LDTT: A LOW LEVEL DRIVER UNIT TESTING TOOL

Poorani Dharmasivam and Kristen R. Walcott

University of Colorado, Colorado Springs

ABSTRACT

In the Linux kernel, SCSI storage drivers are maintained as three different levels. Since Low Level Drivers (LLDs) are hardware specific, they are predominantly developed by hardware vendors only. Thus, LLDs are highly error prone compared to other parts of the SCSI stack. While a few tools exist to test upper and middle levels, there is no tool available to assist developers in verifying the functionality of LLDs at the unit level.

We develop a framework for LLD developers for testing code at the function and unit level. The framework, LDTT is a kernel module with a helper application. LDTT allows LLD writers and designers to develop test cases that can interface between the kernel and device levels, which cannot be accessed by traditional testing frameworks. We demonstrate that LDTT can be used to write test cases for LLDs and that LLD-specific bugs can be detected by these test cases.

KEYWORDS

Unit Testing, Driver Testing, Automated Testing

1. INTRODUCTION

Testing is an integral part of the software development process. However, the majority of software testing research evolves around the user realm. Testing of the operating system and lower level drivers is also critical to system testing. Yet, few tools exist to help in system testing.
A test, in general, is made up of a set of inputs and at least one oracle. For example, a test function for a routine that returns the maximum of two integers could be written as follows:

```
int maxi ( int i1 , int i2 ) {
   return ( i1 >i2 ) ? i1 : i2 ;
}

void t e s t_m a x i ( void )
{  CU_ASSERT( maxi ( 0 , 2 ) == 2 ) ;
   CU_ASSERT( maxi ( 0 , -2) == 0 )
   CU_ASSERT(maxi(2 , 2)==2)
}
```

Within the example test case, a single function is tested based on three sets of inputs. The assert statements act as oracles and will cause the test to fail if any of the oracle assertions are not met.

Test cases can be written in many ways and with different purposes. Multiple assertions can be made, or just one.

No unit testing tools currently exist for automated testing of kernel or device-level components. The Linux community has recognized the need for unit test tool for LLD over the last 20 years, and the LLD code base is increased from 0 to 500 K during that time, but there is no available test tool for validating LLD before acceptance into the kernel source tree [19]. Kernel specific limitations currently prevent development of a generic unit testing framework for Linux kernel modules.

In an attempt to perform unit testing at lower programming levels, CUnit and other testing tools provide a set of assertions for testing logical conditions in low level code. The success or failure of these assertions is tracked by the framework and can be viewed when a test run is complete. Each assertion tests a single logical condition, and the test fails if the condition evaluates to FALSE. Upon failure, the test function continues unless the user chooses the 'xxx FATAL' version of an assertion. In that case, the test function is aborted and returns immediately.

To test kernel code, the code could potentially be moved to user space. However, moving the kernel code to user space is not possible due to the strong dependency between the kernel APIs. For example, the memory allocation is performed in kernel code using the function kmalloc() but whereas the same is done at user space using malloc().

However, all of the unit test tools for C are only available for user space application programs. Unit testing from user space using general applications additionally cannot accurately evaluate kernel-level code. For example, if a function in a kernel module needs to be verified, then particular application level calls which execute kernel module functions after passing through many kernel layers need to be identified and executed. Unfortunately, this does not address the primary purpose of unit testing in which tests are aimed at single statements or methods, not integration between components. A new method is required to directly invoke the kernel function with proper prerequisites set up so that the kernel function alone can be executed with varying arguments.

As a final alternative for testing kernel code, existing tools such as C-Unit or CUTest could be ported to kernel space. However, the test cases cannot be executed directly from a kernel module because existing tools assume that the test code execution is either run in a user level process context or a specific kernel thread. Since the linux kernel has modules ranging from storage drivers to network drivers and beyond, developing a common testing framework either in user space or kernel space is costly and challenging. Also, because kernel modules have a large number of interactions with higher and lower hardware and software levels, applying a generic method without knowing the internal communication between kernel modules may not yield a proper test tool.

In this work, we develop a unit testing framework, LDTT (Low-Level Driver Testing Tool), that is based on CUTest, a unit testing tool for C. The tool is implemented as a kernel module for easy inclusion and reuse. The tool, being at the kernel level, allows for the writing of low-level test cases that interact with the operating system and driver interfaces. LDTT also has a user level helper tool so that it can be used in user space more easily.

We evaluate the framework by creating test cases that are specific to low-level driver concerns (for example, as interacting with the queuecommand interface, which is used to send I/O commands to underlying hardware). We then port an existing mutation testing tool to the kernel to mutate low-level driver code. This must be done carefully as improper mutations may cause kernel panic. Mutation testing is performed on driver code only. We show that tests can be written at the kernel level to reveal generated mutants that are driver oriented.
In summary, the main contributions of this work are as follows:

- A description of the challenges of unit testing at the low-level device driver level (Section)
- A kernel space unit testing framework for low-level device drivers based on a user space unit testing tool (Section)
- A Linux SCSI Low Level Driver Unit Testing kernel module and a helper application to configure test cases, schedule, and execute test cases and collect test reports. (Section)
- An evaluation of the tool and related test cases based on mutation testing or fault seeding.(Section)

2. BACKGROUND

The linux kernel has a heavy emphasis on community testing. Typically, any developer will test their own code before submitting. However, they are often testing both their changes and other people's changes combined together and execute comparatively lower amounts of unit level testing due to lack of support tools available in the kernel. In this section, we discuss unit testing, some tools available for unit testing, and generic details about SCSI and the linux SCSI subsystem architecture in order to explain some of the problems associated with unit testing Linux SCSI LLDs and our proposed tool.

