Detecting Data Races Caused by Inconsistent Lock Protection in Device Drivers

Qiu-Liang Chen, Jia-Ju Bai, Zu-Ming Jiang, Julia Lawall, Shi-Min Hu

Detecting Data Races Caused by Inconsistent Lock Protection in Device Drivers

Qiu-Liang Chen†, Jia-Ju Bai†*, Zu-Ming Jiang†, Julia Lawall‡, and Shi-Min Hu†
†Department of Computer Science and Technology, Tsinghua University, Beijing, China
chenql16@mails.tsinghua.edu.cn, baijiaju1990@gmail.com, jjzuming@outlook.com, shimin@tsinghua.edu.cn
‡Sorbonne University/Inria/LIP6, Paris, France
Julia.Lawall@lip6.fr

***Abstract*—Data races are often hard to detect in device drivers, due to the non-determinism of concurrent execution. According to our study of Linux driver patches that fix data races, more than 38% of patches involve a pattern that we call *inconsistent lock protection*. Specifically, if a variable is accessed within two concurrently executed functions, the sets of locks held around each access are disjoint, at least one of the locksets is non-empty, and at least one of the involved accesses is a write, then a data race may occur.**

In this paper, we present a runtime analysis approach, named DILP, to detect data races caused by inconsistent lock protection in device drivers. By monitoring driver execution, DILP collects the information about runtime variable accesses and executed functions. Then after driver execution, DILP analyzes the collected information to detect and report data races caused by inconsistent lock protection. We evaluate DILP on 12 device drivers in Linux 4.16.9, and find 25 real data races.

***Index Terms*—Data race, inconsistent lock protection, device driver, runtime analysis**

I. Introduction

To improve system performance, modern OS kernels (such as Linux, Windows and FreeBSD) support multithreading to utilize multicore processors. Device drivers thus use concurrency to efficiently communicate with the kernel and hardware. But concurrent execution can introduce concurrency problems. Studies [1]–[3] have shown that concurrency problems occupy a large part of reported bugs in device drivers, and these concurrency problems are often hard to reproduce and detect [2]. Data races are a common kind of concurrency problem. A data race occurs when multiple threads access the same memory location without proper synchronization and at least one access is a write [4]. Data races can introduce non-determinism, and may cause serious runtime problems such as null-pointer dereferences and use-after-free errors.

Many approaches [5]–[14] have been proposed to detect data races in user-level applications. To detect data races in kernel-level device drivers, some approaches [15]–[19] use static analysis, but they often report false positives due to lacking exact runtime information. Several approaches [20]–[23] use dynamic analysis and can detect data races in device drivers. They are based on sampling or lockset analysis. The effectiveness of the sampling-based approaches [20], [21] heavily relies on the sampling frequency. These approaches may miss real data races when the sampling frequency is low, and they may introduce much runtime overhead when the sampling frequency is high. Lockset-based approaches [22], [23] detect data races based on the set of locks protecting shared-variable accesses. But these approaches often report many false data races, because the related shared variables are not actually concurrently accessed when they are written.

The traditional lockset-based approach, as proposed in Eraser [22], is based on the assumption that all accesses to a shared variable can potentially be executed concurrently and thus need to be protected by locks. Accordingly, for each possible shared variable v, globally across the execution, Eraser maintains a set $C(v)$ that contains the set of locks that are held across all previous observed accesses to v. $C(v)$ is initially all of the locks available in the system. As an access to v occurs, $C(v)$ is updated to the intersection of $C(v)$ with the currently held set of locks. An error is reported if $C(v)$ becomes the empty set. In practice, however, it is not the case that all accesses to a given variable v can occur concurrently with each other. As a typical example, initialization may occur before the variable is shared, in which case a data race is impossible. Thus, the traditional lockset-based approach results in many false positives.

To address these issues, we refine the dynamic lockset-based approach by adding more constraints on the set of variables that are considered and the locksets that are compared. Rather than keeping track of a global lockset for a variable, we collect the specific sets of locks held at each accesses. Furthermore, we only compare locksets of accesses that occur in functions that are actually executed concurrently, thus providing stronger evidence that the accesses may conflict. We only perform this check when one of the accesses is a write and when one of the accesses is protected by at least one lock (we refer to the accessed variable as a *possible raced variable*), reflecting developer understanding that a concurrent access is possible. *If a variable is accessed within two concurrently executed functions, the sets of locks held around each access are disjoint, at least one of the locksets is non-empty, and at least one of the involved accesses is a write, then a data race may occur.* In fact, this data race is caused by *inconsistent lock protection* for the possible raced variable in the two concurrently executed functions. As compared to Eraser, we can drastically reduce the set of false positives, because we

* Jia-Ju Bai is the corresponding author.

only report races when there is strong evidence that a race is possible. Furthemore, by focusing on variables that are protected by at least one lock, and are thus more likely actually shared, we are able to keep the runtime overhead manageable.

Based on the above idea, we propose a dynamic approach named DILP, to detect data races caused by inconsistent lock protection in device drivers. Overall, DILP consists of three phases. Firstly, at compile time, with LLVM [24], DILP instruments some places in the driver code, such as variable accesses and driver functions. Secondly, during driver execution, DILP intercepts variable accesses and monitors executed functions. It identifies and records the information about runtime variable accesses and concurrently executed functions. Finally, after driver exeuction, DILP analyzes the recorded information to detect data races caused by inconsistent lock protection. We have implemented DILP for Linux device drivers.

Overall, we make three main contributions in this paper.

- We perform a study of Linux driver patches, and find that over 38% of data-race-fixing patches involve inconsistent lock protection. This study indicates that many reported data races in device drivers are caused by inconsistent lock protection.
- We propose a dynamic approach named DILP, to automatically detect data races caused by inconsistent lock protection in device drivers.
- We evaluate DILP on 12 device drivers in Linux 4.16.9. DILP in total finds 25 real data races, and 11 of them have been confirmed by driver developers. The evaluation also shows that the runtime overhead of DILP is lower than many previous approaches.

The rest of this paper is organized as follows. Section II presents the motivation of this paper. Section III introduces DILP in detail. Section IV presents the evaluation on Linux drivers. Section V compares our work to previous approaches. Section VI makes a discussion about DILP. Section VII shows the related work, and Section VIII concludes this paper.

II. Motivation

In this section, we first motivate our work using a real example in Linux driver code, and then present our study of Linux driver patches.

A. Motivating Example

Figure 1 presents a fixed data race in the *rtl8723ae* device driver. This data race was first introduced in Linux 3.18 (released in December 2014). During driver execution, the function `rtl_ps_set_rf_state` can be concurrently executed with the function `rtl8723e_dm_watchdog`. In the function `rtl_ps_set_rf_state`, the variable `ppsc->rfchange_inprogress` is assigned on line 167, and this write operation is protected by a spinlock acquired on line 166. But in the function `rtl8723e_dm_watchdog`, the variable `ppsc->rfchange_inprogress` is read as a condition value on line 845 without holding the spinlock. Thus, a data race on `ppsc->rfchange_inprogress` may occur. This data race was first fixed in Linux 4.8 (released in October 2016), by acquiring the spinlock in `rtl8723e_dm_watchdog`.[1] This data race persisted over 10 mainline releases (nearly 2 years).

