JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 1-20 (2018)
DOI: 10.6688/JISE.2018.34.1.1

Analyzing the Storage Defects From the Perspective of Synthetic Fault Injection

BALGEUN YOO[1], SEONGJIN LEE[2] AND YOUJIP WON[1]
[1]*Department of Computer and Software*
Hanyang University
Seoul, 04763 Korea
[2]*Department of Aerospace and Software Engineering*
Gyeongsang National University
Gyeongsangnam-do, 52828 Korea
E-mail: [1]*{starhunter; yjwon}@hanyang.ac.kr;* [2]*insight@gnu.ac.kr*

In the era of digital media where users create multimedia, and the size and number of media files increase, the importance of robustness of storage devices is also increasing. Even a fault in storage may lead to not only damage of a file but also cause fatal errors that affect the system. To prevent the storage systems from such errors, it needs to be tested for physical errors and analyze their failure behaviors to take appropriate actions to prevent them. SOAR is an independent fault injection system with an SATA multiplexer to inject faults to not only storages used by desktop applications but also CE devices. SOAR runs real-time on top of Linux. SOAR injects both physical and logical errors on storage devices or device driver layer depending on the type of errors user wants to generate, respectively. Along with SOAR, we designed three test frameworks for multimedia players, file system benchmark tools, and file system. As the case study of the usage of proposed fault injection system, we tested and verified five multimedia players with a CE device, two file system benchmark tools, and four file systems

Keywords: fault injection, physical fault, logical fault, fault tolerance, fault robustness

1. INTRODUCTION

Physical failure of a storage system is inevitable consequence due to initial failure or accumulation of operation time. The digital information which is stored in the storage system has priceless value. So, enterprise system has tolerance for physical failure and recovery mechanism for data loss [1-4]. However, the cost of ownership increases for the personal systems to have the mechanisms of an enterprise system. Typically, enterprise systems rely on multiples of storage devices on RAID system to keep user data safe. Most of the private systems have single-storage. The current tools for a storage device in detecting and handling the faults are ECC and sector remapping. Single storage system needs following steps to get a strong tolerance for physical failure: (1) verify the tolerance level of a system for physical failure; (2) modify the system according to the verification results. To identify and understand potential failures, we use an experiment-based approach for studying the dependability of a system [5]. To take an experiment-based approach, we must first understand a system's architecture, structure, and behavior. Specifically, we need to know its tolerance for faults and failures, including its built-in detection and recovery mechanisms. We also need specific instruments and tools to inject fau-

Received June 18, 2016; revised September 17 & November 27, 2016; accepted February 17, 2017.
Communicated by Shiao-Li Tsao.

lts, create failures or errors, and monitor their effects. Moreover, these instruments and tools have to use the reliable method to verify for the reliability of the verification result.

Fault injection tools proposed in the field have three fundamental problems. First, the error injection is an offline process and cannot be done in real-time. Second, the errors created are either virtual or leaves permanent damage to the storage device. Third, they disregard the fact that application and the system behave differently to virtual errors and to physical errors. To address the shortcomings of existing work, we develop SOAR that allows analyzing the fault tolerance of a storage device. SOAR exploits SATA multiplexer to inject errors to a target device which is mounted to an appliance or a host system in real time without having to detach the target device from the system. SOAR provides recovery mechanism to restore the state of the storage before errors are introduced to the device. Thus, it does not leave permanent damage to the storage, yet it can inject logical errors in device driver level and physical errors on the hardware level. SOAR provides two modes of operation that is ECC CREATOR and ERROR CREATOR for injecting physical errors and logical errors, respectively. Physical errors alter the sectors in the storage which later processed as bad sectors. Logical errors, on the other hand, does not change the data in the storage but changes the error register in the device driver to return error codes to an application. ECC CREATOR is mainly used for testing the read failures and ERROR CREATOR is used for testing read and write failures. Since a storage device remaps a sector if it finds a bad sector, we cannot use ECC CREATOR for injecting write failures.

This paper is an extended version of the papers published in Journal of KIISE 2008 [6] and presented in the 2015 IEEE 5th International Conference on Consumer Electronics Berlin [7]. Kim *et al.* [6] described the fault injection software that runs on Linux Kernel v2.4, which injects physical/logical fault to a target sector address of an HDD. Yoo *et al.* [7] is a short version of this paper which omits much of the detailed information. This paper has four major improvements over earlier works. Firstly, with the help of SATA Multiplexor, we provide an ability to inject real-time errors on the target device while an application is using the target device. Secondly, this work implements a standalone system that can inject errors and recover the injected errors on consumer electronics devices that uses an HDD as a storage device. For example an external HDDs, a set-top box and multimedia player, whereas the earlier work [6] is limited to a storage device connected to the fault injection system. Thirdly, we revised the framework and descriptions about the fault injection techniques. We provide detailed information about the low-level interrupt data structure for PATA and SATA interfaces in Linux, disk sector layout for the WRITE LONG SATA command for ECC Creator, and how physical and logical errors are created using the new hardware. Finally, this work provides new cases studies on DVICO Set-top box and two new file systems (EXT4 and F2FS).

2. RELATED WORK

Failure of the storage system may be classified into a physical and a logical. Logical failure is a failure that occurs in the software part such as registry keys, file system corruption, and other critical data deletion. Physical failure is a failure that occurs in the mechanical part or electrical parts such as an actuator, and circuit board [8]. Logical failure can be resolved by using the file system which supports copy on write (COW) or

journaling. Physical failure cannot be addressed by the file system because the problems are in the storage. Enterprise system which uses multi-storage can recover the loss data with parity information when the physical failure occurs [9]. However, single storage does not have a solution to retrieve the lost data due to physical failure because they do not store additional information such as parity. When a physical failure occurs, single storage can only remap the failure area to spare area. Also, a single storage system may suffer a variety of problems such as read/write delay, and application freeze/termination [10]. Therefore, the single storage system must have a tolerance for physical fault. Applications must have the fault tolerance which accurately identify the type of failure and adopt the countermeasure for physical failure. For example, an application does not wait for a retry of storage to skip that area when a read failure occurs, or application will notify a kernel error message to the user when the application terminates due to the physical failure. Enterprise system has resolved the problems of physical failure with the use of multi-storage. Therefore, the target of this paper is limited to a single storage system. The first step to study the fault tolerance of storage is the development of a method which injects the various faults to intend address. The existing logical fault injection method has a limit of emulation. Also, the current physical fault injection method has a fatal disadvantage that it cannot inject a fault to intend address. Therefore, it needs to develop the technique which injects and recovery a physical fault to propose address.

