Fault Injection based Failure Analysis of CentOS, Anolis OS and OpenEuler

Hao Xu  
Tongji University  
2131483@tongji.edu.cn

Yuxi Hu  
Alibaba Inc.  
huyuxi.pt@alibaba-inc.com

Bolong Tan  
Alibaba Inc.  
bolong.tbl@alibaba-inc.com

Xiaohai Shi  
Alibaba Inc.  
xiaohai.sxh@alibaba-inc.com

Zhangjun Lu  
Tongji University  
2131482@tongji.edu.cn

Wei Zhang  
Tongji University  
1910134@tongji.edu.cn

Jianhui Jiang*  
Tongji University  
jhjiang@tongji.edu.cn

Abstract—The reliability of operating system (OS) has always been a major concern in the academia and industry. This paper studies how to perform OS failure analysis by fault injection based on the fault mode library. Firstly, we use the fault mode generation method based on Linux abstract hierarchy structure analysis to systematically define the Linux-like fault modes, construct a Linux fault mode library and develop a fault injection tool based on the fault mode library (FIFML). Then, fault injection experiments are carried out on three commercial Linux distributions, CentOS, Anolis OS and openEuler, to identify their reliability problems and give improvement suggestions. We also use the virtual file systems of these three OSs as experimental objects, to perform fault injection at levels of Light and Normal, measure the performance of 13 common file operations before and after fault injection.

Index Terms—CentOS, Anolis OS, openEuler, failure analysis, fault injection

I. INTRODUCTION

Failure of any component of a software system generally results in system failure. Ideally, the operating system (OS) should be able to ensure a good quality of experience in presence of faults, such as isolating a failed component without affecting the reliability and responsiveness of the system.

Reliability is one of the important attributes of product or system quality. Failure analysis can find out the causes of system reliability problems, which provides a target for the application of reliability technology. Failure analysis increases the system reliability by judging the system failure modes, finding the failure causes and mechanism and proposing the measures to prevent the re-failure.

Fault injection techniques are designed to make a system fail in a controlled environment so that developers can study the chain of events leading to system failure and improve the system to prevent similar failures or even damage. Unfortunately, due to the increasing complexity and cost of the systems, developers rarely put fault injection techniques into practice, especially when they are difficult to determine which faults (defects) or errors to inject. Failure analysis based on fault injection is one of the effective means to analyze the system reliability. For example, Rakshith Amarnath [1] and Stefan Winter [2] et al. inject faults into CPU registers and drivers to evaluate system reliability. Takeshi Yoshimura et al. [3] inject faults into application processes to study whether errors would propagate to the kernel. Jean Arlat [4] et al. analyze the representativeness of OS fault injection and take Linux as an example to compare the similarities and differences between real faults and injected faults. In the development and application of fault injection tools, since the 1990s, academia has published many achievements, such as fault injector in FTAPE [18], fast fault injection FSFI [21], part of the work is OS oriented fault injection, such as SockPFI [19], TFI [20]. Most of these works have problems in the application of analysis methods or coverage. For example, in the combination method of TFI and fault diagnosis for distributed systems, fault injection part has the shortcomings of too coarse granularity, inaccuracy and low coverage [20]. The tools developed by industry for OS fault injection include FailViz [26], DICE [27], ZOFI [5] and so on. However, ZOFI has some problems, such as inaccurate fault injection, narrow application scope and only supports fault coverage analysis [5].

The purpose of this paper is to provide OS developers with a more effective definition, execution and analysis method and a failure analysis tool based on fault injection and use it to perform fault injection experiments for CentOS 8, Alibaba’s Anolis OS and Huawei’s openEuler. Suggestions for improvement of reliability are also provided based on the analysis results.

The contributions of our work are as follows: 1) Construct a Linux fault mode library (covering kernel and system calls, glibc/basetools, system services) by using the fault mode generation method based on the Linux abstract hierarchy. And provide a systematic, lightweight guidelines, which is applicable to large software systems. 2) Develop a fault injection tool based on the fault mode library (FIFML). 3) Use FIFML to perform fault injection experiments on CentOS 8, Anolis OS and openEuler distributions, give suggestions to improve their reliability. The virtual file systems of these three OSs are also used as the objects to perform fault injection at levels of Light and Normal, measure the performance of 13 common file operations before and after fault injection.

