The Spruce System: quality verification of Linux file systems drivers

Karen Tsirunyan
Russian-Armenian (Slavonic) University, RAU
Yerevan, Armenia
ktsirunyan@gmail.com

Vahram Martirosyan
Russian-Armenian (Slavonic) University, RAU
Yerevan, Armenia
vmartirosyan@gmail.com

Andrey Tsyvarev
Institute for System Programming of RAS
Moscow, Russia
tsyvarev@ispras.ru [1]

***Abstract* — This paper is dedicated to the problem of dynamic verification of Linux file system drivers. Alongside with some existing solutions, the Spruce system is presented, which is dedicated to verification of drivers of certain Linux file systems. This system is being developed in the System Programming Laboratory of Russian-Armenian (Slavonic) University in Armenia. Spruce provides a large variety of tests for file system drivers. These tests help not only verify the file system functionality, but also watch the behavior of the driver in case of system failures and in rare paths.**

***Keywords* - Linux, file system, driver, Spruce, KEDR, fault simulation.**

I. Introduction

Linux-based operating systems are widely used all over the world. Linux supports several file systems, including not only own file systems (ext2, ext3, ext4) and those of other Unix-like OSs (XFS, BtrFS, JFS), but also MS Windows file systems (FAT32, NTFS). As Linux is a Free Software, it is not developed by a specific company or a developer, but rather by a community of developers, which includes thousands of system programmers. Each kernel module is developed by a certain programming team, which is responsible for its technical support. This brings us to the situation where there is no centralized quality control of source code. This makes it possible for some kinds of errors to appear in Linux. For example, no one performs integrity verification of the system in its entirety. That can usually be avoided in other operating systems, which are developed and maintained by certain companies.

File system modules are one of the most widely used components of Linux. Each Linux user needs these modules to operate correctly. Errors in file system drivers can lead to serious consequences - from data distortion and loss to critical security vulnerabilities. The change log of Linux kernel [1, 6] shows that errors in file system modules are still quite common.

Linux is a monolithic OS kernel with support of dynamically loadable modules (Loadable Kernel Modules – LKM). Usually there can be hundreds of threads working concurrently in Linux. It leads to the well-known problems of race conditions. Besides, there are no protection mechanisms between modules and the kernel itself. All the code is being executed in privileged mode which allows any of the modules to have direct access to all the others modules and the kernel. Also there is no garbage collector in Linux. All these factors make Linux module development a very hard process from the point of view of correctness. It is really easy to make mistakes in such situations but it is really hard to find those mistakes later.

Linux file system modules usually consist of two parts: common functionality and FS-specific part. The common functionality implements the back end of general-purpose system calls while the FS-specific part implements such operations as defragmentation (on-line or off-line), partition resize, migration, etc. Each of these parts in its turn consists of two parts: normal execution and error checking and handling.

Because of the complexity of Linux kernel modules there are several kinds of critical errors usually found there. First of all there can be fatal errors, which make the module no longer operational (e.g. dereferencing a null pointer). These errors usually occur because of low quality code; for example, when the error checking and handling is not there.

There are certain resources (such as memory or system objects) which need to be manually returned to the system; otherwise a leak of resources in the kernel modules is said to have occurred. The reason for the occurrence of such errors is that Linux is developed in “C” programming language which does not have its own garbage collector. On the other hand the kernel itself is not responsible for freeing resources after the module is unloaded. Next, as there are multiple threads working in the kernel concurrently, the race conditions are usual. It is one of the hardest errors to discover in kernel modules. This problem is troublesome even in user-space.

Also, there is another kind of errors in kernel modules which probably are the easiest to find and correct – incompliance with the documentation.

1 This work is partially supported by RFBR 10-07-1047a, 11-07-12075-ofi-m, Minobrnauki RF 07.514.11.410.

Testing is one of the existing methods of maintaining the desired quality level of software components. Naturally, the kernel itself and its parts are tested prior to the release, but these tests cover only the basic functionality. A system which allows to check a file system driver more thoroughly and to reveal above errors would be useful for quality assurance of Linux-based OS. This system would be used both by developers and maintainers of existing file system drivers, and by developers of the new ones.

II. Overview

Let us briefly present some of the most popular Linux testing systems.

Autotest [7] is a framework for fully automated testing. It 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.

The Linux Test Project (LTP) [8] is a joint project started by SGI TM and maintained by IBMR, that has a goal to deliver test suites to the open source community that validate the reliability, robustness, and stability of Linux. The LTP test suite contains a collection of tools for testing the Linux kernel and related features.

The Phoronix Test Suite (PTS) [9] is the most comprehensive testing and benchmarking platform available that provides an extensible framework for which new tests can be easily added. The software is designed to effectively carry out both qualitative and quantitative benchmarks in a clean, reproducible, and easy-to-use manner.