Zhu *et al.* [11] suggested the replace recovery algorithm for fault tolerance of single storage. Existing recovery solution uses XOR-based erasure code which reads all of the data of information disk and needs to perform an XOR operation. Therefore, this method has the disadvantage that a colossal computational overhead. The replace recovery algorithm reduces the computational cost to minimize the amount of data read from the disk. It searches a set of parity symbol for recovery with the hill-climbing approach in each parity disk, compare the set of first parity disk and the set of another parity disk, and make up the set that requires a minimum read by replacing a parity symbol. Xiang *et al.* [12] were exploring that RDP (providing the availability of the two disk failure) [13] is poorly optimized and inadequate for single disk failure. Also, Xiang *et al.* proposed a Row-Diagonal Optimal Recovery (RDOR) which is optimized recovery method for single disk failure. If the failure occurs in the two information disk, RDP recovers each erasure symbol by combining the information symbols and row parity with the rest of the disk (except for the failure disk). When a failure occurs in the two parity disk, recovery process is the same as the encoding process of the parity. If the failure takes place in an information disk, RDP recovers the failure by only using single parity column. However, all the data blocks are protected by two parity groups. Therefore, RDP has to read the entire two parity disk for the recovery of each data block. RDOR reduces the number of the read operation and provide a load balancing for all the surviving disks by only read the parity which is required to the error disk. However, these algorithms are not the way to recover the disk failure with single storage but the way to increase the performance and efficiency of the recovery process in a multi-storage system. Therefore, it cannot be applied to this paper.

Fault Injection Tool proposed by Adriaens *et al.* [14] creates physical faults on IDE based storage devices. Although the behavior of faults is like that of real physical faults, they are virtual physical faults injected on device driver layer. A logical address is translated into a physical address in the device driver layer to mark artificially some of the

physical addresses are with faults. “Request” structure in the Kernel is the key data structure in this scheme. One advantage of Fault Injection Tool is that the original information in the storage device is intact, but a user can validate the behavior of the storage device when faults are injected. The faults on this scheme, however, are only virtual, and the fault information does not persist after a reboot of the system.

Other fault injection mechanism proposed by Arlat *et al.* [15] and Gunneflo *et al.* [16] are based on pin-level fault injection and generating heavy-ion on storage devices, respectively. Their methods are difficult to specify the exact location of the faults and cause fatal damage to the storage device.

Marinescu *et al.* [17] showed that the fault tolerance testing work of the storage is hard work that requires manual labor. Marinescu *et al.* proposed LFI which is an easy-to-use tool to inject the fault in the storage. LFI is a tool to inject a fault between shared libraries and application according to predefined scenarios. Also, it is not necessary to study the library of the source code and does not require a platform such as Linux or Windows. However, this tool can be injected an only logical fault, not physical fault. Juszczyk *et al.* [18] inject a fault and observed the reaction of the target in real-time. Juszczyk *et al.* inserted into the code for real-time monitoring on the target and the injection tool. SOAR is possible to inject physical and logical fault unlike the method of Marinescu *et al.*, and implemented a more accurate real-time fault injection method by using the hardware (multiplexer card) in contrast to the software method of Juszczyk.

Kim *et al.* [19] introduces a dedicated Flash Memory Virtual Platform Layer (FMVP) that emulates the errors on the NAND Flash memory. It consists of three layers, virtual platform layer, system software layer, and test environment layer. Fault injection and fault snapshot feature is provided in the test environment. They show that bit flips in the Flash memory may overwhelm the file system. However, it cannot be used in real world storage systems because it is an emulation based approach to analyze the errors on SSDs. SOAR, on the other hand, can inject errors to any PATA and SATA storage devices.

There are three significant differences between SOAR and prior error-injection methods. The first difference is that SOAR can inject real-time errors while an appliance or an application is in use without having to detach the target device from the host system. Secondly, SOAR allows injecting an error on any specific location of a device which can be specified by an LBA or set of LBAs. The injected errors are either a logical or a physical error. Thirdly, SOAR provides recovery mechanism to restore the errors on the device.

3. BACKGROUND: DEVICE DRIVER AND ECC AREA

The standard interfaces on a private computing system are PATA (Parallel ATA) or also known as IDE and SATA (Serial ATA) which is faster than PATA interface. Note that commands in two interfaces are compatible. In this paper, we only consider SATA and PATA interface for desktop systems and do not deal with SCSI/SAS/FC interface for enterprise systems.

Low-level interrupts for PATA and SATA interface in Linux is defined in Linux/drivers/block/hd.c (Kernel v3.19). Table 1 shows the definition of HD errors on hd.c. It describes the definition of errors, for example, bad address, it could not locate a track or ID field, and it has uncorrectable ECC error and CRC error during transfer. SOAR can

inject all errors defined in hd.c and verify the behavior of the storage device. SOAR can receive both block number and LBAs to specify locations of faults, and it also can generate read and write failures and interface errors.

Table 1. linux/drivers/block/hd.c (Linux Ver. 3.19).

/* Bits for HD_ERROR */			
#define	MARK_ERR	0x01	/*Bad address mark*/
#define	TRK0_ERR	0x02	/*couldn't find track 0*/
#define	ABRT_ERR	0x04	/*Command aborted*/
#define	MCR_ERR	0x08	/*media change request*/
#define	ID_ERR	0x10	/*ID field not found*/
#define	MC_ERR	0x20	/*media changed*/
#define	ECC_ERR	0x40	/*Uncorrectable ECC error*/
#define	BBD_ERR	0x80	/*block marked bad*/
#define	ICRC_ERR	0x80	/*new meaning: CRC error during transfer*/

We describe the physical sector layout of an HDD to understand how ECC CREATOR injects physical failures. The length of each field may differ depending on the manufacturers. HDDs have one or more platters, and one platter is divided into many concentric circles which form tracks. The track consists of sectors. A sector is the unit of storage, and each is assigned an address. HDDs read and write in units of sectors. In general, 512 byte is allocated to a sector [20]. Fig. 1 illustrates the physical layout of a sector. It consists of a preamble, SYNC, data, ECC information, *etc.* [21]. Binary strings called Preamble and Postamble is recorded at the beginning and the end of each sector, which is used for synchronization at forward and backward reading process, respectively. SYNC is used to locate the start of a data block in the sector which is different from Run Length Limited (RLL) pattern [22]. Data block stores RLL transformed user's data. CRCC and ECC correspond to error detection code and error correction code, respectively.