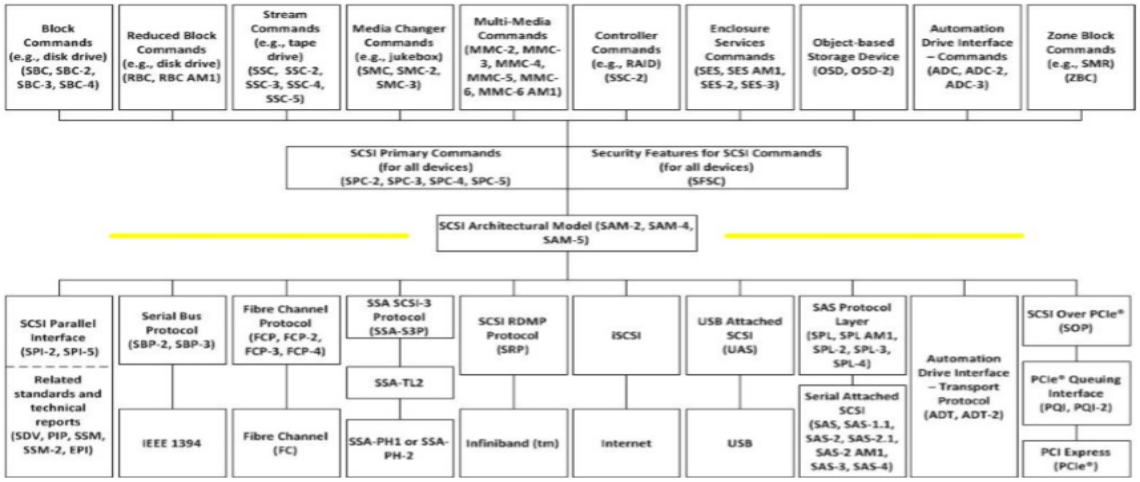

Figure 1: SCSI Architecture Model

2.1. UNIT TESTING

Unit testing is an important part of software development in which developers seek to uncover new software bugs hidden at individual units of the software. A unit may be associated with a single line of code, a function, or, in some instances, a small class.Unit testing can guide the

writing of code and it is useful in increasing assurance that individual lines of code and components work as expected. Individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures are tested to determine if they are fit for use in this form of testing.

There are many tools available to ease the unit testing process [1]. For most popular languages, at least a few tools have been designed. For developing a tool at the kernel level, tools for C are needed. CUnit, a unit testing framework for C [2], is a tool which aids unit testing of C applications. CUnit uses the standard C language, with plans for a Win32 GUI implementation that has not yet been developed. Its testing mechanism is helpful in a kernel realm because it does not currently fork or otherwise protect the address space of unit tests, leading to fewer issues in the kernel space.

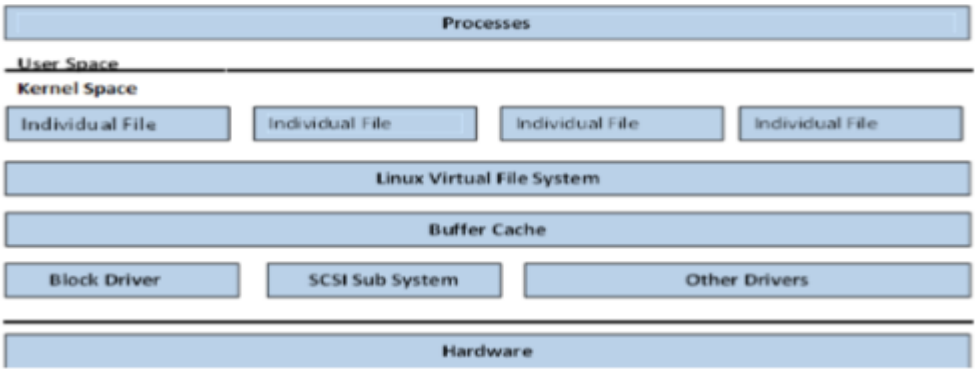

Figure 2: Linux Kernel I/O Stack

CuTest [3] is similar tool that is also used for unit testing C applications. CuTest is simpler than CUnit as it is only requires a single source and header file to be included for use. This simplicity is also useful in modifying the tool for kernel and low-level driver use.

2.2. SCSI (SMALL COMPUTER SYSTEM INTERFACE)

While there are many low-level device drivers, SCSI drivers are some of the most commonly used drivers. SCSI is a collection of ANSI standards and proposed standards which define I/O busses primarily intended for connecting storage subsystems or devices to hosts through host bus adapters.Originally intended primarily for use with small (desktop and desk-side workstations) computers, SCSI has been extended to serve most computing needs, and it is arguably the most widely implemented I/O control structure in use today. The SCSI Architecture Model (SAM) is an ANSI standard that defines the generic requirements and overall framework upon which other SCSI standards are defined.

The first two SCSI standards (SCSI and SCSI-2) were all-in-one standards. They include standards including everything from cables and connectors to protocols to command sets all in one standard. The InterNational Committee for Information Technology Standards (INCITS) T10 Technical adopted a layered standards approach much like the International Organization for Standardization (ISO) Reference Model for networking standards. This approach divides SCSI into multiple layers of standards. The lowest layer deals with physical interfaces (also called transports). The next layer up deals with transport protocols usually directly associated with one

physical transport standard. The top layer consists of command sets associated with specific devices such as disk drives or tape drives.

Figure 1 represents a T10 SAM specification [4] and depicts how the different standards of SCSI are layered in the SAM. From the picture below, it can be seen that majority of the storage devices, whether it is SAS, SATA, SCSI, FC, iSCSI or the latest PCIe all are covered under the big umbrella of SCSI.

