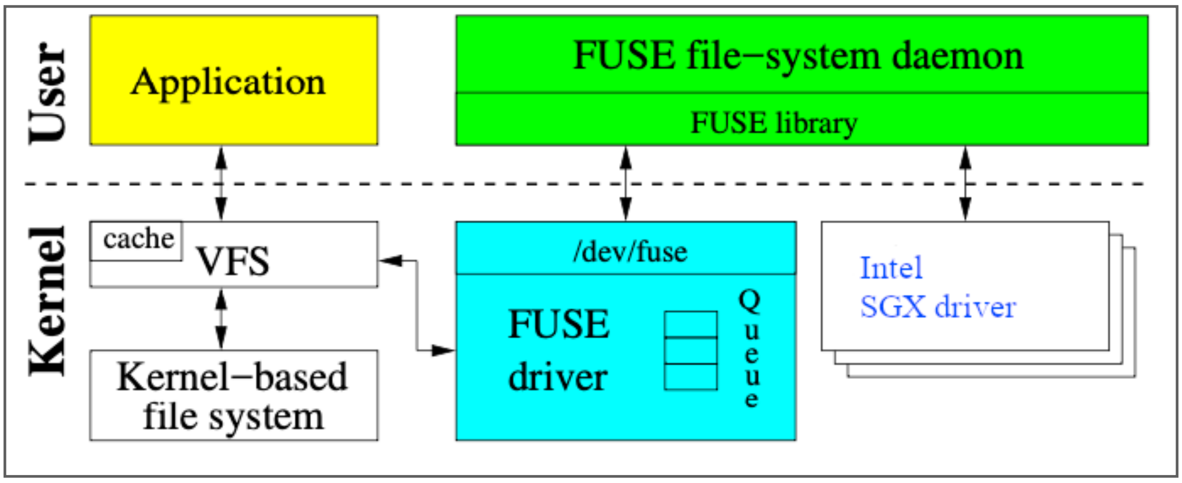


Image 3.3: The modified representation of a basic FUSE fs, as shown in image 1.1, to include SGX service, as a component for it's functionality.

FUSE fs is shown in image 3.3. Each operation will be forwarded from user space to kernel space then FUSE driver and, in case of a large data request will be divided into multiple smaller requests then to the user space file system implementation, which in return will forward the request to an enclave to determine if the operation should be allowed or not and then if no errors occur, to fulfil the desired operation and return the result back to the user space. The result would traverse the opposite direction that was previously stated. It should be also worth noting that SGX enclave side, could potentially also need to operate outside the secure environment, before completing the request and thus communicating with the kernel or requesting additional information or data.

The first design choice we need to make is, how can we represent the superblock of the file system inside the enclave (list or vector), the length of it, if it is better to dynamically reserve and allocate memory, to represent each node, or allocate a constant sized vector onto the stack, which will be used to store our nodes. To be able to mount several file system from the same or different user, we need to define a maximum size for our superblock and the maximum data blocks that each enclave can store.

Following, we need to determine what kind of cache policy should be used. The most common and well documented scheme is the LRU one. We keep inserting data blocks as long as enclave has available space and when the space is maxed out, we evict the blocks that reside on the back of the cache array. Throughout the execution of the enclave we move data blocks to the front each time the block is requested by an ECALL, without taking into account if the the array is filled or not. However if only a portion of blocks is frequently used, a basic LRU implementation can result on continuous changes in the blocks at the beginning of the array, which will not have any real usage, except further non required operations. To solve this, there might also be needed a way to divide the array of the cached blocks to two parts, one that have high frequency accesses and one that has lower, except from the already implemented LRU policy. This way continuous move inside the first partition can be avoided or delay by several by requiring several more accesses.

If we need to support a multithreaded file system, we need to rely on spin-locks inside the enclave instead of mutexes that use OCALLs to achieve synchronisation. Multithreaded file system can benefit the mount time, by utilizing multiple enclave to verify each files data.

To avoid file swapping attacks, we can assign to each file a unique id with proper length, that changes each time the file is saved and enclave closes. For this purpose the sgx sdk library that provides random() functions should be linked at compile time.

