This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3138735, IEEE Transactions on Software Engineering

Hybrid Static-Dynamic Analysis of Data Races Caused by Inconsistent Locking Discipline in Device Drivers

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

Abstract—Data races are often hard to detect in device drivers. According to our study of Linux driver patches that fix data races, about 39% of patches involve a pattern that we call *inconsistent locking discipline*. 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 hybrid static-dynamic analysis approach, named SDILP, to detect data races caused by inconsistent locking discipline in device drivers. SDILP has a dynamic lockset analysis to detect data races at runtime, and a static lockset analysis to detect more data races based on the dynamic-analysis results. It also performs a static taint analysis to reduce the number of variable accesses monitored by the dynamic analysis. Compared to our previous dynamic approach DILP [1], introducing static analysis allows SDILP to achieve better performance and find more data races. We evaluate SDILP on 12 drivers in Linux 5.4, and find 117 real data races, 50 of which have been confirmed by driver developers.

Index Terms—Data race, inconsistent locking discipline, device driver, static analysis, dynamic analysis.

1 Introduction

CONCURRENT execution in device drivers improves program performance on modern multi-core processors, but can introduce concurrency problems. Studies [2]–[4] have shown that many of the reported driver bugs are due to concurrency problems, and that these concurrency problems are often hard to detect [3]. A common kind of concurrency problem is a data race. A data race occurs when multiple threads access the same memory location without proper synchronization and at least one access is a write [5]. Data races can introduce non-determinism, making them hard to debug, and they can cause serious problems such as functionality errors and undefined behaviors.

Most previous approaches [6]–[19] to detecting data races target user-level applications, and these approaches have some important requirements satisfied by user-level code. For example, many approaches, such as Helgrind [17], DPST [18] and FSAM [19], require to start concurrency analysis from a fixed entry point, such as a `main()` function. Besides, many approaches, such as Acculock [12], Chord [10] and CTrigger [11], require to completely monitor thread creation and deletion, in order to infer a happens-before relation [20].

However, kernel-level device drivers do not satisfy these requirements, and avoiding these requirements would require significant modification to the design and implementation of the existing user-level approaches. More importantly, without these requirements, existing user-level approaches cannot work or they may report lots of false results. On the one hand, existing user-level approaches that start concurrency analysis from one fixed entry point cannot test device drivers, as a driver does not have such a fixed entry point [21], [22]. A possible solution is to regard kernel-driver interfaces and interrupt-handler functions as multiple entry points (similar to DCUAF [23]), but the concurrency analysis of these user-level approaches would have to be redesigned for two important reasons. First, alias analysis starting from one entry point is ineffective for identifying alias relationships of data flows starting from multiple entry points. Second, analyzing data-flow interleavings from multiple entry points is more complex than from one entry point. On the other hand, device drivers typically do not run in their own threads, so it is not feasible to completely monitor thread creation and deletion for happens-before-relation inference by analyzing only driver code. A possible solution is to use memory watchpoints to monitor the order of accesses based on physical clocks. However, observing a given order is not sufficient to ensure that the same order must always occur. Instead, a happens-before relation should be inferred based on logical clocks determined by several clock conditions [14], [24], such as thread creation and deletion order of accessed memory. Memory watchpoints can also introduce much runtime overhead. Finally, a practical difficulty of driver race detection is that the driver code can be interrupted, by an interrupt handler that shares the same stack. In this case, a dynamic analysis that does not distinguish between regions of the stack may incorrectly consider that locks held by the driver are providing protection for the accesses of the interrupt handler.

To detect data races in kernel-level device drivers, some previous approaches [25]–[29] use static analysis, but they often report many false positives, due to lacking exact runtime information about memory accesses and synchronization. Several previous approaches [30]–[34] use dynamic

- *Jia-Ju Bai, Qiu-Liang Chen, Zu-Ming Jiang and Shi-Min Hu are with the Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, China. E-mail: baijiaju@tsinghua.edu.cn, {chenql16, jzm18}@mails.tsinghua.edu.cn, shimin@tsinghua.edu.cn.*
- *Julia Lawall is with Inria, Paris, France. E-mail: julia.lawall@inria.fr.*

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

analysis to detect data races in drivers. They are based on sampling or lockset analysis. Sampling-based approaches, such as DataCollider [30], DRDDR [31] and KCSAN [34], check variable accesses in real time at a given sampling frequency. However, they may miss real data races when the sampling frequency is low, and they may introduce much runtime overhead when the sampling frequency is high. To avoid the dependency on the sampling frequency and detect more data races, lockset analysis approaches, such as Eraser [32] and KernelStrider [33], check the set of locks protecting shared-variable accesses.