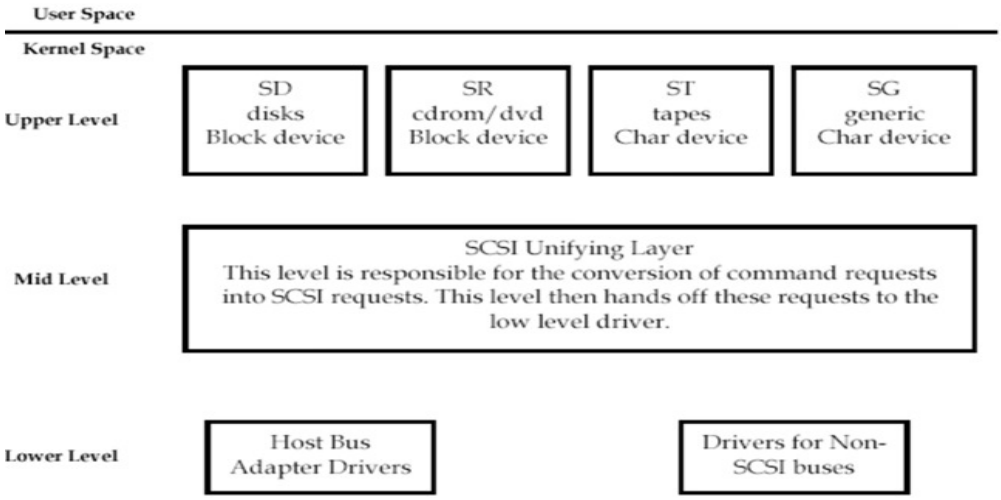

Figure 3: SCSI Subsystem

2.3 LINUX KERNEL'S SCSI DEVICE DRIVER ARCHITECTURE

Another SCSI architecture variant can be represented as a layered SCSI architecture. Operating systems use layered driver structures such as SCSI Common Access Method (CAM). This architecture has one or more class drivers that share one or more host bus adapter drivers. This allows both disk drives and tape drives to coexist on the same parallel SCSI bus. It also permits a disk class driver to control both parallel SCSI and Fiber Channel disk drives. This protects software investment as people migrate to newer physical interfaces.

The Linux kernel I/O subsystem consists of multiple layers of drivers, as seen in Figure 2. The SCSI subsystem is part of the Linux I/O subsystem and handles majority of storage devices that belong to different storage protocol families. The Parallel SCSI (SPI), Fibre Channel (FC), Serial Attached Scsi (SAS) and Serial ATA (SATA) devices are managed by the SCSI subsystem of Linux kernel. The SCSI subsystem is a layer between the File System Drivers and actual storage Host Bus Adapter (HBA) hardware. The SCSI subsystem itself is a layered architecture with three different layers of drivers (or kernel modules). The architecture of the I/O subsystem and SCSI subsystem are represented in Figures 2 and 3.

The SCSI subsystem in Linux consists of three layers namely Upper Level (ULD), Mid Level (SML) and Low Level (LLD) as seen Figure 3. The SCSI subsystem uses a three layer design with upper, mid, and low level layers. Every operation involving the SCSI subsystem (such as reading a sector from a disk) uses one driver at each of the three levels: one upper level driver, one lower level driver and the SCSI midlevel.

The SCSI upper level provides the interface between user space and the kernel in the form of block and char device nodes for I/O and IOCTL. The low level contains drivers for specific hardware devices. In between is the SCSI midlevel, analogous to a network routing layer such as the IPv4 stack.

The SCSI midlevel routes a packet based data protocol between the upper levels device nodes and the corresponding devices in the lower level. It manages command queues, provides error handling and power management functions, and responds to I/O control (IOCTL) requests. The upper level and midlevel drivers are developed by Linux kernel SCSI developers. There will be at least one type of upper level driver for a type of SCSI device. For example the SD driver handles all disk drives and the ST driver handles tape devices and similarly other upper level driver handles a particular type of device. The midlevel driver named scsi mod handles the entire house keeping activities like timing the SCSI requests and handling errors etc and it is a core for the SCSI driver model in Linux. The midlevel driver also is maintained by SCSI developers.

The low level drivers are hardware specific drivers, and they translate the SCSI commands sent by the midlevel into the commands that can be understood by the hardware. Hence the low level drivers are usually developed and maintained by hardware manufacturing companies. The Linux kernel community does rigorous code reviews before accepting driver code submission from the hardware vendor into kernel source tree.

The SCSI subsystem is a major class of device drivers of the Linux operating system. It provides support for peripheral devices that support the SCSI interface, such as high performance hard drives. The code for the SCSI subsystem can be found in the “drivers/scsi” directory of the source code distribution. SCSI support has been present in Linux since the first official release of the kernel in 1994. Like the rest of the kernel, the SCSI drivers are open source. However most of the low level drivers have been developed and then donated by the companies that developed the SCSI controllers themselves. That is, unlike much of the core of the Linux kernel, the original development of most of the low level driver code was generated within a company of full time developers, and not by the open source community.

One noteworthy feature of the SCSI subsystem is that it is designed and implemented as a strict three level architecture: the top and middle levels provide a set of unified and consistent commands for the kernel to control various drivers while the low-level drivers are concrete implementations of this functionality peculiar to the specifics of the particular hardware. Abstractly, the low-level drivers all implement the same functionality, such as hardware initialization, I/O handling, and error checking; they interact with the rest of the operating system only through the upper two levels.

The controlled interfaces between the low level and midlevel and most common functionalities between various low level drivers make low level drivers (LLD) as ideal candidates for unit testing using a generic unit test tool. The unit testing of LLD also has similar limitations to other kernel modules.

In this work, we create a framework to test LLDs at the kernel space level based on existing user level tools. The LLDs are developed by hardware vendors, and testing of the LLDs is typically performed after integrating the driver into the complete stack and by running applications to execute the LLD functionalities from user space. The application calls need to pass through all the layers above LLD to test a particular functionality of LLD, which creates unnecessary

overhead and somewhat nullifies the purpose of unit level testing. If there is any failure, much debugging effort is required to isolate a fault at unit level.

Also, testing a LLD interface thoroughly is not possible today as it is not standalone. Certain functionalities cannot be tested due to restrictions imposed by the upper levels. For example if there is an interface implemented by LLD named queuecommand, the queuecommand interface of an individual LLD will be invoked by the midlevel when it wants to send a new command to a hard drive. The LLD is expected to validate the command and return an error when the command is sent to an unknown device. However, because the SML keeps track of only known devices, that condition can never be tested through SML.

