Kernel Data Race Detection using Debug Register in Linux

Yunyun Jiang[1], Yi Yang[1], Tian Xiao[1], Tianwei Sheng[2], Wenguang Chen[1*]

[1]Department of Computer Science and Technology, Tsinghua University, Beijing, China
[2]Department of Computer Science and Engineering, University of California San Diego, USA
jiangyy09, yang-yi12, xiaot04@mails.tsinghua.edu.cn, tisheng@eng.ucsd.edu, cwg@tsinghua.edu.cn

Abstract: Data races in parallel programs are notoriously difficult to detect and resolve. Existing research has mostly focused on data race detection at the user level and significant progress has been made in this regard. It is difficult to apply detection methods designed for user-level applications to identify OS kernel level races. In this paper, we present a new detection tool that is able to effectively detect race conditions in the Linux kernel environment. We use a dynamic detection approach, employing hardware debug registers available on commodity processors, to catch races on the fly during runtime. Preliminary experimental results show that our tool can effectively identify real data race instances.

Keywords: data race detection, Linux kernel, synchronization, debug register, sample

I. MOTIVATION

Parallel programming stands to play a critical role in improving computing performance in the near future. Writing parallel programs, however, is complicated error-prone work. One of the most elusive concurrency bugs, notoriously hard to detect and fix, is the data race condition which occurs when two threads (at least one of which is a write operation) simultaneously access the same shared memory space without proper synchronization constraints. Data races not only affect program results, leading to incorrect output, but may also induce system behaviors, such as halting or crashing the system.

Considerable research[1-2] has been conducted on data race detection and significant progress has been made. However, most research efforts have focused on user-level data races and very little work has been done at the OS kernel level. Compared with in user-level applications, race conditions in the kernel may result in more severe consequences. And in contrast to user-level applications, kernel-level code employs vastly more complicated synchronization mechanisms, including different types of locks, hardware and software interrupts, widely used signal/wait primitives and low-level shared resources. This makes it difficult to apply detection methods originally designed for user-level applications in this case.

Recently, because of the large overhead of software based approaches, more and more researches are studying the low overhead of hardware based method[3-4]. DataCollider[5] is the first detection tool aimed at identifying data race conditions in the kernel. This solution bypasses the traditional detection mechanism by employing a ubiquitous structure debug register in modern hardware and checks the multi-threaded kernel codes for data races. However, DataCollider is designed for the Microsoft Windows operating system, which is proprietory and cannot therefore be applied directly in further related research efforts. In this context, we present DRDDR (Data Race Detection using Debug Register), a lightweight dynamic data race detector, the first work to apply the debug register-based race detection method to the Linux operating system. We furthermore discuss our practical experience with detecting race conditions in the Linux kernel and hope this information will prove of value to both the research and industry communities.

II. DETECTION STRATEGY AND IMPLEMENTATION

DRDDR samples a small amount of memory access operations as detection candidates and utilizes code breakpoints and watchpoints in modern computer architecture to detect race conditions. Our solution does not add extra runtime overhead to the non-sampled code regions, and, by choosing a low sample rate, is able to conduct the detection operation with negligible overhead.

The essential workings of DRDDR are depicted in Fig. 1. Our solution overcomes the aforementioned issues in kernel data race detection as follows: DRDDR first selects some memory access instructions at random and samples a small number of them to insert code breakpoints. The programs are then allowed to run normally. When a breakpoint is triggered, DRDDR opens a short time window, sets a watchpoint on the sampled memory address using the debug register, and briefly monitors. If another thread accesses the critical shared data in this period, the watchpoint is triggered and DRDDR reports a data race bug.

We use a Linux system based on x86 architecture and rely on the code breakpoint and watchpoint mechanisms supported in the processor hardware to implement DRDDR. The architecture of our solution is depicted in Fig. 2. DRDDR employs DebugFS as a user interface to initialize each part of the solution. The sampler selects a small number of memory access instructions from the sampling set and instruments them with code breakpoints. If any of the breakpoints is triggered, DRDDR runs the conflict detection algorithm on the access operations on the spot, and then randomly selects another program address from the sampling

* Corresponding author, Phone number: +86 13601211749

 978-1-4799-3810-0/14/$31.00

set to insert the breakpoint.

