DISKSHIELD: A Data Tamper-Resistant Storage for Intel SGX

Jinwoo Ahn
jinu37@sogang.ac.kr
Sogang University

Junghee Lee
j_lee@korea.ac.kr
Korea University

Yungwoo Ko
ppandol3@sogang.ac.kr
Sogang University

Donghyun Min
mdh38112@sogang.ac.kr
Sogang University

Jiyun Park
owo1113@sogang.ac.kr
Sogang University

Sungyong Park
parksy@sogang.ac.kr
Sogang University

Youngjae Kim
youkim@sogang.ac.kr
Sogang University

ABSTRACT

With the increasing importance of data, the threat of malware which destroys data has been increasing. If malware acquires the highest software privilege, any attempt to detect and remove malware can be disabled. In this paper, we propose DiskShield, a secure storage framework. DiskShield uses Intel SGX to provide Trusted Execution Environment (TEE) to the host, implements the file system into SSD firmware that provides a Trusted Computing Base (TCB), and uses a two-way authentication mechanism to securely transfer data from the host TEE to the SSD TCB against data tampering attacks. This design frees DiskShield from attacks to the kernel. To show the efficacy of DiskShield, we prototyped a DiskShield system by modifying Intel IPFS and developing a device file system on the Jasmine OpenSSD Platform in a Linux environment. Our results show that DiskShield provides strong data tamper resistance the throughput of read and write is on average to 28%, 19% lower than IPFS.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; **File system security**; **Database and storage security**.

KEYWORDS

Trusted Computing, Storage Security, OS Security

ACM Reference Format:
Jinwoo Ahn, Junghee Lee, Yungwoo Ko, Donghyun Min, Jiyun Park, Sungyong Park, and Youngjae Kim. 2020. DISKSHIELD: A Data Tamper-Resistant Storage for Intel SGX. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3320269.3384717

1 INTRODUCTION

Malware attacks that destroy data can cause very large damages including data loses to corporations and public organizations as well as individuals [8, 19, 22]. These attacks include ransomware [7, 8, 15] and wiper attacks [25]. A ransomware attack encrypts user files, takes the encrypted files as a hostage and demands a ransom [7, 8, 15]. On the other hand, a wiper attack indiscriminately destroys files on file systems [25]. Unlike ransomware that holds files for ransom, the wiper has no direct financial motivation. The wiper attacks include Petya [1] and Shamoon2 malware [4, 14]. Petya malware [1], first found in 2016, has the disruptive power that goes beyond simply encrypting user files, further damaging the system. Petya malware acquires root privilege, and then, by using the `ioctl()` system call, it infects the master boot record (MBR) region and encrypts the file system table. Even further, it deletes encryption keys after the encryption process, thus it can destroy the system permanently. Shamoon2 malware [4, 14], appeared in 2016, operates at the rootkit level. It uses a legitimate kernel driver to access files in the storage devices by bypassing the OS file system API. Thus, it can bypass any file system protections to files enforced by the OS and destroy files while the system is still running.

In order to prevent such ransomware and wiper attacks, various studies such as IPFS [9], Pesos Object Store [16], and Inuksuk [30] have been conducted [9, 16, 30]. These systems make a process space a Trusted Execution Environment (TEE) [27] to defend against the aforementioned data tampering attacks. IPFS [9] is an SGX-based file system which guarantees confidentiality and integrity of file data in the protected memory area of a process, called Enclave [6]. Enclave is a TEE, which provides processes with safe area even if the OS is compromised. In order to guarantee integrity, the IPFS produces Message Authentication Code (MAC) [29] at a write procedure and verifies it during the read procedure. Several SGX based file systems [2, 3, 24] have been implemented by using IPFS to guarantee both confidentiality and integrity for file data. However, IPFS only determines whether data has been tampered with. In other words, it does not check the MAC when writing data, but verifies the MAC of data when reading it later. As a result, data tampering attacks in the OS kernel memory in the middle of overwriting data can result in the loss of original data if there is no data backup. In addition, IPFS relies on the native file system of the OS. Malware can bypass IPFS and directly attack data on file systems or storage devices.

Unlike IPFS, Pesos [16] and Inuksuk [30] use a host's side TEE and a storage device together to protect data. They use a storage device to perform the authentication procedure internally for writing files. Therefore, malware such as Wiper that directly attacks data in the storage device is blocked because the device does not authorize authentication. However, Pesos is a SGX-based external third party storage service. Therefore, like other external storage services, it inevitably requires additional storage space. In addition,

ASIA CCS '20, October 5–9, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-6750-9/20/10…$15.00
https://doi.org/10.1145/3320269.3384717

it involves additional network I/O traffic overhead for external storage services. Also, Pesos requires a special disk, called Kinetic disk manufactured by Seagate. The kinetic disk provides object-based authentication to block unauthorized access. However, to use Pesos, users must purchase commercial Kinetic disks. Users can also only use the object storage features provided by Kinetic disks. And even if the user wants to protect the files selectively by creating a secure zone on the disk, it is not possible because the disk does not provide that kind of features.

Inuksuk [30], on the other hand, creates a secure zone on disk and allows users to selectively store files in that secure area. In Inuksuk, the disk can be partitioned into two regions – non-secure and secure zones. User files are stored in non-secure zone at first and then, files selected by users can be migrated to the secure zone. The secure zone provides write protection, so that malware can not tamper with the data in that zone. However, the problem may become catastrophic if malware avoids user's detection and breaks into the host systems. Because the OS is already compromised, newly updated secure files from applications are exposed to malware's attack. When migrating newly updated files between these zones, tampering intermediate data by the man-in-the middle attack in the compromised OS kernel is possible, thus tampered files can be saved in the secure zone.

To solve the aforementioned problems of existing systems, in this paper, we propose DiskShield, a new storage framework that is secure against data tampering attacks. The DiskShield system is implemented using TEE-based host processes and secure storage devices, similar to Pesos and Inuksuk. However, instead of using external storage like Pesos, DiskShield allocates a secure zone to the local SSD. Therefore, additional space and traffic overheads are minimized.

The storage device performs file-level authentication and blocks unauthorized data tampering attacks. Moreover, a lightweight device file system runs inside the SSD, completely eliminating vulnerabilities to metadata attacks of OS file system that can occur in systems like Inuksuk that relies on the OS file system. DiskShield allows users to build highly secure zones on disk. This area is safe from destructive attacks like the Wiper attacks aforementioned earlier. Also, there is no data movement between disk zones in DiskShield. Thus, as in Inuksuk, when OS is compromised, man-in-the-middle attacks that occur during data movement between disk zones never happen in DiskShield.

Specifically, DiskShield consists of three major components – DS_{FS} (Host-TEE), DS_{SSD} (SSD-TCB), and DS_{AE} (Authentication Enclave). $DS_{FS}$is a host-side enclave that extends IPFS. DS_{SSD} is an SSD that implements a file system that performs file-based authentication of users. DS_{AE} is another enclave that plays a key management role for secure key sharing and data communication between DS_{FS} and DS_{SSD}.

DS_{FS} is a user level file system like IPFS, which is implemented as an enclave using Intel SGX. But, the actual file system is implemented on the firmware inside the SSD device (DS_{SSD}). DS_{SSD} is considered a TCB for the following reasons [26]. First, the SSD firmware provides much smaller TCB comparing with upper-layer system including OS, which offers a smaller attack surface. Second, the SSD firmware is isolated from upper-layer. Therefore, even if the OS is compromised, SSD is secure. Finally, SSD is the last barrier to protect data, which cannot be bypassed by malware. DiskShield does not depend on the OS file system, but when I/O is performed from the DS_{FS} to the DS_{SSD}, the data of the files is temporarily copied into the OS kernel memory. In an environment where the OS kernel is compromised, if data resides in memory for a while, malware can destroy the data. This data, which resides in memory for a while, is called *fresh data*. If a fresh data attack occurs when writing to a file in the DS_{SSD} from the DS_{FS}, the data in the kernel's memory can be modified by the malware and then written to the SSD. Moreover, malwares like Wiper attacks can directly attack *persistent data* on the disks bypassing the OS's kernel memory.

To prevent this problem, DiskShield performs file-level authentication procedures inside the DS_{SSD} to prevent unauthorized writes. Unique keys are given per file. This file key is generated in the DS_{FS}. This key must be safely delivered to the DS_{SSD}. This is done with the help of the DS_{AE}. DS_{AE} manages the SSD's unique key to ensure that DS_{FS} and DS_{SSD} share the same key securely per file. DiskShield does not depend on the OS file system. As mentioned earlier, systems that rely on the OS file system are not free from metadata attacks on files residing in kernel memory. Thus, DiskShield implements a simple file system inside the SSD. This file system is a device file system that allocates secure zones inside the SSD and manages files. Because the metadata of files is managed inside DS_{SSD}, they are safe from file system metadata attacks.

We prototyped a DiskShield system by modifying IPFS (for DS_{FS}) in a Linux environment and developing a firmware-based device file system using the Jasmine OpenSSD Platform [18] (for DS_{SSD}). In particular, we modified the Intel SDK library and the Linux kernel to develop a secure two-way data communication mechanism between DS_{FS} and DS_{SSD} and developed the DS_{AE}. In addition, we have also developed a framework that enables secure communication between DS_{FS} and DS_{SSD} through DS_{AE}.

To fairly assess the effectiveness of DiskShield, we evaluated it from a security and performance standpoint. For security evaluation, we have qualitatively proved that DiskShield securely stores files against various attack scenarios in an environment where the OS is compromised. We also compared the I/O performance of DiskShield and IPFS using our own in-house synthetic file system benchmark. The evaluation results showed that DiskShield had on average 28% and 19% lower read and write throughput, respectively, compared to IPFS.