Figure 4: High level interactions between the LLD and SML

Unit testing of LLDs requires two different levels of testing. Unit testing focuses on single lines or sections of code, but it also can deal with small integrations between APIs, devices, etc.... At an integration level of testing, we can test the higher level interactions of the LLD (with SML or ULD). At the same time, we can test interactions with lower levels including the operating system and hardware. Writing a common unit test module that links with the hardware level is not really possible as the hardware interaction, in general, works through proprietary interfaces provided by hardware vendors. Thus, this project focuses on unit testing higher level interactions that deal with how the LLD interfaces with the SML or ULD.

3. IMPLEMENTATION

Low-level drivers are provided by many third party hardware vendors, but no testing tools currently exist to help the unit or integration testing of drivers that execute in kernel space. Such a tool would benefit both the Linux community and the hardware vendors by helping to harness the low level drivers and increasing the overall quality of LLDs. Therefore, we have developed a tool that can exercise various interfaces provided by LLDs with the purpose of identifying issues at the unit level. Integration level test cases will also be considered to assist with testing interfaces between the driver, kernel, and other tools. LDTT will exercise the different units/interfaces of LLD with varying input combinations and make sure the LLD interfaces behave as defined by the Linux SCSI community and as expected by the SML.

The low level driver interacts with SML in two ways. The LLD invokes certain functions of SML and in addition it provides set of interfaces it exposes to SML through a registration process. High level interactions between the LLD and SML are described in Figure 4. As can be seen in the figure, the LLD registers its interface with the SML, which is an upper-level construct. The SML also adds the driver as a host and tries to locate it. The LLD and SML then work together to configure the device and determine I/O requests and error handling.

The LLDs are at the lowest level in the drivers stack just above hardware. Any unit testing needs to be done from one level above LLD, which is at the level of SML. Hence when we developed the LDTT we had to develop it as a kernel module which exists at the level of SML. We also needed an application helper to allow users to configure test cases and execute test cases.

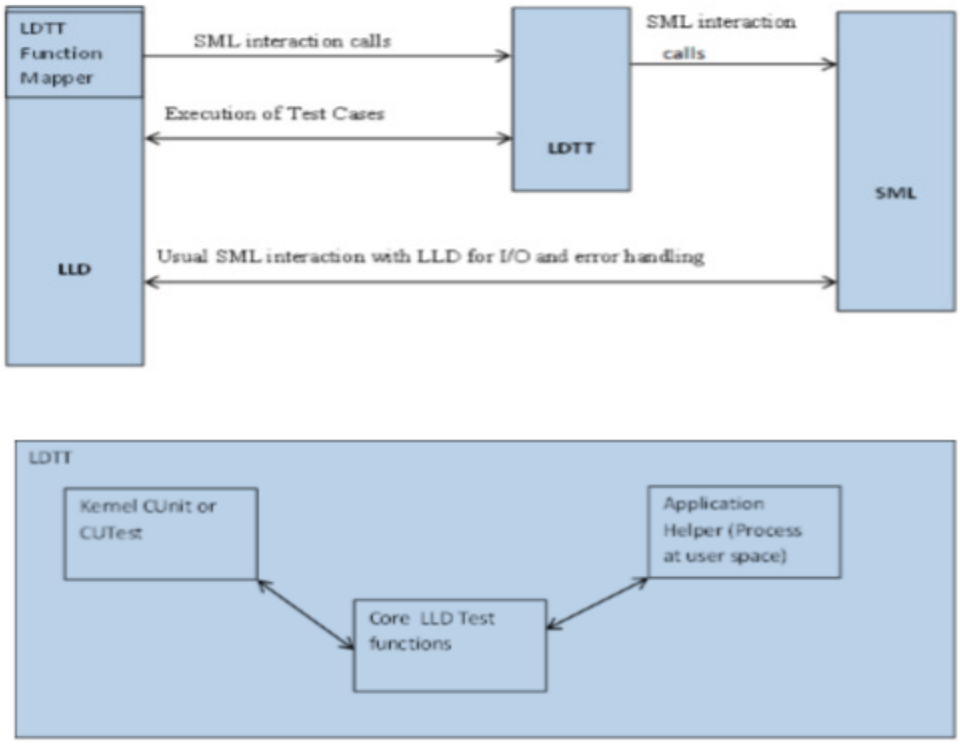

Figure 5: Placement of LDTT in SCSI Driver Stack and Parts of the LDTT

We have developed a new kernel module named LDTT which is layered between the LLD and SML and trap all the communication between LLD and SML. The LDTT module interacts with SML as shown in figure 5. By doing so, the LDTT will get control of the interfaces of LLD and can introduce new test cases or test data and execute LLD in unit test mode. In order to develop this tool, an existing user level C unit testing tool is adapted and ported to the kernel level to work as a part of our LDTT kernel module. The LDTT module is developed based on CuTest ported to kernel level and utilizes the interfaces of the unit test tool for test case management, test suit creation and test status update. Since there is no way to ASSERT in kernel module, the test status is logged into the system logger files.

In the following sections we explain in detail about the CuTest ported to kernel space, the core LDTT module which has defined the LLD SML interaction trapper functions, the character interface to interact with user level applications and the actual user level application.

3.1. KCUTEST THE C UNIT TESTING FRAMEWORK PORTED TO KERNEL SPACE

While doing the background analysis for the tool, we discovered the need for a Linux, kernel level testing tool. We then interacted with several SCSI driver developers through personal e-mail and forums to enquire about the value that could be provided by our tool. We also looked into whether we can use some of the modules in the existing Linux-Test Project, but the Linux-Test

Project has a much defined way of testing and accommodating LLD within their framework. This framework makes the LLD unit testing process heavy. Hence we have decided to create a lightweight tool so that the burden on hardware driver developers will be minimal.

