Comparison of Fault Simulation Over Custom Kernel Module Using Various Techniques

Syeda Huma Jabeen\(^1\), Zhai Gaosho\(^2\), Muhammad Ejaz Sandhu\(^1\), Naima Noor\(^1\), Rana Hakeem Ali Mudassar\(^1\)

\(^1\)Department of Computer Science, The Institute of Management Sciences, Lahore, Pakistan. \(^2\)Beijing Jiaotong University, China.

(Received: 19 July 2021 ; Accepted: 08 Sep 2021 ; Issue Published: 12 Sep 2021)

ABSTRACT

To test the behavior of the Linux kernel module, device drivers and file system in a faulty situation, scientists tried to inject faults in different artificial environments. Since the rarity and unpredictability of such events are pretty high, thus the localization and detection of Linux kernel, device drivers, file system modules errors become unfathomable. ‘Artificial introduction of some random faults during normal tests’ is the only known approach to such mysterifying problems. A standard method for performing such experiments is to generate synthetic faults and study the effects. Various fault injection frameworks have been analyzed over the Linux kernel to simulate such detection. The following paper highlights the comparison of different approaches and techniques used for such fault injection to test Linux kernel modules that include simulating low resource conditions and detecting memory leaks. The frameworks chosen to be used in these experiments are; Linux Test Project (LTP), KEDR, Linux Fault-Injection (LFI), and SCSI.

KEYWORDS: Device Drivers, Fault simulation, KEDR Framework, Linux, Linux Test Project (LTP), Linux Fault-Injection (LFI), memory leak detection, SCSI

1. INTRODUCTION

Linux kernel-based Operating systems (OS) are used widely in the world. More than 90% of the most powerful supercomputers, based on that. Because the performance of Linux is superior as compared to other operating systems [1,2]. However, Android OS-based mobile devices dominate the international market [3]. There is still an increasing opportunity to use Linux in enterprise systems, where users expect very high reliability [4]. One of the critical components of the Linux kernel is device drivers. However, there are many other essential kernel components, such as networking stacks, file systems, sound, video, infrastructures, and subsystems providing virtualization support, hard drives, removable drives, and network adapters. Fault injection (FI) or fault simulation techniques are generally used to minimize or overcome the difficulties by regulating the faults and executing error handling code [5]. Also helps to evaluate the system’s tolerance and predict the system failure [6]. Different fault injection tools or techniques are available for Linux fault injection or fault simulation. Still, none of them can inject various kinds of fault scenarios or are not flexible to inject faults as one wants to inject, which is required for systematic assessment of error handling code [4]. Linux kernel module testing by fault simulation is a critical task. It differs from user-space applications; Linux kernel modules might have direct access to some memory areas used by other parts of the Kernel [7]. Kernel modules may communicate at a lower level with the hardware and may have some other capabilities that are hardly available outside the Kernel. Furthermore, it should be noted that any type of kernel module resource allocation will not be freed automatically even when one unloads the module. So, it should be noted that a faulty kernel module is very harmful if it gets out of control [8].

The rest of the paper is structured as follows. In Section II, we describe Categories and Tools of Fault Injection. Then, in Section III, we present our experimental setup. In Section IV, we provide and discuss the results of our experiments and summarize our contributions. In Section V we propose topics for future work.

2. OVERVIEW OF VARIOUS TOOLS AND CATEGORIES OF FAULT INJECTION

In the following section, we explain the different categories and tools of fault injection.

2.1. Fault Injection Categories

Fault injections are essential techniques for authenticating the reliability of a system by examining the performance of the devices when a fault happens. Numerous efforts have been made to develop methods for inserting faults into a system prototype. Most of the advanced techniques are divided into five main types, as shown in Figure 1. The significant categories of Fault injection are specified below

2.1.1. Hardware-Based Fault Injection

Hardware-based fault injections require expert knowledge and manual testing. However, their results are unquestionable [9]. It’s a kind of physical fault injection. In which faults are injected by heavy-ion radiation, electromagnetic-based interference like voltage or power supply disability, laser [10]. Hardware fault injections can be sub-categorized
based on their faults and locations, i.e., with contact and without contact [11].

2.1.2. Software-Based Fault Injection

Software-based fault injection (SWIFI) alters the contents of memory or registers the occurrence of fault simulation. In Software-based fault injection, different errors or bugs are introduced at a software level that would result in the production of faults in hardware. For speedy coverage, Software-based fault injections depend upon simulations [9]. Software injection can further be categorized on how the Fault is injected:[11].