FILE: linux-3.18/drivers/net/wireless/rtlwifi/rtl8723ae/dm.c

```
829. void rtl8723e_dm_watchdog(...) {
        ......
+++.    spin_lock(&rtlpriv->locks.rf_ps_lock);
845.    if (... && (!ppsc->rfchange_inprogress)) {
          ......
855.    }
+++.    spin_unlock(&rtlpriv->locks.rf_ps_lock);
856.    if (rtlpriv->btcoexist.init_set)
857.      rtl_write_byte(...);
858. }
```

FILE: linux-3.18/drivers/net/wireless/rtlwifi/ps.c

```
 79. bool rtl_ps_set_rf_state(...) {
        ......
165.    if (!protect_or_not) {
166.      spin_lock(&rtlpriv->locks.rf_ps_lock);
167.      ppsc->rfchange_inprogress = false;
168.      spin_unlock(&rtlpriv->locks.rf_ps_lock);
169.    }
170.    return actionallowed;
171. }
172. EXPORT_SYMBOL(rtl_ps_set_rf_state);
```

Fig. 1. A fixed data race in the *rtl8723ae* driver of Linux 3.18.

This example illustrates the pattern of inconsistent lock protection that may cause data races. Specifically, *if a variable is accessed within two concurrently executed functions, the sets of locks held around each access are disjoint, at least one of the locksets is non-empty, and at least one of the involved accesses is a write, then a data race may occur.*

B. Study of Linux Driver Patches

To know about the proportion of reported data races caused by inconsistent lock protection, we perform a study of Linux driver patches in the Patchwork project [25]. We select the Patchwork project, as it is used by a number of Linux kernel maintainers to collect patches pending for the next release. We select the accepted patches from April 2015 to April 2018 that fix data races, by searching the patch title. We focus on the patches of 6 common driver classes, namely wireless controller, Ethernet controller, sound card, multimedia, MMC (multimedia card) and RDMA drivers. We get 252 accepted patches that fix data races. Among these patches, we identify those that involve inconsistent lock protection. The result is shown in Table I. The first column shows the driver class; the second column shows the number of these accepted patches that fix data races; the third and fourth columns, respectively, present the number and percentage of accepted patches that involve inconsistent lock protection.

From Table I, we find that more than 38% of the accepted patches fixing data races involve inconsistent lock protection. The remaining patches target data races of other patterns, such as atomicity violations and forgetting to disable interrupts where needed. In the patches involving inconsistent lock protection, most data races are fixed by: 1) adding new calls to lock and unlock functions to protect the raced variables;

[1]Patch link: https://patchwork.kernel.org/patch/9198639/

TABLE I
STUDY RESULT OF LINUX DRIVER PATCHES

Driver class	Race	Pattern	Proportion
Wireless	52	16	30.8%
Ethernet	50	20	40.0%
Sound	24	10	41.7%
Multimedia	33	16	48.5%
MMC	11	4	36.4%
RDMA	82	32	39.0%
Total	252	98	38.9%

or 2) moving existing calls to lock and unlock functions to a place that can protect the raced variables. These race fixes suggest that the fixed data races were introduced because: 1) the driver developer forgot that the function containing the access can be concurrently executed with another function having access to the same shared variable, which causes the lack of lock protection; or 2) the driver developer remembered to add necessary lock protection in the related function, but the lock protection is not complete. Indeed, these two reasons are difficult to avoid in driver development, because a device driver often has many functions and complex control logic [26]. Thus, inconsistent lock protection is likely to be introduced, which may cause data races in device drivers.

III. APPROACH

A. Basic Idea

To detect data races caused by inconsistent lock protection, we use a runtime analysis involving two steps. The first step identifies possible raced variables. The second step checks all accesses to the identified possible raced variables, by comparing the locksets that protect these accesses. If the intersection between the locksets is empty, data races will be reported.

The first step identifies a variable as a possible raced variable if it satisfies two requirements: 1) it is a possible shared variable, and 2) it is accessed with the protection of at least one lock. The second step records two kinds of information, namely locksets and driver functions that are concurrently executed. Moreover, interrupt status is also recorded. When a hardware interrupt is raised while a driver function *F* is executed, the device driver suspends executing the function *F* to execute the corresponding interrupt handling function. In this case, we need interrupt status to identify that the interrupt handling function, not the function *F*, is executed.

B. Architecture

Based on our basic idea, we propose an automated runtime analysis approach, named DILP, to detect data races caused by inconsistent lock protection in device drivers. We have implemented DILP with the Clang compiler [27]. Figure 2 shows the architecture of DILP, which has three parts:

- ***Driver generator.*** This part compiles the driver source code, performs code instrumentation, and finally generates a loadable driver.

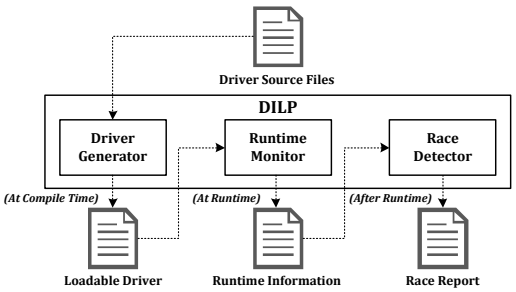


Fig. 2. Overall architecture of DILP.

- ***Runtime monitor.*** This part uses the instrumented code to monitor driver execution, and collects runtime information for subsequent analysis. In order to not modify the OS kernel code, this part is implemented as a kernel module.
- ***Race detector.*** This part analyzes the collected runtime information, and detects data races caused by inconsistent lock protection.

Based on the above architecture, DILP consists of three phases that are introduced as follows.

C. Phases

1) ***Code Instrumentation****:* In this phase, the driver generator performs code instrumentation and generates a loadable driver. Firstly, DILP uses the Clang compiler to compile the driver C code into LLVM bytecode. Secondly, DILP performs code instrumentation on the LLVM bytecode. Thirdly, DILP uses the Clang compiler to compile the modified LLVM bytecode into assembly code. Finally, DILP uses the GCC[2] compiler to compile the assembly code into an object file, and generates a kernel object file as a loadable driver. DILP mainly instruments three kinds of places in the LLVM bytecode:

- *Calls to lock or unlock functions.* By instrumenting these calls, DILP can collect locksets at runtime.
- *Driver functions.* By instrumenting the entry and terminal basic blocks of each driver function, DILP can collect the information about concurrently executed functions.
- *Variable accesses.* By instrumenting *load* and *store* instructions that contain memory accesses, we can monitor read and write operations at runtime.

2) ***Runtime Information Collection****:* In this phase, using the instrumented code, the runtime monitor identifies and records the calling contexts of shared-variable accesses and call paths of concurrently executed functions. The two kinds of information are gotten from possible shared-variable accesses, executed driver functions, locksets and interrupt status.