The rest of this paper is organized as follows: Section II...
presents the methods for fault mode generation based on Linux abstract hierarchy structure analysis (AHSA), construction of the Linux fault mode library and failure analysis of Linux based on fault injection. Section III gives the principle and structure of FIFML. Section IV provides the design and analysis of the fault injection experiments for three OSs, suggestions for improvement of reliability and the impact of the virtual file system faults of three OSs on the performance of 13 common file operations at levels of Light and Normal. Section V concludes the paper and discusses the limitations of our work and future work.

II. Fault Mode Generation Method Based on Linux AHSA and Failure Analysis Based on Fault Injection

A. Failure Analysis Process and Methods

For a computer that uses Linux as the OS, Linux can generally be divided into abstract levels as kernel functions, system calls, libs and system services. The Linux kernel is mainly used for abstraction and access scheduling of hardware resources. The number of Linux kernel functions is large and many kernel functions are complex. The high complexity of Linux makes it difficult to generate fault modes. Different functional modules may behave differently after a fault occurs. It depends on many factors, such as the resources used, the number and types of functions provided and interactions with other parts. For example, functions that need to create an object may be prone to problems such as running out of space and incorrect addresses.

The occurrence of the faults, errors, or failures of runtime Linux are usually caused by code defects being activated, hardware errors or Linux application errors. If an error produced by a code bug propagates to the system call layer, it will cause an error when it interacts with the Linux applications, then causes the Linux to fail. Performing failure analysis directly on Linux code is time-consuming and laborious. We have used software orthogonal defect classification (ODC) and software-implemented fault injection method to perform fault analysis on the Linux 2.6.18 kernel code [6]. The experimental results show that 59.66% of the kernel code increments are related to the Linux system call chain. Linux system call related failures can cover nearly 60% of Linux kernel code failures. So it would be more efficient to generate failure modes at the system call level. Here, we adopt a fault mode generation method based on Linux AHSA to generate Linux fault modes and construct a Linux fault mode library to support Linux failure analysis based on fault injection. The complete failure analysis process is shown in Fig. 1.

Step 1: Analyze the Linux abstract hierarchy structure.

The five main modules of Linux (i.e. file system, interrupt management, IO management, memory management, process management) and the key functions of each module (such as set memory permissions, send signals to processes, open files, manage semaphore sets, send messages, etc.) are identified. Relevant information can be obtained from the analysis of Linux source code and system calls. It does not need to fully understand the internal details of the module, but need to master the interface and parameter information.

Step 2: Determine the Linux failure modes.

Failure modes will be used in Linux failure analysis to help find the direct cause of these failure modes. The types of failure modes considered are defined according to the relevant literature [14-17] and our previous work on modeling the effects of software failures [8-11], including:

1) Kernel function failure: The return value of a system call is wrong, it is unable to provide correct service.
2) Delay: The long delay makes it unable to provide services within the user’s waiting time.
3) Buffer data error: The buffer data is wrong, it cannot provide the correct service.
4) System downtime and restart: The system breaks down or restarts, it cannot provide services.
5) Kernel Denial of Service: The system kernel refuses to provide services to users.
6) System CPU usage increases: The increase in CPU occupancy affects the services provided to users.

Based on past experience with fault injection and combined with the analysis of Linux functional issues listed below, the Linux failure modes can be identified. Although this process may not get to the root cause of the failure, it is more comprehensive, convenient and effective. The issue list includes:

1) Does the function return an abnormal result due to invalid kernel function parameters being called? If yes, then one or more failure modes corresponding to "kernel function failure” are generated.
2) Does the function return an abnormal result due to insufficient permissions of the called kernel function? If yes, then one or more failure modes corresponding to “kernel function failure” are generated.
3) Does the function return an abnormal result due to the wrong address of the called kernel function? If yes, then one or more failure modes corresponding to “kernel function failure” are generated.
4) Does the function return an abnormal result due to the value range of the called kernel function exceeds the
specified data type? If yes, then one or more failure modes corresponding to “kernel function failure” are generated.

5) Is the function unable to provide services within the specified time due to delay? If yes, one or more failure modes corresponding to “delay failure” are generated.

6) Is there no response for a long time during the execution of this function? If yes, then one or more failure modes corresponding to “delay failure” are generated.

7) Does the function still give wrong results due to external data errors? If yes, then one or more failure modes corresponding to “buffer data error” are generated.