In the beginning of creation of our tool, CUnit and CUTest were considered as possible candidates to be ported to Kernel space. These tools are commonly used at the user level for testing C programs. We initially started with CUnit, but it has many user space dependencies. CUTest has proven much more suitable to our needs and contains similar features to CUnit. Hence we ported the CuTest to kernel level. The primary changes involved while porting the tool are modifying memory management and clearing up memory leaks, the CuTest used malloc and for our KCuTest we had to change all memory allocation calls to use kernel memory allocation (kmalloc) and free (kfree) routines, in addition there were many instances of memory leak in CuTest and that needs to be changed too. In many places in CuTest there were huge local variables defined in the stack and with limited kernel stack we have got compile warnings and needed to remove the local variables and place them in heap. We also had to work around use of "set jump" and "jump" routines as jumping across functions is not advised and not possible in kernel code. The CuTest test case structure definition is also need to be modified to include a reference to the ldtt host instance, so that the test case can be associated with one particular SCSI host.

The KCuTest provides functions to create test suite, create test cases, execute tests and record results. The KCuTest also provides different functions to validate different kinds of assertions and logs the assertion messages into the kernel logger.

3.2. CORE LDTT

We have a developed the core LDTT module which contains four other functional units required for LDTT. The LDTT core module provides the LLD trapper function implementation, IO control support for the applications to manage test cases, a kernel thread per SCSI host to execute test cases and sample test cases to validate certain interfaces of the LLD.

3.2.1. MODULE INITIALIZATION

At the init module function, the LDTT registers itself as character device with miscellaneous driver (misc) provided by the Linux Kernel, we had the option of implementing a completely new character or use the misc driver. The misc driver approach is more clean and easily maintainable, hence we decided to register our ioctl interface with misc driver.

We have currently used a fixed unused minor number for registering with the misc driver and in future based on acceptance in LDTT in Linux community, we could either reserve a minor number for LDTT in kernel source or could request a dynamic minor number. The device node registered for interacting with LDTT from user space through misc driver is /dev/LDTT.

3.2.2. LLD-SML INTERACTION TRAPPER FUNCTIONS

When a LLD needs to be tested using LDTT, the LDTT needs to include the ldtt.h file provided by the LDTT, the header file contains preprocessor directives to map a typical SML interface with LDTT. The LLD interacts with SML through certain function calls to register hosts, request for scanning devices, and remove hosts. Those important SML interactions are routed to LDTT

through the preprocessor level redefinition. The Core LDTT module implements the below SML interaction functions to manage any SCSI hosts which needs to be tested through LDTT.

3.2.3. LDTT HOST

ldtt host is the anchor data structure to be used within ldtt. There is one instance of the structure per registerd scsi host. This structure contains, a pointer to scsi host data structure, a pointer to CuTestSuite, the current state of the testing on the host and a work queue and kernel thread to execute test cases.

```
struct  ldtt_host {
            struct Scsi Host *host _ptr ;
            unsigned char hostptr _valid ;
            unsigned char scan _completed ;
            unsigned char ldtt _state ;
            unsigned char ldtt _num ;
            CuSuite *test _suite ;
            char test _work_q _name [20];
           struct workqueue _struct *test _work _q ;
           struct delayed _work test _work ;
};
```

3.2.4. SCSI HOST ALLOC

The LLD when detects an actual host bus adapter, it requests for allocating host data structure within SML and to do so, the LLD calls the scsi host alloc function, with a host template data structure which contains all the interfaces exposed by the LLD for I/O and error handling. The scsi host alloc returns a pointer to a host data structure, which is an anchor data structure used by any LLD. The SML trapper for scsi host alloc is defined as below

```
#define scsi_host_alloc ldtt_scsi_host_alloc
```

The ldtt host alloc functions internally calls the scsi host alloc and if the scsi host alloc returns success then it allocates ldtt host structure and initialize the state appropriately and store it in a global array of ldtt host pointers and increments the global registered host count value.

3.2.5. SCSI ADD HOST

The scsi add host call from LLD indicates that the LLD wishes to add the host with SML. We trapped it and we set the hostptr valid field in ldtt host and start the kernel thread which is required to execute test cases.

3.2.6. SCSI SCAN HOST

This call allows the SML to scan for the devices connected with a particular SCSI hardware and in this call the scan completed field of ldtt host is set and at this point only the LLD is completely ready to accept any commands to any device and after this call the LDTT can execute tests on a SCSI host.

3.2.7. SCSI REMOVE HOST

This call from LLD indicates the LLD wishes to unregister a host, within the trapper for this, the scan completed, hostptr valid bits will be reset and test thread will be stopped.

3.2.8. SCSI HOST PUT

put This call from LLD indicates that the SML can completely clear a host pointer and associated devices, the trapper in LDTT, cleans up the internal reference of a host and clears up all memory Associated With Ldtt Host And Its Associated Test Suite And Test Cases.

3.3. I/O CONTROL (APPLICATION/IOCTL) INTERFACES

The core LDTT implements the below mentioned interfaces which can be used by an application to manage testing through LDTT.

3.3.1. LDTT GET HOSTS

This interface provides the number of SCSI hosts registered with LDTT for testing and provides the host number and name of the each individual hosts registered with LDTT. The associated data structure is as below

```
struct ldtt_get_hosts {
    unsigned char num_hosts;
    unsigned char host_num[LDTT_MAX_HOSTS];
    unsigned char host_name[LDTT_MAX_HOSTS][20];
}
```

3.3.2. LDTT GET TESTS

This interface provides the number of test cases currently available in the LDTT and the test case number and name of the each test case. The associated structure is

```
struct ldtt_get_tests {
    unsigned char num_tests;
    unsigned char test_num[MAX_TEST_CASES];
    unsigned char test_name[MAX_TEST_CASES][100];
};
```

3.3.3. LDTT GET HOST STATE

This interface provides the state of a particular ldtt host. The application needs to provide the host number to retrieve the state, the associated data structure is

```
struct ldtt_get_hoststate {
    unsigned char host_num;
    unsigned char host_state;
};
```

3.3.4. LDTT QUEUE TESTS

This interface configures the tests to be executed, it allocates test suite and adds the user selected test cases to the suite and set the state of the particular LDTT host to TESTS QUEUED, so that the kernel thread can execute the test cases and collect the status. The application needs to provide the host number and selected test case numbers, the associated data structure is

```
struct ldtt_queue_tests {
    unsigned char host_num;
    unsigned char num_tests;
    unsigned char test_num[MAX_TEST_CASES];
};
```