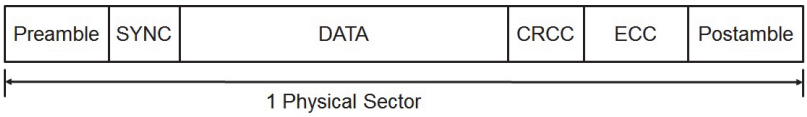


Fig. 1. Physical sector layout.

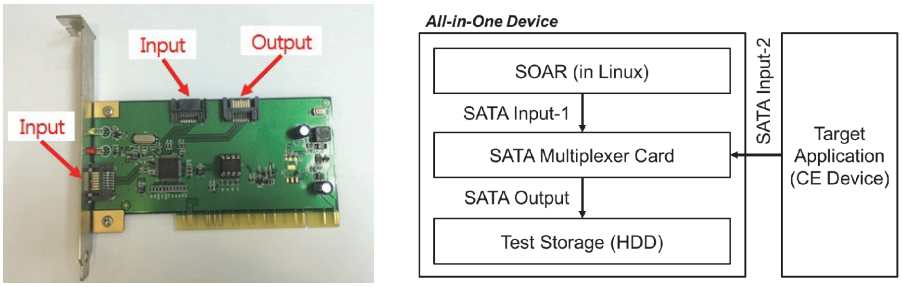
SOAR uses `WRITE LONG` SATA command to inject physical failures on the device [23]. As POSIX `write()` writes sector sized data, the data is recorded data block on a disk sector along with subsidiary information. Note that firmware of the device is responsible for calculating subsidiary information that goes along the data block in a sector. Unlike `write()`, `WRITE LONG` overwrites both data and ECC area of the sector. When `WRITE LONG` command is issued, the data block and ECC area on the sector is filled with `0xff`. `WRITE LONG` command can be used to generate MEDIUM ERROR [24] on specific LBAs. When the device calculates ECC using the data and compares it with the stored ECC in the sector, it returns fail because the two are different. The device marks the sector as a bad sector.

If Automatic Read Remapping (ARRE) or Automatic Write Remapping (AWRE) is configured in the device, the corresponding LBA is remapped to a spare sector, and the bad sector is added to the defect list (PLIST). ARRE and AWRE bits in disks can be modified through sdparm tool [25]. For example, the following command is used to remove AWRE and ARRE bit of a device until next boot of the device: `sdparm-c AWRE, ARRE/dev/sda`

4. SOAR: DESIGN OVERVIEW

4.1 Hardware Description

SOAR is composed of injection fault software, SATA multiplexer card (PMC PM8307 SPS 3GT) [26] and embedded system. Fig. 2 (a) shows the layout of SOAR, and Fig. 2 (b) shows the system scheme of SOAR. Fault injection software is a user application running on Linux, and it injects logical/physical faults on block addresses or LBAs. The multiplexer has two host ports and one target port which allows switching of SATA devices. The multiplexer has two host ports and one target port which allows switching of SATA devices. One of the SATA input port is used by the appliance or the host system and the other input port is used by SOAR. The two systems share a target device connected to the output port. The multiplexer translates SATA commands from either of systems to be processed in the target device; thus, it is possible for SOAR to inject errors to the target device in real-time while the host system to runs any applications on the target storage device. The multiplexer supports NCQ (Native Command Queuing) [27] command and non-queued (ATA, ATAPI-7) Command at the same time. Note that PCI slot supplies 1.8V to operate the device. We packaged the SOAR as the portable embedded system with display, SSD, and customized and optimized kernel for SOAR system for fast boot.



(a) SATA multiplexer card. (b) System scheme of SOAR.

Fig. 2. System scheme and SATA multiplexer card.

4.2 Fault Injection

SOAR has two error injecting modes: ECC and ERROR CREATOR. ECC CREATOR generates physical bad sectors which cause read to fail on specific block numbers or LBAs. ERROR CREATOR makes physical faults on specific block number or LBAs logically. Note that ERROR CREATOR can create not only read failures but also write

Table 2. Comparison between the ECC CREATOR and ERROR CREATOR.

Items and Functions	Function of ECC CREATOR	Function of ERROR CREATOR
Roles	make and recovery of physical bad sector	make and recovery of physical fault
Techniques	Use of SATA COMMAND	manipulating the error register value and construction of virtual environment
Applications	read error	read/write and interface errors

failures and interface errors. Table 2 compares the two modes. We use these two modes to verify robustness and reliability of a storage system.

Fig. 3 illustrates flow diagram of ECC CREATOR and ERROR CREATOR modes of SOAR. ECC CREATOR injects physical faults directly to the storage device exploiting ATA Commands. On the other hand, ERROR CREATOR is processed in the kernel, file system, and device driver to create physical faults in the storage.

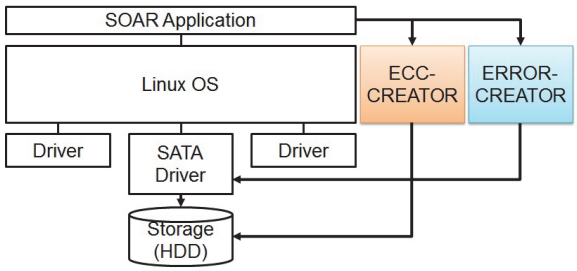


Fig. 3. Fault injection system flow.

4.3 Techniques to Address the Fault

One requirement of using ECC and ERROR CREATOR is ease of specifying error locations. SOAR has four different methods, General, Random, Fault Library, and Fault File, to locate faults, which is described in Table 3. General selection method lets a user choose the location to inject faults.

Table 3. Error location selection techniques.

Name	Explanation
General	Inject errors and recovery on the basis of the user's designation of location
Radom	Inject unstandardized errors (technique based on the random location selection)
Fault Library	Inject errors by using a file containing error information
Fault File	Inject errors at the block of file you want

Random selection method reads LBA ranges from the user and injects faults within the range. Generated fault on the field in Random selection process may affect only a sector or a set of consecutive sectors. In this work, we define a consecutive sector as the sector group. The total number of sectors in the fault potential range is given in Eq. (1), where the start and the end of the address are denoted as i and j, respectively. N_f the

number of faults within the given range. The number of sectors in the sector group and the number of sector groups are denoted as N_s and N_g.

$$S(i, j) = (1 \times (N_f - N_a)) + N_s \times N_a \quad (1)$$

Assuming that the parameters given for the Eq. (1) are i=100, j=7000, N_f=10, Ng=4 and Ns=3 for range start address, end address, faults within the range, number of sector groups, and sector group size, respectively. Ten faults are going to be injected within the LBA range from 100 to 7000; since three sector groups have the size of four, the total number of LBAs affected by the injection is 18. Fig. 4 illustrates the example.