2 BACKGROUND AND MOTIVATION

2.1 Intel SGX and Secure File System

2.1.1 Intel SGX. Intel Software Guard Extension (Intel SGX) provided by Skylake Intel x86 processor provides an instruction set to guarantee confidentiality and integrity of a user process, even if OS is compromised. An isolated and trusted memory region that Intel SGX provides is called *Enclave*. In general, a software developer divides an application into an untrusted part and an enclave part, and then establishes an interface between these two parts. The untrusted part uses `ECALL` to call the enclave and the enclave uses `OCALL` to call the untrusted part. `OCALL` is necessary for the enclave because the enclave cannot directly use system calls. Intel Software Developer Toolkit (Intel SDK) is a basic library that Intel provides for SGX-based applications. It provides developers with

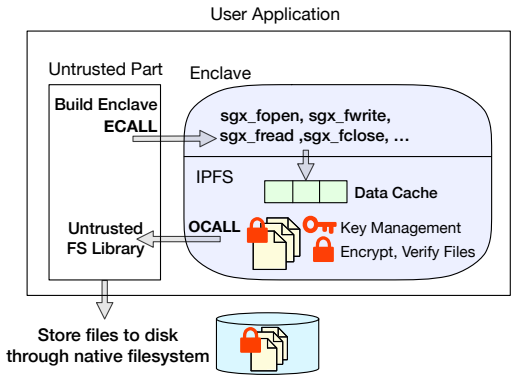


Figure 1: IPFS Software Architecture and Write I/O Flow.

various toolkits. For example, Intel Protected File System (IPFS) is a user level file system included in SDK. It provides a file system API that guarantees confidentiality and integrity of persistent data in the storage device. Enclave memory is protected by Memory Encryption Engine (MEE). This protected memory region is limited to 128 MB. To address this limitation, the SGX driver encrypts the enclave memory and swaps it out to the system memory if the enclave is full. However, the swap inevitably incurs overhead due to frequent encryption and decryption processes. Therefore, as the size of enclave increases, its performance is likely to degrade.

2.1.2 Secure File System based on SGX. Many researchers study various file systems that protect persistent data of the disk in the SGX environment [2, 3, 9, 24]. The Intel Protected File System (IPFS) is a file system to provide confidentiality and integrity of files. Intel builds the IPFS into the SGX SDK library. Figure 1 shows the workflow of an SGX application that stores files using IPFS. The application builds enclave and uses ECALL to enter the enclave. In the enclave, the application can use a set of IPFS APIs to store files. The IPFS employs its own data cache to improve performance. Also, it manages per-file keys. The IPFS encrypts and signs a file using this per-file key. For this, IPFS stores file data in the form of a Merkle hash tree, encrypts and verifies them. When IPFS stores file to a disk, it calls OCALL to call Untrusted FS Library [2] which calls a POSIX API instead of IPFS. This is because Intel restricts the enclave from directly calling system calls related to file operations. Then, the native file system in OS stores the file to the disk.

IPFS has a simple and efficient per-file key management policy [9]. When IPFS creates a file, the file key can be provided by the processor or the user. The processor derives a key from the enclave's identity (MRSIGNER) [10]. Therefore, other applications or enclaves can not write or read the files locked by the key. The user can also provide the key using IPFS API (sgx_fopen()). This is efficient when multiple enclaves have to share a single secure file. Also, IPFS provides API (sgx_fexport_auto_key(), sgx_fimport_auto_key()) to change the owner of files from one enclave to another [9]. To guarantee the confidentiality and integrity of data in the disk, when a file is written, the file is encrypted and a file's MAC is generated. When reading the file, file integrity is checked through MAC verification, and then the file is decrypted. Many SGX based file systems follow the IPFS to guarantee the confidentiality and integrity of the data in a disk. However, in this way, data cannot be protected from

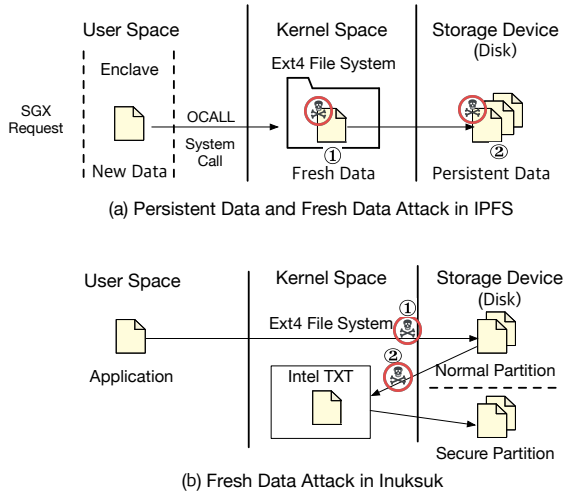


Figure 2: Description of Vulnerabilities in Data Tampering Attacks in IPFS and Inuksuk [30].

data tampering attacks of malware. *Integrity checking only shows whether the data has been tampered or not, but it is unable to block the data tampering attack.*

2.2 Motivation

Previous researches using the host process as a TEE [2, 3, 9, 24] can not defend against data tampering attacks. Figure 2(a) depicts an attack scenario for *persistent data* and *fresh data* in IPFS. Persistent data refers to data stored on disk. Fresh data, on the other hand, is newly updated data that is temporarily stored in kernel memory before storing on disk. If OS is compromised, malware can attack the persistent data. Also, if malware evades user's detection and invades in host system, it can intercept and corrupts fresh data. Then, from user's point of view, all the newly updated information is lost after malware invasion.

When a file write request is called using IPFS APIs inside an enclave, IPFS encrypts the file data, and its MAC is generated. Encrypted data leaves the enclave through OCALL and is conveyed to the user space system call (write()). Then, it is finally stored in the storage device through a native file system in OS. The data in the enclave is safe because SGX protects it. However, malware can use a "man-in-the-middle" attack, which threatens the data (①) that is created by the enclave but not yet stored in the storage device (Fresh Data). For example, malware can hijack the write system call of the native file system in OS, which IPFS calls through OCALL. In this way, malware can obtain the data when the system call is made and corrupt it. In addition, malware can directly attack the data that is persistently stored on the disk (persistent data) (②). Malware with a root privilege can use system calls, such as `pwrite()`, to files in the native file system (Ext4) in OS, making it possible to overwrite them. It can also use the `ioctl()` system call after discovering the file layout information (LBA list), thus making it possible to write directly to LBAs by opening a storage device. IPFS may detect the data tampered when reading it, but cannot prevent it.

Figure 2(b) shows the fresh data attack surface in Inuksuk [30]. Secure Encryption Disk (SED) in Inuksuk is composed of normal partition and secure partition. An application updates a file in

Table 1: Attack Surface of Data Tampering Attack.

Attack	Attack Vector	Description
Malicious Firmware Update	Data, Metadata in Storage	Malware can update malicious code to the firmware to corrupt data stored on SSD.
File based Attack	Data in Storage	Malware with root privilege can overwrite or delete the file that resides in the native filesystem (Ext4) at the user level, by using the system calls like `pwrite()`.
LBA based Attack	Data, Metadata in Storage	Malware can open the device directly, bypassing the filesystem, with functions like `ioctl()`, and modify persistent data by accessing the device's LBA.
System Call Hijacking	Data in OS	Malware can hijack systemcall(write()) to corrupts data buffer.
Malicious Device Driver	Data, Metadata in	Malware can modify device driver to store sensitive data in hidden storage space.
Malicious Enclave	Data, Metadata in Storage	Malware can builds malicious enclave to overwrite existing files generated from authorized enclave.
User Authentication Bypass	Data, Metadata in Storage	Malware can impersonate authorized users to connect correct application which can write the secure files.
Malicious Security Manager	Data, Metadata in Storage	Malware can attack the Security Manager (Authentication Enclave) which makes the secure channel between application and SSD.

the normal (non-secure) partition through the native file system in OS. In Intel TXT, the trusted updater is implemented. Trusted updater copies the user-selected files from the normal partition to the secure partition. This design firmly protects the persistent data already stored inside the secure partition. That is because nothing except the trusted updater can overwrite the data in the secure partition. However, malware can use a "man-in-the-middle" attack, which threatens the data (①, ②) that is created or updated by the application but not stored in the secure partition of the storage device yet. This is because the path between the application and the normal partition (①), and the path between the normal partition and the Intel TXT (②) are vulnerable to the malware. It is because these paths are made through a compromised OS kernel.

Also, neither IPFS nor Inuksuk can protect files from file metadata attacks as both systems rely on the native file system in OS. For example, malware like Wiper [25] can destroy file metadata of file systems in OS. If the file system metadata is attacked in this way, even if the contents of the file are not attacked, file access may be completely impossible. Therefore, in this study, we develop a secure storage framework, called DiskShield that can protect against persistent data and fresh data attacks mentioned above and can block file metadata attacks.

3 THREAT MODEL

Malware can invade host computers and elevate privileges to the highest level (Ring 0 level). Thus, it can compromise the user space and the whole kernel space, including the VMM, OS file systems, and kernel drivers. When the malware invades host, malware can tamper or delete data, metadata and even super block resulting in loss of data forever. Table 1 shows the possible attack surface.

Enclave codes can not be compromised because SGX protects the TEE in hardware level. Also, SSD firmware is a TCB of DiskShield, so the vulnerability of SSD firmware is out of scope in this paper (e.g., malicious firmware update). And, we assume that the SSD firmware update process is secure by a digital signature and secure boot [20]. Malware may attempt to attack the persistent data already stored in the disk (file based attack and LBA based attack). Second, malware can enter the OS in advance, and attempt to tamper fresh data that enclave creates or updates in the OS kernel memory (system call hijacking and malicious device driver). Third, although existing host enclave can not be attacked, the malware can succeed in building another malicious enclave. The malware's enclave can try to overwrite secure files generated from the host's authorized enclave (malicious enclave). Also, the malware can impersonate as a security manager (user authentication bypass) and give wrong

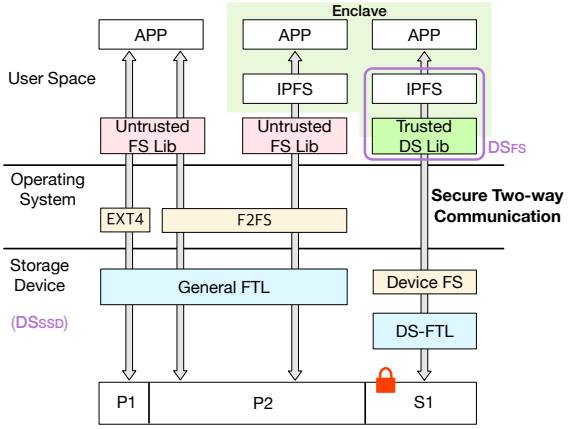


Figure 3: Comparing the software architecture of a DiskShield system to existing OS file systems. P1&P2 are normal partitions whereas S1 is a secure partition where DiskShield is formatted.

file keys to the host (malicious security manager). How to defend against these types of attacks are presented in Section 6.3.

