# On the Implementation of Data-Breakpoints Based Race Detection for Linux Kernel Modules

Nikita Komarov
ISPRAS
Moscow, Russia
nkomarov@ispras.ru

***Abstract*—An important class of problems in software are race conditions. Errors of this class are becoming more common and more dangerous with the development of multi-processor and multi-core systems, especially in such a fundamentally parallel environment as an operating system kernel. The paper overviews some of existing approaches to detect race conditions including DataCollider system based on concurrent memory access tracking. RaceHound, a race condition detection system for Linux drivers based on similar principles as DataCollider is presented.**

***Keywords*—driver verification; race condition; linux kernel; dynamic verification; operating system**

## I. Introduction

The Linux Kernel is one of the most popular and fast-developed projects in the world. Linux Kernel development started in 1991 by Linus Torvalds. The development process of Linux kernel is distributed, about 1,000 people worldwide are involved in the preparation of each new kernel release. The new release comes out every 2-3 months. Changes are submitted by the developers in the form of little pieces of code called patches. Each kernel release consists of about 9-13 thousands of patches, which corresponds to an average of about 7.3 patches per hour. The total source code size of one of the latest versions of the Linux kernel - version 3.2 – is about 15 million lines. These data are given in the latest Linux Foundation report on Linux Kernel development [6].

Linux Kernel development process is described in [7]. There are some other branches of kernel development based on the original Linux Kernel. Some of the Linux distributions developers support their own versions of the kernel - for example, Red Hat [11], openSUSE [12] and Debian [13]. These kernels are different from the original version in that they support some additional functionality and/or contain bug fixes. There are some kernel versions with the significant changes to the basic systems of the kernel, for example, a real-time Linux Kernel [14] or the Android kernel [15]. Over time some changes from different branches of development needed by a broad range of people can get to the original kernel.

As with any programs, there are various errors in Linux Kernel that lead to the incorrect functioning of the OS, freezing etc. The greatest part of the kernel (about 70%) are various device drivers. The results of studies that have been carried out in [16] and [17] in the early 2000s for kernels 1.0 to 2.4.1, showed that drivers contain up to 85% of all errors in the Linux Kernel. A similar study for the Microsoft Windows XP kernel in 2006 also showed that the highest number of errors in the operating system kernel belongs to the device drivers [18]. More recent studies done in 2011 for the Linux Kernel versions from 2.6.0 to 2.6.33 showed that although the number of errors in the drivers became less than in the kernel components responsible for the support of the various architectures and file systems, their share is still high [19].

The task of ensuring the reliability of drivers is important as drivers in Linux work with the same privileges level as the rest of the kernel. Because of this a vulnerability in the drivers can lead to the possibility of execution of arbitrary code with kernel privileges and access to the kernel structures.

## II. Race Conditions

One of the important types of errors in the software is race conditions [20]. A race condition occurs when a program is working wrong due to an unexpected sequence of events that leads to the simultaneous access to the same resource by multiple processes.

As an example of a race condition, consider a simple expression in some programming language: b = b + 1. Imagine that this expression is executed simultaneously by two processes, the variable b is common to them, and its initial value is 5. Here is a possible example of the order of execution of the program:

- Process 1 loads b value in a register.
- Process 2 loads b value in a register.
- Process 1 increases its register value by 1 with a result of 6.
- Process 2 increases its register value by 1 with a result of 6.
- Process 1 stores its register value (6) in the variable b.

- Process 2 stores its register value (6) in the variable b.

The initial value of b was 5, each of the processes added 1, but the final result was 6 instead of the expected 7. Processes are not executed atomically, another process may intervene and perform operations on some shared resource between almost any two instructions. Similarly, the classic example is the simultaneous withdrawal of money from a bank account from two different places: if the check for the required amount in the account by the second process would occur between similar check and amount decrease of the first process, the account balance may become wrong that will cause a loss and significant reputation damage.