3.3.5. LDTT GET RESULTS

This interface collects the results from previously executed test and frees up the test suites and associated test cases. It sets the state of the particular LDTT host to TESTS NOT STARTED, so that the new test cases can be configured to run. The application needs to provide the host number and this function will return the number of test cases executed and pass/fail result of each test case. More details about the failures are logged in the kernel logger, the associated data structure is:

```
struct ldtt_queue_tests {
    unsigned char host_num;
    unsigned char num_tests;
    unsigned char test_num[MAX_TEST_CASES];
};
```

3.4. TEST EXECUTION THREAD AND LDTT TEST APPLICATION

Test execution thread is a kernel thread which polls the state of the LDTT host for a configurable period of time and if the LDTT host state indicates new test cases are queued, it executes those test cases, collect individual test case results and collect the details to the level of line of failure. After the execution of test cases, the thread sets the LDTT host state either as Test Passed or Test Failed.

A test case is a function of prototype void testcaseX (CuTest *Tc) where CuTest is the test case data structure defined by CuTest with an added reference to LDTT host. The test case should have one CuTest Assertion. The function pointers of all the defined test cases need to be added into a global array of test cases with appropriate name. The array will be used to provide the test case details when an application requested for the same through ioctl call.

A sample test case and the test case global array looks as below, when an LLD developer extends this framework by adding more test cases, they need to stick with the prototype and add the new test cases in the global array. Currently the tool supports fifty test cases and that can be changed by redefining the MAX TEST CASES macro.

```
void qcmd_positive (CuTest *tc)
{
<other required code>
  rtn = tc->ldtthost->host_ptr->hostt->queuecommand
  (tc->ldtthost->host_ptr, scmd);

        CuAssert(tc, "Qcmd return failure", rtn == 0);
}
```

Array of test cases can be represented by the following:

```
struct ldtt_tests LDTTTC [ ] =
{
    { "QcmdPositive", qcmd_positive },
    { "QcmdNegative", qcmd_negative },
    { "AbortPositive", abort_positive },
    { "AbortNegative", abort_negative }
};
```

As part of the implementation, we have developed a sample test application which queries the registered hosts and configures test cases on the first host and waits for test execution and collects and prints the result of the same. If some LLD developer wants to extend the application with more configurable options, they can develop based on the sample test application

4. EVALUATION

We have validated the LDTT implementation against the three major objectives;

1.The functional verification of LDTT: We wanted to ensure our LDTT implementation is completely functional and one or more test cases can be constructed using the LDTT tool

2.The LDTT advantage: We wanted to ensure our LDTT implementation is having an edge over other existing tools in terms of validating a unit of LLD which is not possible with any of the existing tool.

3.The effectiveness of test cases developed using the LDTT: We wanted to ensure the test cases we have developed using LDTT are able to identify the mutants or faults injected purposefully in the unit under test.

The functional verification of LDTT requires one or more LDTTs to be used as test code. The SCSI debug driver is a sample LLD provided within Linux kernel and exposes few RAM disks through a SCSI controller. This driver could load and expose few devices without needing any actual hardware.

The LDTT trapper header file ldtt.h is added into the SCSI debug source tree and SCSI debug is compiled with LDTT headers. Once the scsi debug driver is compiled with LDTT headers all the SML interaction from scsi debug will be directed to LDTT. Compiling scsi debug with LDTT headers make it dependent on the LDTT module. So the LDTT needs to be loaded prior to scsi debug driver. For the functional evaluation, we have compiled the scsi debug with LDTT, and then loaded LDTT and scsi debug drivers into the kernel space. The scsi debug by default exposes one pseudo SCSI host controller with two RAM disks attached with it. We have observed that the scsi debug with mapped LDTT functions can be successfully loaded into the kernel space

and able to see the SML calls directed to LDTT and the LDTT is able to trap the communication. The LDTT is able to create a test reference for the scsi debug host successfully.

Once the scsi debug module and LDTT are loaded successfully, we have executed our test application. The test application is able to query the number of hosts registered with LDTT and list the scsi debug along with proper name. We can also query the list of test cases available in LDTT through the test application. We then Using the test application scheduled few test cases to be executed and enabled tracer prints in scsi debug and observed the test cases are successfully getting executed by the test execution test and the code path of scsi debug displays the tracer prints.

Positive and negative test cases for the queuecommand , abort task, target reset and host reset LLD interfaces are executed successfully. The test cases are executed successfully without any kernel panic and the test case results can be retrieved through the test application. Using the kernel logger we can observe the details of the functions executed and the detailed results.

We would like to validate the LDTT with real hardware drivers however obtaining actual SCSI host controllers for this testing is time and cost prone. The guest operating systems running in virtual machines expose one or more pseduo host controllers and the hardware drivers for the operating system can be executed with the pseudo host controllers as similar to running on real hardware.

We have decided to use the pseudo drivers in guest operating system for our second level evaluation. We have used Oracle Virtual Box which exposes a pseudo LSI SCSI Parallel Interface (SPI) host controller and we are able to load mptspi driver provided in the Linux kernel source tree. We have compiled the mptspi driver with the ldtt.h file and loaded it successully. While the LDTT is loaded and scsi debug host is registered with LDTT for testing, with this setup we were able to successfully register two different SCSI host with LDTT and able to execute test cases in parallel on both. We have unloaded one of the drivers and made sure test cases are executed on proper controller, this is done to prove the hosts can dynamically register and unregister with LDTT for testing. Throughout our validation we didnt observe any kernel panic or any unexpected results, this confirms the basic framework is functional and using LDTT we are able to execute tests one more than one SCSI controllers as defined in this project.