We restrict the situation to a remote adversary that uses software to seize the victim's privileges and attack the user. Therefore, DiskShield cannot defend against physical attacks such as power analysis or chip modification [5, 17]. Also, if malware pretending as a user successfully authenticates with the enclave, it can try to corrupt secure files protected by an enclave. However, the success of this attack depends on the user authentication and file management policies provided by SGX application. Therefore, this attack is out of scope in this work because it is an SGX application implementation issue. Existing SGX file systems also do not address attacks that pass authentication between user and enclave [2, 9]. The purpose of the existing SGX based file system and DiskShield is to protect the data generated by SGX application. Finally, DiskShield cannot detect or remove malware. It also cannot defend DDOS and side-channel attacks. DiskShield aims to protect the user's sensitive data from being tampered with by malware.

4 DISKSHIELD STORAGE SYSTEM

4.1 Overview of DiskShield

DiskShield is a secure storage framework. Especially, it extends IPFS to implement DS_{FS}, a user-level DiskShieldfile system, and implements DS_{SSD}, a file system with firmware inside the SSD to completely block attacks from malware tampering with users' files. DiskShield provides a highly secure partition on SSDs from

Table 2: Notations for DiskShield Problem Formulation.

Component	Description
DS_{FS}	Extension of IPFS implemented as an Enclave on a host
DS_{SSD}	Updated SSD that performs file-level authentcation
DS_{AE}	Process that performs key management
rq_{Fi}	File related request parameters which should be transferred to DS_{SSD}
$PKList$	List of public keys from all manufacturers
Key	**Description**
K_{Fi}	File key generated by DS_{FS}
K_{SSD}	Unique Device Key
K_{MF}	The manufacturer key
Function	**Description**
$E_K(M)$	Encrypt the message (M) using Key (K)
$D_K(M)$	Decrypt the message (M) using Key (K)
$MAC_K(M)$	Generate Message Authentication Code generated from message (M) using key (K)

data tampering attacks, allowing users to selectively store files in their secure areas. Figure 3 illustrates DiskShield's software stack. DiskShield allows users to create normal and secure partitions on SSDs (P1, P2, S1). P[1-2] can be formatted using the OS file system. DiskShield, on the other hand, can be used by formatting S1 as DS_{SSD}. Users can safely read and write files selected in this secure area on DS_{SSD}.

4.1.1 Problem Formulation. DiskShield consists of a mix of hardware and software components. The first component is the user-level DiskShieldFile System (DS_{FS}). DS_{FS} is an extension of IPFS that is a user-level file system implemented as an enclave on a host. The second component is the DiskShieldSSD (DS_{SSD}). DS_{SSD} implements the file system inside the SSD and performs file-level authentication inside the device. The third component is the Authentication Enclave (DS_{AE}). DS_{AE} is a process that mainly performs reliable key management implemented as an enclave to share file keys K_{Fi} between DS_{FS} and DS_{SSD}. DiskShield uses three types of keys: The file key (K_{Fi}) generated by DS_{FS}, the unique device key (K_{SSD}) of DS_{SSD}, and the manufacturer key (K_{MF}) assigned by the SSD manufacturer to the DS_{AE}. DS_{FS} internally generates a file key (K_{Fi}). Each SSD is assigned a unique (K_{SSD}) by the manufacturer. Each SSD manufacturer has a unique key (K_{MF}). DS_{AE} is an SGX-based process provided by the manufacturer to the host, which has the manufacturer's key (K_{MF}) and SSD's key (K_{SSD}) inside the enclave. Since these keys are performed inside the enclave, they cannot be leaked outside. In addition, DS_{FS} includes $PKList$, which is a list of all manufacturers' public keys, and $PKList$ is permanently stored on disk through the sealing process. The public key in $PKList$ is used to verify DS_{AE} during attestation [12]. A detailed description of each key and component of DiskShield are given in Table 2.

4.1.2 Secure File Create I/O Flows. DiskShield ensures integrity for all files created using DS_{FS}. Read requests in DiskShield work the same as traditional IPFS. Using this file write request as an example, Figure 4 describes the I/O flow for write requests from the users and associated security procedures in DiskShield. In particular, DiskShield has the following assumptions about the threat model: (i) If needed, the user can safely update the SSD firmware through the legal firmware update procedure provided by the manufacturer. The manufacturer signs all data to be sent for firmware updates. The SSD firmware then uses digital signatures to verify the data [20]. (ii) The manufacturer provides the user with

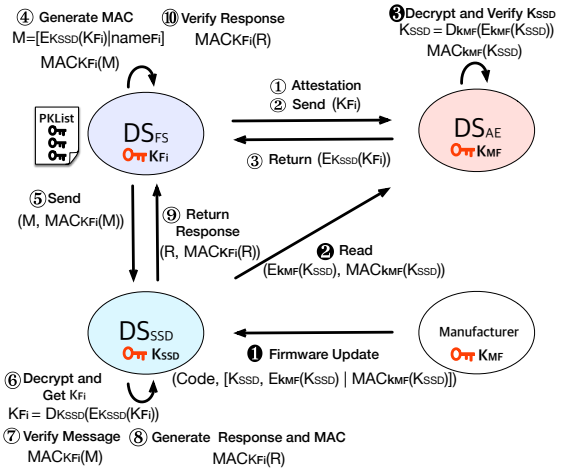


Figure 4: Description of the procedures that guarantee securely creating files in DiskShield.

a security manager process called Authentication Enclave (DS_{AE}). (iii) The $PKList$ is embedded into DS_{FS} by application developers.

DiskShield is basically a system that determines whether an SSD allows file access and blocks file access if it is illegal. To achieve this, SSD implements a file-based authentication system. In DiskShield, each file is given a unique key at file creation. And files on SSD are allowed to be accessed with file keys. To do this, the file keys created in DS_{FS} must be shared with the DS_{SSD} when the file is first created. DS_{FS} and DS_{SSD} share file keys securely using a symmetric key (SSD device key, K_{SSD}) with the help of DS_{AE}. In our DiskShield design, the symmetric key (K_{SSD}) is managed by DS_{AE}, so DS_{FS} can obtain the symmetric key through attestation with DS_{AE}. Thus, this symmetric key (K_{SSD}) must already be shared by DS_{AE} and DS_{SSD} before the file is created. In the following, we describe the procedure for how DS_{AE} and DS_{SSD} securely obtain copies of K_{SSD}.

To build a DiskShield system, the user must ask the manufacturer to update the SSD's firmware. Of course, if the manufacturer's firmware is already installed inside the SSD, this step is not necessary. When updating the firmware of an SSD, the manufacturer sends DiskShield's firmware code, K_{SSD} and a device file to DS_{SSD} (❶). The device file contains K_{SSD} encrypted with K_{MF} and the Message Authentication Code ($MAC_{K_{MF}}$ (K_{SSD}). The transfer for this firmware update from the manufacturer to the SSD is done securely with a digital signature [20]. DS_{SSD} updates the firmware with the code provided by the manufacturer and permanently stores K_{SSD} and the device file in the secure zone. Now, we describe the process of how DS_{AE} acquires K_{SSD}. DS_{AE} reads the device file from DS_{SSD} (❷), and finds K_{SSD} through decryption and integrity check (❸). Finally, DS_{AE} and DS_{SSD} can safely share K_{SSD}.

Next, we describe the procedure for a user to safely create a file from DS_{FS} to DS_{SSD}. DS_{FS} creates a file key (K_{Fi}) in the enclave. First, when opening the DS_{FS} file, DS_{SSD} checks whether the file exists. For this, DS_{FS} sends file name ($name_{Fi}$) and $MAC_{K_{Fi}}(name_{Fi})$ to DS_{SSD}. Here, MAC is required since tampered file names during the transfer should be filtered DS_{SSD} checks whether the file exists by received file name. Since DS_{SSD} has internal file system, it can check the file's existence. File system implementation details of

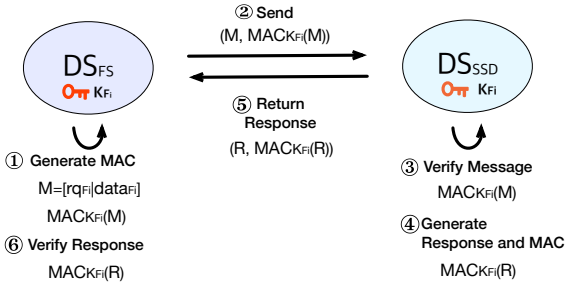


Figure 5: Description of the procedures that guarantee securely writing files in DiskShield.

DS_{SSD} will be explained in Section 5.3.2. If a file exists, then MAC is verified with DS_{SSD}'s key. If verification fails, DS_{SSD} signals an error that alerts that file name has been tampered. Moreover, DS_{FS} checks an Error with Read After Write method. Implementation details about Read After Write method will be explained in Section 5.2.1. When verification succeeds, file's existence is checked. DS_{FS}'s file open request succeeds if the file exists in DS_{SSD}. If not, DS_{SSD} alerts a missing file error. After that, DS_{FS} checks an Error with Read After Write method. DS_{FS} performs the following procedures for file creation.

First, DS_{FS} requests DS_{AE} for attestation [12] to establish a secure channel between them (①). Here, the attacker can install malicious DS_{AE} on the host and attempt to succeed in attestation with DS_{FS}. To defend against it, DiskShield adds the following process to SGX's attestation: First, during attestation process, DS_{AE} signs all messages sent to DS_{FS} with K_{MF}. Second, DS_{FS} reads public key corresponding to the manufacturer of DS_{AE} from *PKList*. Third, DS_{FS} verifies all messages received from DS_{AE} with a public key during attestation. If verification fails, then DS_{FS} determines it to be a malicious DS_{AE}, and rejects attestation.