The traditional lockset analysis (Eraser) [32] 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 variable that may be shared (referred to subsequently as a *possible shared variable*) *v*, globally across the execution, the analysis maintains a set *C(v)* that contains the set of locks that are held across all previous observed accesses to *v*. *C(v)* initially contains all of the locks available in the system. When an access to *v* occurs, *C(v)* is updated to the intersection of *C(v)* with the currently held set of locks. A race is reported if *C(v)* becomes empty. This lockset is based on an over-aggressive hypothesis that all accesses to a given variable *v* can occur concurrently with each other. However, this hypothesis is false in many cases. For example, initialization may occur before the variable is shared, in which case a data race is impossible. Thus, the traditional lockset analysis often reports many false positives. Moreover, to maintain locksets, this analysis instruments all variable accesses and monitors them, which can introduce much runtime overhead. Besides, this analysis may miss data races involving variables that are not accessed in a particular execution. To address these issues, we propose two new techniques.

First, to reduce false positives, we refine the lockset analysis by adding more constraints on the set of variables that are considered and the locksets that are compared. Specifically, we only compare locksets of accesses occurring in functions that are actually executed concurrently, thus providing stronger evidence that the accesses may conflict. Besides, we perform this check only when one of the accesses is a write and when one of the accesses is protected by at least one lock (we refer to such an accessed variable as a *possible raced variable*), reflecting that developers understand 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.* We call the pattern causing this data race *inconsistent* locking discipline, analogous to the consistent locking discipline used to describe Eraser [32]. Compared to the traditional lockset analysis, our strategy can drastically reduce the number of false positives, as it only reports data races when there is strong evidence that a race is possible.

Second, we introduce static analysis to enhance dynamic analysis. On the one hand, to reduce the runtime overhead of monitoring variable accesses, we perform a static taint analysis of the driver code to identify possible shared-variable accesses, and only instrument such accesses for runtime monitoring, instead of all variable accesses. On the other hand, we observe that some variable accesses have the same contexts as the found data races, namely they access the same variables, occur in the same function and hold the same locks, but these accesses are not covered by dynamic analysis. Such accesses can also cause data races, but are missed by dynamic analysis. Based on this observation, according to the data races found by dynamic analysis, we perform a static lockset analysis of the driver code to detect data races missed by dynamic analysis.

Based on the above ideas, we propose a hybrid static-dynamic approach named SDILP, to detect data races caused by inconsistent locking discipline in device drivers. SDILP consists of three phases. First, at compile time, SDILP performs a static taint analysis of the driver code, to identify and instrument the variable accesses affected by global variables and pointer-typed function arguments, because they are often the source of shared variables [26], [35]. Second, during driver execution, SDILP monitors the instrumented variable accesses and executed driver functions. It records information about each instrumented variable access, including the location, lockset, accessed variable, etc., and identifies the driver functions that are concurrently executed at runtime. With this runtime information, SDILP performs a dynamic lockset analysis to detect data races caused by inconsistent locking discipline. Finally, after driver execution, according to the data races found by the dynamic analysis, SDILP performs a static lockset analysis of the driver code to detect data races missed by the dynamic analysis. We have implemented SDILP using the Clang compiler [36].

The hybrid static-dynamic analysis of SDILP is novel for data-race detection in device drivers. In particular, this analysis is the first to extend the race-detection results of dynamic analysis via static analysis. Moreover, different from existing hybrid static-dynamic approaches [37], [38] that use static analysis to identify possible raced instruction pairs, SDILP uses static taint analysis to identify possible shared-variable accesses, which can help to find more data races involving complex alias relationships missed by these approaches. Compared to our previous approach DILP [1], SDILP can detect more data races with less runtime overhead, benefiting from this hybrid static-dynamic analysis. Overall, we make four main contributions:

- We study Linux driver patches and find that about 39% of the patches fixing data races involve inconsistent locking discipline, indicating that many reported data races in device drivers are caused by this issue.
- We propose to use static analysis to enhance dynamic analysis in data-race detection, in two respects: 1) reducing the runtime overhead, and 2) extending the race-detection results.
- We implement a hybrid static-dynamic approach named SDILP, to effectively detect data races caused by inconsistent locking discipline in device drivers.
- We evaluate SDILP on 12 device drivers in Linux 3.3.1 (April 2012) and 5.4 (November 2019). SDILP finds 165 real data races in Linux 3.3.1, and 64 of them have been fixed in Linux 5.4. SDILP finds 117 real data races in Linux 5.4, and 50 of them have been confirmed by driver developers. We also compare SDILP to two existing kernel race checkers (KernelStrider and KCSAN), and SDILP finds many data races missed by these checkers.