• Compile-time fault injection
• Runtime fault injection

2.1.3. Simulation-Based Fault Injection

The technique is applied to high-level fault models, for example, to VHDL models. It is beneficial in evaluating the model when it is just in the initial state. It tests the various abstraction levels by utilizing distinct kinds of descriptive languages [10]. Faults propagated during simulation by changing the model’s logical values. This approach mainly focuses on injecting Fault primarily on VHDL models and systems [12].

2.1.4. Emulation Based Fault Injection

This technique is used to study the actual behavior of the circuit in the environment application considering real-time interactions. Its FPGAs are used for speeding up fault simulation and effective circuit emulation [13]. It can be very time-consuming [17].

2.1.5. Hybrid Fault Injection

This approach is a combination of both software-implemented fault injection and hardware. The Hybrid Fault injection technique can be grouped into the following:

• Invasive Techniques: The techniques that leave behind footprints during testing. The problem of sufficiently complex systems, especially time-dependent systems, is that it is usually impossible to remove the footprint of the testing processes from system behavior, independent of fault injections.
• Non-invasive Techniques: These can hide their presence without affecting a system other than the fault injection [15].

3. METHODOLOGY AND EXPERIMENTAL SETUP

Several techniques and tools are being used for the Linux kernel module testing; some of the unique tools are reviewed in this paper, as shown in Figure 2. Fault injection tools can be categorized as follows:

• Static Analysis Tools: The tools that only analyze the source code of the module.
• Dynamic Analysis Tools: The tools, which perform runtime, post mortem, or both. The Fault is injected into the circuit, and then the different input vectors are simulated in the circuit to compute faults [16].

It is helpful to use both kinds of analysis, i.e., static and dynamic kernel modules testing [17].
block devices. The Linux fault injection framework supports runtime patterns. LFI framework is directed in both ways, i.e., user-space (application-level) and five system-level targets. There is a configuration to activate the Linux FI framework, i.e., a set of files in the debug file-system to divert the kernel toward a specific point where faults should be injected. This can be done automatically by an application or through a script. According to user interest, fault injection mechanisms can be altered and enhanced for specific points of interest because Linux is free software and open-source. Like Figure 3, the workflow of LFI shows how we have used this framework to insert faults. As of version 4.9.5., the framework can allocate memory in the Linux kernel module mainly in three ways:

- Page allocations errors:
- Slab errors:
- Disk I/O errors:

Various systems are now included in Linux kernel like Kmemleak, Kmemcheck, FI framework, etc. [18].

- The latest and easy way to compile the Kernel.

3.2. Experiment on Linux Fault-Injection (LFI)

To implement the fault injection by using LFI, (i) VMware station 10.0 is installed on our machine, and a virtual machine is built, and (ii) Ubuntu 15.04 with Linux-header-3.19.0-15-generic is installed on that virtual machine. Then source codes of (iii) LFI are downloaded, compiled and installed on the virtual machine.

3.2.1. Compilation of custom kernel in Linux Fault-Injection (LFI)

The compilation of the custom kernel consists of two ways:

- The traditional way to compile the Kernel.

3.2.2. Fault Injection Capabilities and Configuration of Capabilities Behavior in Linux Fault-Injection (LFI)

After reviewing the behavior of LFI, it was acknowledged that the Linux Fault Injection Framework is required for Debugfs entries to satisfy the fault injection capabilities. There are mainly two ways to make Kernel satisfied for fault injection capabilities, as shown in Table 1 [20]:

- Boot time configuration of fault capabilities.
- Runtime configuration of fault injection.

### Table 1. Fault Injection Behavior

<table>
<thead>
<tr>
<th>Available Fault Injection Capabilities</th>
<th>Boot Time Configuration of Capabilities</th>
<th>Runtime Configuration of Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faillab</td>
<td>o Faillab</td>
<td>Compile a Custom Kernel As per capabilities of Fault Injection</td>
</tr>
<tr>
<td>Fail page alloc</td>
<td>o Fail page alloc =</td>
<td></td>
</tr>
<tr>
<td>Fail fmt</td>
<td>o Fail fmt =</td>
<td></td>
</tr>
<tr>
<td>Fail to make a request</td>
<td>o Fail to make request =</td>
<td></td>
</tr>
<tr>
<td>Fail MDC request</td>
<td>o Fail MDC request =</td>
<td></td>
</tr>
</tbody>
</table>

3.2.3. Selection of Faults and Parameters in Linux Fault-Injection (LFI)

After successfully compiling the custom kernel, a fault can be selected to inject by utilizing different types of faults in LFI, i.e. Fail-Page, FailSlab and DiskI/O, as shown in Table 2.