8) Does the function give an error result of unknown type? If yes, then one or more failure modes corresponding to “buffer data error” are generated.

9) Does the function cause data synchronization error due to unexpected system shutdown and restart? If yes, then one or more failure modes corresponding to “system shutdown and restart” are generated.

10) Does the function cause abnormal data errors due to unexpected system shutdown and restart? If yes, then one or more failure modes corresponding to “system shutdown and restart” are generated.

11) Does the function cause an exception due to the process being killed? If yes, then one or more failure modes corresponding to “kernel denial of service” are generated.

12) Does the function affect the service due to the system overload? If yes, then one or more failure modes corresponding to “system CPU usage failure” are generated.

Step 3: Construct the Linux Fault Mode Library.

According to the list of failure modes, we obtain the corresponding fault modes by analyzing related functions. It is necessary to consider the system calls and related parameters used by the functions. The fault modes are generated based on system call characteristics and they are added into the fault mode library. Repeat the above process until no new fault mode is generated. Thus, the construction of the fault mode library is completed.

Step 4: Perform fault injection and failure analysis.

The fault injection tool is used to inject faults into the OS. After a fault is injected into the OS, it is necessary to track the propagation process of errors generated by the injected fault, monitor the behavior of the OS and analyze the failure caused by the injected fault. The fault injection tool should accurately simulate the fault modes and the OS source code should not be changed during the fault injection process.

B. Examples of Linux Fault Mode Generation

The “Set memory permission” function of Linux will fail due to kernel function failure, so one or more failure modes corresponding to “kernel function failure” are generated according to issues 1) - 4) in issue list. The system call corresponding to “Set memory permission” is mprotect(), and the related fault modes are analyzed from the characteristics of mprotect(). The function description of mprotect():

\[
\text{SYSCALL_DEFINE3(} \text{mprotect, unsigned long, start, size_t, len, unsigned long, prot) }
\]
mprotect does this by calling function do_mprotect_pkey(), the function description of do_mprotect_pkey() (The pkey value is fixed to -1 when mprotect() is executed):

\[
\text{static int do_mprotect_pkey(unsigned long start, size_t len, unsigned long prot, int pkey) }
\]

For example, the generation process of Linux fault modes caused by a parameter error in do_mprotect_pkey() ia as follows:

1) If the first parameter start is wrong, the fault mode of “invalid address parameter error when setting memory permission” may occur. When memory access permission is set, start is an invalid pointer, or is not an integer multiple of the page size. It is not aligned with the memory page. The function returns -EINVAL for “Invalid argument”.

2) If the second parameter len is wrong, the fault mode of “memory overflow error when setting memory permissions” may occur. When memory access permission is set, the specified address space [start, start+1-len] exceeds the actual process address space (this is also related to the start parameter). The function returns -ENOMEM for “Out of memory”.

3) If the third parameter prot is wrong, the fault mode of “permission conflict error when setting memory permissions” may occur. After the mmap() is used to map a read-only file to memory, if you try to give this memory PROT_WRITE permission (write permission), the service will be denied due to a permission conflict error. The function returns -EACCES for “Permission denied”.

C. Linux Fault Mode Library

We constructed a Linux fault mode library with 2880 fault modes by analyzing Linux source code and their 152 system calls. The attributes of each fault mode include: simulation_method_id, fault_mode_id, fault_mode_name, simulation_method_type and attach_data, where simulation_method_id is used as the key to uniquely identify each fault mode, fault_mode_id gives the type of each fault, fault_mode_name describes the specific fault mode, and simulation_method_type indicates the type of simulation method used for this fault mode, attach_data contains various additional parameters required for injection of this fault mode.

III. FAULT INJECTION TOOL BASED ON THE FAULT MODE LIBRARY

FIFML was developed to avoid modifications to the OS source code, thus the cost of rebuilding the code base is eliminated. The system structure of FIFML is shown in Fig. 2. It includes control module, fault mode library management module, fault injection scheme generation module, fault injection module, log module and a Linux fault mode library.
A. Control Module

The control module is used for the initialization of the FIFML. It controls the entire fault injection process and provides a user interface. It starts the fault injection scheme generation module and calls the fault injection module to inject faults, according to the commands and parameters provided by the user. In addition, it can also use the fault mode library management module to operate the fault mode library. The control module is divided into four parts:

1) The command parser is used for receiving and parsing user commands, and controlling the work of other parts according to the parsed commands.
2) The injection manager is used for interacting with the fault injection module and the fault injection scheme generation module, and handles operations related to fault injection.
3) The mode manager is used for interacting with the fault mode library management module and processing operations related to the management of the fault mode library.
4) The log manager is used for interacting with the log module and processing log-related operations.