After this, DS_{FS} sends K_{Fi} to DS_{AE} through this securely established channel (②). DS_{AE} encrypts K_{Fi} with K_{SSD} ($E_{K_{SSD}}(K_{Fi})$) and returns it to DS_{FS} (③). DS_{FS} creates a Message (M) using $E_{K_{SSD}}(K_{Fi})$ and the file name ($name_{Fi}$). In addition, it generates $MAC_{K_{Fi}}$ (M) using K_{Fi} (④). Then, DS_{FS} sends this information (M and $MAC_{K_{Fi}}$ (M)) to DS_{SSD} when creating a file (⑤). DS_{SSD} decrypts K_{Fi} encrypted from Message (M) using K_{SSD} (⑥). DS_{SSD} then uses K_{Fi} to generate $MAC_{K_{Fi}}$ (M) to prove its integrity (⑦). If verification fails, DS_{SSD} alerts an error to a message (M) that tells tampering has happened. After that, DS_{FS} uses Read After Write method to check an error and file open fails. If the verification is successful, the key (K_{Fi}) sharing of the file between DS_{FS} and DS_{SSD} is complete. And DS_{SSD} creates a file on the internal file system. DS_{SSD} creates a response (R) that has file-success return value and creates MAC ($MAC_{K_{Fi}}$ (R)) by using R as K_{Fi}. After that, DS_{FS} gets response (R) and $MAC_{K_{Fi}}$ (R) created by DS_{SSD} with Read After Write method. DS_{FS} proves $MAC_{K_{Fi}}$ (R) with K_{Fi}. When verification succeeds, the file open procedure ends with success (⑩). If verification fails, it means that response (R) or $MAC_{K_{Fi}}$ (R) was tampered in the middle of the procedure. Here, file-open ends with a failure, however file system semantic mismatch between DS_{FS} and DS_{SSD} occurs since file is correctly created in DS_{SSD}. Nevertheless, user can attempt to open a file again and solve the mismatch problem.

4.1.3 Secure File Write I/O Flows. Figure 5 shows the file write/update procedure. DS_{FS} locks data using K_{Fi}, generates $MAC_{K_{Fi}}$ (M) (①). Then sends them to DS_{SSD} (②). DS_{SSD} uses K_{SSD} to decrypt the message, prove its integrity, and decide whether to allow the write request (③). If verification fails, DS_{SSD} alerts an error to a message (M) that tells tampering has happened. When verification succeeds, write is performed with device file system. After that, DS_{SSD} creates a response (R) that has file-success return value, and creates a MAC ($MAC_{K_{Fi}}$ (R)) by using R as a K_{Fi}(④). DS_{FS} achieves response (R) and $MAC_{K_{Fi}}$ (R) created by DS_{SSD} with Read After Write method. On verification success, file-write sucessfully ends (⑥). If verification fails, it means that response (R) or $MAC_{K_{Fi}}$ (R) was tampered in the middle of the procedure. Here, file-write ends with a failure, however, file system semantic mismatch between DS_{FS} and DS_{SSD} occurs since file is written by DS_{SSD}. However, user can attempt to write a file again and solve the mismatch problem.

5 DISKSHIELD IMPLEMENTATION

5.1 DS_{FS} Module

DS_{FS} implements Trusted DS Libaray (Figure 3). This library differs from the existing FS library provided in IPFS for the following reasons. IPFS uses the untrusted FS Library to utilize native file systems [2]. The library uses a set of POSIX API for native file system requests to store files created by IPFS on disk. This library is an untrsuted library because it goes through the untrusted kernel's I/O stack. On the other hand, DS_{FS} provides a Trusted DS Library. DS_{FS} communicates directly with DS_{SSD} without going through the untrusted I/O stack, allowing I/O requests to the device file system, which is isolated inside of DS_{SSD}. This process is executed by secure two-way commnunication, so it is called trusted DS Library.

5.1.1 Attestation. DS_{FS} contains *PKList* which is a list of public keys from all manufacturer keys(K_{MF}). *PKList* is provided to the application developer in the form of a file using SGX sealing [11]. Therefore, only DS_{FS} can safely read the *PKList*. The file size is very small(less than 10 KB) because there are currently fewer than 80 SSD manufacturers [28]. DS_{FS} uses public keys to verfy all the message received from DS_{AE} during attestation. Therefore, the DS_{FS} can trust that the target of attestation is the correct DS_{AE}. We then build a secure channel between DS_{FS} and DS_{AE} by implementing local attestation provided by Intel SGX [10].

5.1.2 MAC Verification and Data Tunneling. DS_{FS} implements MAC Verification and Data Tunneling for secure data transfer when writing files to DS_{SSD}. When an application updates files, the Mac Verification Modules (MV) creates MAC [29] of the file's updated data and requests parameters that should be transferred. Request parameters include file descriptor, file offset and data size. This MAC is used in DS_{SSD} later to authenticate file update requests. Also, when DS_{SSD} returns the response message to the Enclave, MV verifies the response message to check whether it correctly came from DS_{SSD}. If it is successful, it returns the response to application.

DS_{FS} performs tunneling of user data buffer and requests parameters. For secure file I/O, there are additional request parameters that should be transferred through device driver (e.g., Linux ATA

Table 3: A set of DiskShield APIs.

DiskShield API	Description
sgx_fopen	Creates or opens a protected file
sgx_fopen_auto_key	Creates or opens a protected file
sgx_fwrite	writes the given amount of data to the file, and extends the file pointer by that amount
sgx_fread	reads the requested amount of data from the file, and extends the file pointer by that amount
sgx_ftell	Returns the current value of the position indicator of the stream
sgx_fseek	Sets the position indicator associated with the stream to a new position
sgx_fflush	forces a cache flush, and if it returns successfully, it is guaranteed that your changes are committed to a file on the disk
sgx_feof	tells the caller if the file's position indicator hit the end of the file in a previous read operation
sgx_fclose	closes a protected file handle
sgx_remove	deletes a file from the file system
sgx_fexport_auto_key	exporting the latest key used for the file encryption
sgx_fimport_auto_key	importing a Protected FS auto key file created on a different Enclave or platform
sgx_ferror	returns the latest operation error code
sgx_clearerr	attempts to repair a bad file status, and also clears the end-of-file flag
sgx_fclear_cache	clearing the internal file cache
sgx_flist	shows the list of file owned by Enclave

driver). As we will discuss in the experimental section, we have implemented DiskShield (especially, DS_{SSD}) on the OpenSSD Jasmine development board [18]. The OpenSSD Jasmine development board uses the SATA protocol. Therefore, this paper describes our implementation of the extension of the SATA protocol. However, our design is also applicable to the NVMe protocol. For example, the file descriptor and offset should be transferred so that device file system in DS_{SSD} can find files and position of updated data. Also, MAC (32 Bytes) should be transferred for authentication. We used tunneling to transfer the request parameter information as well as the user data buffer.

After MV generates MAC, the updated data buffer and request parameters (MAC, version, file name, key, etc) are enveloped on a single buffer. This single buffer is transferred to the storage device through a SATA driver.

5.1.3 User API. Table 3 shows a list of APIs provided by DiskShield. DiskShield basically supports as same API as IPFS [9]. User can creates secure file using sgx_fopen and sgx_fopen_auto_key [9]. If a file is created using sgx_fopen_auto_key, the file key is generated automatically by DS_{FS}. Here, the processor derives file key from enclave identity (MRSIGNER). Otherwise, the user can provide file key using sgx_fopen. In this way, multiple enclaves can have authority to access same file becuase user can use same file key for different enclaves. Also, there are APIs (sgx_fexport_auto_key, sgx_fimport_auto_key) that move the authority of file from one enclave to others. Additionally, the DiskShield provides new API named sgx_flist to show all files list owned by Enclave, similar to 'ls' commands in Linux.

Figure 6 shows the sequence of file creation and file write in DiskShield using sgx_fopen_auto_key and sgx_fwrite respectively, . For file close (sgx_fclose), the DS_{FS} asks DS_{SSD} to close the file. For file read (sgx_fread), the DS_{FS} asks DS_{SSD} to read the file, and get the data. Read request works the same as IPFS, because the IPFS provides read verification.

Table 4: Parameters and data transferred b/w DS_{FS} and DS_{SSD}.

File Request	Parameter	Data	Response
Open/Create	Cmd	MAC, Name, Key	MAC, Fd, Version, File size
Close	Cmd, Fd	MAC, Version	MAC, Retmsg, Version
Write	Cmd, Offset, Size, Fd	Data, MAC, Version	MAC, retmsg, Version
Read	Cmd, Offset, Size, Fd		Data

5.2 Secure Two-Way Communication Module

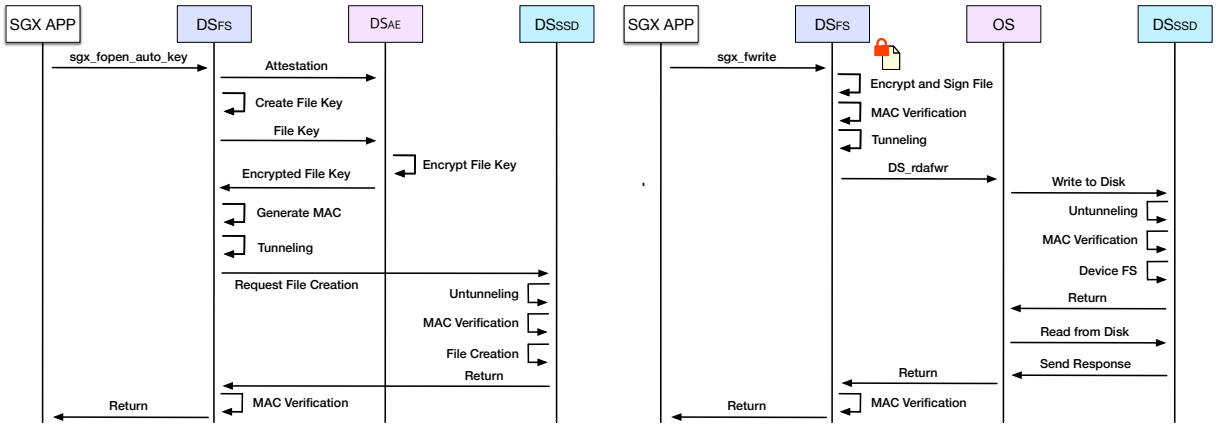
For secure two-way communication between DS_{FS} and DS_{SSD}, we implement a read-after-write method. We also modify the disk device driver to request the disk for filesystem operations of DS_{SSD}.