Given a parallel program encoded in the form of a binary file, DRDDR first uses the decoder to transform it into a sampling set consisting of all instructions that access memory data. This decoding process is accomplished by sending concurrent debug symbols to the binary file. There are some equivalent choices of decoder such as the QEMU[6] emulator and the KVM[7] virtual machine. QEMU is a set of emulators widely used on the GNU/Linux platform that run in both user-mode and system-mode and support many kinds of architectures, but it is not suited for kernel-mode. In comparison, KVM is a much better match for the requirements of DRDDR. It is an open-source kernel virtual machine with concise functions and good transplant ability. KVM can also run in kernel mode with far less functional dependencies, helping reduce the processing time for certain complex DRDDR functions. We therefore choose KVM for our implementation and transplant a portion of its features.

DRDDR sets a watchpoint on the shared variables in the code breakpoint handler after decoding. The current thread is then suspended and there is a waiting period to see if any other threads access that particular memory location. Two strategies are used for detection: watchpoint and value-comparison.

1) Watchpoint: Modern computer architecture has functionality for trapping read and write operations to processor memory addresses, which plays a key role in effectively setting watchpoints in debuggers. DRDDR utilizes the four debug registers provided by x86 hardware to efficiently monitor other memory access operations that may conflict with the current one. The detailed progress is shown in Fig. 3.

2) Value Comparison: The value-comparison strategy is very simple: if a write operation conflicts with the sampled operation and modifies the data value at a location in the memory, DRDDR can verify the change by reading the shared data before and after the operation. This strategy has an obvious shortcoming that it cannot detect the situation where a read operation conflicts with a write one. Similarly, it cannot identify conflicting write operations in which the final data value equals the initial value. Even so, the value-comparison strategy is still very effective in practice.

III. RESULTS

We verify the correctness of DRDDR by screening all data race related bugs for typical, deeply discussed and perfectly solved instances using the official kernel code bugs management tool in Linux, Kernel Bugzilla[8]. Reproducing these bugs involved locating suitable kernel versions and configuring the relevant environments. We present two examples in Fig. 4 and 5 and summarize their properties in Table I. Experiments show that DRDDR detects these real-world kernel data race bugs accurately and efficiently.

We have freely released our solution to the open source community[9], for the benefit of programmers who require such tools, and to assist scholars in related fields to further their research in a much more easier and efficient way.

In the future, we can improve performance in terms of using more-efficient sampling strategies. Random sampling has certain shortcomings, including that it is insufficient in accuracy and creates extra overhead in some cases. If we could obtain information about which portions of a program are more likely to form race conditions, using methods such as program tagging or pre-analysis[10], we could preferentially pick those locations in the sampling process and improve detection performance.

[1] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21, 1978, 7: 558--565.

[2] Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems (TOCS) 15, 1997, 4: 391--411.

[3] Sheng T W, Vachharajani N, Eranian S, Hundt R, Chen W G, Zheng W M. RACEZ: a lightweight and non-invasive race detection tool for production applications. In Proceedings of the 33rd International Conference on Software Engineering, ICSE'11 2011. 401--410

[4] Arulraj J, Chang P C, Jin G, Lu S. Production-run software failure diagnosis via hardware performance counters. In Proceedings of the 18th international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'13. ACM, 2013: 101-112.

[5] Erickson J, Musuvathi M, Burckhardt S, Olynyk K. Effective Data-Race Detection for the Kernel. In Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10, vol. 10, 2010. 1--16.

[6] Open Source Processor Emulator QEMU, http://wiki.qemu.org/Main_Page

[7] Kernel Based Virtual Machine KVM, http://www.linux-kvm.org/page/Main_Page

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

[9] DRDDR source code on GitHub, https://github.com/ahyangyi/DRDDR

[10] Sheng T W. Researches on key technologies of data races detection in concurrent programs, Dissertation for the Doctoral Degree. Beijing: Tsinghua University, 2010. 49--65

 978-1-4799-3810-0/14/$31.00

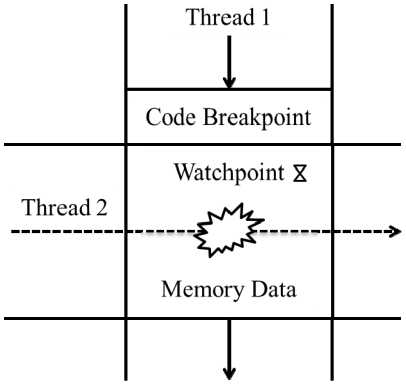


Fig. 1 The basic principle of DRDDR.