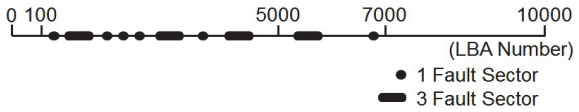


Fig. 4. Random positioning.

By default SOAR can take either block numbers or LBAs to set faults on the device. We use Fault Library method to let SOAR interchangeably use block numbers and LBAs as the location for faults. Fault Library method uses a file which describes the information in Table 4. The file can be utilized in both ECC CREATOR and ERROR CREATOR mode and the file describes list sectors and blocks with command or error types. For example, ECC CREATOR in Table 4 injects crash command to sectors 111, 222, and 333 and blocks 100, 200, 300 on /dev/hda1. The command can be set to recover the faults in specified blocks and sectors. ERROR CREATOR provides options to specify a particular device (*ex*. /dev/sda1), sector number, block number, and type of an error and fault. ERROR CREATOR uses physical error types defined in device driver area. Device driver specifies various error types and they are identified by a hexadecimal number as shown in Table 4. For example, the errortype:64 denotes Command aborted error (0x04). There are three fault types in ERROR CREATOR, which are read fault, write fault, and read/write fault.

Table 4. How to apply the fault library function.

Function	ECC CREATOR	ERROR CREATOR
File Processing Rule	device: `/dev/hda1` sector: 111,222,333 block: 100,200,300 command: crash	device: `/dev/hda1` sector: 111,222,333 block: 100,200,300 errortype: 64 action: 1 command: crash

Finally, Fault File-base selection method injects physical faults in a file. SOAR exploits IOCTL and RMAP of a file system to locate starting block number of a file. IOCTL is used to find the file descriptor of a given file, and RMAP returns the block number of giving file descriptor. Fig. 5 shows the flow.

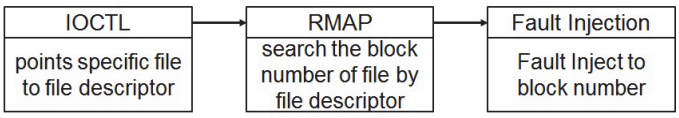


Fig. 5. File fault flow.

5. DETAILS OF FAULT INJECTION

In this paper, we provide two different error injection methods, ECC CREATOR and ERROR CREATOR. ECC CREATOR exploits WRITE LONG SATA command to generate a bad sector on the device. It is used to create a physical error. When an application tries to read the sector, device freezes because of the read failure. The Device driver is responsible for handling the read failure, for example, limiting the number of read retries can be one solution. When an application tries to write the sector, the device first remaps the bad sector to a new sector and writes user data to the sector successfully. ERROR CREATOR, on the other hand, exploits error registers in device driver. It can emulate both read and write failures without having to access the device. More importantly, the device driver returns an error code to the application; thus, the application is responsible for handling the read or write failures. The reason we have two different error injection modules in the paper is that the behavior for physical and logical errors are different and not all errors are reproducible with only one of the methods. This section describes the details of ECC CREATOR and ERROR CREATOR.

5.1 Physical Fault

ECC CREATOR, which is to verify the behavior of read failures, makes use of SATA Command `WRITE LONG` [28] to inject ECC errors to a target sector. The upper layer attempting to access the sector recognizes it as a bad sector on the disk. Thus, the device returns read failure when an application tries to read the sector. For a write attempt to the bad sector, the device tries to write on the sector for a predefined number of times and remaps the sector address to a spare area.

Since the target sector is overwritten with 0xff via `WRITE LONG` command, the original data is lost in ECC CREATOR mode. To recover the original data, ECC CREATOR mode offers an option to back-up the original data and recover after the test. Note that block number based injection has two disadvantages. First, although ECC CREATOR affects a sector, the sectors associated with the block number have to be backing-up. Second, SOAR exploits `write()` system call which makes the test device dependent to the file system. SOAR makes use of ATA Command `WRITE` to operate on sector level and avoids the two potential limits on block number based injection.

To verify that ECC CREATOR injected faults properly, SOAR uses S.M.A.R.T tool [29] and `read()` system call. S.M.A.R.T tool is an open source tool which can check bad sectors on the device. `read()` system call performs read on the target sector, which then leaves a kernel message upon failure. Since ECC CREATOR injects a bad sector on the device, the sector will be in the same state even after the reboot or attached to another system.

5.2 Logical Fault

ERROR CREATOR mode in SOAR allows injecting other types of physical errors but the original data intact. ERROR CREATOR mode creates virtual errors defined in the device driver. ERROR CREATOR uses following steps to inject an error. Upon receiving the location of an error type from the user (read, write, *etc.*), ERROR CREATOR modifies the error register on the device driver. When the host requests an access device driver checks whether the sector has a fault, and if the fault type matches with access type SOAR blocks the access to the storage and emulates the error. Adriaens *et al.* [30] also provide similar virtual error injection schemes. However, they have a coarse set of errors. For example, read and write operation on error-marked sector returns error messages regardless of their error types. SOAR can trigger all of the error types defined in the device driver, which are described in Table 1. ERROR CREATOR manipulates the error registers in the device driver to create physical faults on access sector types, *i.e.*, read/write/interface access. Although ERROR CREATOR provides pseudo physical errors, it provides a variety of errors regardless of their access types and physical fault types.

ERROR CREATOR keeps a list of injected errors. For read and interface errors, the list also records the recovery method for respective errors. In the case of the read error, ERROR CREATOR removes the error information from the list upon write operation on the target sector. Since interface errors last only temporarily, we trigger an error on the first attempt to access the target sector and remove the error for the next tries.

Since ERROR CREATOR injects pseudo physical errors on the device through manipulation of a device driver, it does not support the use of S.M.A.R.T tool. However, it is possible to verify the errors on the target sectors through kernel messages while accessing the sector through system calls.