5.2.1 Two-Way Communication using Read-After-Write. We design a new system call (`DS_rdafwr`) to implement bidirectional data transmission between DS_{FS} and DS_{SSD}. This system call is called after DS_{FS} makes a single buffer using tunneling. It first sends the file update request to DS_{SSD}. Also, it receives response message including MAC from DS_{SSD} and sends it to DS_{FS}. However, the MAC (32 Bytes) returning as the device's response value is too big to be sent in the restricted response size (less than 20 Bytes) supported by the device driver. For this reason, we constructed Read-After-Write flow in the system call handler. The Read-After-Write (RAW) flow first sends write request to update files. Then, it sends read requests to the storage again. Then, it can read the long response message which includes MAC as a data buffer from DS_{SSD}. The system call handler of the storage device reads this data buffer and sends it to the DS_{FS}.

5.2.2 Device Driver Implementation. New device driver commands are needed to update secure files because DS_{SSD} has a different communication flow compared with normal I/O. DiskShield communicates with SSD through SATA device driver. SATA protocol transfers data by the unit of Frame Information Structure (FIS). Figure 7 shows the structure of Register FIS. Register FIS has a size of 20 Bytes. Register FIS includes request parameters such as command type, logical block address, and size of data to send. On the other hand, data FIS stores the data to send or data to receive. We have modified the SATA driver and constructed for the safe transmission of data and request parameters to the device. It also helps DS_{SSD} distinguish secure file update request from an existing normal file I/O. To this end, DiskShield first defines new SATA commands sets(e.g., DS_CREATE, DS_OPEN, DS_WRITE, DS_READ, DS_CLOSE). Table 4 exhibits request information and user data buffer transmitted to SSD through system call (`DS_rdafwr`). Furthermore, DiskShield optimizes request information when marshalling it in Register FIS. For instance, the reserved (4 Bytes) field of Register FIS is not utilized. Thus, instead of wasting the field, DiskShield utilizes it as a space to save additional parameters such as a file descriptor. Moreover, since DiskShield is not a block-unit-based request, LBA domain remains unused. File offset parameters are stored in the space.

5.3 DS_{SSD} Implementation

DS_{SSD} implements (i) unmarshalling and Mac generation to authenticate files and (ii) a file system inside the SSD for managing files in a secure zone using internal CPU and DRAM in the SSD.



(a) File Creation (b) File Write

Figure 6: Description of sequence diagrams of file creation and file write Operations.

	Byte3	Byte2	Byte1	Byte0
Word0	Features	Command	C|R|R|Reserved	FIS Type (27h)
Word1	Device	LBA(23:16)	LBA(15:8)	LBA(7:0)
Word2	Features(exp)	LBA(47:40)	LBA(39:32)	LBA(31:24)
Word3	Control	ICC	Count(15:8)	Count(7:0)
Word4	Reserved(0)	Reserved(1)	Reserved(2)	Reserved(3)

Figure 7: Register FIS structure.

5.3.1 Unmarshalling and MAC Generation. Using a file key shared by DS_{FS}, DS_{SSD} executes unmarshalling and MAC verification of secure file I/O requests. The version counter during MAC verification makes it possible to defend the replay attack.

First, the DS_{SSD} distinguishes the types of I/O (normal file or secure file I/O) by looking at the SATA commands. If it is secure file I/O, DS_{SSD} should unmarshalls request parameters that are hidden in register FIS (File descriptor, offset) and data FIS (name, key, MAC, version). SATA driver first sends register FIS to DS_{SSD} and sends Data FIS to DS_{SSD} afterward. Therefore, the DS_{SSD} first classifies data requests based on SATA command received from Register FIS. For normal data I/O requests, DS_{SSD} calls the existing General FTL (G-FTL) to service the request. If the SATA command is a secure file command (DS_OPEN, DS_WRITE, etc), DS_{SSD}, it first extracts request parameters (command, offset, size, file descriptor) from Register FIS and stores it in an event queue [18] for a while. After then, when data FIS arrives, the other request parameters (MAC,version) piggybacked on the data are untunneled. Then, the previously accumulated data in the event queue are extracted.

Second, after Unmarshalling the data, DS_{SSD} verifies the integrity of data in MV. The MV on the device side obtains the file key from device file system and then authenticates the host's file requests. If the authentication is successful, file update is executed through the internal filesystem. After the file system performs the file request, MV gets the response value from it, generates the MAC, and sends it to the host. The response data and MAC are checked on the host-side MV later.

We also design a version counter in MV to defend malware's replay attack. If a malware attacks data, DS_{SSD} can detect it by authenticating MAC, so it can prevent falsified data from being overwritten. However, malware can return the data saved in the device to the old version through a replay attack. For instance, if DS_{FS} performs a DiskShield write request in the compromised OS, a malware monitoring through system call can steal user data and parameters sent through `DS_rdafwr` system call. If the malware uses `DS_rdafwr` system call and re-transfer to SSD old data and parameter information sent before, it can not be blocked by DS_{SSD} because the data is signed by proper file key. In order to prevent this, DS_{SSD}'s MV has a version counter module. The version counter module of MV in DS_{SSD} updates file versions and stores version information inside the file inode of the filesystem. On the other hand, DS_{FS} saves version information received from DS_{SSD} in the `protected_fs_file`. `protected_fs_file` is a IPFS file metadata structure which is loaded on the safe enclave. When this metadata should be updated on disk, it becomes a part of the user data buffer. Because the user buffer is authenticated in DS_{SSD}, the version counter in `protected_fs_file` is also safely stored.

The following shows the flow in blocking process of replay attack; First, when making a file update request, DS_{FS} packs file version information in user data domain and sends them (see Table 4). Then, if MAC authentication is successful in DS_{SSD}'s MV, version Counter compares if the version information sent by DS_{FS} agrees with that saved in file inode. If they do not match, it is regarded as a replay attack. Such a command is rejected and error code is returned to the host. If they match, on the other hand, inode version is updated (simply by adding 1 to the previous version) and the updated version information is sent to the host as a part of response. DS_{FS} compares the returned version information with its previously-held version. If the returned version is a one-notch-upgraded version from the existing one, authentication is successful. Then, DS_{FS} keeps the upgraded version in `protected_fs_file` and sends back when making an update request later.

5.3.2 DiskShield Device File System. DS_{SSD} implements a file system inside. The file system manages per-file key inside DS_{SSD} and assists authentication in the unit of file. Moreover, it directly performs requests sent by DS_{FS}. Since this is a filesystem running within the limited SSD resources, however, the required memory size must be minimized. DS_{SSD} constructs DS-FTL in the unit of file and reduces the indexing conversion process from offset-LBA-PBA to offset-PBA. Therefore, it removes the necessity for an additional

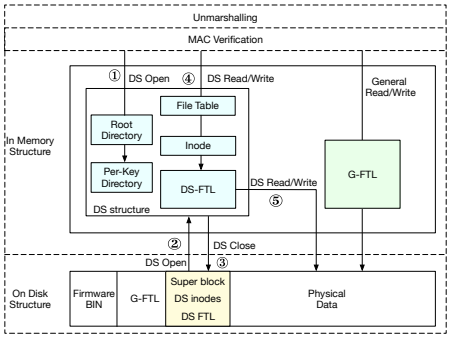


Figure 8: DS_{SSD} in-memory and on-disk structure.

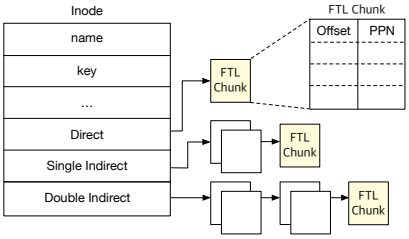


Figure 9: Inode and DS-FTL structure.

file allocation table. Also, DS-FTL itself does not require additional memory space. When host updates SSD to implement DS_{SSD}, it can set the maximum size of the secure file area. Then, DS_{SSD} reduces the size of the existing FTL (G-FTL) which is unusable, and allocates DS-FTL accordingly. For example, assume that SSD size is 128GB, and the host allocates secure file area as a 8GB. Because the G-FTL only has to map 120GB now, DS_{SSD} reduces the size of G-FTL, and allocates this space for DS-FTL which have to map until 8GB. The part below explains the overall structure of DS_{SSD} file system and how a set of file operations requested by DS_{FS} are executed.

File System Structure. Figure 8 shows the in-memory and on-disk structure inside the DS_{SSD}. Note that here the in-memory means internal DRAM of the SSD. Information about the superblock, inode, and DS-FTL is saved inside the on-disk fixed area. The super block is loaded into the memory during the booting sequence, and the inode and DS-FTL are loaded into the memory when the file is opened upon a request. The following shows the DiskShield Device File System (DDFS) structure within the in-memory structure:

- **Super block** saves the information, such as device key (K_{SSD}), root directory pointer, and the number of files. The device key is an unique key that each SSD owns.
- **Inode** is the metadata information of a file. It does not only saves the necessary information, such as a file name, size, inode number, but also saves the file key (K_{Fi}), version, and the DS-FTL pointer. The file key is a per-file key that is shared with an Enclave and transmitted from the host when the file is created. When the Enclave sends a file update request, the version is updated to block the replay attack.
- **DS-FTL** differs from the regular FTL (G-FTL), which is accessed through the logical block address. Figure 9 shows the DS-FTL structure which is allocated per file. The DS-FTL maps the file offset to the physical block address (PBA). Like the previous UNIX filesystem, inode manages the DS-FTL as a pointer-based file indexing using the direct and indirect node blocks. Each FTL chunk maps the offset to the physical address where data is stored. This mechanism minimizes overhead because the FTL chunk is dynamically allocated as the file size grows. For instance, a file under 4MB is accessed through the direct field, so that only one FTL chunk is allocated. If the file size is 128 MB, the FTL chunk is allocated up to a single indirect area, and if the file size is 4 GB, it is allocated up to a double indirect area.
- **Root directory** is the root directory of DDFS. Root directory consists of per-key directories that own different keys. When file open request (DS_OPEN) arrives in DDFS, DDFS first searches root directory to find the specific per-key directory that matches the key from host.
- **Per-key directory** has all files which have the same file key. It maps the file name to the inode number as a hash table. DDFS searches inode here when file open request (DS_OPEN) arrives. Also, this directory is used for the sgx_flist API (see Section 5.1.3) which shows the list of files owned by the same key.
- **File table** maps the file descriptor to the inode pointer.