The rest of this paper is organized as follows. Section 2 motivates our work. Section 3 introduces SDILP. Section 4 introduces the evaluation. Section 5 gives some discussion. Section 6 presents related work, and Section 7 concludes.

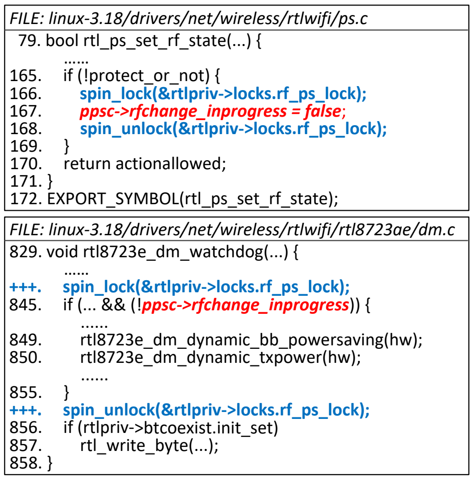
2 Motivation

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

2.1 Motivating Example

Figure 1 shows a real data race in the *rtl8723ae* device driver. This data race was first introduced in Linux 3.18 (released in December 2014). 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, while holding a spinlock acquired on line 166. However, in the function `rtl8723e_dm_watchdog`, the variable `ppsc->rfchange_inprogress` is read on line 845 without holding the spinlock. Thus, a data race on `ppsc->rfchange_inprogress` may occur. This data race has been confirmed to be harmful by the developer. Indeed, the raced variable `ppsc->rfchange_inprogress` is checked in the branch condition on line 845 and affects the execution of various functions related to power management (line 849 and 850) that should not be used when a driver state change is in progress. This data race was first fixed in Linux 4.8 (released in October 2016), by acquiring the spinlock in `rtl8723e_dm_watchdog`.[1] Namely, it persisted over 10 mainline releases (nearly 2 years).

This example illustrates the pattern of inconsistent locking discipline 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.* Compared to cases where the two involved accesses hold no locks, inconsistent locking discipline provides stronger evidence of concurrent execution, because at least one of the involved accesses is protected by a lock.



```
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);
```

```
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)) {
             ......
849.         rtl8723e_dm_dynamic_bb_powersaving(hw);
850.         rtl8723e_dm_dynamic_txpower(hw);
             ......
855.     }
+++.     spin_unlock(&rtlpriv->locks.rf_ps_lock);
856.     if (rtlpriv->btcoexist.init_set)
857.         rtl_write_byte(...);
858. }
```

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

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

TABLE 1
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%

2.2 Study of Linux driver patches

To understand the proportion of reported data races caused by inconsistent locking discipline, we study Linux driver patches in the Patchwork project for OS kernel [39], which is used by a number of Linux kernel maintainers to collect patches pending for the next release. We select the accepted race-fixing patches from April 2015 to April 2018, by searching the patch titles. We focus on the patches of 6 driver classes: wireless controller, Ethernet controller, sound card, media, MMC (multimedia card) and RDMA drivers. We find 252 accepted patches that fix data races. Among these patches, we manually identify those that explicitly involve inconsistent locking discipline in the driver code. The results are shown in Table 1, including the driver class, the number of the accepted race-fixing patches (Race), and the number (Pattern) and percentage (Proportion) of accepted patches that involve inconsistent locking discipline.

From Table 1, we find that 39% of the accepted race-fixing patches involve inconsistent locking discipline. The remaining patches target other patterns of data races, such as accessing the involved variable without holding any lock or without disabling interrupts where needed. In the patches involving inconsistent locking discipline, most data races are fixed by: 1) adding new calls to lock and unlock functions to protect the raced variables; or 2) moving existing calls to lock and unlock functions to a place that can protect the raced variables. These fixes suggest that the fixed races were introduced because: 1) the driver developer forgot that the function containing the access can be concurrently executed with another function accessing the same shared variable, and thus did not add lock protection; or 2) the driver developer remembered to add some lock protection, but the lock protection is incomplete. Indeed, these two reasons are difficult to avoid in driver development, as whether two variable accesses can be concurrently executed is often hard to statically determine [21], [22], [40], [41]. Thus, inconsistent locking discipline is likely to be introduced, which may cause data races. Considering that the percentage 39% is not small, even though it might not precisely reflect the rate of such races in the Linux kernel, we believe that it is worthy to design a specific and effective approach to accurately detect the data races caused by inconsistent locking discipline in device drivers.