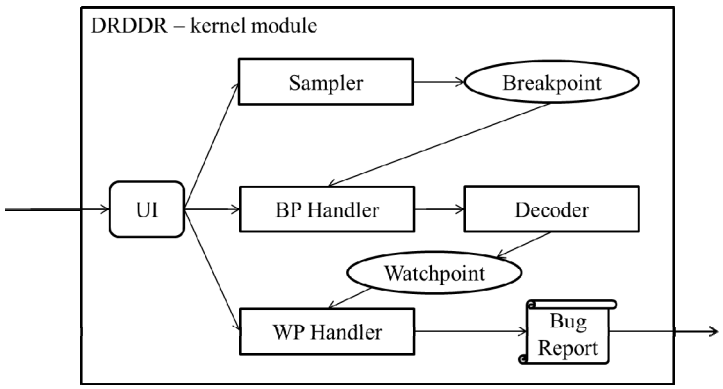
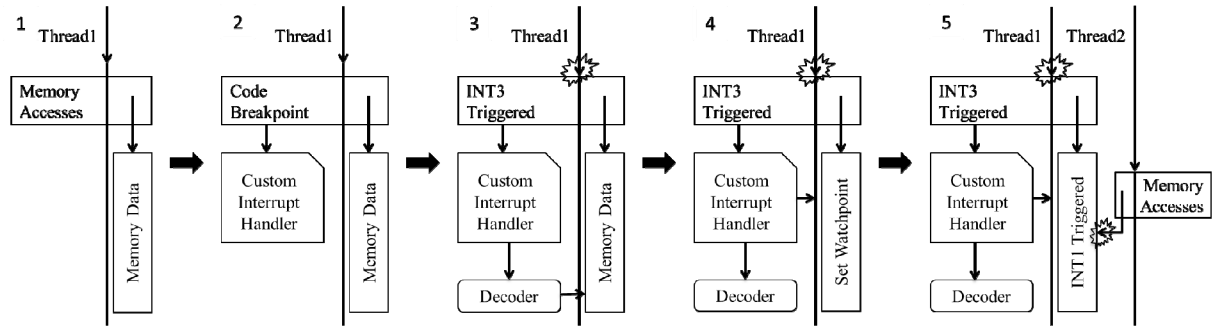


Fig. 2 The architecture diagram of DRDDR.



1. Sampling memory access.

2. Replacing interrupt handler and inserting code breakpoint.

3. Decoding current instruction after INT3 triggered.

4. Setting watchpoint on the memory data accessed by Thread1.

5. Reporting race condition when INT1 triggered by Thread2 on the shared data.

Fig. 3 The basic principle of setting a code breakpoint and watchpoint in DRDDR.

Thread 1:	**Thread 2:**
…	ext2_link();
ext2_rename();	…
…	ext2_unlink();

```
static inline void inc_nlink(struct inode *inode){
        inode->i_nlink++;
}
static inline void drop_nlink(struct inode *inode){
        inode->i_nlink--;
}
```

Fig. 4 Race1 in *ext2* file system.

Thread 1:	**Thread 2:**
ecryptfs_get_inode(){	ecryptfs_write(){
…	
iget5_locked()	…
…// Initialization	iget5_locked()
unlock_new_inode()	i_size_write()
i_size_write()	unlock_inode()
…	…
}	}

Fig. 5 Race2 in *eCryptfs* system.

Table I
Description of the Linux kernel races detected by DRDDR.

Property	Race1	Race2
Location	*ext2* file system	*eCryptfs* system
Kernel version	2.6.38-rc8	3.0-rc1
Type	Atomicity violation	Order violation
Description	Changing file link number without lock.	Rewriting file size unsynchronized.
Breakpoint	`inode->i_nlink`	`i_size_write()`

978-1-4799-3810-0/14/$31.00 ©2014 IEEE