File Operation. In Figure 8, the DS_{FS} accesses the per-key directory from the root directory in the open procedure (①). Then, it obtains the inode number from the per-key directory and loads the inode and DS-FTL from the NAND Flash(②). Finally, it registers the new file descriptor and inode pointer to the file table and returns the file descriptor. During the close procedure, the loaded inode and DS-FTL are saved to the NAND Flash(③) and returns a response to the host. When performing a read or write procedure, the DDFS obtains the inode pointer from the file table, which is mapped to the file descriptor(④). Then, the DDFS accesses to DS-FTL through an inode and accesses the data through a Physical Block Address (PBA) that is mapped to the file offset (⑤). Then it accesses to PBA to write or read data.

Recovery from Power Failure. DS_{SSD} guarantees consistency of DDFS. The inconsistency issues generated by power failure can occur when a single command on the DS_{FS} is broken up into muliti-ple operation in DS_{SSD}. Cause of file system inconsistency in DS_{FS} includes (i) sgx_fwrite command makes DDFS to update not only data, but also inode, and DS-FTL and (ii) file creation (sgx_fopen, sgx_fopen_auto_key), file removal (sgx_remove), and file key modification (sgx_fexport_auto_key, sgx_fimport_auto_key) makes DS_{FS} update not only inode but also per-key directory and root directory. If there is power failure during these operations, consistency can be broken. To solve this problem, DS_{SSD} utilizes OOB (out-of-band) area. OOB refers to a few bytes of region additionally allocated to each NAND flash physical page. This area is used to store the metadata of the physical page, such as logical address. Since this OOB area is transparent to Host system, security of the OOB area is preserved. To prepare for power failure, when page write occurs for secure file, the DS_{SSD} records inode number and offset in the OOB metadata area instead of traditional logical address.

When the power-failure occurs, the recovery is performed with following steps – Step(1): As in the conventional method, DS_{SSD} searches full NAND flash and recovers mapping table of G-FTL by searching logical address in each OOB. Step(2): For secure file, DS_{SSD} reads inode and offset from OOB space during NAND flash

search and recover the inode and DS-FTL. Then, with this information (Inode, Offset, PBA), it can recover the inode and DS-FTL. Step(3): The DS_{SSD} searches full inode which has file key. With this information (inode, file key), it can reorganize per-key directory and root directory.

6 EVALUATION

We performed our experiment on Intel(R) Core(TM) i7-8700 CPU @ 3.70GHz with 16 GB RAM (128 MB for EPC). DiskShield was implemented on Linux 4.10.16 and DS_{SSD} was implemented on the SATA2.0 based OpenSSD Jasmine [18], which is composed of ARM7TDMI-S core running at 87.5MHz, 96KB SRAM and 64MB DRAM, and 64GB NAND Flash memory chip. To prototype DiskShield, we modified 2,661 LoC (lines of code) on Jasmine Open SSD, 791 LoC on SGX SDK, and 243 LoC on Linux kernel.

To implement DS_{FS}'s two-way communication module, we implemented a system call handler to send direct requests to the device in a READ-AFTER-WRITE (RAW) mechanism. The system call inserts a request to the DS table, which is a data structure in a kernel, responsible for moving parameters of DS file I/O requests to the SATA driver. Specifically, the system call handler inserts parameters into the DS table, and the SATA driver retrieves them from the DS driver. In the SATA driver, the new SATA command for DiskShield is defined and the requests popped from the DS table are sent to the device. DS_{SSD} implemented all the flows for DDFS, but we used a fixed HMAC time delay of an FPGA module for authentication [23] since the OpenSSD does not include the hardware module for HMAC authentication unlike the latest SSDs [21]. The delay used is 23.01 us for authenticating 4,608 bytes [23], which is the size of a basic unit for transmission in IPFS. Delay is calculated based on the Xilinx Virtex 6-3 which is HMAC SHA-256 IP core.

6.1 Performance Evaluation

To measure the performance overhead of DiskShield, we developed a multi-threaded in-house synthetic workload generator. For a fair comparison against direct I/O implementation of DiskShield, the DS_{FS} uses direct I/O policy which bypasses the page cache in Virtual File System.

Figure 10, Figure 11 and Figure 12 show throughput comparisons betweeen DiskShield and IPFS (baseline). Note that DiskShield offers both secure zone (DiskShield S-zone) and non-secure (normal) zone (DiskShield R-zone) on the disk. DiskShield R-zone means the IPFS-based regular file I/O zone using DS_{SSD}. This measures the I/O overhead of DS_{SSD}'s regular zone. DiskShield S-zone means the DS_{FS}-based secure file I/O zone using DS_{SSD}. Only the DiskShield S-zone files are protected from data-tampering attacks.

Figure 10 and Figure 11 show the I/O throughput comparison for big file (16 MB) and small file (4 KB) workloads with respect to increased number of I/O threads. The in-house synthetic workload first creates files and then write the files. Then, overwrite and read throughput are measured. As expected, a small file workload shows lower performance compared to a big file workload.

From Figure 10, we observe the overall throughput of Baseline and DiskShield R-zone is almost equal. This result shows that the DS_{SSD} incurs negligible overhead for DiskShield R-zone. Also, for the single thread, the read throughput of DiskShield S-zone is

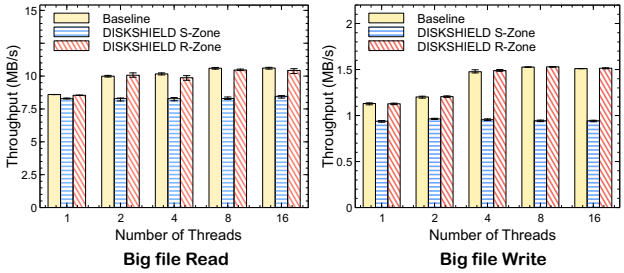


Figure 10: Comparisons for IPFS and DiskShield with big files.

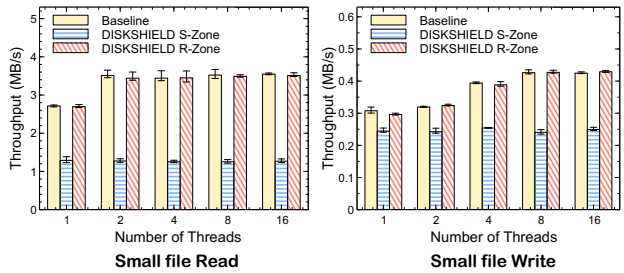


Figure 11: Comparisons for IPFS and DiskShield with small files.

similar to the baseline, whereas the write throughput of DiskShield S-zone is 17% lower than the baseline. The low write throughput of DiskShield S-zone results from the DS_{FS} using the RAW requests to implement the two-way communication between the enclave and DS_{SSD}.

As the number of threads increases, the write and read throughput of baseline and DiskShield increases and converges to 1.5MB/s and 10.5MB/s. On the other hand, the DiskShield S-zone does not vary in performance at all as the number of threads increases. This is due to the serialization of RAW processes between multiple I/Os. Specifically, in our implementation, we used RAW process with locks because the write requests and read requests for receiving SSD response must be processed sequentially. If the driver supports sufficiently large response size, this problem may be solved because there is no need to implement RAW.

Figure 11 shows similar results in Figure 10. In particular, we observe that DiskShieldS-zone's read performance is quite low in the case of a single thread. This was not the case for big file workloads. The overhead mainly comes from the overhead for loading and storing a large number of inode metadata in DiskShield. If the file is small, this overhead becomes exceptionally large. In the baseline, the Ext4 file system loads multiple inodes in the main memory at once. Therefore, multiple inodes are cached in the main memory, eliminating the need of loading them from the storage. In contrast, the DiskShield S-zone only loads a single inode to minimize memory usage. Therefore, an inode should be loaded from the storage every time a file is opened. This is the trade-off between the memory space usage and the metadata hit ratio. If the DiskShield S-zone loads inode to memory in advance like native file system, then the temporal overhead will fade away. However, it will need more memory space in SSD.

Figure 12 shows the performance evaluation for different I/O patterns. We see that the overall performance of write is lower than read. This is because Jasmine OpenSSD Jasmine's block-level write performance (50MB/s) is 5x lower than its read performance

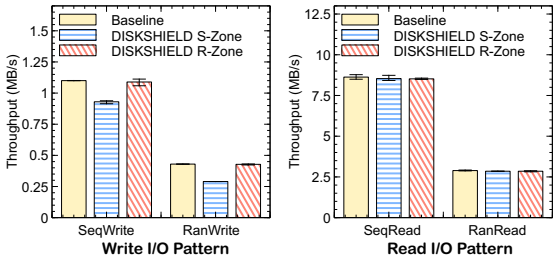
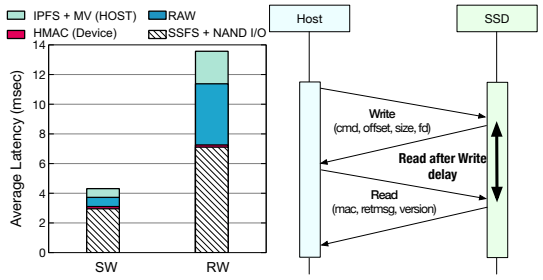


Figure 12: Comparisons for IPFS and DiskShield with various I/O patterns.