With the development of multicore and multiprocessor systems race condition related errors including the Linux Kernel, becomes even more important than before. For example in the study [1] it was concluded that the race conditions are the most frequent type of error in the Linux Kernel and make up about 17% of typical errors in the Linux Kernel (on the second and third place are specific objects leaks and null pointer dereference – 9% both). The study was conducted by analyzing the comments to the changes in the Linux Kernel. From the above it can be concluded that race conditions are an important and common class of errors, including those in the Linux Kernel, and the task to find them is relevant.

## III. Existing methods for detecting race conditions

There are various ways to detect race conditions in programs. Most of the dynamic methods are based on two principles: Lockset and Happens-before [4] [5]. Lockset based tools check if there is synchronization between threads when accessing shared variables. This makes it possible to find a large number of potential errors, but the number of false positives is high too.

Happens-before based instruments find accesses from different threads to a specified area of memory that have no specified order, meaning they can be in a different order. These instruments depend on how the access is done in a real system operation, so they identify a smaller subset of errors but have greater accuracy than the Lockset method. An alternative to this method is a direct test for simultaneous memory access by placing breakpoints - this method is implemented in the DataCollider system (see sect. II.C). In most real systems, a combination of two methods is used. Let's consider some examples of such systems.

### *A. Helgrind*

Helgrind is a tool for analyzing user mode programs for race conditions, based on the Valgrind framework [10]. This system can detect three types of errors:

- Improper pthreads API use;
- Possible deadlocks that occur due to incorrect order of synchronization mechanisms;
- Race conditions.

Helgrind detects race conditions by monitoring all accesses to the memory of the process and all use of synchronization primitives. Then the system builds a graph, based on which it makes a conclusion that there is a «happens-before» relationship between accesses. If the access to a certain area of memory happens in two different threads, the system checks whether there can be found the «happens-before» relationship between them, that is, whether one of accesses happen before the other. The system makes conclusions of the presence or absence of such a connection based on the presence or absence of various synchronization primitives. If the memory access occurs in at least two different threads and the system cannot find a relationship «happens-before» between them, it concludes that there is a data race between them.

### *B. ThreadSanitizer*

ThreadSanitizer is another engine that finds race conditions in user space programs. The algorithm of this system is similar to the Helgrind algorithm and is described in [23]. The system instruments the program code adding calls to its functions before each memory access and every time the program uses some synchronization tools. The system then tries to figure out which of the memory accesses occur with inadequate synchronization and may conflict with other memory accesses. ThreadSanitizer also has an offline mode in which it can be used to analyze traces created by some other tools such as Kernel Strider for Linux Kernel [24].

### *C. DataCollider*

The system is designed for dynamic race conditions detection in Microsoft Windows kernel. It was developed in Microsoft Research and is described in [3]. The system uses the principle which is slightly different from other described systems and is as follows:

- The system periodically sets up software breakpoints in random places of studied code.
- When the software breakpoint is triggered, the system decodes the triggered instruction getting the memory address and sets a hardware breakpoint on access to this address. Then it stops the process execution for a short time to increase the chance of another access to this address.
- After the delay the system removes the hardware breakpoint.
- If the hardware breakpoint is triggered, data race is reported. Also data race is reported if the value at the address has changed – to take the possible use of direct memory access (DMA) into account.

DataCollider system is used in Microsoft, it helped to find about 25 errors in the Windows 7 kernel. In [3] low overhead advantage of the system is noted: it can find some errors in the kernel even with the settings causing overhead of less than 5%.

## IV. Linux driver verification

The main features of driver design and verification are the direct work with the hardware, a common address space, limited set of user space interfaces and multithreading. This makes it difficult to debug the drivers and to determine the causes of errors. Let's consider some software products that are used to verify the Linux drivers.

### *A. Kmemleak, kmemcheck*

These systems are the most known and widely used. They are included in Linux Kernel. Kmemleak [8] is a system for finding memory leaks. Its principle of operation is similar to those in some of the garbage collectors in high level languages. For every memory allocation, the information about the selected memory area (address, size etc.) is stored, and when this area is deallocated the corresponding entry is removed. The system can be interacted with via a character device in debugfs. With every access to this device the following steps are conducted:

- The "white" list of the allocated and not freed memory areas is created.
- Certain areas of memory are scanned for pointers to the memory of the "white" list. If the system finds such a pointer, the memory is transferred from the "white" list to the "gray" list. The memory in the "gray" list is considered accessible and not leaked.
- Each block of the "gray" list is also scanned for pointers to the memory of the "white" list.
- After this scanning all memory left in the "white list" is considered to memory leaks.