3 Approach

In this section, we first introduce our core idea of hybrid static-dynamic race detection, then present the architecture and phases of SDILP, and finally discuss its improvements over our previous approach DILP [1].

3.1 Core Idea

To improve the performance and race-detection coverage of dynamic analysis, our core idea is introducing static analysis to achieve two benefits. First, inspired by Spindle [42], which uses static analysis to reduce the runtime overhead of dynamic analysis in memory-error detection, we use a static taint analysis to identify which variable accesses are related to possible shared variables; indeed, other variable accesses are useless in data-race detection. In this way, we can monitor only the identified accesses, to effectively reduce the runtime overhead of dynamic analysis. Second, we use a static lockset analysis to detect data races that involve the same contexts as the reported data races but are not covered by the dynamic analysis. Besides data races, we believe this idea is applicable to detecting other kinds of problems via dynamic analysis.

Based on our core idea, we perform three steps to detect data races caused by inconsistent locking discipline. First, at compile time, we perform a static taint analysis of the driver code, to identify and instrument the variable accesses affected by global variables and pointer-typed function arguments, because they are often the source of shared variables [26], [35]. We also instrument driver-function entry and exit points, to help identify concurrently executed driver functions. Second, during driver execution, we monitor the instrumented variable accesses and executed driver functions. We record information about each instrumented variable access, including location, lockset, accessed variable, etc., and identify concurrently executed functions. With this information, we perform a dynamic lockset analysis to detect data races caused by inconsistent locking discipline. Finally, after driver execution, according to data races found by the dynamic analysis, we perform a static lockset analysis of the driver code to detect data races missed by the dynamic analysis.

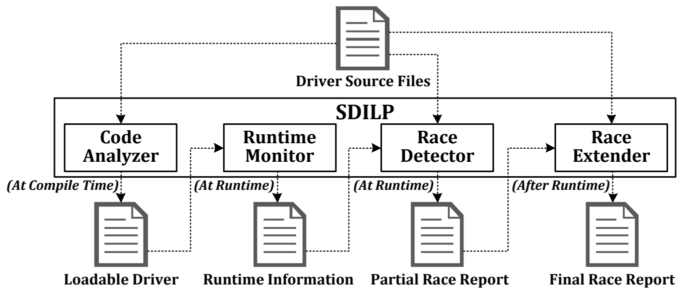


Fig. 2. Overall architecture of SDILP.

3.2 Architecture

Based on the idea in Section 3.1, we design a hybrid static-dynamic approach named SDILP, to detect data races caused by inconsistent locking discipline in device drivers. We have implemented SDILP with Clang [36], and perform analysis and code instrumentation on driver LLVM bytecode. Figure 2 shows the architecture of SDILP, which has four parts:

- ***Code analyzer.*** This part compiles the driver source code, performs static taint analysis and instrumentation of the driver LLVM bytecode, and finally generates a loadable driver.
- ***Runtime monitor.*** This part uses the instrumented code to monitor driver execution and collect runtime information. To avoid modifying the OS kernel, this part is implemented as a kernel module.
- ***Race detector.*** This part uses dynamic lockset analysis to analyze the collected runtime information and detect data races during driver execution.
- ***Race extender.*** This part uses static lockset analysis to detect data races missed by dynamic analysis after driver execution, by analyzing the driver code and the data races found by dynamic analysis.

3.3 Phases

3.3.1 Code Analysis

In this phase, SDILP performs two main tasks:

Identifying possible shared-variable accesses. In a driver function, some kinds of variables cannot be shared by multiple threads, such as local variables and non-pointer-typed function arguments, and thus the accesses to such variables are not useful to detecting data races. In fact, shared variables are often related to global variables and pointer-typed function arguments [26], [35]. Thus, if a variable access is related to such variables, SDILP identifies it as a possible shared-variable access and instruments this access for runtime monitoring; otherwise, SDILP does not handle this access. For this purpose, SDILP performs a static taint analysis of the driver code to identify possible shared-variable accesses at compile time.

Because SDILP is designed to detect data races, the main challenge of our static taint analysis is how to accurately and efficiently identify possible shared-variable accesses in driver code. On the one hand, to reduce false negatives of data-race detection in dynamic analysis, it is important to identify all shared-variable accesses in static taint analysis. For this purpose, our static taint analysis performs *flow-sensitive* taint analysis to accurately identify possible shared variables. On the other hand, a device driver has many functions and variable accesses, and thus it is important to reduce the analysis time. For this purpose, our static taint analysis performs efficient *intra-procedural* analysis of each driver function, and avoids infinite looping on loops and recursive calls.