(a) Time latency (b) RAF flow

Figure 13: Performance analysis of DiskShield for write I/Os.

(225MB/s). Generally, regardless of the I/O pattern, the read throughput is almost same between baseline, DiskShield R-zone, and DiskShield S-zone. It shows that the DiskShield's read flow does not incur much overhead. In sequential and random write patterns, the throughput difference of baseline and DiskShield R-zone is negligible. This means that DS_{SSD} does not disturb regular file I/O and offers comparable performance with baseline. However, the sequential and random write throughput of DiskShield S-zone is 13% and 31% lower than the baseline.

To analyze the overhead of DiskShield S-zone, we measured the response time of sequential write and random write requests. We also broke down the response time. Figure 13(a) shows the time break-down of the response time for writing 64 MB files in sequential and random patterns. The ratio of the response time of HMAC operation in DS_{SSD} is negligible because the delay (23.01us) is much smaller than the amount of response time. The RAF response time is the delay of two-way communication in DS_{FS}. Because the two-way Communication is implemented using the RAW pattern in the system call handler, the unnecessary delay occurs as depicted in Figure 13(b). This overhead occupies 14%, 30% of sequential and random write response time. The reason for the bigger overhead in the random write is that the IPFS sends more write requests to the storage in random write. This is because data cache hit ratio in IPFS is lower.

Table 5 compares the open and close latency of IPFS and DiskShield S-zone for different file sizes. The latency of IPFS is all similar, regardless of the file size. However, in the case of DiskShield, the open and close latency significantly increase as the file size increases. This is due to overhead from loading and storing DiskShield's DS-FTL. When a file is opened, the inode and DS-FTL are loaded from NAND flash, and when the file is closed, they are stored in NAND flash. Also, the size of DS-FTL linearly increases as the file size grows. For example, when file size is 4KB, the total size of DS-FTL is only 32B. When the file size becomes 512MB, the total size of DS-FTL is about 64KB. This leads to increment in data size that should be loaded and stored.

Table 5: Open/close latency (ns) comparisons for different file size.

File Size		4KB	2MB	32MB	512MB
Open	IPFS	9.501	12.455	11.563	10.169
	DiskShield S-zone	163.86	191.70	351.36	4412.3
Close	IPFS	17.877	22.215	18.076	18.494
	DiskShield S-zone	192.51	218.66	512.21	5061.6

Table 6: Security analysis on the SGX-based filesystem.

Name	IPFS	Obliviate	Graphene	SPICHER	Inuksuk	DiskShield
Confidentiality guarantee	✓	✓	✓	✓	✓	✓
Integrity verification	✓	✓	✓	✓	✓	✓
Freshness verification	×	✓	×	✓	✓	✓
Side-channel attack	×	✓	×	×	×	×
Persistent data attack	×	×	×	×	✓	✓
Fresh data attack	×	×	×	×	×	✓

Table 7: DiskShield's defense to data-corruption attacks.

Attack Scenario	DiskShield Solution
File based Attack	Secure file can be overwritten only by authorized DS_{FS} which has right file key.
LBA based Attack	Secure file is hidden from OS, so LBA based approach is impossible.
System Call Hijacking	Even if the data buffer is corrupted, DS_{SSD} deny this request by verifying MAC. Here, DS_{FS} verifies error message from DS_{SSD}.
Malicious Device Driver	As same as system call hijacking
Malicious Enclave	Existing IPFS does not permit unauthorized enclave to open secure files.
Malicious Securiy Manager	DS_{FS} verifies DS_{AE} using *PKList* during attestation.
Replay Attack	DS_{SSD} authenticate replay attack through I/O Version Counter.

6.2 Space overhead analysis in device

In this part, we explain the space overhead of DS_{SSD} for implementing the file system (DDFS) in the device. The main space overhead generated by DDFS is super block, inode, and the DS-FTL. Because DiskShield directly maps an offset to a physical block address, it does not need an additional indexing table for mapping an offset to a logical block address. When the secure partition is allocated, the normal partition size in which G-FTL maps reduces. Therefore, the size of G-FTL also decreases. DiskShield takes advantage of this free space. Within this free space, the DDFS dynamically allocates memory for inodes, superblocks, and DS-FTLs. Therefore, DDFS does not need additional memory space in the SSD. Currently, this metadata is loaded when file opens, and be flushed when file closes. In future work, to reduce flushing metadata, We will implement a cache mechanism that utilizes this memory space efficiently.

6.3 Security Evaluation

We analyzed the security of diverse TEE-based local data storage systems that protect the persistent data. Pesos is an external third party storage service, we do not compare the DiskShield with Pesos because the target environment is different. Table 6 presents a security analysis of the TEE-based file system and the key-value store. IPFS [9] is a basic file system that is implemented in the SGX SDK library. It guarantees file confidentiality and checks integrity through the read-verification process. Obliviate [2] is an in-memory filesystem that is specialized in the side-channel attack defense. It checks integrity using the Merkle hash tree, same as IPFS. Graphene [24] provides a library OS that does not require modification of the original application, but rather, loads it directly

into the SGX. Graphene checks integrity by comparing the hash values when a file is opened. SPEICHER [3] is a key-value store that is specialized in defending replay attacks. It also guarantees confidentiality and checks integrity, similar to IPFS. It builds a direct I/O library that is based on Intel SPDK [13]. Using this system, the data created by an Enclave is saved in the user buffer by an OCALL and is transferred directly to the device while bypassing the kernel. However, malware not only can attack the persistent data, but it can also damage the fresh data by modifying the user data buffer, which is outside of an Enclave. On the other hand, the inuksuk protects persistent data already stored in the secure zone even if malware has invaded. However, if the malware avoids user detection and succeeds in invading the host system, the situation would be catastrophic. This is because user continuously updates secure files without knowing the existence of malware. All kinds of newly updated fresh data can be tampered by malware before arriving at secure zone. Then, the user loses updated information forever. To sum up, none of these systems can protect the integrity of persistent data from corruption attacks.

DiskShield protects persistent data through DS_{SSD} authentication. In addition, when the fresh data is tampered by malware, DS_{SSD} denies it by verifying MAC. Also, the DS_{FS} verifies response received from DS_{SSD} through two-way communication. If the MAC verification or response fails, DS_{FS} returns write error message to the application.

Table 7 shows DiskShield's solution to the file modification attack scenarios and solutions described in Section 3. Using pwrite() for file based Attack is impossible because the secure file is hidden from a native file system. On the other hand, the malware can use system calls such as `DS_rdafwr` provided from DiskShield and try to attack secure zone. However, DS_{SSD} denies this request since the request verification will fail. System call hijacking and malicious device driver are a kind of man-in-the-middle attack, for which DiskShield can defend using DS_{SSD}'s authentication and secure two-way communication. Also, a malicious enclave can try to open the secure file using sgx_fopen(). However, existing IPFS denies opening files because the file key is not matched.

7 CONCLUSION

This paper presented DiskShield, a tamper-resistant storage for Intel SGX. The DiskShield is an SGX-based versatile data protection system optimized for local platform. We prototyped DiskShield to evaluate security and performance analysis using the Jasime OpenSSD Platform in Linux. The evaluation shows the overheads of protecting data as tamper-resistant state is reasonable compared to the baseline IPFS.

8 ACKNOWLEDGMENTS

This research was supported in part by Samsung Semiconductor research grant and by Next-Generation Information Computing Development Program through National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7080243). Y. Kim is the corresponding author.

REFERENCES

[1] 2017. Petya/NotPetya Ransomware Analysis. https://idafchev.github.io/writeup/2017/07/21/petya_ransomware_analysis.html.
[2] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. Obliviate: A data oblivious file system for intel SGX. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*.
[3] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*.
[4] Osborne Charlie. 2018. Shamoon data-wiping malware believed to be the work of Iranian hackers. https://www.zdnet.com/article/shamoons-data-wiping-malware-believed-to-be-the-work-of-iranian-hackers/.
[5] Chintan Chavda, Ethan C Ahn, Yu-Sheng Chen, Youngjae Kim, Kalidas Ganesh, and Junghee Lee. 2017. Vulnerability analysis of on-chip access-control memory. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.
[6] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.
[7] Joseph Goedert. 2016. Meeting the Threat of Ransomware. https://www.healthdatamanagement.com/author/joseph-goedert.
[8] Dan Goodin. 2017. A New Ransomware Outbreak Similar to WCry is Shutting Down Computers Worldwide. https://arstechnica.com/security/2017/06/a-new-ransomware-outbreak-similar-to-wcry-is-shutting-down-computers-worldwide/.
[9] Intel. [n.d.]. Intel Protected File System Library. https://software.intel.com/sites/default/files/managed/76/8f/OverviewOfIntelProtectedFileSystemLibrary.pdf.
[10] Intel. [n.d.]. Intel Software Guard Extensions Developer Guide. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf.
[11] Intel. [n.d.]. Introduction to Intel SGX Sealing. https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing.
[12] Intel. [n.d.]. Local (Intra-Platform) Attestation. https://software.intel.com/en-us/node/702983.
[13] Jonathan S. (Intel). 2016. Introduction to the Storage Performance Development Kit (SPDK). https://software.intel.com/en-us/articles/introduction-to-the-storage-performance-development-kit-spdk.
[14] Devika Jain. 2017. Shamoon 2: Back On the Prowl. https://nsfocusglobal.com/shamoon-2-back-on-the-prowl/.
[15] Elliot Kass. 2016. Roundtable: Ransomware. , 25–32 pages. Health Data Management.
[16] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference*.
[17] Junghee Lee, Kalidas Ganesh, Hyuk-Jun Lee, and Youngjae Kim. 2017. FESSD: A fast encrypted ssd employing on-chip access-control memory. *IEEE Computer Architecture Letters* 16, 2 (2017), 115–118.
[18] Sang-Phil Lim. 2016. The Jasmine OpenSSD Platform: Technical Reference Manual (v1.4, in English). http://www.openssd-project.org/mediawiki/images/Jasmine_Tech_Ref_Manual_v.1.4e.pdf.
[19] F. Mercaldo, V. Nardone, and A. Santone. 2016. Ransomware Inside Out. In *Proceedings of the 2016 11th International Conference on Availability, Reliability and Security (ARES)*.
[20] Micron. 2016. Protecting Your SSD and Your Data. https://www.datasheetarchive.com/whats_new/1c1a884377ab1954f2efc54b614636ec.html.
[21] Inc Micron Technology. 2017. FIPS 140-2 Cryptographic Module Non-Proprietary Security Policy. https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp2848.pdf.
[22] C. Moore. 2016. Detecting Ransomware with Honeypot Techniques. In *Proceedings of the 2016 Cybersecurity and Cyberforensics Conference (CCC)*.
[23] Mercora Technologies. 2017. HMAC SHA-256 Fast IP Core. http://www.mercoratech.com/products/hmac-sha256-fast-core.
[24] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17)*.
[25] VENTURA VITOR. 2018. WIPER MALWARE: ATTACKING FROM INSIDE. https://talos-intelligence-site.s3.amazonaws.com/production/document_files/files/000/033/904/original/Talos_WiperWhitepaper.v3.pdf.
[26] Xiaohao Wang, Yifan Yuan, You Zhou, Chance C Coats, and Jian Huang. 2019. Project Almanac: A Time-Traveling Solid-State Drive. In *Proceedings of the 14th EuroSys Conference*.
[27] Wikipedia. 2018. Trusted Execution Environment. https://en.wikipedia.org/wiki/Trusted_execution_environment.
[28] Wikipedia. 2019. List of solid-state drive manufacturers. https://en.wikipedia.org/wiki/List_of_solid-state_drive_manufacturers.
[29] Wikipedia. 2019. Message authentication code. https://en.wikipedia.org/wiki/Message_authentication_code.
[30] Lianying Zhao and Mohammad Mannan. 2019. TEE-aided Write Protection Against Privileged Data Tampering. In *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*.