Possible shared-variable accesses. DILP performs a dynamic taint analysis from two kinds of basic possible shared variables, namely global variables and pointer-type function arguments. Figure 3 shows the procedure of this analysis for a function *func*. DILP maintains two sets of possible shared

[2]In some cases, a driver completely built by Clang cannot be successfully loaded, thus we use GCC here.

```
Procedure: Identifying possible shared variables in the function func
 1: shared_set := ø;
 2: foreach arg in GetArgPtrSet(func) do
 3:     AddSharedSet(arg, shared_set);
 4: end foreach
 5: foreach inst in GetInstSet(func) do
 6:     ret_val := GetReturnVal(inst);
 7:     foreach operand in GetOperandSet(inst) do
 8:         if operand is a global variable then
 9:             AddSharedSet(operand, global_set);
10:         end if
11:         if operand ∈ shared_set or operand ∈ global_set then
12:             AddSharedSet(ret_val, shared_set);
13:             CollectAccess(operand, ret_val, func, ...);
14:         end if
15:     end foreach
16: end foreach
17: delete shared_set;
```

Fig. 3. Dynamic taint analysis of identifying possible shared variables.

```
FuncA: tmp1 = load var1;
FuncA: store tmp1, var2;
FuncA: call mutex_lock();
FuncA: store tmp1, var3;
* Interrupt handling begin
IntrFunc: tmp = load irq_var1;
IntrFunc: store tmp, irq_var2;
* Interrupt handling end
FuncA: call mutex_unlock();
```

Fig. 4. Example of interrupt handling.

variables, namely *shared_set* for *func* and *global_set* for the driver. Before *func* is executed, DILP initializes *shared_set* to the set of pointer-typed arguments of the function. While *func* is executed, DILP intercepts each instrumented instruction *inst* in this function (lines 5-16). For *inst*, DILP gets its result value *ret_val*, and then handles each operand *operand*. If *operand* is a global variable, DILP adds it in *global_set* (lines 8-10). Then, DILP compares *operand* with each stored variable in *shared_set* and *global_set* according to the memory address (lines 11-14). If *operand* belongs to either of the two sets, *operand* is indicated to be a possible shared variable. Accordingly, the affected variable *ret_val* is regarded as a possible shared variable for the function *func*. In this case, DILP adds *ret_val* in *shared_set*, and collects the calling context of variable accesses for *ret_val* in *shared_set*. *func* exits, DILP deletes the set *shared_set* (line 17).

Executed driver functions. DILP intercepts the starting and ending basic blocks of each driver function, to collect the call paths of variable accesses and the call paths of concurrently executed functions. It maintains a list that contains the names of currently executed functions and the IDs of currently running threads. For example, when a driver function *F1* begins to be executed, DILP records the function name and running thread ID as a node in the list. While *F1* is executed, if another driver function *F2* begins to be executed, DILP searches the list and finds the node of *F1*. If the running thread ID of *F1* is different from that of *F2*, it indicates that *F1* and *F2* are executed in different threads at the same time, so *F1* and *F2* can be concurrently executed. In this case, DILP records the call paths of *F1* and *F2* as a call-path pair of concurrently executed functions. When *F1* returns and exits, the corresponding node is deleted from the list.

Lockset. DILP intercepts the calls to lock and unlock functions. It records the function name and arguments, which are used to differentiate the held locks for variable accesses.

Interrupt status. DILP uses specific kernel interfaces (like `in_interrupt` in the Linux kernel) to check interrupt status. Figure 4 uses an example to illustrate the necessity of checking interrupt status. In the example, `FuncA` performs a load (read) operation and a store (write) operation, and then calls the lock function `mutex_lock` to protect the store operation on the variable `var3`. At this time, a hardware interrupt is raised, and the interrupt handling function `IntrFunc` is executed on the thread of running `FuncA`. The function `IntrFunc` performs a load operation on the variable `irq_var1` and a store operation on the variable `irq_var2`, and then returns. The `FuncA` continues to be executed on this thread, and it calls the unlock function `mutex_unlock` to release the lock. If DILP would ignore interrupt status, it would collect two pieces of incorrect information: (1) `IntrFunc` is incorrectly considered to be called by `FuncA`. (2) The load and store operations of `irq_var1` and `irq_var2` are incorrectly considered to be protected by the lock. This is a form of stack ripping [28]. By checking interrupt status when monitoring each function's execution, DILP can avoid such errors. Specifically, when interrupt status is true, DILP creates a separate function call path and the corresponding lockset to record variable accesses and lock usages in interrupt handling.

Calling context. According to possible shared variables, executed functions, locksets and interrupt status, DILP maintains and collects calling contexts of shared-variable accesses. The calling context contains function call path, held locks (lock type and lock object's memory address), accessed variable (variable's memory address) and access type (read or write). It is used to identify possible raced variables in the next phases.

*3) **Race Detection**:* In this phase, with the collected runtime information, DILP first identifies possible raced variables and their accesses, and then detects data races caused by inconsistent lock protection. Finally, DILP produces data-race reports containing the locations of raced variables and information about the context in which the possible race occurs.

To reduce overhead, this phase is performed after driver execution. Specifically, DILP records the runtime information into a log file during driver execution, and performs race detection according to this log file after driver execution.

Figure 5 shows the procedure of race detection. Firstly, according to the calling context of each shared-variable access, DILP identifies a shared variable as a possible raced variable if its access is protected by at least one lock (lines 1-6). After that, DILP gets the set of calling contexts of the possible-raced-variable access (presented as *pos_ctx_set*). Secondly, DILP compares the calling contexts of each possible-raced-variable access and each shared-variable access (line 7-20), and reports a data race if: 1) their call-path pair is in the set

```
Procedure: Detecting data races from collected runtime information
Input: ctx_set – the set of calling contexts of shared-variable accesses
       path_set – the set call-path pairs of concurrently executed functions
 1: pos_ctx_set := ø;
 2: foreach ctx in ctx_set do
 3:     if GetLockSet(ctx) != ø then
 4:         AddSet(ctx, pos_ctx_set);
 5:     end if
 6: end foreach
 7: foreach pos_ctx in pos_ctx_set do
 8:     pos_path := GetCallPath(pos_ctx);
 9:     foreach ctx in ctx_set do
10:         path := GetCallPath(ctx);
11:         if ( (pos_path, path) ∈ path_set and
12:              GetAccessVar(pos_ctx) == GetAccessVar(ctx) and
13:              GetLockSet(pos_ctx) ∩ GetLockSet(ctx) == ø ) then
14:             if (GetAccessType(pos_ctx) == WRITE or
15:                 GetAccessType(ctx) == WRITE) then
16:                 ReportDataRace(pos_ctx, ctx);
17:             end if
18:         end if
19:     end foreach
20: end foreach
```

Fig. 5. Data-race detection with collected runtime information.

of call-path pairs of concurrently executed functions; 2) their accessed variables reference the same memory address; 3) the intersection between their locksets is empty; 4) one of the accesses is a write.

D. Example

To illustrate the behavior of DILP, we consider the Linux *e1000e* driver code in Figure 6(a).