Kmemcheck [9] is a simple system that keeps track of uninitialized memory areas. Its principle of operation is as follows:

- The system intercepts all the memory allocation. Instead of each area an area 2 times as big is allocated, these additional ("shadow") pages are initialized with zeroes and are hidden.
- The allocated memory area is returned to the caller with cleaned “present” flag. As a result, any reference to this memory will result in page fault.
- When such a memory access happens, kmemcheck determines the address and size of the corresponding memory access. If the access is for writing, the system populates the corresponding bytes of "shadow" page with 0xFF, then successfully completes the operation.
- If the access is for reading, the system checks the appropriate bytes of "shadow" pages. If at least one of them is 0, uninitialized memory access is reported.

### *B. KEDR*

KEDR (short for KErnel-mode Drivers in Runtime) is a system for the dynamic analysis of Linux Kernel modules [21]. This system can replace some kernel function calls with its wrappers which can produce some additional actions such as saving the information of function calls or just returning errors. Users can create their own systems based on this system, and solutions for some specialized tasks are included, such as:

- Memory leaks detection. To solve this problem, the system keeps track of calls to different functions that allocate and free the memory. After unloading the tested module the system creates a report containing all memory locations that have been allocated, but have not been freed, along with the call stack for each of the memory allocation functions calls. The report also includes any attempts to free the memory that has not been allocated. The scenario is different from the Kmemleak system in that the memory leak detection happens after the unloading of the target module, which simplifies the algorithm.
- Fault simulation. To solve this problem given functions are replaced by a wrapper which returns errors on defined scenario.
- Call tracking. Information about calls to given functions (including arguments, return values etc.) is stored in a file for later analysis.

This system has been used successfully and helped to find about 12 errors in various Linux drivers [22].

### *C. Static methods*

Static program verification is the analysis of program code without actually executing it, as opposed to dynamic analysis. It does not require setting up the test environment, and provides the ability to analyze all the possible execution paths of the program, even those which need the coincidence of several rare conditions. When applied to the OS Kernel, static verification is particularly useful because in many cases creating a test environment and analyzing some of the execution paths can be a non-trivial task. However, static analysis has many limitations. The main part of the paper is devoted to the dynamic verification system, so we will not examine the static verification methods in detail. A more detailed review of these methods is given in [2].

## V. RACEHOUND

As part of the Google Summer of Code 2012 [25], the author developed a lightweight race detection system for Linux Kernel. The algorithm used by this system is similar to the algorithm used by the DataCollider system (see sect. II.C). The system is designed not only to find race conditions, but also to confirm the data obtained with other systems that can produce false alarms, for example, ThreadSanitizer (see section II.B). At present the system supports the x86 and x86-64.

The originally planned principle is as follows:

- The system randomly plants software breakpoints (there is a Linux Kernel API called Kprobes [26] for that) in various places of the investigated kernel module, periodically changing them.
- When the software breakpoint is triggered, the system decodes the instruction on which the breakpoint was planted, using the decoder of the Linux Kernel modified in KEDR project. Then it determines the memory address which the instruction tries to access and sets the hardware breakpoint on this address (there is also an API in Linux Kernel for this [27]), and then stops the process for a short time to increase the chance of access from another process to this address.
- After the delay the system removes the hardware breakpoint.

- If the hardware breakpoint was triggered in the elapsed time, the race condition is reported. The race condition is also reported if the value at the address has changed – to cover the case of direct memory access.

Software breakpoints in x86 architecture work as follows. The first byte of instruction at the specified address is replaced by the 0xCC byte – interrupt INT3 – preserving the original byte in some place. When the CPU executes the instruction, an interrupt is triggered and control is passed to the interrupt handler in Linux Kernel, which searches the list of software breakpoints for the appropriate address. When the address is found, it transfers control to the appropriate handler. After the handler finishes the original instruction is executed.