The command parser obtains the command input by the user and identifies it as a fault injection command, and then calls the injection manager to control the fault injection scheme generator to generate a simulation method ID, fault mode ID, target process PID, fault occurrence time, fault duration, target file name and other information to control the work of the fault injection module.

B. Fault Mode Library Management Module

The fault mode library management module is responsible for managing the fault mode library. It supports query, insert, delete, modify, import and export operations. After the module parses the control commands issued by the control module, it performs corresponding operations on the fault mode library according to the control commands.

C. Fault Injection Scheme Generation Module

The fault injection scheme generation module is responsible for receiving the fault injection control information, and combined with the acquired fault simulation method data to generate a fault injection scheme as the input of the fault injection module. The fault injection scheme generation module is divided into two parts: 1) the schema queryer is used for querying specific fault mode data from the fault mode library management module according to the control command; 2) the scheme generator is used for generating the fault injection scheme according to the fault mode data.

The fault injection scheme generation module executes the control command issued by the control module, generates a fault injection scheme according to the fault mode library and the control information in the control command, and sends it to the fault injection module. The fault injection scheme determines the fault set to be simulated, the start time and duration of each fault, and the scope of influence of each fault. The generated fault injection scheme includes: fault mode information, fault simulation method information, fault occurrence time, fault duration, target process PID, target file name and other information. The fault occurrence time is the time when the fault injection operation starts execution, in the format of “YYYYMMDD:HHMMSS”; the fault duration is the duration of the fault injection operation, in seconds. The fault specified by the target process PID only activates on the process corresponding to the specific PID. The fault injection operation specified by the target file name only activates on the specific file. For example, we can simulate a fault within 60 seconds from 10:00:00 on March 20, 2021, and the fault only survives when the process with PID 1234 accesses the file a.dat.

D. Fault Injection Module

The fault injection module is responsible for implementing the fault injection scheme, and executes the specified fault simulation operation according to the fault injection scheme. The fault injection module is divided into three parts: 1) the license checker is used for checking whether the tool license is valid; 2) the parameter checker is used for checking whether the received control commands are legal; and 3) the injector is used for performing fault injection according to the fault injection scheme and managing the faults injected into the system. After receiving the control command from the control module, the fault injection module interacts with the Linux kernel to realize fault injection.

E. Log Module

The log module is responsible for recording the operation, intermediate data and internal error information of Linux and the tool itself. The log module is divided into two parts: 1) the log collector is used for collecting and formatting log data; and 2) the log storage is used for storing log data to log files. The log module receives and executes the control commands issued from the control module, and records log information including user instructions and system feedback.
F. Comparison with ZOFI

We compared FIFML and ZOFI in terms of the accuracy and coverage of fault injection. The experimental object is CentOS 8.18.0.

ZOFI supports fault simulation by changing the values of registers or instruction pointers. FIFML simulates a fault by modifying the return code of the OS system call. They can accurately inject faults on their respective fault injection objects.

The fault modes provided by ZOFI are limited to operations affected by value changes in registers and instruction pointers. The coverage of FIFML depends on the fault mode library which covers the fault modes produced by the file system, interrupt management, IO management, memory management and process management parts. So the coverage of FIFML is better than that of ZOFI.

IV. EXPERIMENTS AND ANALYSIS

We perform two types of experiments for CentOS-stream-8 (kernel 4.18.0-383.el8.x86_64), AnolisOS-8.4-GA (kernel 4.18.0-305.am8.x86_64) and openEuler-20.03-LTS-SP3 (kernel 4.19.90-2112.8.0.0131.oe1.x86_64), one is the failure analysis experiments based on fault injection for the three parts of the Linux-like OS, i.e. kernel function and system call, lib (including glibc and basic compilation tools), system service (including key components that Linux provides services), the other is the performance comparison experiments of three virtual file systems of Linux-like OSs without fault injection and with fault injection.