6. EXPERIMENTAL SETUP

The host system for SOAR uses Intel Dual Core 2 2.9GHz CPU with 4GB main memory and runs Linux kernel v3.19. The fault injection target employed in the case study is SATA 3.0 HDD (WD4001FFSX) with 4TB of capacity. Although existing test frameworks can verify the robustness of storages, they are not capable of testing massive and high-density storage devices and file systems; they are focused on verification of logical faults and robustness of file systems. This work implements the framework to verify not only the logical faults but also physical faults of systems.

Fig. 6 illustrates the framework. SOAR is composed of three main components: Fault Positioning, Fault Injection, and Fault Verification. SOAR is capable of testing three targets systems, *i.e.*, multimedia devices/players, file system benchmark tool, and file system. SOAR injects faults and verify the robustness of system while application launch, write test, and running a workload.

Fig. 7 illustrates the conventional processes of SOAR for three target systems, *i.e.*, set-top box and multimedia players, file system benchmark tools, and file systems. After the selection of the target system, we find LBA of target system's storage device via read and write workload. Before injecting errors, we unmount and mount the storage device to ensure that caches are cleared. If we are to verify the behavior of read failure, we use ECC CREATOR to inject physical errors to the storage device; and to verify write fail-

ure, we use ERROR CREATOR mode which injects logical errors on the device driver. Finally, we run the initial workload on the storage device and monitor kernel messages or notifications from applications.

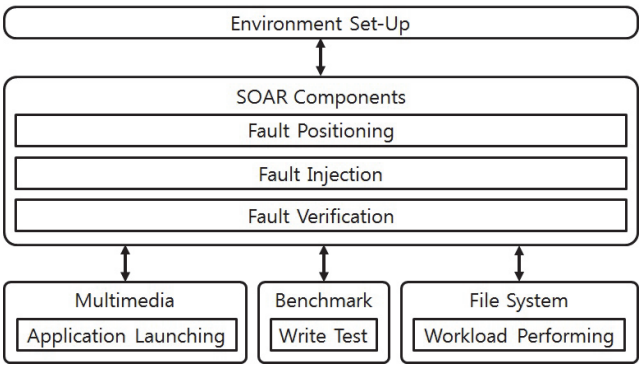


Fig. 6. Test framework component.

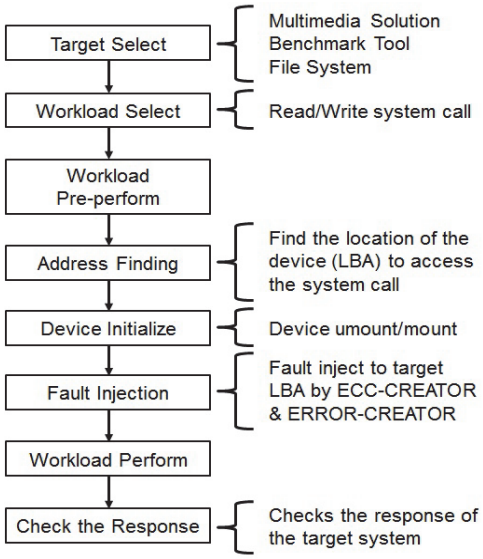


Fig. 7. Basic test flow of SOAR.

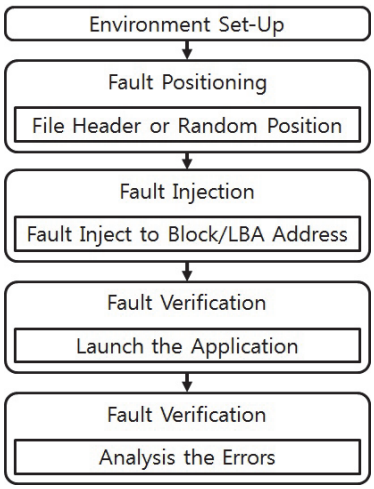


Fig. 8. Multimedia player test framework.

7. TEST FRAMEWORK AND CASE STUDY

7.1 Multimedia Player

Video players and mp3 players are probably the most widely used multimedia players. Since multimedia files vary in size, a small number of bad sectors in them may not be a significant problem; however, when bad sectors are placed in the header of the files or number of I-frames on the video files, the players may have difficulty in playing the

file correctly. On the other hand, MP3 files which tend to be a lot smaller than video files may suffer from only a few bad sectors.

Concerning characteristics of the multimedia files, SOAR uses test framework for the multimedia files which is shown in Fig. 8. SOAR uses FAULT FILE location selection method to specify the test file and to determine the position of faults. Then, ECC CREATOR or ERROR CREATOR mode is used to inject physical or other types of errors. In general, headers of a multimedia file are placed at the beginning of the file; we use 0 block number as the position to inject error, and use arbitrary block number within the file to inject faults. Then, we use a multimedia application to run the files and analyze the errors and notification messages while playing the files.

In the multimedia and MP3 players, we used ECC CREATOR mode to inject physical faults and monitored the behavior and error messages of applications after introducing the faults to the data. We chose to use ECC CREATOR (physical error) over ERROR CREATOR (logical error) in multimedia and MP3 players to show how different multimedia players behave to physical errors that do not return error codes. Table 5 lists the error messages and the occasions.

Table 5. Screen result and error message with read error.

Screen Result		Error Message	
Sign	Result	Sing	Result
A	Retry	a	Correct message
B	Skip the read failure	b	Incorrect message
C	Freezing	c	No message
D	Application Termination		
E	Application Crash		

MP4 encoded video file is used in the experiment, which is composed of Group of Pictures (GOP) structures [31]. Suppose a GOP consists of fifteen frames, there are one I-frame, five P-frames, and nine B-frames. I-frames are encoded independently of other frames. P-Frames makes use of the information from the latest preceding I or P frame, and B frames exploit information from the latest preceding or successive I or P frame. The most important frame in a GOP is I frame. In our experiment, the max, the average, the min size of GOP is 100 Kbyte, 16 Kbyte, and 900 bytes, respectively. Note that SOAR injects faults in units of a sector (512 bytes), applications read in units of a block which is 4 Kbyte. It means that a fault on a random sector may affect about four frames of a GOP on the average, which blocks the application from playing the file on that location.

For the case study of the multimedia environment, we use three video player (AVIfile, MTV, and Mplayer), two MP3 player (RealPlayer and Freeamp) and Divico Tvix Player PVR M-6510A, which is a Linux-based multimedia set-top box with a single SATA HDD.