All of the testing systems mentioned above cover only that part of file system drivers, which is responsible for normal execution. These systems do not test the driver behavior in case of system failure and other rare execution paths.

Along with the testing systems 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***. Because operations with hardware are performed via drivers corresponding to this hardware, these systems also check drivers.

Most certification systems for Linux simply use testing for verification of hardware. They use external test suites, including Autotest, LTP and PTS, and test suites specially developed for a specific system. So, such systems suffer from the same problems as described above for test suites.

But some certification systems pay more attention to checking device drivers.

While testing storage hardware, ***SUSE Yes Certified Program***[12] examines the work of its driver for memory leaks (chunks of memory which have been requested by the driver, but have not been freed by it) and accesses memory areas outside of allocated ones. Also, some tests are performed in a mode in which some memory requests from the driver may return failure. Such tests verify the operability of the driver in case of memory pressure.

For checks and for memory pressure emulation instrumentation of driver object file is used. Allocation/deallocation function calls are replaced with calls of special stubs.

Certification program ***Oracle Linux Test (OLT)***[13], beside testing of OS in normal conditions, also performs testing in conditions of system-wide memory pressure.

The checking of internal properties of driver work during testing gives much for quality driver verification in comparison with testing only.

For example, the rate of leaked memory per one device operation may be low. So, for revealing a leak, a normal test should perform many device operations before the total memory leak becomes sufficient for making the test fail. Indeed, simply increasing the number of device operations may be insufficient for triggering test failure, because not all operations may cause memory leaks (and even not in any condition).

Checking memory leaks while the driver is working changes the situation dramatically. In that case a single operation is guaranteed to be sufficient for revealing memory leaks. Moreover, error reporting in that case becomes more informative than that in case of test failure caused by memory exhaustion.

In SUSE Yes Certified program internal properties of driver work are checked for memory leaks and write-past-end/write-before-begin errors. The disadvantages of the implementation are: small number of intercepted allocation/deallocation functions (this leads to missing memory leaks) and inability to reuse implementation of those checks separately from the certified program itself.

Testing under memory pressure also improves the quality of driver verification, allowing triggering driver code which is responsible for error-processing. This, in turn, allows for verification for otherwise unexecuted code.

Linux kernel and kernel modules, which implement drivers, request memory from a common pool, and are strongly interconnected with one another.

Because of this, system-wide memory restriction applied by Oracle Linux Test is not very effective for single driver testing – there is no guarantee that concrete failure in memory allocation affects the given driver.

Memory restriction simulation based on making only those requests fail, which are performed by the driver, is much more effective. Every such failed request affects the guarantees of driver execution.

There are the following disadvantages of such simulation implementation in SUSE Yes Certified program: restriction in simulation scenarios choice (only scenarios based on random generator are available), and again inability to reuse implementation separately from the certified program itself.

The certification systems mentioned above are summarized in Table1.

TABLE 1.

	Whole modules checking	**Tests**
Suse	Exists	Own
Red Hat	not exists	Own
Oracle	Only out-of-memory imitation in a whole system	Own
Canonical (Ubuntu)	not exists	mainly external(LTP, Phoronix, ...) + own shell
Google (Chrome OS)	not exists	own + external(LTP, Autotest, Unixbench, ...)

There are also systems for dynamic analysis of Linux kernel.

Kmemleak is a memory leak detector included in the Linux kernel. It 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 will trap 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 part of Linux kernel.

Fault Injection Framework which is 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.

KEDR Framework [5] is an extensible system for dynamic analysis of kernel modules (device drivers, file system modules, etc.) in Linux on x86 systems. KEDR tools operate on the modules chosen by the user and can detect memory leaks, perform fault simulation as well as other kinds of data collection and analysis. KEDR-based tools have already proven their effectiveness by finding errors in several widely used kernel modules [3]. KEDR framework is Free Software and is distributed under the terms of GNU General Public License Version 2.

All of these tools have different abilities. None of them is strictly superior to the others. The advantages of the KEDR Framework include the following:

- Operation with a specific module.
  On execution KEDR waits for the target module to be loaded into the kernel. On module load KEDR finds out the calls of the kernel functions from the target module and replaces them with calls to the corresponding functions from the KEDR framework.
- Possibility to define well-tuned scenarios of fault simulation.
  KEDR tools support configuration files which can define some specific conditions for the fault simulation to be activated. For example, it can simulate failures for a single kernel function with some probability or exact frequency or even when some conditions on function arguments are met.
- Possibility to extend the functionality. KEDR framework and KEDR Tools can be extended by new call monitors, fault simulators, trace analyzers etc.

Our goal is to verify both the normal functioning of the drivers and their behavior in critical situations. The revealing of driver errors is vital, insofar as a single error in the driver can result in the failure of the whole system. Additionally, approximately half of errors found in the kernel are statistically in the drivers and file systems [9].