For each type of experiments, the workload is first run to warm up system for 30 seconds. Then, FIFML performs fault injection. Finally, the system is rebooted to the same initial state of the experiment. Log files are collected at the end of each experiment and used later to analyze whether the fault has been injected and whether the injected fault affects the OS performance.

A. The Experiment of Failure Analysis Based on Fault Injection

The failure analysis experiment results are divided into five levels: 1) Crash: The system crashed after injecting a fault; 2) No responding: The system does not give response after injecting a fault; 3) Affect: Some processes are affected and cannot run normally after injecting a fault; 4) Light: The system and processes are slightly affected after injecting a fault; 5) Normal: The system is running normally after injecting a fault.

1) Kernel function and System call: The same fault is injected into the kernel functions and system calls of three OSs for each experiment, and each fault is injected repeatedly three times. Fig. 3 shows the aggregated results of fault injection for the three systems. Each bar represents the corresponding fault distribution.

For each OS, 1250, 229, 621, 173 and 597 fault injection experiments are performed on the modules of file system, interrupt management, IO management, memory management and process management parts. The number of fault injection experiments depends on the number of fault modes in the fault mode library. The proportion of file system and IO management module seriously affected by injected faults (Crash or No responding) is relatively low, and the proportion of interrupt management module and process management module seriously affected by injected faults is relatively high. The comparison of the three OSs shows that Anolis OS performs well in file system and IO management, while openEuler performs better in memory management and process management. The three OSs performed similarly in interrupt management, Anolis OS crashes less frequently.

In the related types of failures in the experiments, one type of failure is related to the read file operation. In these experiments, when the system provides the function of reading files, system call sys_read() returns error codes to the caller (such as EIO, EINVAL, EWOULDBLOCK) to simulate IO errors, invalid parameter errors, operation blocking, etc. Fig. 4 shows a situation when a system call returns an error code resulting in a runtime exception. CentOS and openEuler will
crash directly without providing any meaningful information to the user, so it may bring a bad experience to the user. Anolis OS will prompt that the library file cannot be read and provide a message of “Input/output error”.

Another type of function failure in the operation of obtaining the file status according to the file descriptor is generated by returning EFAULT, ENOTDIR, ELOOP from sys_newfstat(). CentOS and openEuler cannot execute the file operations. It prompts that the used library file cannot be loaded and gives “Error 40” information. Some functions of Anolis OS, such as linking files produced during the compilation process, will prompt Error, but the overall performance of the system is stable. Anolis OS is more robust, and it can provide some help for subsequent failure recovery.

In addition, some of the differences are highly related to the component version. A total of 39 fault modes which are related with sys_munmap(), sys_getdents64(), and the remaining 8 system calls, will cause some versions of systemd components to fail. It causes core dumps, then results in the system to crash. The faults of sys_fadvise64() will introduce corresponding faults when the access mode of the file data is pre-declared. This type of faults will affect cat. For example, the 8.30 version will report a “Bad address” error, but the 8.32 version will not.

2) glibc/basetools: The default gcc versions of CentOS, Anolis OS and openEuler are 8.5.0 20210514 (Red Hat 8.5.0-15), 8.5.0 20210514 (Anolis 8.5.0-10.0.1), and 7.3.0, respectively. We performed fault injection on the default versions of gcc for three OSs. We construct a fault mode library with 385 fault modes for gcc. Each fault mode is injected separately, and the performance of the three OSs is compared at the lib layer.

The experimental results show that the three OSs perform similarly, and the injected faults at lib layer only affect the functions related to the library functions and do not spread to other parts of the system. For example, a fault is injected into do_spec, then it returns a wrong value (like reversed at the last bit). The failure of processing the SPEC specification and the execution of instruction is manifested as being unable to read the shared library file of mpfr. It only affects some processes that call this function, but does not seriously affect the system.

3) System service: We selected 11 common system services, such as dbus.service, NetworkManager.service, sshd.service, systemd-journald.service. Two hundred fault modes are selected, and 20 fault modes are randomly formed into a test group. There are 10 test groups in total. The experimental results show that the performance of these three OSs is the same. They are divided into two categories: Normal and Affect, as shown in Fig. 5. More than half of the services will be affected, such as firewalld.service, a dynamic firewall daemon, which will behave abnormally with “python-nftables fail”. From the system’s point of view, the affected process will not affect the system operation, but key services, such as systemd-coredump, which will cause a core dump, will affect the system operation and make the system unable to respond.