### Table 2. Types of Faults.

<table>
<thead>
<tr>
<th>Parameter of Faults in /sys/kernel/debug/</th>
<th>Types of Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fail Slab</td>
</tr>
<tr>
<td>Verbose</td>
<td>3</td>
</tr>
<tr>
<td>Times</td>
<td>3</td>
</tr>
<tr>
<td>Task-filter</td>
<td>3</td>
</tr>
<tr>
<td>Stacktrace-filter</td>
<td>3</td>
</tr>
<tr>
<td>Space</td>
<td>3</td>
</tr>
<tr>
<td>Require-start</td>
<td>3</td>
</tr>
<tr>
<td>Require-end</td>
<td>3</td>
</tr>
<tr>
<td>Probability</td>
<td>3</td>
</tr>
<tr>
<td>Interval</td>
<td>3</td>
</tr>
</tbody>
</table>
3.2.3. Injection of Fault on Module and Tests in LFI

The Linux Fault Injection Framework has a shell script of fault injection with the name failcmd inside the fault injection directory. An individual may inject a fault on a Single module by providing a module parameter. In addition, it can be injected on Run tests. To inject a fault, fault type may be defined at run time by using a command given below:

```
# env FAILCMD TYPE=fail_page alloc ./tools/testing/fault-injection/failcmd.sh -- times=100 -- make -C tools/testing/selftests/ run tests
```

In this command, parameters are used 100 times, fault type is selected as a Fail - page, and applied to run tests. But if the fault type is not specified, by default, it will Failslab.

3.3. SCSI Fault Injection tool framework

The SCSI fault injector is capable of fault injections in a realistic pattern, and it is comprised of a set of various test programs to cover different fault conditions. It is implied as a set of System Tap scripts. System Tap is mainly used to track information when an I/O request is passed between layers within a kernel [4]. The SCSI fault injection works in two steps.

- It identifies the target SCSI command by matching user-specified.
- Inject Fault in the processing of the target SCSI command.

3.3.1. Testing Error Handler Using SCSI Fault Injection

As Figure 4, the Workflow of SCSI illustrates how the SCSI framework is used for fault injection. To get a fruitful result, it is required to have a set of test programs during tests systemic testing for error handling source code by using fault injections for any target kernel component. This set of test programs makes sure either that the Fault is occurring or not while the target kernel component is being handled appropriately. For specific test programs, it is compulsory to have a target kernel component in a preferred state. The specific (desired) Fault is injected in the desired place where it will trigger by test programs, and then the result is tested whether it is correctly injected or not. For such goals, the Tool provides a platform to specify the kind of Fault, which will be injected, and then finally, it will cause injection.

3.3.2. Specifying the Type of SCSI Fault

The SCSI HDD fault patterns are classified to define a specific fault pattern generated by the fault injector. SCSI fault can be classified into two patterns. First, “Response of SCSI device with an error pattern”, meaning when the drive returns an error condition, it is returned to the OS. For example, it can be caused by a media error. Second, “The unresponsive device pattern”, in this case, the drive does not return any error to the OS 15, resulting in a timeout. For example, it can be caused by SCSI cable faults. Instead of that, HDD hardware faults can be classified into (see Table 3):

1. **Temporary Fault**: It can be generated by an accidental way and recoverable HDD fault.
2. **Permanent Fault**: A severe HDD fault can cause it.

3.3.3. Experiment on SCSI

To inject a fault with a single disk once the target command is found, the SCSI fault injector modifies the target. System Tap provides a platform for the SCSI fault injector to insert a hook inside the Kernel that dynamically also alters the value of variables [4]. While using SystemTap, a false is created from a SCSI device if this reports an error in OS [13]. SCSI tool faults patterns and triggers conditions are obtained from a test program as an argument. Figure 4 illustrates a Workflow of SCSI. For example, RAID drivers software were tested by injecting different SCSI faults to RAID drivers and to check if the software RAID driver and SCSI middle layer error handling code work properly.
### Table 3. Types of faults offered by SCSI