A SECURITY EVALUATION USING MALWARE SAMPLE

To evaluate with some attack vectors discussed in Table 7, we have created attack scenarios based on the behavior of real malware samples. We assume that a remote attacker invades host systems and runs malware. We test the following scenarios.

A.1 Persistent Data Attack

A.1.1 File-based attack using DiskShield system call.

Assumption. The attacker has successfully invaded into the host system and installed a user-level malware. The attacker also knows that the foo.txt file is safely stored in the DS_{SSD} secure zone.

Attack Scenario. To corrupt the secure file named foo.txt, the malware uses DiskShield system call (`DS_rdafwr`) to send a write command to DS-SSD. Malware tries to open the secure file, read it, encrypt it, and then write the encrypted data.

Evaluation Result. For evaluation, DS_{FS} creates a 4KB file. DS_{FS} sequentially performs sgx_fopen_auto_key, sgx_fwrite, and sgx_fclose (Figure 14(b)). Then, DS_{SSD} performs DS_OPEN, DS_CREATE, DS_WRITE, and DS_CLOSE operation in sequence (Figure 14(c)). The reason why DS_CREATE is performed after DS_OPEN is that the DS_{FS} sends the open request first, and then sends the create request again if the file does not exist. Here, the kernel messages of DS_rdafwr system call and SATA device driver are shown in Figure 14(a).

Figure 14(d), (e), and (f) show the messages generated by each component during malware intrusion. First, the malware uses the DiskShield system call to open the secure file, foo.txt. Although this command is transferred to DS_{SSD}, DS_{SSD} verifies MAC and denies opening the file (Figure 14(f)). This is because the malware cannot generate the right MAC without the correct file key. Therefore, since the file open request is denied, the malware fails to tamper with a secure file, foo.txt (Figure 14(d)).

A.1.2 File-based attack using pwrite() and LBA-based attack using ioctl().

Attack Scenario. To corrupt secure file, foo.txt, the malware opens the file and uses pwrite() to overwrite it. Also, the malware can open the device (DS-SSD) directly and use ioctl() to overwrite data in the SSD.

Evaluation Result. The attacks mentioned above access the persistent data by logical block address (LBA). However, the secure zone is isolated in the SSD. The secure zone is a hidden space that cannot be accessed by LBA. Therefore, malware cannot attempt to corrupt the secure zone.

A.2 Fresh Data Attack

A.2.1 System Call Hijacking.

Assumption. The attacker has invaded the host system and succeeded in gaining the root privilege. Also, the attacker avoided user detection and installed the rootkit as a kernel module. In this OS compromised environment, the attacker tries to hijack system calls when DS_{FS} performs file open, write, flush, and close to update the secure file (foo.txt).

Attack Scenario. The rootkit firstly reads the system call table from the kernel symbol table. Then it finds the position where the DiskShield system call (`DS_rdafwr`) is stored. It overwrites this position with the address of the rootkit module. Therefore, when DS_{FS} calls DS_rdafwr, the rootkit module is called instead. It encrypts the data buffer the host is trying to write. It calls the original DS_rdafwr to overwrite the secure zone.

Evaluation Result. For evaluation, DS_{FS} creates a 4KB secure file (foo.txt) as mentioned above (Figures 14(a), (b), and (c)). The attacker loads the rootkit into the kernel. The rootkit performs system call hijacking when installed (Figure 15(b)).

When DS_{FS} opens a secure file and calls DS_rdafwr for writing, the rootkit hijacks the system call and modifies the user data buffer (Figure 15(b)). However, DS_{SSD} verifies MAC and denies write requests (Figure 15(c)). On the other hand, DS_{FS} fails to perform fflush(), and returns an error message to the application and exits (Figure 15(a)). In summary, the malware not only fails to tamper with the data stored in the secure zone, but through the two-way communication, DS-FS immediately notices the failure and succeeds in returning an error message.

A.2.2 Malicious Device Driver.

Assumption. The attacker has invaded the host system and succeeded in gaining the root privilege. Also, the attacker avoided user detection and installed the rootkit as a kernel module. In this OS compromised environment, the attacker tries to hook the SATA device driver modules before DS_{FS} performs file open, write, flush, and close to update the secure file (foo.txt).

Attack Scenario. The rootkit hooks a function in the SATA driver that packs commands into the registerFIS (Section 5.2.2). When a secure file write command is received, the rootkit converts the command into the regular write command. It tries to invalidate the data protection by storing data in the regular zone instead of the secure zone of DS_{SSD}.

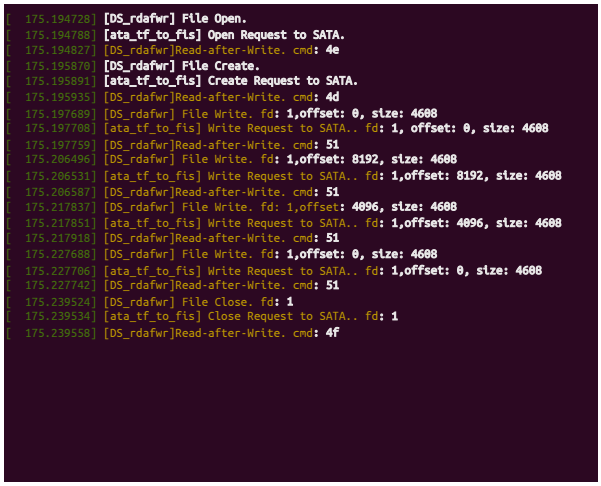
Evaluation Result. As mentioned above, DS-FS creates 4KB foo.txt (Figures 14(a), (b), and (c)).

The attacker loads the rootkit into the kernel. The rootkit installation goes through the following process.

First, The rootkit reads the address of the variable (ahci_ops) which holds a function pointer to the driver's registerFIS packing function (achi_qc_prep).

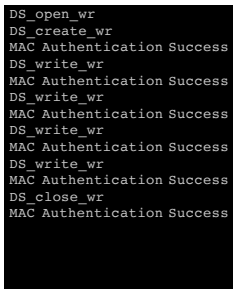
Second, The rootkit overwrites the address of the malicious rootkit module with the position where the ahci_qc_prep function pointer is loaded (ahck_ops->ac_prep).

Third, Figures 15(d), (e), and (f) show the messages generated by the three components. If DS_{FS} opens a secure file and attempts to write the secure file, the rootkit module hooked from the SATA driver replaces the secure command (DS_write) with a regular write command (write). In Figure 15(f), the message with the [sata_hooking] tag is a kernel message generated from a hooked rootkit module, not an existing driver module. As a result, DS_{SSD} accepts the regular write command and writes data to the regular zone(Figure 15(d)). Then it transmits a response to DS-FS in the host (Figure 15(e)).



(a) Secure File Generation: Kernel Messages

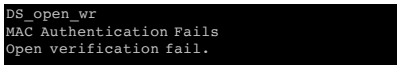
(b) Secure File Generation: DS_{FS} Messages



(c) Secure File Generation: DS_{SSD} Messages

(d) File Attack: Malware Application Messages

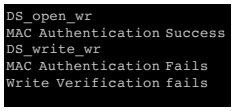
(e) File Attack: Kernel Messages



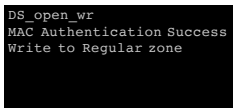
(f) File Attack: DS_{SSD} Messages

Figure 14: Messages from the Application, Kernel and Storage against File-based Attack using DiskShield System Call

(a) System Call Attack: DS_{FS} Messages

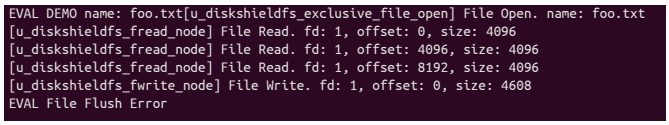


(c) System Call Attack: DS_{SSD} Messages



(d) Driver Attack: DS_{SSD} messages

(b) System Call Attack: Kernel Messages



(e) Driver Attack: DS_{FS} Messages

(f) Driver Attack: Kernel Messages

Figure 15: Messages from the Application, Kernel and Storage against System Call Hijacking and Malicious Device Driver Attack

However, since this response is wrong, DS_{FS} returns an error message to the application and exits. To sum up, even though the regular zone is overwritten with the wrong data, the data stored in the secure zone is kept safe. Also, through the secure two-way communication, DS_{FS} recognizes this immediately and succeeds in returning an error message.