The function `e1000e_get_stats64` calls a lock function `spin_lock` on line 5965, and then calls `e1000e_update_stats`. In `e1000e_update_stats`, the variable `hw->mac.tx_packet_delta` is written on line 4970. Because this variable is accessed from the argument `adapter` of `e1000e_update_stats`, during driver execution, DILP identifies this variable as a possible shared variable, and records the calling context of this variable's access (presented as CTX_q in Figure 6(b)). Besides, the function `e1000_watchdog_task` calls `e1000e_update_adaptive` on line 5349. In `e1000e_update_adaptive`, the variable `mac->tx_packet_delta` is read on line 1780. Because this variable is affected by the function argument `hw`, during driver execution, DILP identifies this variable as a possible shared variable, and records the calling context of this variable's access on line 1780 (presented as CTX_p in Figure 6(b)). Moreover, during driver execution, DILP finds and records that `e1000e_update_adaptive` and `e1000e_update_stats` are concurrently executed with the function call paths (presented as $PairPath_x$ in Figure 6(b)).

After driver execution, DILP analyzes the collected runtime information, shown in Figure 6(b). Because in CTX_p, the access to variable `hw->mac.tx_packet_delta` is protected by a lock, DILP identifies this variable as a possible raced variable. Then, DILP compares CTX_p with other recorded calling contexts. By comparing CTX_q and CTX_p, DILP finds that: 1) their call-path pair $PairPath_x$ is in the recorded call-path pairs; 2) their accessed variables are identical; 3) the lockset in CTX_p is empty but the lockset in CTX_q is non-empty; 4) The access type in CTX_q is write. Thus, DILP reports a data race.

E. Reducing Runtime Overhead

Though DILP focuses on detecting data races and it does not regard performance as a major goal, we still try to reduce its runtime overhead. The overhead is mainly introduced by intercepting variable accesses. We exploit three techniques to reduce this overhead:

1) Dropping repeated information. During driver execution, a variable may be repeatedly accessed in the same calling context. If DILP completely handles all of these variable accesses, much repeated information will be recorded, which can introduce unnecessary overhead and complicate race detection. To solve this problem, DILP compares each calling context of the intercepted variable access with all recorded calling contexts, and drops this calling context if there is a match.

2) Separate recording thread. If the runtime information is recorded in the driver thread, two problems will occur. Firstly, the driver thread will execute much extra code, introducing much runtime overhead. Secondly, as writing files in the kernel can sleep, when the driver thread is executed in atomic context [29] (such as holding a spinlock) and DILP records the runtime information into the log file, a sleep-in-atomic-context bug will occur and can cause system hang or crash [30]. To solve these problems, DILP records the runtime information in a separate kernel thread. When the driver is installed, DILP creates a recording thread and makes this thread sleep. During driver execution, DILP wraps each piece of the runtime information as a message and stores this message in a message queue, and then wakes up the recording thread. The recording thread fetches each message from the message queue, and records the information in this message. When the message queue is empty, the recording thread sleeps again.

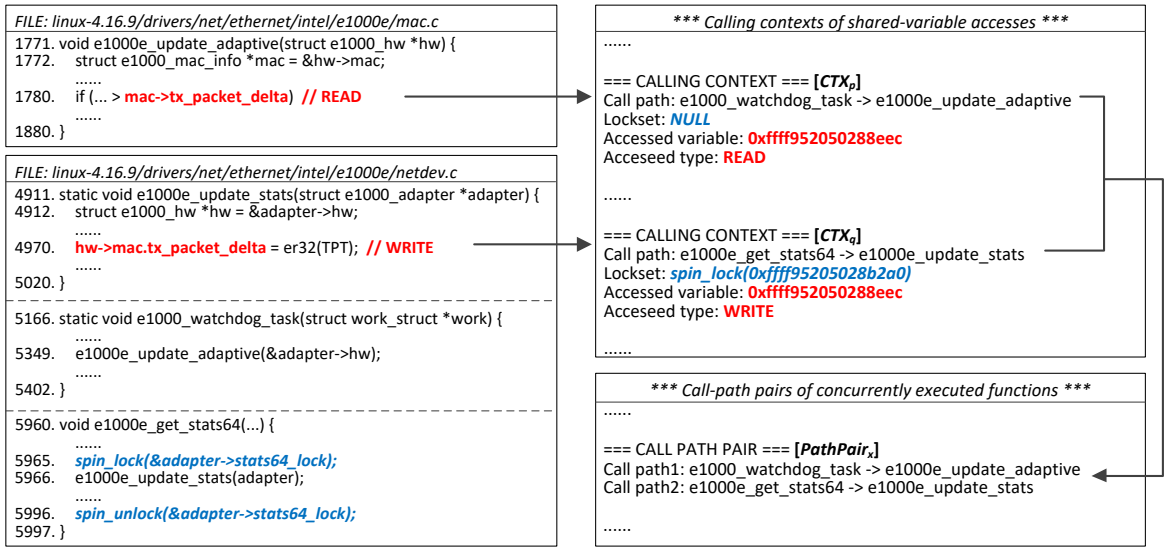
3) Buffer caching. To reduce the runtime overhead of frequently writing to the log file, DILP first caches the runtime information in a memory buffer, and writes to the log file when the buffer is full or the driver is removed.

IV. Evaluation

A. Experimental Setup

To validate the effectiveness of DILP, we evaluate it on real Linux device drivers. The tested drivers are selected according to three criteria: 1) they should be commonly used in practice; 2) they should be within the driver classes in the study in Section II-B, which we have found to have many data races caused by inconsistent lock protection; 3) they should run as kernel modules, so we can enable DILP by installing the DILP kernel module before installing the driver. According to these criteria, we select 12 Linux device drivers, including 6 Ethernet controller drivers, 3 wireless controller drivers and 3 sound card drivers. Table II lists the tested device drivers.

The experiment runs on a common Lenovo PC with four Intel i7-3770@3.40G processors and 8GB physical memory.



(a) Part of the driver source code (b) Part of the collected runtime information

Fig. 6. Example of detecting a data race in the *e1000e* driver.

TABLE II
TESTED DRIVERS

Class	Driver	Controlled Device	LOC
Ethernet	e100	Intel 82559 Ethernet Controller	3.2K
	dl2k	ICPlus IP1000 Ethernet Controller	2.3K
	8139too	Realtek RTL8139 Ethernet Controller	2.7K
	3c59x	3Com 3c905B Ethernet Controller	3.4K
	e1000e	Intel 82572EI Ethernet Controller	29.6K
	tg3	Broadcom BCM5721 Ethernet Controller	21.8K
Wireless	iwl4965	Intel 4965AGN Wireless Controller	29.1K
	b43	Broadcom BCM4322 Wireless Controller	57.1K
	ath9k	Atheros AR5418 Wireless Controller	87.8K
Sound	cmipci	C-Media CM8738 Sound Card	3.4K
	maestro3	ESS ES1988 Allegro-1 Sound Card	2.8K
	ens1371	Ensoniq ES1371 Sound Card	2.5K

TABLE III
WORKLOADS OF TESTING DRIVERS

Class	Workload Description	Commands
Ethernet	Network configuration	*ifconfig, dhcp, nmcli, route*
	Data transmission	*ping, ssh, scp, ftp, wget*
Wireless	Network configuration	*iwconfig, dhcp, nmcli, route*
	Data transmission	*ping, ssh, scp, ftp, wget*
Sound	Sound playing	*aplay, mplayer*
	Sound recording	*arecord*

The driver code is compiled using Clang 5.0 and GCC 5.4. For each tested driver, we install it in the system, and run it with a workload on four threads, and finally remove it. The workloads are shown in Table III.