Table 6 shows the experiment results. A proper error handling mechanism for the multimedia players is to skip the byte with a bad sector and continue to play the file. Most of the applications failed to operate properly. AVIfile and MTV froze when reading the bad sector, and they did not give any error messages. Mplayer, however, after some retries to read the bad sector, it skipped the block which has a fault and displayed an error message that it is retrying to read the data.

Table 6. Error message of multimedia player.

Target	Screen	Error Message	Header
AVIfile	C	c	a
MTV	C	c	b
Mplayer	A/B	a	a
RealPlayer	D	b	b
Freeamp	D	c	b
Tvix	C	c	c

When bad sectors are injected to the header of the metadata file, only AVIfile and Mplayer showed the correct error message that it could not open the media file because the header is corrupted. MTV, on the other hand, displayed just a warning message.

RealPlayer and Freeamp, the MP3 player, terminates when reading the bad sector in the media file. Bad sector in the header forces RealPlayer to quit, but Freeamp gives the pop-up error message. In either case of termination, RealPlayer displayed "Bus error". On the other hand, Freeamp did not give any error message when a bad sector is in the data area, but it shows an error notification for a bad sector in the header case. Both MP3 players failed to operate properly when the file contains a bad sector.

Error handling routine of the Tvix is not also ideal. When a bad sector is injected into the data area of the media file, Tvix tries to read the data for about a minute and skips the block with a fault. Before it skips to a readable region, Tvix freezes. When there are five or more consecutive bad sectors in the file, Tvix freezes for 15 seconds and jumps to next file. In either case, Tvix does not show any error messages. In either case, Tvix does not show any error messages. Tvix behaves differently to logical and physical errors. When we inject logical errors (read failure) to the Tvix while playing a media, it stops playing and jumps back to default menu. When a bad sector is injected on the header of the media file, Tvix freezes for 15 seconds and jumps to next file failing to show any error messages.

7.2 File System

File system metadata describes various information about the user data. As a means to test the integrity of file system, SOAR uses a framework illustrated in Fig. 9 which injects physical and logical errors on the file system to test its behavior in read and write failures.

Before we begin the test, we first unmount and format the target storage device and mount the device for both read and write test. Read fail occurs when the file system is unable to read the metadata of files and directories or metadata of file system itself. In the file system test framework, we first create some directories and files and use ECC CREATOR or ERROR CREATOR mode to inject physical or logical errors on the file system metadata, respectively. After the injection, we check for following items: (i) Files and directories exist after the test, (ii) Partial loss in the files, (iii) Data in a file are corrupted, and (iv) The file system can be mounted properly. To verify the state of the file system, we first save the contents of target location then inject a fault in the location. If the content of the target location is different from the saved content, then we mark it as damaged. Next verification method in the experiment utilizes *l*s command that lists the

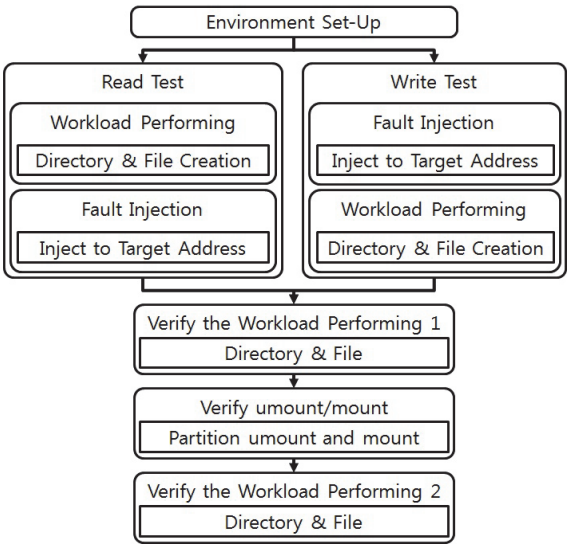


Fig. 9. File system test framework.

files and directories in a file system. We check for the number of files or directories missing in the file system as the result of injecting a fault in the storage.

We test read and write robustness test with well-known Linux file systems Ext3 [32], Ext4 [33], and JFS [34]. We also test recently released Flash friendly files system F2FS [35]. We use SOAR to inject faults in data structures and metadata of core file system and analyze their behaviors. After faults are introduced faults to the file system, the results are categorized as follows: Data Damage (DD) – diff of before and after of a file system block is not null; Data Loss (DL) – Reading a block returns block read fail; Object Loss, (OL) – Created file or directory disappeared; Mount Fail (MF) – Failed to mount the file system; Umount Fail (UF) – Failed to unmount the file system; and, System Crash (SC) – System crashed.

7.2.1 Ext3 and Ext4

Table 7 shows the result of robustness test on Ext3 file system. It shows that mount of the file system fails when physical faults are injected to superblock, group descriptor, and inode table. When read failure occurs on any metadata stored in a journal area, the file system fails to mount. Introducing physical faults on another file system succeeds on mounting the file system, however, the user data is damaged. It shows that Ext3 file system is prone to physical faults. On the other hand, when to write fails due to physical errors, file system succeeds on mounting and unmounting without giving any error messages. There was no system crash during the experiment.

Table 8 illustrates the result of running SOAR on Ext4 file system. The results of Ext4 are analogous to that of Ext3 except for few cases. For example, when physical faults are injected to inode table, Ext4 does not fail in mount and unmounting operation, whereas Ext3 does fail. Another notable difference is that files do get lost on Ext4 file system. When logical errors are introduced to the file system, Ext4 shows fewer errors for write tests than Ext3.

Table 7. Results regarding read and write failure in Ext3 file system.

File System	Data Structure	State of file system					
		DD	DL	OL	MF	UF	SC
		Read Failure					
Ext3 (Data)	superblock	×	○	–	○	–	–
	group descriptor	×	○	–	○	–	–
	block bitmap	×	○	–	–	○	–
	inode bitmap	×	○	○	–	–	–
	inode table	×	○	–	○	○	–
Ext3 (Journal)	super block	×	×	×	○	×	–
	descriptor block	×	×	×	○	×	–
	data block	×	×	×	○	×	–
	commit block	×	×	×	○	×	–
		Write Failure					
Ext3 (Data)	superblock	–	–	–	–	○	–
	group descriptor	–	–	–	–	○	–
	block bitmap	○	○	–	–	–	–
	inode bitmap	–	–	○	–	–	–
	inode table	×	–	○	–	–	–
Ext3 (Journal)	super block	○	–	○	–	–	–
	descriptor block	○	○	○	–	–	–
	data block	○	○	○	–	○	–
	commit block	○	○	○	–	–	–

Notes: (○: Possible to occur, –: No occurrence, ×: Impossible to verify)

Table 8. Results regarding read and write failure in Ext4 file system.