B. The impact of faults at levels of Light and Normal on performance

To evaluate the impact of faults on system performance, faults at levels of Light and Normal are injected into the virtual file systems of CentOS 8, Anolis OS and openEuler. The performance (operation time) of 13 common operations is measured before and after fault injection. The total number of injected fault modes $N$ is 832.

The 13 common operations are as follows:

1) Create and initialize a new inode object under the given superblock(sb_alloc_inode)
2) Obtain the status of the file system corresponding to the superblock(sb_stat_get)
3) Synchronize the file system with the disk(sb_sync_fs)
4) Create and delete hard links(inode_link)
5) Create and delete symbolic links(inode_symlink)
6) Create and delete a new directory(inode_mkdir)
7) Move/rename files(inode_rename)
8) Update file offset pointer(file_lseek)
9) Read data from a file(file_read)
10) Write data to a file(file_write)
11) Maps the given file to the specified address space(file_mmap)
12) Writes all cached data from the file back to disk(file_sync)
13) Make zero copies of files(file_sendfile)
In the performance measurement experiment, the required time of each operation without fault injection and the required time of each operation with fault injection $PAF$ are obtained respectively, the impact level of faults on operation performance is divided according to the difference of the required time of each operation before and after fault injection. In order to rationally divide influence level, reduce the impact of fluctuations in performance, we define performance threshold $PT$ for estimating the maximum performance change for each operation without fault injection. The standard deviation of performance of each operation without fault injection $MPD$ is also defined and the worst performance of each operation without fault injection $WP$ is obtained from the required time of each operation before fault injection. According to the 3 Sigma criterion in statistics, 99.73% of the data are within 3 standard deviations of the mean if the performance fluctuation follows a normal distribution. Therefore, in order to further improve the confidence level, the maximum value of the performance fluctuation is allowed to be the sum of $MPD$ and $WP$ under the condition that the standard deviation and the worst performance are known. We have

$$PT = MPD + WP$$ (1)

According to the relationship between $PAF$ and $PT$, three influence levels are defined as follows.

1) No Influence: $PAF < PT$
2) Mild Influence: $PT \leq PAF < 5 \times PT$
3) Serious Influence: $PAF \geq 5 \times PT$

“No Influence” indicates that there is almost no difference in operation performance before and after fault injection. “Mild Influence” indicates that the operation performance deteriorates before and after fault injection. However, the performance difference is tolerable for users and hardly affects the performance of user processes and system services running on the system. “Serious Influence” indicates that operational performance is significantly degraded before and after fault injection, as well as affecting the performance of user processes running on the system and system services.

The experimental results are shown in Fig. 6 and Fig. 7. Fig. 6 describes the number of faults that slightly affect the performance after fault injection for 13 common operations of virtual file systems of CentOS 8, Anolis OS and openEuler. Anolis OS has fewer “Mild Influence” on operations such as file write and file mmap. CentOS 8 has advantages in file sendfile operations and openEuler has fewer “Mild Influence” on inode link operations. Fig. 7 describes the number of faults that seriously affect the performance after fault injection for 13 common operations of virtual file systems of CentOS 8, Anolis OS and openEuler. The three OSs have similar performance in file entries and superblock operations. However, in terms of operations related to the inodes, openEuler suffers a lot of performance degradation due to fault injection.

To quantitatively evaluate the influence of fault injection on
the performance of virtual file systems of CentOS 8, Anolis OS and openEuler, we define the performance level affected by fault $PL_f$ and the fault impact degree $EF$. $EF$ is related to the number of faults causing “Mild Influence” $SLA$ and the number of faults causing “Serious Influence” $SEA$, but their impacts are different. To distinguish this difference, different weights are assigned to $SLA$ and $SEA$. Thus, we have

$$EF = 0.4 \times SLA + SEA$$

The larger the value of $EF$ is, the more faults that affect the performance, the smaller the $PL_f$ is. The larger the value of $EF$, the smaller the impact on $PL_f$. $PL_f$ could be represented as follows.

$$PL_f = e^{-EF \times N} \times 100\%$$

The $PL_f$ of virtual file systems of CentOS 8, Anolis OS and openEuler are shown in Fig. 8. Anolis OS has the best performance and CentOS 8 has a slightly lower score than that of Anolis OS. Since openEuler suffers from severe performance degradation due to fault injection, it has a lower score.