B. Detecting Data Races

To validate whether DILP can find known data races, we first use it to test the 12 drivers in an old Linux version 3.3.1 (released in April 2012). To validate whether DILP can find new data races, we test the 12 drivers in a recent Linux version 4.16.9 (released in May 2018). We also count the variable accesses and concurrently executed function pairs at runtime. Table IV shows the results of detecting data races. The column *"Variable access"* shows the numbers of all variable accesses (A), possible shared-variable accesses (S) and distinct possible shared-variable accesses (D) recorded by DILP. The column *"Path Pair"* shows the number of call-path pairs of concurrently executed functions recorded by DILP. The column *"Data race"* presents the number of data races found by DILP. From Table IV, we have three findings:

1) DILP can filter out much unnecessary runtime information. For example, for the drivers of Linux 4.16.9, DILP in total intercepts 242M variable accesses, and identifies 31M possible shared-variable accesses by using its dynamic taint analysis. From these variable accesses, DILP in total identifies and records 1.8M distinct possible shared-variable accesses, amounting to 5.8% of possible shared-variable accesses and 0.7% of all variable accesses. Thus, many unnecessary variable accesses are not recorded, which can effectively reduce the runtime overhead and simplify race detection.

2) The results in the column *"Path pair"* of Table IV indicate the degree of the concurrency of the tested drivers. Wireless controller drivers have more call-path pairs of concurrently executed functions than Ethernet controller and sound card drivers, thus wireless controller drivers have higher concurrency with the tested workloads. Indeed, in the experiment, compared to the tested Ethernet controller and sound card drivers, the tested wireless controller drivers use more code and provide more kinds of functionalities that can be concurrently performed.

3) DILP finds 13 and 25 data races in the tested drivers of Linux 3.3.1 and 4.16.9 respectively. We manually check the detected data races, and find all of them are real. By comparing the results and driver code of Linux 3.3.1 and 4.16.9, we find that the two data races found in the *dl2k* driver of Linux 3.3.1

TABLE IV
RACE DETECTION RESULTS.

Class	Driver	Linux 3.3.1			Linux 4.16.9		
		Variable access (A/S/D)	*Path pair*	*Data race*	*Variable access (A/S/D)*	*Path pair*	*Data race*
Ethernet	e100	16887K / 1937K / 91K	164	2	17189K / 2945K / 391K	217	2
	dl2k	16901K / 2972K / 186K	111	2	21513K / 2690K / 144K	103	0
	8139too	17607K / 2610K / 5K	68	3	21794K / 3210K / 7K	80	7
	3c59x	14716K / 1952K / 63K	85	5	26431K / 3739K / 178K	110	7
	e1000e	40420K / 3873K / 179K	938	1	47692K / 4120K / 188K	1155	1
	tg3	10632K / 1092K / 171K	1538	0	10046K / 1017K / 170K	1709	2
Wireless	iwl4965	25064K / 3759K / 424K	10369	0	26372K / 3380K / 455K	11721	6
	b43	33053K / 6364K / 102K	4318	0	35486K / 6111K / 120K	4488	0
	ath9k	24363K / 2694K / 167K	3416	0	26783K / 3104K / 150K	3618	0
Sound	cmipci	927K / 97K / 0.5K	8	0	1153K / 106K / 0.7K	10	0
	maestro3	3437K / 389K / 1.3K	31	0	3255K / 362K / 1.1K	27	0
	ens1371	3892K / 396K / 0.5K	27	0	4184K / 406K / 0.4K	31	0
Total		208M / 28M / 1.4M	21073	13	242M / 31M / 1.8M	23269	25

```
FILE: linux-4.16.9/drivers/net/ethernet/intel/e100.c
 872. static int e100_exec_cb(..., int (*cb_prepare)(...)) {
      ......
 879.   spin_lock_irqsave(...);
      ......
 891.   err = cb_prepare(...);  // call a function pointer
      ......
 925.   spin_unlock_irqrestore(...);
      ......
 928. }
-----------------------------------------------------
1097. static in e100_configure(...) {
      ......
1148.   if (nic->flags & multicast_all)
      ......
1185. }
-----------------------------------------------------
1606. static void e100_set_multicast_list(...) {
      ......
1625.   e100_exec_cb(..., e100_configure);
      ......
1627. }
-----------------------------------------------------
1713. static void e100_watchdog(...) {
      ......
1758.   nic->flags &= ~ich_10h_workaround;
      ......
1762. }
```

Fig. 7. A detected data race in the *e100* driver.

```
##### Data Race #####
READ: e100.c, 1148: e100_set_multicast_list -> e100_exec_cb -> e100_configure
WRITE: e100.c, 1758: e100_watchdog
LOCK: e100.c, 879: e100_set_multicast_list -> e100_exec_cb -> spin_lock_irqsave
```

Fig. 8. Example data-race report generated by DILP.

have been fixed in Linux 4.16.9. Thus, DILP can find known data races. We have also reported the data races found in Linux 4.16.9 to driver developers. 11 of them have been confirmed. We have not yet received a reply for the others. Thus, DILP indeed finds new data races.

Figure 7 shows a data race found by DILP in the *e100* driver of Linux 4.16.9. The function `e100_set_multicast_list` calls `e100_exec_cb` with a function pointer argument referencing `e100_configure` on line 1625. The function `e100_exec_cb` calls `spin_lock_irqsave` to acquire a spinlock, and then calls `e100_configure` via the function pointer argument on line 891. The function `e100_configure` reads the variable `nic->flags` in a branch condition on line 1148. The mentioned function call path is concurrently executed with that of the function `e100_watchdog` during driver execution. This function writes the variable `nic->flags` on line 1758 without lock protection, thus a data race may occur. Figure 8 shows the report generated by DILP for this data race. The report contains useful information about the data race, including the locations of variable accesses and held lock(s) as well as related call paths.

By reviewing the data-race reports generated by DILP, we have some interesting findings about the detected data races:

1) Many of the detected data races involve interrupt handling. Specifically, 9 data races in Linux 3.3.1 and and 14 data races in Linux 4.16.9 are in this case. The main reason is that interrupt handling functions are often concurrently executed with other driver functions, because hardware interrupts are frequently raised when hardware devices work.

2) 2 data races in Linux 3.3.1 and 5 data races in Linux 4.16.9 involve function pointer calls. The data race shown in Figure 7 is such an example. Without runtime information, it is often difficult to correctly identify the set of functions that may be referenced by a function pointer, making such data races difficult to find by statically checking the driver code.