<table>
<thead>
<tr>
<th>Script Name</th>
<th>Fault type description</th>
</tr>
</thead>
<tbody>
<tr>
<td>diskless.e</td>
<td>Permanently read error simulation</td>
</tr>
<tr>
<td>diskwrite.e</td>
<td>Permanently read/write error simulation</td>
</tr>
<tr>
<td>sectored.e</td>
<td>Read error correctable by writing simulation</td>
</tr>
<tr>
<td>temporary.e</td>
<td>Temporary read error simulation</td>
</tr>
<tr>
<td>/kernel.e</td>
<td>Temporary write error simulation</td>
</tr>
<tr>
<td>/memorie.e</td>
<td>Temporary no response to reading access simulation</td>
</tr>
<tr>
<td>/power.e</td>
<td>Temporary no response to write access simulation</td>
</tr>
<tr>
<td>/missing.e</td>
<td>Permanent no response on both read/write access simulation</td>
</tr>
</tbody>
</table>

### 3.3.4. Testing Procedure

The following procedure needs to be evaluated.

- Installing the OS on one of the SCSI disks.
- Injection of various patterns of HDD fault.
- Check the results.

### Make custom kernel to support Fault Injection with SCSI

The SCSI fault injector frame is flexible in behaviour, i.e., easy to inject by making simple changing in the Kernel because of its SystemTap scripts [4]. Virtual configurations for the Kernel are given below:

#### General setup

- a. [*] Kprobes
- b. [*] Kernel - user space relay support (formerly relayfs)
- c. [*] Enable loadable modules

#### Kernel hacking

- a. [*] Kernel debugging
- b. [*] Compile the Kernel with debug info

### Redundant Array of Independent Disks - RAID

Disk I/O performance can be improved by creating a single logical unit of multiple hard disks. RAID is a storage technology that mainly focuses on combining multiple disks to create one disk over which Fault can be injected to improve its functionality.

* **Raid 1 - Mirroring** RAID1 array software setup (also recognized as a “mirroring” array), in which the array forms of data are written over two disks. Although it also can generate a RAID-1 on a single hard disk by making a partition over it. It has a drawback with it; failure may occur if that single hard disk faces with Fail.

* **MDADM - multiple disks-admin on Linux**

Mdadm is mainly a tool that helps to generate a RAID of different levels. This tool provides the functionality to manage, assemble, create, and monitor a RAID. It is available by defaults on LINUX distros such as CentOS, RHEL, Fedora or Arch Linux platforms.

* **Creation of RAID device** To create RAID 1, firstly, it verifies the number of disks available under the Ubuntu server, using a (fdisk -lu) list of all the available disks. It has found only a single disk sda (system disk).

* **Hard Disk Partition** While the hard disk has been added successfully; it still has not been mounted on virtual systems. You need to do a hard disk partition and use it. Check the current hard drives and their corresponding partitions, open Terminal and type command: Sudo fdisk -lu. The terminal will show the two prompts given below:

```
Disk /dev/sdb doesn’t contain a valid partition table
Disk /dev/sdc doesn’t contain a valid partition table
```

* **Create a sdb system disk** After reboot, verify the above functionality has been done successfully. It must add a partition table for sdb, format and automatically mounted in a virtual system. Sdb is created by using a given command:

```
“fdisk /dev/sdb”
```

* **Create a sdc system disk** The partition table is open by using a given command:

```
“fdisk/dev/sdc”
```

### 3.3.5. Install SCSI fault injector source and mdadm

Download, SCSI tool source forms the GitHub repository, extract it on desires place. To install the mdadm command below is used:

```
Sudo apt-get install mdadm
```

### 3.3.6. Create md RAID1 array

To create an md RAID1 array, mdadm is used with multiple options.

```
mdadm -C /dev/md0 -l1 -n2 -f /dev/sd[bc]1
```

Status verification of RAID arrays is done by using the command below

```
cat /proc/mdstat
```

### 3.3.7. Injection of Fault on RAID1 /dev/md0

Injecting a fault on RAID1 using a SCSI tool needs to follow the steps below:

1. **Create a filesystem** After creating RAID1 success, it is time to create a file system for md0. The command below is used to create a file-system mkfs -t ext3 /dev/md0

2. **Mount the md device** After successfully creating the file system, it must mount md0 to the desired place. The command given below is used to mount /dev/md0 successfully.
mount -t ext3 /dev/md0/home/SCSI/Desktop/SCSI-fault-injection-test-tool-1.0.1