C. Suggestions

According to the experimental results, we have the following suggestions:

1) The failure analysis based on fault injection can identify more accurately the reliability problems existing in the system. System call based fault injection can be used to effectively analyze the relationship between the abnormal system functions and their corresponding system calls, whether there is a corresponding system call error handling method in functions that are highly dependent on system calls. In these cases, obvious failures (crash, no responding, fewer error messages) can greatly impact the final user experience. By providing corresponding processing methods, such as adding exception handlers in the library or adding fault tolerance methods to key functions, the system crashes can be effectively avoided. Performing regression tests with fault injection to check exception handling is also an effective method.

2) Some differences are related to the component version and such differences are related to the selection of the component package. Therefore, in the system development process, not only the functions and performance provided by the component package, but also its reliability should be considered. In addition, components should be optimized and upgraded in time.

3) Strengthening the reliability of key components of the system has an obvious effect on improving the reliability of the system.

V. CONCLUSION

In this paper, we first apply the fault mode generation method based on Linux AHSA to systematically generate a large number of fault modes of an OS, construct a fault mode library and develop a fault injection tool FIFML. Then, we perform the fault injection experiment on three commercial Linux-like OSs, identify the reliability problems and give improvement suggestions. In addition, we use the virtual file systems of these three OSs as objects, to perform fault injection at levels of Light and Normal, measure the performance of 13 common file operations before and after fault injection.

The bottleneck of using FIFML for Linux failure analysis is the completeness of the fault mode library. The complexity of the system failure behavior is high, the failure analysis time is long and it is difficult to ensure the coverage and accuracy of the fault mode library. Our current work is to improve the rationality and completeness of the fault mode library by analyzing the impact of injected fault modes on Linux based on practical experience. Future work is to reduce the manual effort required to construct a library of Linux fault modes, combine practical experience with automated tools and employ fault injection based reliability testing for OS reliability estimation.

REFERENCES

[1] R. Amarnath, S. N. Bhat, P. Munk, E. Thaden, “A Fault Injection Approach to Evaluate Soft-Error Dependability of System Calls,” Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops, Memphis, pp. 71–76, Oct. 2018.

[2] S. Winter, O. Schwahn, R. Natella, N. Suri, D. Cotroneo, “No pain, no gain? The utility of parallel fault injections,” Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, Florence, vol. 1, pp. 494–505, May 2015.

[3] T. Yoshimura, H. Yamada, K. Kono, “Using Fault Injection to Analyze the Scope of Error Propagation in Linux,” Information and Media Technologies, vol. 8, no. 3, pp. 655–664, 2013.

[4] J. Arlat, J. C. Fabre, M. Rodriguez, “Dependability of COTS microkernel-based systems,” IEEE Transactions on Computers, vol. 51, no. 2, pp. 138–163, Feb. 2002.

[5] V. Porpodas, “ZOFIT: Zero-Overhead Fault Injection Tool for Fast Transient Fault Coverage Analysis,” CoRR, abs/1906.09390, 2019.

[6] R. Chillarege, S. Biyani, “Identifying risk using ODC based growth models,” Proceedings of the 5th International Symposium on Software Reliability Engineering, Monterey, pp. 282–288, Nov. 1994.

[7] D. Cotroneo, A. K. Iannillo, R. Natella, S. Rosiello, “Dependability Assessment of the Android OS Through Fault Injection,” IEEE Transactions on Reliability, vol. 70, no. 1, pp. 346–361, March 2021.

[8] X. L. Chen, J. H. Jiang, W. Zhang, X. Z. Xia, “Fault diagnosis for open source software based on dynamic tracking,” Proceedings of the 7th International Conference on Dependable Systems and Their Applications, Xi’an, pp. 263–268, Nov 2020.

[9] X. Z. Xia, W. Zhang, J. H. Jiang, “Ensemble methods for anomaly detection based on system log,” Proceedings of IEEE 24th Pacific Rim International Symposium on Dependable Computing, Kyoto, pp. 93-94, Dec. 2019.
[10] A. Jin, J. H. Jiang, J. W. Hu, J. G. Lou, “A PIN-based dynamic software fault injection system,” Proceedings of the 9th International Conference for Young Computer Scientists, Zhangjiajie, pp. 2160-2167, Nov 2009.