3) The detected raced variables are all data structure fields. The raced variable `nic->flags` in Figure 7 is an example. The main reason is that, to share variables in different driver functions, the driver often wraps these variables in specific data structures, and passes the pointers of these data structures as function arguments. Thus, the driver often accesses shared variables by accessing related data structure fields.

4) Many of detected data races are caused by reading the raced variable in a branch condition without lock protection. Specifically, 7 data races in Linux 3.3.1 and 9 data races in Linux 4.16.9 are in this case. The data race shown in Figure 7 is an example. Indeed, reading a shared variable in a branch condition is a small and often-overlooked operation, so driver developers sometimes forget to protect it with a lock.

C. Runtime Overhead

We measure the runtime overhead introduced by DILP, to check whether it can heavily affect driver execution. To quantify the runtime overhead, we use common benchmarks

TABLE V
PERFORMANCE RESULTS.

Driver	Original		DILP		DILP_slow	
	Throughput	*CPU*	*Throughput*	*CPU*	*Throughput*	*CPU*
e100	94.1Mb/s	1.5%	7.5Mb/s	12.7%	0.1Mb/s	15.39%
dl2k	94.0Mb/s	2.2%	12.3Mb/s	10.4%	0.4Mb/s	18.59%
8139too	90.6Mb/s	1.4%	43.6Mb/s	8.5%	7.9Mb/s	11.0%
3c59x	94.1Mb/s	1.7%	16.5Mb/s	16.4%	0.5Mb/s	19.7%
e1000e	93.9Mb/s	1.3%	7.8Mb/s	10.9%	0.1Mb/s	11.9%
tg3	94.1Mb/s	1.3%	24.7Mb/s	11.5%	0.5Mb/s	14.0%
iwl4965	13.5Mb/s	1.7%	1.2Mb/s	12.1%	0.2Mb/s	13.5%
b43	12.6Mb/s	1.6%	2.2Mb/s	12.8%	0.3Mb/s	18.5%
ath9k	13.4Mb/s	1.7%	1.5Mb/s	12.7%	0.1Mb/s	15.6%
cmipci	-	0.5%	-	1.7%	-	3.3%
maestro3	-	0.7%	-	3.1%	-	5.6%
ens1371	-	0.7%	-	3.5%	-	5.8%

to measure the performance of the original drivers and the drivers instrumented by DILP. Moreover, to quantify the value of the techniques that are used for reducing runtime overhead (described in Section III-E), we also measure the performance of the tested drivers when handled by a modified version of DILP, *DILP_slow*, in which we drop these techniques.

For the Ethernet controller drivers and wireless controller drivers, we use *netperf* [31] to measure the network throughput and CPU utilization when sending 128-byte TCP bulk data blocks (TCP_STREAM). For sound card drivers, we measure the CPU utilization when playing and recording a wave file for thirty seconds. We test each device driver in Linux 4.16.9 five times, and then calculate the average value of the network throughput and CPU utilization. Table V shows the results, and we find that:

1) The runtime overhead introduced by DILP is around 7.2x on average. The network throughput of Ethernet controller and wireless controller drivers is decreased by about 1/7.5 (the overhead is 7.5x) on average, and the CPU utilization of all tested drivers is increased by about 6.8x on average. This runtime overhead is lower than many previous runtime analysis approaches of detecting data races in kernel-level programs, such as Intel's Thread Checker [32] that introduces 200x runtime overhead and Eraser [22] that introduces 10x-30x runtime overhead.

2) The techniques that are used to reduce runtime overhead in DILP are effective. Without these techniques, DILP_slow introduces about 125.9x runtime overhead (281.7x in the network throughput and 9.0x in the CPU utilization) on average, which is much larger than DILP. The high runtime overhead of DILP_slow is introduced by recording repeated possible shared-variable accesses and frequently writing the log file.

In fact, most of the runtime overhead of DILP is caused by intercepting variable accesses during driver execution. We believe that static analysis can help to reduce this runtime overhead. For example, by analyzing the data flow and control flow of the driver code, static analysis can identify the variables that are never shared between different driver threads. Then, the accesses to these variables do not need to be instrumented. This would decrease the amount of the instrumented code, which can help to reduce the runtime overhead.

V. COMPARISON TO EXISTING APPROACHES

Many approaches [15], [33]–[37] use static analysis to detect data races without actually running the program. Among them, RacerX [15] is a well-known static lockset-based approach that can detect data races and deadlocks in OS kernel code. RacerX first extracts control flow graphs and variable information from source files. Then, it exploits an inter-procedual, flow-sensitive and context-sensitive analysis to compute locksets in code paths and detect data races. Finally, it post-processes and ranks the results to generate data-race reports. RacerX finds 13 data races in Linux 2.5, and 6 of them are false positives, giving a false positive rate of 46.2%. Similar to RacerX, DILP is a lockset-based approach, but it is a dynamic approach and exploits exact runtime information for lockset analysis through runtime monitoring. Thus, DILP reports fewer false positives than RacerX.

Several approaches [20]–[23] use runtime analysis to detect data races in kernel-level programs. Among them, we select KernelStrider [23] to make a detailed comparison, for three reasons: 1) it has been used to test some Linux device drivers and found some real data races; 2) its approach is similar to that of DILP, namely intercepting variable accesses to collect runtime information during driver execution and detecting data races according to the collected information after driver execution. 3) its source code is publicly available. KernelStrider is implemented based on KEDR [38], which is a framework of binary-code instrumentation for Linux kernel modules. During driver execution, KernelStrider collects runtime information, and after driver execution, it detects data races according to the collected information.

In design, DILP has three important improvements compared to KernelStrider:

1) KernelStrider relies on the specific static information about kernel interfaces, such as the function type and function name. In the implementation of KernelStrider, this information is hard-coded. But this information is specific to each kernel version. By checking the code history of KernelStrider,[3] we find that the most recent Linux kernel version that it supports is Linux 4.5. We try to run KernelStrider on the tested drivers of Linux 4.16.9, but it fails because the static information of many kernel interfaces is not matched. DILP does not rely on such information, and it automatically analyzes the driver code to perform code instrumentation. Thus, DILP can conveniently test device drivers of different kernel versions.

2) KernelStrider does not identify which driver functions are concurrently executed at runtime. For this reason, it may report false data races when the involved driver functions are never concurrently executed. DILP monitors the execution of driver functions, and identifies call-path pairs of concurrently executed functions to reduce false positives in race detection.

3) KernelStrider uses the interrupt status to maintain function call paths, but does not use this status to maintain locksets. Thus, the locksets maintained by KernelStrider may be incorrect when interrupts occur. DILP uses the interrupt

[3]https://github.com/euspectre/kernel-strider/issues/6

TABLE VI
RACE DETECTION RESULTS OF DILP AND KERNELSTRIDER.

Driver	DILP		KernelStrider	
	Detected	*Real*	*Detected*	*Real*
e100	2	2	6	0
dl2k	2	2	15	1
8139too	3	3	4	0
3c59x	5	5	16	1
e1000e	1	1	85	3
tg3	0	0	47	0
Total	13	13	173	5

status to maintain locksets, which can address this issue. Thus, DILP can reduce the possibility of reporting false data races involving interrupt handling.