Hardware breakpoints are implemented as four debug registers on Intel x86 processors. This system is described in the Intel Developer Manuals [28]. An addresses can be written in these registers, and there will be an interrupt on an access to these addresses. The interrupt is then processed by the Linux Kernel which transfers control to the appropriate hardware breakpoint handler.

### *A. Implementation features*

There were some problems in the implementation of system. Software breakpoint handlers are executed in an atomic context, and therefore it was impossible to properly set the hardware breakpoint for all available CPUs. This problem was solved by installation and removal of hardware breakpoint not from the software breakpoint handler, but from the function in the task queue. Unfortunately, this decision led to a time gap between the beginning of the delay and the hardware breakpoint setup, and therefore may reduce the probability of concurrent memory access to occur within the delay (for a very small time delay - down to 0) and to lower detection accuracy. This effect, however, requires a special study.

Another problem was the execution of the original instruction in the software breakpoint handler. This execution takes place inside an interrupt handler, but the original instruction refers to the address on which the hardware breakpoint has just been set. In some cases, removal of hardware breakpoint does not yet happen at the time of the original instruction execution, and the hardware breakpoint was triggered. However, the software breakpoint handler works in an atomic context and the interrupt is forbidden. This caused some faults and unusual behavior. This problem was solved by dropping Kprobes API and implementing similar functionality manually. Instead of executing the original instruction separate from the module code this instruction was restored and the control was transferred to the investigated module. To reset the breakpoints after that the timer has been set, which reset the breakpoints at frequent intervals replacing their first bytes with 0xCC. This decision, however, also has a drawback: there is a period in which a software breakpoint is not set at the needed place. This can also reduce the accuracy of error detection.

The system consists of a kernel module which has an interface based on the debugfs and some auxiliary scripts. The interface is a character device in debugfs which allows a user to set the possible breakpoints range, from which N is randomly chosen, in the format <function name>+<offset>. If the both parameters or just the offset are equal to *, the complete module or the complete function is added, respectively.

An important limitation of the system is the inability to work on single-core systems. This problem is caused by the software breakpoint handler execution: it is performed in an atomic context, so putting the process to sleep is impossible. Therefore, instead of the function msleep() the mdelay() function is used, which waits a specified period of time, leaving the thread in running state. For this time no other tasks can run on the same processor. Therefore the process which could cause a race condition should run on another core to be able to execute at a delay time.

The system requires a Linux Kernel 2.6.33 or later (this version introduced the Hardware Breakpoints API). The build system is based on CMake. At present the system is in the state of working prototype. It requires testing on some real drivers to identify potential errors and defects, adjust the parameters of the system (number of breakpoints, time intervals, etc.) and to evaluate the effectiveness of the system in the real world. For testing it is necessary to pay attention to the choice of test cases for drivers – they should include some parallel and concurrent testing, because the system just increases the probability of errors, being useless if the concurrent access is impossible.

Another direction of the system development may be the development of interfaces and integration with other race condition detection systems. For example, the system can be useful when working together with static methods which provide a significant number of false positives to confirm these data with them. However, dynamic methods which can produce some false positives can also benefit from such integration.

## VI. Conclusion

Race conditions are an important problem. This paper reviews some methods for detecting race conditions, including those in the operating system kernel, and some features of Linux drivers verification. The race condition detection system created by author is described.

Most of the race condition detection systems are based on one of the two methods: LockSet and Happens-before, or some kind of their combination. From a theoretical point of view, the direction of future work could be some more detailed review of existing methods for detecting race conditions in order to integrate the developed system with some of them.

Directions of further practical work should be testing the developed system on some real drivers and its integration with other systems, including those based on static methods. Testing on real drivers will help to identify errors or omissions, find some valid settings of various parameters of the system (number of breakpoints, time intervals etc.) and to evaluate the effectiveness of the system in real conditions.