However avoiding whole file system superblock rollback attacks, we need to rely on external services, like a remote trusted server to provide us with encryption keys or nonces that we store together with our superblock each time we unmount the file system. In case that our system has a TPM, we can use this TPM to store encryption keys that are retrieved each time to decrypt the superblock and recover the nonce. This way a remote server is no longer need, but we must rely on TPM ability to prevent data leakage and also take into account any potential wear off from frequent usage.

To avoid any malicious return values tamper, each time an ECALL is made, the operation should be checked for validity, together with each file or directory id.

To test our file system, we can first run it in simulation mode without entering real enclaves by compiling with SGX_MODE=SIM and when we are sure about our implementation we can change the SGX_MODE=HW to run in hardware mode that effectively will create an enclave. We can also provide the flag SGX_DEBUG=1 to be able to debug our file system in case of potential crashes, because otherwise there is no mechanism to properly debug it, which would actually defeat the purpose of the secure and isolated execution. We most likely will use such file system for educational purposes so we can leave SGX_PRERELEASE=1. To run our file system in release mode we need Intel to also sign our file system code.

4. Conclusion

Safe cloud file storing and retrieving was already achieved by encrypting on client before sending it to remote file storage, which guarantees the integrity and confidentiality of files, assuming a secure channel. Client and server can leverage the SGX remote attestation to establish a secure channel, initialise any necessary variables and begin exchanging data. Further security assurances, like freshness or protection from known side channel attacks, can be provided by software layer addition on top of already implement file system schemes. Even under a malicious cloud OS or Hypervisor, the exchange of data between a honest client and a malicious server can be achieved with remote attestation and encryption inside an enclave, guaranteeing confidentiality and integrity. (leaving only dos door) However, under a local OS scope, Intel TEE itself cannot be considered a solution for secure file system implementations without the assistance of secure IO paths, trusted drivers and trusted containers. Without these components, a local SGX file system can be reduced to an ordinary file system that just performs some functions inside an isolated space and relies on OS for security assurances. Integrity and confidentiality of already encrypted stored data can be achieved by either SGX or non-SGX implementations, but secure retrieval or modification cannot be ensured by either of them under a malicious OS, without the assistance of a secure container and secure IO. It should be worth noting that although TEEs provide a secure environment, they cannot fully protect against all software fault injection attacks without support of additional hardware components.

Bibliography / References

1. Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Cloud-Native File Systems, USENIX Association, Boston, Jul 2018. pp 4-5.
2. Dr. Brijender Kahanwal. File System Design Approaches. https://arxiv.org/ Mar 2014. pp 1-3.
3. Bharath Kumar Reddy Vangoor, Vasily Tarasov, Erez Zadok. To FUSE or Not to FUSE Performance of User-Space File Systems. Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17), March 2017, Santa Clara, Ca USA. pp. 59-62.
4. Victor Costan and Srinivas Devadas. Intel SGX Explained, IACR Cryptology ePrint Archive 2016.
5. Intel(R) Corporation. Intel Protected FS. https://www.intel.com/content/dam/develop/external/us/en/protected/IntelProtectedFileSystem_Reference.pdf. pp 2-5.
6. Tate, Hongliang Tian Ph.D. Understanding SGX Protected File System. http://www.tatetian.io/2017/01/15/understanding-sgx-protected-file-system/, 2017.
7. Dorian Burihabwa, Pascal Felber, Hugues Mercier, Valerio Schiavoni. SGX-FS Hardening a File System in User-Space with Intel SGX, IEEE, Dec 2018.
8. Sandeep Kumar, Smruti R. Sarangi. SecureFS A Secure File System for Intel SGX. Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses, October 2021.
9. Shweta Shinde, Shengyi Wang and Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury and Prateek Saxena. BesFS A POSIX Filesystem for Enclaves with a Mechanized Safety Proof. Proceedings of the 29th USENIX Security Symposium, August 2020.
10. Samuel Weiser, Mario Werner. SGXIO Generic Trusted I/O Path for Intel SGX. https://arxiv.org/, Jan 2017.
11. Intel(R) Corporation. Intel(R) Software Guard Extensions for Linux* OS. https://github.com/intel/linux-sgx