File System	Data Structure	State of file system					
		DD	DL	OL	MF	UF	SC
		Read Failure					
Ext4 (Data)	superblock	×	○	–	○	–	–
	group descriptor	×	○	–	○	–	–
	block bitmap	×	○	–	–	○	–
	inode bitmap	×	○	–	–	–	–
	inode table	×	○	○	–	–	–
Ext4 (Journal)	super block	×	×	×	○	×	–
	descriptor block	×	×	×	–	×	–
	data block	×	×	×	–	×	–
	commit block	×	×	×	–	×	–
		Write Failure					
Ext4 (Data)	superblock	–	–	–	○	–	–
	group descriptor	–	–	–	○	–	–
	block bitmap	○	–	–	○	–	–
	inode bitmap	–	–	–	–	–	–
	inode table	×	×	–	–	–	–
Ext4 (Journal)	super block	○	–	–	○	–	–
	descriptor block	○	–	–	–	–	–
	data block	○	–	–	–	–	–
	commit block	○	–	–	–	–	–

Notes: (○: Possible to occur, –: No occurrence, ×: Impossible to verify)

When sparse_super flag is set, the copy of superblock is in the group 0 and groups in powers of 3, 5, and 7. When the flag is not set, then the copy of the superblock is copied on all groups. However, the result of our experiment on injecting faults on superblock and group descriptor of block group 0 shows that Ext3 and Ext4 did not make use of the duplicated superblocks to recover from the errors. For Ext3 and Ext4 to have better tolerance against faults, it needs to identify the error and make use of the copy of superblocks in other groups.

7.2.2 JFS

Table 9 shows the result of JFS file system, which is very different from that of Ext3 and Ext4 file system. What is different is that JFS recovers errors by overwriting the failed data with the backup copy when there is a physical error in the device and unable to read from the storage. JFS fails to mount the file system when original and backup copy contains error and fails to read from the device; it shows that JFS runs thorough check-ups on a partition and its metadata.

However, when there is write fails in journal related metadata, JFS exhibits all errors other than system crashes, which shows that JFS does not consider failures in the journal area.

Table 9. Results regarding read and write failure in JFS file system.

File System	Data Structure	State of file system					
		DD	DL	OL	MF	UF	SC
Read Failure							
JFS (Meta)	super block	–	–	–	–	–	○
	control page	–	–	–	–	–	○
	inode allocation map	–	–	–	–	–	○
	aggregate inode table	–	–	–	–	–	○
	block allocation map	–	–	–	–	–	○
JFS (Journal)	journal super block	×	×	×	×	×	○
	journal data block	×	×	×	×	×	○
Write Failure							
JFS (Meta)	super block	–	–	–	○	–	○
	control page	–	–	–	○	–	–
	inode allocation map	–	○	○	○	–	–
	aggregate inode table	–	○	○	○	–	–
	block allocation map	○	○	○	○	–	–
JFS (Journal)	journal super block	×	×	×	×	○	○
	journal data block	○	○	○	○	○	–

Notes: (○: Possible to occur, –: No occurrence, ×: Impossible to verify)

7.2.3 F2FS

Table 10 illustrates the result of running SOAR on various data structures of F2FS. Of all file systems tested in this paper, F2FS shows the most resilience to the device failures. Among five metadata areas in the F2FS, namely superblock, checkpoint, Node Address Table (NAT), Segment Summary Area (SSA), and Segment Information Table (SIT), only checkpoint area failed to mount when there are physical and logical faults in

Table 10. Results regarding read and write failure in F2FS file system.

File system	Data Structure	State of file system					
		DD	DL	OL	MF	UF	SC
Read Failure							
F2FS	check point area	–	–	–	○	–	–
	segment info table	–	–	–	–	–	–
	node address table	–	–	–	?	–	–
	segment summary area	–	–	–	?	–	–
	main area	–	–	–	–	–	–
Write Failure							
F2FS	check point area	–	X	X	○	X	X
	segment info table	–	–	–	–	–	–
	node address table	–	–	–	?	–	–
	segment summary area	–	–	–	?	–	–
	main area	–	–	–	–	–	–

Notes: (○: Possible to occur, –: No occurrence, ×: Impossible to verify)

the metadata. Note that physical and logical failures in NAT and SSA caused F2FS to freeze for dozens of seconds before successfully mounting the file system. F2FS keeps a shadow copy of important file system metadata that is checkpoint, node address table, and segment information table. With the shadow copied metadata, F2FS recovers the errors introduced to the file system.

8. CONCLUSIONS

Existing fault injection and verification tools can only inject logical errors to the storage system which does not persist upon reboot of the system and unable to create physical errors. In this paper, we introduced fault injection and verification tool, SOAR, for a storage device to test the robustness of the system against read and write errors. SOAR can either inject physical errors on the storage device or create logical errors in the device driver layer. SOAR exploits an SATA multiplexer card to inject errors in real-time. We designed three test frameworks to analyze the integrity and error handling capabilities of multimedia applications, file system benchmarks, and file systems. We tested five multimedia players including a set-top box, two file system benchmark tools, and four file systems using SOAR.

ACKNOWLEDGEMENT

This research was supported by Basic Research Lab Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (No. 2017R1A4A1015498). This work was supported by the fund of research promotion program, Gyeongsang National University, 2017. Seongjin Lee (insight@gnu.ac.kr) is the corresponding author.