[1] V. Mutilin, E. Novikov, A. Khoroshilov
Analysis of typical errors in Linux operating system drivers.
Proceedings of Institute for System Programming of RAS, vol.22, 2012

[2] M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. V. Khoroshilov, and P. E. Shved
Using Linux Device Drivers for Static Verification Tools Benchmarking
Programming and Computer Software, 2012, Vol. 38, No. 5

[3] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk
Effective Data-Race Detection for the Kernel
9th USENIX Symposium on Operating Systems Design and Implementation, 2010
http://static.usenix.org/event/osdi10/tech/full_papers/Erickson.pdf

[4] Cormac Flanagan, Stephen N. Freund
FastTrack: Efficient and Precise Dynamic Race Detection
http://slang.soe.ucsc.edu/cormac/papers/pldi09.pdf

[5] Nels E. Beckman
A Survey of Methods for Preventing Race Conditions
http://www.cs.cmu.edu/ nbeckman/papers/race_detection_survey.pdf

[6] Jonathan Corbet, Greg Kroah-Hartman, Amanda McPherson
Linux Kernel Development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It (2012)
http://go.linuxfoundation.org/who-writes-linux-2012

[7] Jonathan Corbet
How to Participate in the Linux Community. A Guide To The Kernel Development Process (2008)
http://www.linuxfoundation.org/content/how-participate-linux-community

[8] Jonathan Corbet
Detecting kernel memory leaks
http://lwn.net/Articles/187979/

[9] Jonathan Corbet
kmemcheck
http://lwn.net/Articles/260068/

[10] Valgrind Manual. 7. Helgrind: a thread error detector.
http://valgrind.org/docs/manual/hg-manual.html

[11] S. M. Kerner
The Red Hat Enterprise Linux 6 Kernel: What Is It? (2010)
http://www.serverwatch.com/news/article.php/3880131/The-Red-Hat-Enterprise-Linux-6-Kernel-What-Is-It.htm

[12] OpenSUSE Kernel
http://en.opensuse.org/Kernel

[13] Debian Kernel
http://wiki.debian.org/DebianKernel

[14] OSADL Project: Realtime Linux
https://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html

[15] Jonathan Corbet
Bringing Android closer to the mainline
https://lwn.net/Articles/472984/

[16] A. Chou, J. Yang, B. Chelf, S. Hallem, DR Engler
An Empirical Study of Operating System Errors. Proc. 18th ACM Symp. Operating System Principles, 2001

[17] M. Swift, B. Bershad, H. Levy
Improving the reliability of commodity operating systems. In: SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003

[18] A. Ganapathi, V. Ganapathi, D. Patterson
Windows XP kernel crash analysis. Proceedings of the 2006 Large Installation System Administration Conference, 2006

[19] N. Palix, G. Thomas, S. Saha, C. Calves, J. Lawall, and Gilles Muller
Faults in linux: ten years later. Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS ’11), USA, 2011

[20] David Wheeler
Secure programmer: Prevent race conditions (2004)
http://www.ibm.com/developerworks/linux/library/l-sprace/index.html

[21] KEDR Manual
http://code.google.com/p/kedr/wiki/kedr_manual_overview

[22] KEDR wiki: Problems Found
http://code.google.com/p/kedr/wiki/Problems_Found

[23] Thread Sanitizer Manual
http://code.google.com/p/thread-sanitizer/w/list

[24] Kernel Strider Manual
http://code.google.com/p/kernel-strider/wiki/KernelStrider_Tutorial

[25] Project: Implement a Lightweight Data Race Detector for Linux Kernel Modules on x86
Google Summer of Code 2012
http://www.google-melange.com/gsoc/project/google/gsoc2012/nkomarov/7001

[26] Linux Kernel Documentation: Kprobes
http://www.mjmwired.net/kernel/Documentation/kprobes.txt

[27] Prasad Krishnan
Hardware Breakpoint (or watchpoint) usage in Linux Kernel. Ottawa Linux Symposium, 2009
http://kernel.org/doc/ols/2009/ols2009-pages-149-158.pdf

[28] Intel 64 and IA-32 Architectures Software Developer Manuals
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html