III. Spruce System

All the testing systems mentioned above are operating only with the part of normal execution of FS modules (they verify the kernel drivers' functionality according to documentation).

One of the most important parts of the drivers – the one that is not covered by existing systems – is the error checking and handling. It is clear that any function call (in our case kernel functions) can fail. For example, there can be lack of resources (memory, internal kernel objects) which makes resource allocation functions fail. In theory, everyone should check the return values of all the functions called in code. In practice, however, developers often forget to add checks and handle error cases. This is very dangerous in case of drivers, since an unhandled error can result in the corresponding driver module becoming unloadable or disabled. This, in turn, will lead to undefined behavior of the corresponding device.

It is therefore very important to be certain that the file system driver handles all possible errors and faults.

The Spruce project [2, 3] is designed to verify several Linux file system drivers, including Ext4, BtrFS, XFS, JFS. The system consists of several modules.

Every module has two execution scenarios – normal execution and fault simulation. In normal execution mode, the module verifies the functionality of the driver according to the documentation. In this case each error in the driver is deemed as a test failure. This case is more or less included in existing testing systems.

The main advantage of the Spruce system is that it can also cover the error checking and handling part of the code, which comprises about one third of the whole source code.

Below is the description of the Spruce system modules:

- ***Main module***. Implements the user interface. Allows to define some configurable values (which modules should be executed, which file system drivers should be checked, where to store the execution log).
- ***System call checker***. This is the module which provides the major part of the code coverage. It verifies the system calls which concern to the file systems such as *creat, open, fsync*. The verification is done based on the POSIX manual pages. This module covers almost all the code except some of the error handling lines. (Only with normal execution scenario).
- ***Common operations***. This module checks the functionality of the common system utilities which concern to the file systems such as *cp, mkdir, ln*.
- ***Benchmark***. This module provides benchmark testing of a wide variety of operations including creating and removing large files, compression and decompression of files, reading and writing large amount of data etc.
- ***File system specific modules***. This is not just a module but a set of modules, which cover all the FS-specific code in the corresponding kernel modules. The specific features include online resize, online defragmentation, delayed allocation, migration from other file systems.

The Spruce system is implemented mainly in C++ language using object-oriented design. Each test is executed in its own process. It makes the whole system much more stable. If any of the tests crashes or runs for too long it will not affect other tests. The parent process takes care of the test.

Each module can be configured to be executed in different ways. For instance, the user can decide which system calls should be verified, or which test should be performed.

In an ordinary environment, it is almost impossible to simulate fault situations, error paths, rare execution paths etc. In order to solve this problem the KEDR framework is to be used.

There are of course several tools which could also be used to achieve this goal. They are KmemLeak, KmemCheck and Fault Injection tools provided by the Linux kernel itself. Let's see why KEDR framework is preferable to these tools. Verification of a kernel module (in our case a single file system driver) must be done in separation from the other parts of the system. This means that if something must be modified in the system it had better concern only the module to be verified.

Except for the module-based issues, Spruce system needs some mechanism to make the file system driver execute all the rare execution paths and error handling parts of code. This means that there is a need for some kernel functions to fail under certain conditions. For example, to make the file system driver execute all its memory allocation error handlers, it is not enough to make all the calls of *kmalloc* (and similar functions) to fail. In that case usually only the first error handling code would be activated. That is because such errors (memory allocation failures) make the driver's current function execution impossible. Such error handlers usually stop the function execution, returning some error code to the user.

Besides the memory allocation failures there can be other kinds of kernel functions which would also need to be simulated. This means that there can be need to extend the set of the supported functions.

It is usual for Linux file system drivers to check for some capabilities prior to the operation execution. That is why some parts of the file system driver source code are really rarely executed, because users usually have some basic capabilities. So Spruce needs the corresponding kernel function (*capable*) also to be simulated.

The list of the necessary functionality is quite similar to the feature list of the KEDR Framework and KEDR-based Tools. It makes KEDR really suitable for the Spruce system. On the other hand, all the other kernel module analyzing tools provided by the Linux kernel cover only some of the presented needs. That's why KEDR was chosen as a supporting framework for the Spruce system.

With KEDR, one can artificially simulate memory allocation errors (and potentially any other errors in kernel functions used by the driver). KEDR can be configured to make the driver pass through all errors which would be impossible in normal execution. Nevertheless, in normal execution mode, the Spruce system cannot cover more than 70% of file system driver code. This is because in any well-designed and well-implemented system, approximately one third of the code is dedicated to error checking and handling. However, the system call testing module covers some error cases, since it analyzes a number of argument value sets for system calls. So with KEDR-based tools Spruce system could be able to cover also part of the remaining 30% of code.