We also actually run KernelStrider to test the 6 Ethernet controller drivers of Linux 3.3.1 that are tested by DILP, with the same workloads (shown in Table III) as for DILP. We also manually check the data races detected by KernelStrider, and compare them to DILP's results. The results of data race detection are shown in Table VI.

We find that most data races detected by KernelStrider are false, for two reasons: 1) the related driver functions are never concurrently executed; 2) the maintained locksets are incorrect when interrupts occur. Among the five real data races detected by KernelStrider, one of them is detected by DILP. The other four data races are not detected by DILP, because the related variable accesses are not protected by any lock, and DILP cannot detect data races in such pattern. On the other hand, 12 real data races found by DILP are missed by KernelStrider. Thus, DILP has a lower false positive rate than KernelStrider, and finds many data races missed by KernelStrider.

VI. DISCUSSION

A. Threats to Validity

The main threats to internal validity are about the lock synchronization mechanism and concurrency analysis of DILP. Firstly, because DILP is a lockset-based approach, it cannot detect data races involving other synchronization mechanism, such as memory barriers. Secondly, we analyze driver concurrency at function-level granularity, which is a trade off between accuracy and performance. Many previous approaches (such as [20], [21]) analyze driver concurrency at instruction-level granularity, to achieve good accuracy and reproduce real data races, but they often introduce much overhead.

The main threats to external validity are about the introduced overhead and code coverage. Firstly, though DILP uses some techniques to reduce runtime overhead, it still introduces 7.2x runtime overhead on average in our evaluation. In this case, the driver concurrency may be affected during execution, and thus DILP may miss some real data races. To mitigate this threat, we will introduce static analysis to reduce the amount of instrumented code. Secondly, as DILP is a dynamic approach, its code coverage heavily relies on the tested workloads and driver configuration. In the experiment, we only use common workloads and common configurations to test drivers in normal cases. Thus, only some normal execution code paths are covered and related data races are found. To improve code coverage and find more data races, we will introduce fault injection and fuzzing techniques to improve code coverage and detect more data races.

B. Generality

Besides the tested drivers in the evaluation, DILP can be easily applied to other device drivers, because DILP works automatically and does not require specific information of the tested drivers. Besides Linux device drivers, DILP can be ported in other operating systems (such as FreeBSD and NetBSD) to test their device drivers, because DILP does not rely on specific features of the OS kernel.

VII. RELATED WORK

A. Dynamic Analysis

Many approaches use dynamic analysis to detect data races with runtime information. They are based on the happens-before relation [39], sampling [40] or lockset algorithm [22].

Happens-before-based approaches [41]–[44] track memory addresses and synchronization events to infer the temporal happens-before relation between two events. When two conflicting memory accesses α and β are on the same memory location, and neither α happens before β nor β happens before α, a data race may occur. For example, Djit$^+$ [41] exploits vector time frames to track each shared-variable access, and checks whether this access has the happens-before relation with prior accesses to the shared variable. These approaches report no false positives, but they often miss many real data races, and introduce much overhead due to tracking and infering the happens-before relation during program execution. To our best knowledge, no happens-before-based approach has been used to test device drivers.

Sampling-based approaches [5], [7], [14], [20], [21] monitor variable accesses at intervals instead of tracking all variable accesses, and thus they can achieve better performance than happens-before-based approaches. For example, LiteRace [14] is an effective sampling-based approach to detect data races in user-level applications. It uses adaptive sampling to track infrequently accessed regions in the program, and detects related data races. DataCollider [20] is a well-known sampling-based approach to detect data races in the Windows kernel. It randomly samples a small set of memory accesses. To increase the possibility of capturing concurrent accesses to the identical memory addresses, it delays the current thread for a short time. It also uses hardware breakpoints to set data breakpoints at the access location to trap any second access during delay. If the second access happens, a real data race is detected. However, the effectiveness of sampling-based approaches heavily relies on the sampling frequency. They may miss real data races when the sampling frequency is low, and may introduce much overhead when the sampling frequency is high.

Lockset-based approaches [11], [22], [23], [45]–[47] maintain locksets of shared variables and running threads, and detect data races by computing the intersection between the

locksets of each accessed shared variable and its runing thread. Eraser [22] was the first lockset-based approach. It performs binary-code instrumentation to achieve runtime monitoring of shared-variable accesses for each thread, and detects data races by checking whether consistent locksets are used. Eraser can test both user-level programs and kernel-level programs. However, lockset-based approaches often report false positives, because they cannot ensure that the involved shared variables of reported data races are actually concurrently accessed during program execution.

Our approach DILP uses the idea of locksets but it is different from previous lockset-based approaches. Firstly, DILP focuses on data races caused by inconsistent lock protection, and compares the locksets of the accesses to the same variables in different threads. Secondly, DILP identifies the functions that are concurrently executed, to help reduce false positives. Thirdly, specific to device drivers, DILP considers interrupt handling to reduce the possibility of reporting false races when interrupts occur.

B. Static Analysis

Many approaches use static analysis to detect data races without actually running the program. They are based on flow-insensitive type-based analysis (such as [33]–[35]) or flow-sensitive lockset-based analysis (such as [15], [36], [37]). For example, Flanagan et al. [33] propose a type-based analysis approach for Java programs. This approach introduces formal type annotations to capture synchronization patterns, and detects data races according to these patterns. RacerX [15] is a well-known lockset-based analysis approach that can detect data races and deadlocks in OS kernel code, and we have introduced RacerX in Section V.

Static analysis can conveniently detect data races without running the tested programs. But due to lacking exact runtime information, it often reports many false positives. For DILP, it could be interesting to introduce static analysis to identify the variables that are never shared between different driver threads. Accesses to these variables do not need to be instrumented. This would decrease the amount of the instrumented code, which can help to reduce runtime overhead.

C. Symbolic Execution

Some approaches [17], [48]–[50] use symbolic execution to detect data races. Basically, symbolic execution uses a symbolic value to replace the concrete value of a variable, and explores code paths by recording and solving path constraints. When exploring code paths, symbolic execution can perform a lockset analysis and detect data races. For example, WHOOP [49] is a symbolic execution approach of detecting data races in device drivers. It uses over-approximation and a symbolic pairwise lockset analysis to attempt to prove a driver race-free. It also uses some optimizations based on device-driver-domain knowledge to reduce the amount of analyzed memory regions.

Symbolic execution is in effective in achieving high code coverage for data race detection. However, for symbolic execution, solving path constraints and exploring code paths are often time-consuming when analyzing complex and large-scale driver code.

VIII. Conclusion

In device drivers, many data races are caused by a common pattern that we call inconsistent lock protection. To detect such data races, we propose a runtime-analysis approach, named DILP. It uses code instrumentation to monitor driver execution and collect runtime information. Then, it performs an offline analysis of the collected information, to detect data races caused by inconsistent lock protection. We evaluated DILP on 12 Linux device drivers. It found 25 new real data races, and 11 of them have been confirmed by driver developers.