3. **Create a target file to cause a fault**

   The file created to inject a fault might be cached in memory, but access to the target device wants to be made to inject a fault. For being sure, must drop all pages by using a command:
   
   `echo 1 >/proc/sys/VM/drop caches`
   
   Check the inode number of the file by command:
   
   `ls -hil`
   
   It created two prompts as a result of `ls -hil`:
   
   ```
   Total 28K
   11 drwx—2 root 16K 2007-12-21 18:22 lost+found
   12 -rw-r--r-- 1 root21 2007-12-21 19:39 test.txt
   ```
   
   These prompts showed that:
   
   ```
   “test. the text has inode number 12 in the md0 device.”
   ```

3.3.8. **Install SystemTap**

   SystemTap is a crucial feature for the SCSI injector, so ensure whether the platform is installed; if it is not installed, install it by the apt-get SystemTap command. SystemTap takes almost 5-10 minutes, depending upon the processor.

3.4. **LTP Fault Injection tool framework**

   Linux Test Project (LTP) is an open-source project, which brings test suites to the open-source developers or the users that authenticate the steadfastness, strength, and constancy of Linux.

   The Linux operating system has various aspects that can be tested with the LTP test suite because it contains automatic and semi-automatic tests.

   The workflow of LTP is described in Figure 5, which shows how we have used this framework to insert faults.

   ![Figure 5. Workflow of LTP](image)

   3.4.1. **Compile a custom kernel for Linux Test Project (LTP)**

   Mainly LTP is needed to compile a kernel to work with LTP to injection fault with the support of LFI.

3.4.2. **Installation of LTP by Quick Start**

   To install the LTP quick start has been chosen, which will reduce the overall installation time and give a fast install.

   ![Table 4. Required Kernel Configuration (KC) for LTP](image)
3.4.3. Linux Test Project (LTP) supports Linux Fault-Injection (LFI)

LTP is joined with LFI to inject a fault. Table 4 shows the kernel configuration for LTP.

3.4.4. Test cases in Linux Test Project (LTP)

LTP mainly provides various test cases that can be used to verify the Kernel. It provides the network test, disk I/O test, ID check and other kernel code tests under the Runltp directory.

3.4.5. Runltp Test-cases for fault injection

Runltp test cases have a script to run an actual test. The test script can be selected manually depending upon the requirements. The script is called runalltest.sh but it does not mean that all tests in the LTP will run at once.

3.4.6. Injection of Fault on default Test in Linux Test Project (LTP)

We have chosen a default test (Run test) to inject a fault. However, we need to run a script. Runltp with root privilege.

3.4.7. A shell script and parameters to inject Fault

There are mainly two parameters, Loop and Probability, supports for LTP Test suites for fault injection over tests. LTP test shell script is made by Perl; LTP command will be working as input, which will create a final file on the way given below:[12]

{loop1} <binary name is tested>
{loop2} <insert _kernel faults.sh to test binary name>
{loop3} <label passes from loop3 to loop (n-1)>
{loopn under kernel fault} <restore kernel faults default.sh to test binary name>

3.4.8. Injection of faults over choose Test

Test Case 1
We have selected the memory management (mm) test as the first test to run under the faulty Kernel; the command is given below:

```bash
# ./runltp -f mm -F 6,15 -o ltp with fault injection.out
Loop → 6
Probability → 15
```

Test Case 2

We selected mmapstress as a second test to run under a faulty kernel, a command is given below:

```bash
# ./runltp -f mmapstress -F 8,40 -o ltp with fault injection stress test.out
Loop → 8
Probability → 40
```

Test Case 3

We choose dma thread diotest as a third test to run under a faulty kernel; the command is given below:

```bash
# ./runltp -f dma thread diotest -F 6,60 -o ltp with fault injection dma thread dio test.out
Loop → 6
Probability → 60
```

3.5. KEDR Framework

We have already worked with this Tool and have published our work. We have used KEDR for the device’s read and write operations and memory allocation to inject faults into the related custom kernel module. We found the KEDR framework is efficient and helpful in injecting and simulating a fault [19].

4. RESULTS

After successfully injecting a fault by different tools, it is observed which Tool is better in what way. The rating level is selected from 0 - 5 numbers, 0 being the lowest and five being the highest.

Table 5 illustrates the comparison made based on Feature, Support and, Ease. LTP is consistently higher than others except for feature and design, where KEDR is the highest. However, overall, LTP is better. LFI is slower in all aspects, indicating that it is the slowest Tool. On the other hand, SCSI is moderate in all aspects.