REFERENCES

1. L. Spainhower and T. A. Gregg, "IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective," *IBM Journal of Research and Development*, Vol. 43, 1999, pp. 863-873.
2. S. Mejjaouli and R. F. Babiceanu, "Holonic condition monitoring and fault-recovery system for sustainable manufacturing enterprises," *Service Orientation in Holonic and Multi-Agent Manufacturing and Robotics*, Springer, 2014.
3. W. Cox and T. Considine, "Grid fault recovery and resilience: Applying structured energy and microgrids," in *Proceedings of IEEE Innovative Smart Grid Technologies Conference*, 2014, pp. 1-5.
4. A. Rajimwale and W. Hsu, "Large scale data storage system with fault tolerance," US Patent, 2014, 8,713,282.
5. M. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *IEEE Computer*, Vol. 30, 1997, pp. 75-82.
6. K. Youngjin, W. Youjip, and K. Rakie, "SOAR: Storage reliability analyzer," *Computer Systems and Theory*, Vol. 35, 2008, pp. 248-262.
7. Y. Balgeun and W. Youjip, "SOAR: Storage based system reliability analyzer," in *Proceedings of IEEE International Conference on Consumer Electronics*, 2015, pp. 340-344.
8. J. Elerath, "Hard disk drives: The good, the bad and the ugly!" *ACM Queue*, Vol. 5, 2007, pp. 28-37.
9. D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *ACM SIGMOD Record*, Vol. 17, 1988, pp. 109-116.
10. H. Huang and K. Shin, "Partial disk failures: Using software to analyze physical damage," in *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, 2007, pp. 185-198.
11. Y. Zhu, P. Lee, Y. Hu, L. Xiang, and Y. Xu, "On the speedup of single-disk failure recovery in XOR-coded storage systems: theory and practice," in *Proceedings of the 28th IEEE Symposium on Mass Storage Systems and Technologies*, 2012, pp. 1-12.
12. L. Xiang, Y. Xu, J. Lui, and Q. Chang, "Optimal recovery of single disk failure in RDP code storage systems," *ACM SIGMETRICS Performance Evaluation Review*, Vol. 38, 2010, pp. 119-130.
13. P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *Proceedings of the 3rd Usenix Conference on File and Storage Technologies*, 2004, pp. 1-14.
14. J. Adriaens and D. Gibson, "A software layer for IDE disk fault injection," in *Proceedings of System Lacking Originality Workshop*, 2005.
15. J. Arlat, Y. Crouzet, and J. Laprie, "Fault injection for dependability validation of fault-tolerant computing systems," in *Proceedings of the 19th IEEE International Symposium on Fault-Tolerant Computing*, 1989, pp. 348-355.
16. U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *Proceedings of the 19th IEEE International Symposium on Fault-Tolerant Computing*, 1989, pp. 340-347.

17. P. Marinescu and G. Candea, "LFI: A practical and general library-level fault injector," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*, 2009, pp. 379-388.
18. L. Juszczyk and S. Dustdar, "Programmable fault injection testbeds for complex SOA," in *Proceedings of the 8th International Conference on Service-Oriented Computing*, 2010, pp. 411-425.
19. S. K. Kim, J. Choi, D. Lee, S. H. Noh, and S. L. Min, "Virtual framework for testing the reliability of system software on embedded systems," in *Proceedings of ACM Symposium on Applied Computing*, 2007, pp. 1192-1196.
20. J. Gim and Y. Won, "Extract and infer quickly: Obtaining sector geometry of modern hard disk drives," *ACM Transactions on Storage*, Vol. 6, 2010, pp. 1-26.
21. T. Saito, T. Yamaguchi, A. Kanamaru, and W. Ichihara, "Rotating disk storage device and recording method," US Patent, 2007, 11/652,387.
22. P. Siegel, "Recording codes for digital magnetic storage," *IEEE Transactions on Magnetics*, Vol. 21, 1985, pp. 1344-1349.
23. P. Gutmann, "Secure deletion of data from magnetic and solid-state memory," in *Proceedings of the 6th USENIX Security Symposium*, Vol. 14, 1996, pp. 77-89.
24. L. Bairavasundaram, G. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," *ACM SIGMETRICS Performance Evaluation Review*, Vol. 35, 2007, pp. 289-300.
25. J. Qian, C. Meyers, and A. Wang, "A Linux implementation validation of track-aligned extents and track-aligned RAIDs," in *Proceedings of the USENIX Annual Technical Conference*, 2008, pp. 261-266.
26. I. Davies, "System and method for sharing SATA drives in active-active RAID controller system," US Patent, 2009, 7,536,508.
27. B. Dees, "Native command queuing-advanced performance in desktop storage," *IEEE Potentials*, Vol. 24, 2005, pp. 4-7.
28. K. Grimsrud and H. Smith, *Serial ATA Storage Architecture and Applications: Designing High-Performance*, Cost-Effective I/O Solutions, Intel Press, 2003.
29. H. W. Gottinger, "SMARTD: An intelligent decision support tool for strategic management on technology diffusion," *Strategische Planung*, Vol. 1, 1985, pp. 261-275.
30. G. Le and J. Didier, "The MPEG video compression algorithm," *Signal Processing: Image Communication*, Vol. 4, 1992, pp. 129-140.
31. S. Tweedie, "Ext3, journaling filesystem," in *Proceedings of Ottawa Linux Symposium*, 2000, pp. 24-29.
32. M. Cao, S. Bhattacharya, and T. Tso, "Ext4: The next generation of ext2/3 filesystem," in *Proceedings of Linux Storage and Filesystem Workshop*, 2007, pp. 1-37.
33. S. Best, "Journaled File System (JFS) for Linux" *O'Reilly Open Source Convention*, *Sand Diego*, 2002, pp. 1-39.
34. C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2015, pp. 273-286.

Balgeun Yoo received the BS degree in Computer Engineering from Dankook University in 2008. He received his Ph.D. in Computer Engineering in 2016 from Hanyang University. His research interests include operating systems. He is currently with LG Electronics.

Seongjin Lee (李聖眞) is an Assistant Professor at Department of Aerospace and Software Engineering, Engineering Research Institute, Gyeongsang National University, South Gyeongsang Province, Korea. He did his BS and MS in Department of Electronics and Computer Engineering, Hanyang University, Seoul Korea in 2006 and 2008, respectively. He received his Ph.D. in Computer Engineering in the same university in 2015. His research interests include system performance, measurements, analysis, characterization, and classification.

Youjip Won (元裕集) is a Professor at Department of Computer Science, Hanyang University, Seoul Korea. He is leading Embedded Software System Lab. He did his BS and MS in Department of Computer Science, Seoul National University, Seoul, Korea in 1990 and 1992, respectively. He received his Ph.D. in Computer Science from University of Minnesota in 1997. Before joining Hanyang University in 1999, he worked at Intel Corp. as Server Performance Analyst. His research interests include network traffic modeling, analysis and characterization, multimedia system and networking, file and storage subsystem, lower power storage system. In 2006, Multimedia File System Project funded by Samsung Electronics was awarded “Best Academy-Industry Collaboration Practice in Samsung Electronics”. In 2007, he was awarded “National Research Lab” grant which is highly selective and prestigious governmental grant.