[11] A. Jin, J. H. Jiang, “Fault injection scheme for embedded programs at machine code level and verification,” Proceedings of 15th IEEE Pacific Rim International Symposium on Dependable Computing, Shanghai, pp. 55-62, Nov. 2009.

[12] M. R. Lyu, Handbook of software reliability engineering. NY: McGraw-Hill, 1996.

[13] A. Silberschatz, P. B. Galvin, G. Gagne, Operating system concepts. John Wiley & Sons, 2006.

[14] D. Cotroneo, A. Lanzaro, R. Natella, “Faultprog: Testing the accuracy of binary-level software fault injection,” IEEE Transactions on Dependable and Secure Computing, vol. 15, no.1, pp. 40-53, 2018.

[15] R. Natella, D. Cotroneo, H. S. Madeira, “Assessing dependability with software fault injection: A survey,” ACM Computing Surveys, New York, vol. 48, no. 3, pp. 1-55, 2016.

[16] M. Cinque, D. Cotroneo, R. Della Corte, et al., “Assessing direct monitoring techniques to analyze failures of critical industrial system,” Proceedings of the IEEE 25th International Symposium on Software Reliability Engineering. IEEE, Naples, pp. 212-222, Nov. 2014.

[17] J. A. Duraes, H. S. Madeira, “Emulation of software faults: A field data study and a practical approach,” IEEE Transactions on Software Engineering, vol. 32, no. 11, pp. 849-867, 2006.

[18] T. K. Tsai, R. K. Iyer, “Measuring Fault Tolerance with the FTAPe fault injection tool,” Beilner, H., Bause, F. (eds) Quantitative Evaluation of Computing and Communication Systems. TOOLS 1995. Lecture Notes in Computer Science, Berlin, vol. 977, pp. 26-44, May 1995.

[19] S. Dawson, F. Jahanian, T. Mitton, “A software fault injection tool on real-time Mach,” Proceedings of 16th IEEE Real-Time Systems Symposium, Pisa, pp. 130-140, Dec. 1995.

[20] C. Pham, et al., “Failure Diagnosis for Distributed Systems Using Targeted Fault Injection,” IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 2, pp. 503-516, Feb. 2017.

[21] O. Schwahn, N. Coppik, S. Winter, N. Suri, “FastFI: Accelerating Software Fault Injections,” Proceedings of the IEEE 23rd Pacific Rim International Symposium on Dependable Computing, Taipei, pp. 193-202, Dec. 2018.

[22] M. Sullivan, R. Chillarege, “Software defects and their impact on system availability-a study of field failures in operating systems,” Digest of Papers. Twenty-First International Symposium on Fault-Tolerant Computing, Montreal, pp. 2-9, 1991.

[23] W. I. Kao, R. K. Iyer, D. Tang, “FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults,” IEEE Transactions on Software Engineering, vol. 19, no. 11, pp. 1105-1118, Nov. 1993.

[24] A. Thakur, R. K. Iyer, L. Young, I. Lee, “Analysis of failures in the Tandem NonStop-UX Operating System,” Proceedings of 6th International Symposium on Software Reliability Engineering, Toulouse, pp. 40-50, Oct. 1995.

[25] I. Lee, R. K. Iyer, “Software dependability in the Tandem GUARDIAN system,” IEEE Transactions on Software Engineering, vol. 21, no. 5, pp. 455-467, May 1995.

[26] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, N. Bidokhti, “FailViz: A Tool for Visualizing Fault Injection Experiments in Distributed Systems,” Proceedings of the 15th European Dependable Computing Conference, Naples, pp. 145-148, Sep. 2019.

[27] C. Sheridan, D. Whigham, M. Artač, “DICE Fault Injection Tool,” Proceedings of the 2nd International Workshop on Quality-Aware DevOps, Germany, pp. 36-37, July 2016.

[28] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun, “Experimental analysis of the errors induced into Linux by three fault injection techniques,” Proceedings of the International Conference on Dependable Systems and Networks, Bethesda, pp. 331-336, June 2002.

[29] J. Christmannsson, P Santhanan, “Error injection aimed at fault removal in fault tolerance mechanisms-criteria for error selection using field data on software faults,” Proceedings of 7th International Symposium on Software Reliability Engineering, 1996, pp. 175-184.

[30] F. Salles, M. Rodriguez, J. C. Fabre and J. Arlat, “MetaKernels and fault containment wrappers,” Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, NY, pp. 22-29, Nov. 1999.