Table 5. Comparison made on Feature, Support Ease

<table>
<thead>
<tr>
<th>Tool</th>
<th>Feature</th>
<th>Support</th>
<th>Ease</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEDR</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>SCSI</td>
<td>3.5</td>
<td>3</td>
<td>3.7</td>
<td>2</td>
</tr>
<tr>
<td>LFI</td>
<td>2.5</td>
<td>4</td>
<td>2</td>
<td>2.5</td>
</tr>
<tr>
<td>LTP</td>
<td>4.5</td>
<td>5</td>
<td>4.5</td>
<td>5</td>
</tr>
</tbody>
</table>

A better fault injector tool is the one which supports as many as possible faults injected into it. Kma-alloc, Fail-slab, DiskI/O, Raid and, Char Device were injected in LFI, KEDR and, SCSI to check which tools support which Fault to inject. Table 6 illustrates LFI supported three faults. At the same time, we were succeeded to inject two faults in KEDR and SCSI. Only LFI supports maximum injectors.

Table 6. Fault Based Comparison

<table>
<thead>
<tr>
<th>Tool</th>
<th>Kma-alloc</th>
<th>Fail-slab</th>
<th>DiskI/O</th>
<th>RAID</th>
<th>Char Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFI</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>KEDR</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SCSI</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 7 illustrates the comparison made based on the KEDR-sample-target module. The KEDR exemplary module is verified under different conditions to check whether the KEDR-sample module is supported under different conditions or not. The first task was under normal conditions (no simulator); in this condition, the KEDR-sample target loaded successfully with its device driver but failed to inject the Fault. After that, it was verified in LFI; the KEDR-sample target loaded successfully with its device driver in this condition. However, it didn’t support the injection of Fault. The condition KEDR-sample target loaded successfully along with its device driver in KEDR, with no error.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Successfully load module</th>
<th>Load the char Devices (eth0), (eth1)</th>
<th>Verify the devices</th>
<th>Successfully injected Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>LFI</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Do not support module</td>
</tr>
<tr>
<td>KEDR</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Every test on each Tool is needed for a different space level, as shown in Figure 6. LTP needs maximum space between them while KEDR needs the least space that is 20 GB. SCSI and LFI space depend upon custom kernel configuration, and it is clear that the custom kernel needs more significant than 2 GB so, SCSI needs an additional space of 15 G (25 GB + 15 GB = 40) while LFI requires more significant than 30 GB (25 GB + 5 GB = 30 GB).

The time required for each Tool depends upon the kind of test being tested at that time. As shown in Figure 7, a significant amount of time is required for LTP, some test of under the fault kernel takes about 20 hrs and some test just finished under 1 hr. The least time is required for the KEDR fault injector. SCSI required time is also dependable over the test 1/2 – 3 hr. custom kernel time is processor and configuration dependent if the processor is fast and configuration are less in number than time can be 5 hrs. Still, otherwise, it may lead to ≥ 7 hrs.

5. CONCLUSION

For LTP (Linux Testing Project), we performed three tests under the faulty Kernel: probability percentage and numbers of loops varied each time. KEDR tool was used to study fault simulation and detection of memory leaks overbuilt in the module (Kedr-sample-target) and custom kernel module (Example LKM). SCSI tool was used to inject Fault under the RAID1. The mdadm approach was used to create RAID1.

Comparative experiments have been done based on i) Feature, Support, and Ease ii) Fault support iii) KEDR-sample-target over different conditions iv) Space requirements v) Time requirements of each Tool and reviewing various tools for runtime analysis of Linux kernel modules testing by fault simulation brought us to the conclusion that there are no definitive solutions to this problem.

6. FUTURE WORK

Fault can be injected at a specific kernel module or device driver, while a different fault can be specified each time. However, there is still a need for more research in this area of study because the existing Fault Injection Frameworks and tools can not satisfy the requirement entirely and correctly. To improve and make it more ideal, the goal is to test some real devices such as pcnet32, speaker, camera, etc. This experiment will further help to design a fault injection tools better than the existing one used in this experiments, with aspects to time (designed Tool take less time to evaluate an experiment compared to existing), space (take less space to occupy will installation and injecting a fault over a target), support (must be Linux supportive more than others so it will be easy to set up a platform), design and feature ( must be versatile in feature and design that supports for fault injection over the kernel source codes, actual device drivers and custom kernel indeed) and efficient and comfortable.
REFERENCES