After analyzing all the above mentioned aspects of file system driver verification, the quality verification system can be defined. Such a system must make the driver pass through all the possible execution paths (even those not developed in the source code). This means the following:

- Make the driver perform all the normal execution paths.
- Make the driver operate in out-of-resources and other faulty situations to make sure that the driver does not fail.
- Make the driver confront some really rare situations during the execution.

So, our purpose is to develop a verification system which can test several Linux file system drivers in scenarios mentioned above. That can be achieved by testing all the possible use cases of the drivers (according to the documentation) and using the KEDR framework and KEDR tools (if necessary extending the KEDR tool set).

It is clear that if a verification system covers only some parts of the driver source code, missing such important parts as error checking/handling and rarely executed code, it does not qualify to be a high quality verification system. On the other hand a 100% code coverage does not necessarily mean high quality verification. There can be pieces of code which ***should be there*** but are missing. For example, if the driver does not check the status code returned by a function, even a 100% code coverage cannot find out that error. To reveal such errors the testing system should do something more: make the called function fail. None of the mentioned testing systems are able to do such a thing.

Still, it can be stated that (under even conditions) the verification system which brings to higher percent of code coverage is better than the others.

IV. Current State

As of now, the modules ***Main, Benchmark,*** and ***Syscall*** have been implemented (normal execution mode only). Verification methods for FS-specific drivers capabilities are being investigated.

Code coverage analysis has been performed to calculate the Spruce system quality. That could show how the Spruce system is competitive with the existing Linux testing systems.

Table 2 and Table 3 present the coverage values for drivers of several file systems in kernel version 3.2.9 according to the execution of above mentioned verification systems and the Spruce system. The data has been acquired by means of Gcov tool [11], which is a part of GCC.

TABLE 2.

OS x64	Ext4	Btrfs	Xfs
LTP	40.1%	42.9%	42.9%
PTS	34.6.%	36.2%	32.3%
Spruce	40.9%	36.8%	44.0%

TABLE 3.

OS x86	Ext4	Btrfs	Xfs
LTP	42.8%	39.2%	39.3%
PTS	34.5%	35.7%	32.5%
Spruce	40.7%	35.4%	40.8%

The figures in tables show that even the partially developed Spruce system is already competitive with the existing solutions (coverage analysis for Autotest is not done because our goal was not to get those values but to be able to compare Spruce with some of the leading testing systems). Moreover, in its current state, the Spruce system execution takes less than 4 seconds. For a sample comparison, it takes LTP two minutes. Of course the figures are acquired by running the systems under the same conditions, i.e. LTP is executed only on those tests which check those and only those system calls that Spruce does. Also in those conditions LTP gives only 34% code coverage, when Spruce gives 41%.

V. Future Directions

We are planning on implementing the incomplete modules of Spruce system. It is also planned to utilize the KEDR-based tools for fault simulation scenarios in those kernel functions which are often used in file system drivers. If necessary, we will upgrade KEDR to provide broader functionality convenient for the Spruce system. It will allow for gaining higher quality of Linux file system drivers verification using the Spruce system.

VI. Conclusion

In this paper we have analyzed the errors which usually occur in Linux kernel modules and especially in file system drivers. Later several testing and certification systems were presented and analyzed to find out in what way and how well they perform verification. Also for that reason the source code coverage was calculated for several file system drivers. Then the Spruce system was presented which is designed to perform high quality verification of Linux file system drivers. The goal is to be achieved using the KEDR framework and tools.

References

[1] A.V. Khoroshilov, V.S. Mutilin, E.M. Novikov, P.E. Shved, A.V. Strakh. Linux Driver Verification Architecture. Proceedings of the Institute for System Programming of RAS, volume 20, 2011 г. ISSN 2220-6426 (Online).

[2] Martirosyan V., Shatokhin E., Gishyan S. Dynamic verification of linux file system drivers. Proceedings of Computer Science and Information Technologies conference, Yerevan, 2011.

[3] Rubanov V., Shatokhin E.. Runtime Verification of Linux Kernel Modules Based on Call Interception. Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST'11), Berlin, Germany, March 2011.

[4] Spruce system, https://code.google.com/p/spruce.

[5] KEDR framework, http://code.google.com/p/kedr/

[6] The Linux Kernel Repository - Change Logs, http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git

[7] Autotest Framework, http://autotest.kernel.org.

[8] Linux Test Project, http://ltp.sourceforge.net.

[9] Phoronix Test Suite, http://www.phoronix-test-suite.com.

[10] Linux Kernel Bugzilla, https://bugzilla.kernel.org

[11] Project Gcov, http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[12] Storage Test Tools for SUSE YES Certified Program, http://www.novell.com/developer/ndk/storage_test_tools.html

[13] Oracle Linux Test Project, http://oss.oracle.com/projects/olt/