5. EVALUATION OF THE LDTT ADVANTAGE

During the conceptualization and background analysis for the project, we wanted to make sure whether this project will be a useful project and whether it provides more value than other currently available tools. The closest possible tool for LDTT is the sg3 utils [5], the sg3 utils is the user level tool for sending I/O and other kind of commands to a host controller or to an attached drive from user level application. When sg utils sends SCSI commands, the commands pass through different layers of the SCSI driver stack similar to a normal file system I/O. However the sg3 utils provides variety of test options beyond normal I/O and typically used by many LLD developers for validating different interfaces exposed by their LLD.

We have decided to compare our tool against sg3 utils and we verified whether all the functions provided by sg3 utils can be executed through LDTT. During our evaluation we found that out we are able to cover majority of sg3 utils function with the exception of validating I/O passed from user space because the LDTT generates all I/O requests from kernel space. Our comparative

study between sg3 utils and LDTT proves that at least there is one important test method which is not available in sg3 utils can be supported in LDTT.

Abort handler interface provided by LLD is an interface which is executed very frequently to recover any timed out I/O request. However there is no tool including sg3 utils provides a method to send an abort request to verify the code in abort handler directly. The best available method existing today to validate abort request is to seed an error at I/O protocol level in the hardware to create I/O timed out condition. The error seeding requires additional hardware like protocol analyzers and error injectors. Using LDTT, the abort handler testing can be executed in a less complex way. The LDTT abort test case issues a new SCSI I/O request to the driver under test and immediately executes abort task handler provided by the driver to abort the command.

We are able to validate we can write test cases to send abort command without any change at hardware level. We executed positive abort test cases across both scsi debug and mptspi drivers. Another advantage of LLDT is to send a malformed SCSI commands directly to LLD bypassing the usual validation checks done at SML and Upper levels on a SCSI command sent by the sg3 utils. We were able to the change different fields of SCSI commands and observed that we hit failures at LLD level.

Through our testing we have proved the LDTT has advantage over other existing tools and since it is implemented at a level very close to LLD and have unlimited access over the functions exposed by LLD and able to execute test cases which is otherwise not possible without LDTT.

There is no standard test suite available for validating of Linux kernel modules especially the LLD. The evaluation of LDTT requires developing various test cases to exercise all possible interface of LLD. We wanted to prove the developed test cases are valid enough to identify defects in the driver, thus we decided to add mutants in the LLD interfaces and to verify whether our test cases identify and kill the mutant.

For generating mutants, we tried to use the mutation tool MiLu [6]. We tried it on scsi debug.c which is a sample driver used with LDTT. We tried to create a mutant for the function ”scsi debug queuecommand lck”. This did work partially and the problem is the mutated code was not having some important data structure pointers and initializations. The first thing we noticed was that MiLu requires the source file to be run with for example “gcc -E”. This is stated in the information about MiLu.

This means that the source file needs to be run through the preprocessor before MiLu is able to mutate the file but the problem is we are compiling the kernel code and we do not have all the headers require for generic preprocessing to be successful. Another observation we had is the dowhile statements were removed. So we looked into this and found that MiLu lacked support for do while statements, this is quite important since many kernel macro use do-while. Hence we were not able to use the tool for our evaluation and the efforts required to fix the tool is estimated as very high.

We have decided to use another simple tool named mutate.py which is a simple python script from [7] for mutation. We had to modify the tool to generate mutants within a specific function block, the tool generates only one mutant at a random number of line within the code block. The mutant type is determined based on the line of code which is getting mutated.

We have developed ten to fifteen test cases around the commonly used LLD interfaces like queuecommand, abort task, device reset, target reset and host reset. We have executed the test cases on mutated code. In the process of evaluation using mutate.py we found that there are many mutants created are equivalent mutants and the outcome of testing is not modified by the equivalent mutants. Only the logical operator mutation yields few defects.

In the abort task interface testing with mptspi driver when we mutated certain logical operator checks we end up in kernel panic which shows the test case is able to hit the mutant. In addition to the mutants, we manually added debug prints in LLD under test and confirms the code path is covered to a great extent.

For example, if you take queuecommand interface, the expected return value of the interface for successful queuing of I/O request is 0 and all other failure cases the queuecommand has to return non -zero values. We wrote a test case to validate whether a wrong invocation of queuecommand will properly return error value but to validate whether our test properly reports the error we need to modify the queuecommand interface under test to return 0 for failures and make sure the test case prints the failure properly.

6. THREATS TO VALIDITY

All our evaluation is done through a sample RAM disk driver and pseudo LLD on virtual machines, we didn't have a chance to run our tool against actual SCSI HBA and associated driver. We may some issues related with DMA, physical memory access, hardware access and actual Interrupt Service Routines (ISRs).

Another threat to our evaluation is we have used a very minimal mutation tool and focused on manual fault seeding for the evaluation of sample test cases. Though we tried to refrain from mutating based on test cases, there is a high possibility that our mutants are biased towards our test cases. However the basic core functionality of the LDTT and the usefulness of the LDTT with the available drivers are proved.

The LDTT generates all I/O requests from kernel level and typical I/O request to LLD is generated at user level and that impose a thread to validity our tool at the current version and we would like to address this in our future work.

7. RELATED WORK

There are numerous Unit testing frameworks and tools available for various different languages. The [1] lists majority of available unit testing frameworks.

Most of current research is happening on unit testing framework for object oriented languages especially Java. The Junit [7] is most famous framework available for unit testing Java based object oriented code and researches are happening in the direction of optimizing test suite based on automated unit testing of Java code using Junit. Junit way of unit testing [8] and unit testing concurrent programs using Junit [9] .

There are some existing frameworks for unit testing C code, but there are no noticeable researches which we are aware in unit testing for C code. One of the famous available unit testing frameworks for C is Cunit [2] . The CuTest is a simple and lightweight framework [3]. However

the existing majority of frameworks are available in user space and good in unit testing user level programs but not kernel code.

The kernel itself and its parts are tested prior to the release, but these tests cover only the basic functionality. There are some testing systems which perform testing of Linux Kernel:

The Linux Test Project (LTP) delivers test suites to the open source community that validates the reliability and stability of Linux. The LTP [15] test suite contains a collection of tools for testing the Linux kernel and related features

Autotest – a framework for fully automated testing. It [10] is designed primarily to test the Linux kernel, though it is useful for many other purposes such as qualifying new hardware, virtualization testing, and other general user space program testing under Linux platforms. It's an open-source project under the GPL and is used and developed by a number of organizations, including Google, IBM, Red Hat, and many others.