DILP can be still improved in some aspects. Firstly, as a dynamic approach, DILP does not cover the entire driver code at runtime and may miss data races in the uncovered code. To address this limitation, we will introduce fault injection and fuzzing techniques to improve code coverage and detect more data races. Secondly, the runtime overhead of DILP can be further reduced. To achieve this goal, we will introduce static analysis to reduce the amount of instrumented code. Finally, we only implement DILP to test Linux device drivers. We will port DILP in operating systems to test their device drivers.

Acknowledgment

We would like to thank the Linux driver developers who gave helpful feedback on our data-race reports. This work was supported by the Natural Science Foundation of China (project number 61872210).

References

[1] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser, "Dingo: taming device drivers," in *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*, 2009, pp. 275–288.

[2] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[3] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008, pp. 329–339.

[4] "Data race," https://software.intel.com/en-us/inspector-user-guide-linux-data-race.

[5] Y. Cai, J. Zhang, L. Cao, and J. Liu, "A deployable sampling strategy for data race detection," in *Proceedings of the 2016 International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 810–821.

[6] P. Fonseca, C. Li, and R. Rodrigues, "Finding complex concurrency bugs in large multi-threaded applications," in *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, 2011, pp. 215–228.

[7] M. D. Bond, K. E. Coons, and K. S. McKinley, "PACER: proportional detection of data races," in *Proceedings of the 31st International Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 255–268.

[8] S. Lu, S. Park, and Y. Zhou, "Detecting concurrency bugs from the perspectives of synchronization intentions," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, no. 6, pp. 1060–1072, 2011.

[9] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *Proceedings of the 27th International Conference on Programming Language Design and Implementation (PLDI)*, 2006, pp. 308–319.

[10] S. Lu, S. Park, and Y. Zhou, "Finding atomicity-violation bugs through unserializable interleaving testing," *IEEE Transactions on Software Engineering (TSE)*, no. 4, pp. 844–860, 2012.

[11] C. Von Praun and T. R. Gross, "Object race detection," in *Proceedings of the 16th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001, pp. 70–82.

[12] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," in *Proceedings of the 30th International Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 121–133.

[13] X. Xie, J. Xue, and J. Zhang, "Acculock: accurate and efficient detection of data races," *Software: Practice and Experience (SPE)*, vol. 43, no. 5, pp. 543–576, 2013.

[14] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: effective sampling for lightweight data-race detection," in *Proceedings of the 30th International Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 134–143.

[15] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 237–252.

[16] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and accurate static data-race detection for concurrent programs," in *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*, 2007, pp. 226–239.

[17] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler, "Static race detection for device drivers: the Goblint approach," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, 2016, pp. 391–402.

[18] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang, "Static data race detection for concurrent programs with asynchronous calls," in *Proceedings of the 2009 International Symposium on Foundations of Software Engineering (FSE)*, 2009, pp. 13–22.

[19] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher, "Model checking concurrent Linux device drivers," in *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE)*, 2007, pp. 501–504.

[20] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel." in *Proceedings of the 9th International Conference on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 151–162.

[21] Y. Jiang, Y. Yang, T. Xiao, T. Sheng, and W. Chen, "DRDDR: a lightweight method to detect data races in Linux kernel," *The Journal of Supercomputing*, vol. 72, no. 4, pp. 1645–1659, 2016.

[22] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[23] "KernelStrider: detecting data races in Linux kernel modules by collecting runtime information," https://github.com/euspectre/kernel-strider.

[24] "LLVM compiler infrastructure," https://llvm.org/.

[25] "Patchwork project for the Linux kernel," https://patchwork.ozlabs.org/, https://patchwork.kernel.org/, https://patchwork.linuxtv.org/.

[26] A. Kadav and M. M. Swift, "Understanding modern device drivers," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 87–98.

[27] "Clang compiler," http://clang.llvm.org/.

[28] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management." in *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002, pp. 289–302.

[29] J. Corbet, "Atomic context and kernel api design," 2008, https://lwn.net/Articles/274695/.

[30] J.-J. Bai, Y.-P. Wang, J. Lawall, and S.-M. Hu, "DSAC: effective static analysis of sleep-in-atomic-context bugs in kernel modules," in *Proceedings of the 2018 USENIX Annual Technical Conference*, 2018, pp. 587–600.

[31] "Netperf benchmark," http://www.netperf.org/netperf/.

[32] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas, "Accurate and efficient filtering for the Intel thread checker race detector," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006, pp. 34–41.

[33] C. Flanagan and S. N. Freund, "Type-based race detection for Java," in *Proceedings of the 21st International Conference on Programming Language Design and Implementation (PLDI)*, 2000, pp. 219–232.

[34] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller, "Automated type-based analysis of data races and atomicity," in *Proceedings of the 10th International Symposium on Principles and Practice of Parallel programming (PPoPP)*, 2005, pp. 83–94.

[35] C. Flanagan and S. N. Freund, "Detecting race conditions in large programs," in *Proceedings of the 2001 International Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001, pp. 90–96.

[36] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: context-sensitive correlation analysis for race detection," in *Proceedings of the 27th International Conference on Programming Language Design and Implementation (PLDI)*, 2006, pp. 320–331.

[37] S. Keul, "Tuning static data race analysis for automotive control software," in *Proceedings of the 11th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2011, pp. 45–54.

[38] V. V. Rubanov and E. A. Shatokhin, "Runtime verification of Linux kernel modules based on call interception," in *Proceedings of the 4th International Conference on Software Testing, Verification and Validation (ICST)*, 2011, pp. 180–189.

[39] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[40] M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code," in *Proceedings of the 22nd International Conference on Programming Language Design and Implementation (PLDI)*, 2001, pp. 168–179.

[41] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded C++ programs," in *Proceedings of the 9th International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003, pp. 179–190.

[42] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *Proceedings of the 39th International Symposium on Principles of Programming Languages (POPL)*, 2012, pp. 387–400.

[43] A. K. Rajagopalan and J. Huang, "RDIT: race detection from incomplete traces," in *Proceedings of the 2015 International Symposium on Foundations of Software Engineering (FSE)*, 2015, pp. 914–917.

[44] M. Prvulovic and J. Torrellas, "ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes," in *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, 2003, pp. 110–121.

[45] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: a race and transaction-aware java runtime," in *Proceedings of the 28th International Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 245–255.

[46] O. Shacham, M. Sagiv, and A. Schuster, "Scaling model checking of dataraces using dynamic information," in *Proceedings of the 10th International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005, pp. 107–118.

[47] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu, "Efficient data race detection for distributed memory parallel programs," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 51:1–51:12.

[48] M. Said, C. Wang, Z. Yang, and K. Sakallah, "Generating data race witnesses by an SMT-based analysis," in *Proceedings of the 3rd International Symposium on NASA Formal methods (NFM)*, 2011, pp. 313–327.

[49] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and precise symbolic analysis of concurrency bugs in device drivers," in *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, 2015, pp. 166–177.

[50] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: static race detection on millions of lines of code," in *Proceedings of the 2007 International Symposium on Foundations of Software Engineering (FSE)*, 2007, pp. 205–214.