Additionally, there are certification systems developed by some major GNU/Linux distribution companies. These systems usually check complete GNU/Linux distributions for compatibility with hardware. There are certification systems developed by Novell, Red Hat, Oracle, Canonical, Google.

There are also systems for dynamic analysis of Linux kernel:

1) Kmemleak is a memory leak detector included in the Linux kernel. It [13] provides a way of detecting possible kernel memory leaks in a way similar to a tracing garbage collector with the difference that the orphan objects are not freed but only reported via /sys/kernel/debug/kmemleak.

   Kmemcheck [14] traps every read and write to memory that was allocated dynamically (i.e. with kmalloc()). If a memory address is read that has not previously been written to, a message is printed to the kernel log. Kmemcheck is also a part of Linux kernel.

2) Fault Injection Framework (included in Linux kernel) allows for infusing errors and exceptions into an application's logic to achieve a higher coverage and fault tolerance of the system.

Though there are complex test suits like LTP, Autotest are available, there are thousands of combination of devices, kernels, device drivers possible for testing and testing everything is beyond the scope of a centralized automatic test suite, hence majority of Linux kernel testing is dependent on developers who develop a particular kernel module or device drivers.

As stated previously, the LLD are developed by developers employed in hardware vendor organization and dedicating their time in validating using complex framework like LTP is highly impossible and the LLD tester is light weight and specifically targeted for LLD.

Lachesis [12] a testsuite for Linux based real-time systems is designed with portability and extensibility as main goals, and it can be used to test Linux, PREEMPT RT, RTAI and Xenomai real-time features and performances. It provides some tests for SCHED DEADLINE patch, too. Lachesis is a similar active research however it is focused on complete real time OS system testing of complete kernel.

Junaid et all [11] proposes present a comprehensive model for executing unattended tests covering both the host side build system testing and target side runtime testing including benchmarking of an embedded Linux but that paper covers overall embedded system benchmarking and not unit level testing of SCSI stack.

Martin et al [18] examine Linux device driver used on IBM mainframes and focus on the misuse of unsigned integers. First, we develop a specialpurpose tool which analyzes the parse-tree of the source code. It reported easy cases of misuses immediately but it is focused only on a specific case.

Avinux is an integrated software verification tool [17] chain that comes as an Eclipse2 plugin which significantly improves the automation in Linux Device Driver code analysis and error checking.Hendrick et al successfully used Avinux for the automatic analysis of Linux device drivers reducing the immense overhead of manual code preprocessing, but it does the verification at code level and not running actual tests.

In this paper [16] the authors aim at porting the verifications techniques developed by Microsoft for SDV(Static Driver Verifier) to the Linux kernel project for Integrated Static Analysis for Linux Device Driver. This does testing by annotating drivers and it is generic verification of device drivers and not focused on specific aspects of LLD.

8. CONCLUSION

In this project, we developed a unit testing tool for the SCSI Low Level Driver in Linux Kernel; The tool is based on CuTest and is ported into the Linux kernel to create a new framework for testing between the operating system, low level drivers, and hardware. We developed the complete infrastructure to test the LLD and we have developed sample test cases and a sample test application. To do this, we incorporated two SCSI Low level drivers available. In our evaluation, we observed that LDTT can be used to write test cases to cover LLD specific mutations based on common LLD concerns.

In future work, we will evaluate LDTT with non-virtualized SCSI hardware to show our tools can be used when working between hardware, the kernel, and drivers. We will also develop more sophisticated test cases to cover various types of hardware and drivers and provide our tool as single package. Finally, because LDTT operates mostly at the kernel level, we also nee to ensure that no security holes are introduced by this framework.

REFERENCES

[1] http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks.

[2] http://cunit.sourceforge.net/.

[3] http://cutest.sourceforge.net/.

[4] http://www.t10.org/members/w_sam5.htm.

[5] http://sg.danny.cz/sg/sg3_utils.html.

[6] http://www0.cs.ucl.ac.uk/staff/y.jia/Milu/.

[7] http://junit.org/.

[8] http://www.ist.tugraz.at/teaching/pub/Main/QS/cheon-leavens04_jmlunit. pdf, November 2007.

[9] http://www.cs.rice.edu/~javaplt/drjava/papers/ PPPJ2009-Ricken-ConcJUnit.pdf, November 2007.

[10] J.Admanski and S. Howard. Autotest-testing the untestable. In Proceedings of the Linux Symposium, 2009.

[11] M.J.Arshad, A. Anwar, and A. Hussain. Automated framework for validation of embedded linux build system in continuous integration environment. JOURNAL OF FACULTY OF ENGINEERING & TECHNOLOGY, 20(1), 2013.

[12] A.Claudi and A. F. Dragoni. Lachesis: a testsuite for linux based real-time systems. In 13th Real-Time Linux Workshop, Prague, Czech Republic, 2011.

[13] J.Corbet. http://lwn.net/Articles/187979/.

[14] J.Corbet. http://lwn.net/Articles/260068/, November 2007.

[15] S.Modak and B. Singh. Building a robust linux kernel piggybacking the linux test project.

[16] H.Post and W. Ku¨chlin. Integrated static analysis for linux device driver verification. In Integrated Formal Methods, pages 518–537. Springer, 2007.

[17] H.Post, C. Sinz, and W. Ku¨chlin. Avinux: Towards automatic verification of linux device drivers. In Submitted to: Computer Aided Verification, 19th Intl. Conf., CAV, 2007.

[18] M.Rathgeber, C. Zengler, and W. Ku¨chlin. Verifying the use of unsigned integers in linux device driver code. 2011.

[19] W.Wang and M. W. Godfrey. A study of cloning in the linux scsi drivers. In 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM),pages 95–104. IEEE, 2011.