Figure 3 presents the static taint analysis for a driver function *func*. It produces a set *access_set* that collects all instructions of possible shared-variable accesses in the function. The analysis first initializes *access_set* to the empty set (line 1), and then analyzes each possible code path (lines 2-23). To avoid infinite looping on loops and recursive calls, the analysis of a code path ends when encountering a basic block that is already been handled in this code path. When analyzing a code path, the analysis uses a set *val_set* to store all possible shared variables at the LLVM bytecode level, and initializes this set with the pointer-typed function arguments and global variables (lines 3-9). Then, the analysis checks each instruction *inst* in the code path (lines 10-22). For *inst*, the analysis gets its result variable *res_val* (if *inst* has no result variable, *res_val* is a null value), the set of operands *op_val_set* and the instruction type *inst_type*. The analysis checks whether *op_val_set* and *val_set* have a non-empty intersection, to judge whether *inst* has an operand identified

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3138735

Procedure: Identifying possible shared-variable accesses in the function *func*

```
1:  access_set := ø;
2:  foreach code_path in GetCodePathSet(func) do
3:      val_set := ø;
4:      foreach arg_val in GetArgPtrSet(func) do
5:          AddSet(arg_val, val_set);
6:      end foreach
7:      foreach global_val in GlobalValSet(func) do
8:          AddSet(global_val, val_set);
9:      end foreach
10:     foreach inst in GetInstSetInPath(code_path) do
11:         res_val := GetResultVal(inst);
12:         op_val_set := GetOperandVal(inst);
13:         inst_type := GetInstType(inst);
14:         if CheckSetIntersect(op_val_set, val_set) then
15:             if GetValType(res_val) == PointerType then
16:                 AddSet(res_val, val_set);
17:             end if
18:             if inst_type == LOAD or inst_type == STORE then
19:                 AddSet(inst, access_set);
20:             break;
21:         end if
22:     end foreach
23: end foreach
```

Fig. 3. Taint analysis of identifying possible shared-variable accesses.

```
FILE: linux-5.4/drivers/net/ethernet/broadcom/tg3.c
7915. static netdev_tx_t tg3_start_xmit(struct sk_buff *skb,
7916.                                   struct net_device * dev) {
7917.     struct tg3 *tp = netdev_priv(dev);  // read dev
7918.     u32 base_flags;
          ......
7954.     base_flags = 0;  // write base_flags
          ......
8171.     tp->tx_dropped++;  // read and write tp->tx_dropped
8172.     return NETDEV_TX_OK;
8173. }
```

Fig. 4. Example of identifying possible shared-variable accesses.

as a possible shared variable. If so, *inst* is considered to be tainted. If its result variable *res_val* is pointer-typed, *res_val* is identified as a possible shared variable and added into *val_set* with no repetition. In this case, if *inst* is a read or write operation (namely a *LOAD* or *STORE* instruction in the LLVM bytecode), *inst* is identified as an instruction accessing a possible shared variable and is added into *access_set* with no repetition. After the analysis, *access_set* stores all identified instructions for possible shared-variable accesses, which will be instrumented and monitored.

Figure 4 illustrates the taint analysis using the *tg3* driver code. In the function `tg3_start_xmit`, the variables `skb` and `dev` are pointer-typed function arguments and thus are considered as possible shared variables. Accordingly, on line 7917, the read operation of `dev` is identified as a possible shared-variable access. Moreover, `tp` is assigned by invoking a function on `dev`, and thus `tp` is tainted by `dev` and identified as a possible shared variable. On line 7954, the variable `base_flags` is written, but it is not tainted by any pointer-typed function argument or global variable, and thus its access is not identified as a possible shared-variable access. Finally, as `tp` is identified as a possible shared variable, the read and write operations of `tp->tx_dropped` on line 8171 are identified as possible shared-variable accesses.

In fact, several static approaches [23], [25] can identify possible raced instruction pairs, and their accessed variables are considered to be possible shared variables. This strategy could be useful to further reduce the runtime overhead of dynamic analysis. However, as these approaches neglect alias relationships in lockset analysis, many real shared variables involving alias relationships could be unidentified, causing related data races to be missed in dynamic analysis. Thus, SDILP does not use this strategy, and instead uses the taint analysis in Figure 3 to reduce false negatives.

Code instrumentation. SDILP instruments three kinds of places in the driver LLVM bytecode:

- *Calls to lock and unlock functions.* SDILP uses them to collect the locksets of variable accesses at runtime. Specifically, SDILP uses the names of common lock and unlock functions (like `spin_lock` and `mutex_lock`) as described in Linux kernel documents about locking [43].
- *Driver functions.* SDILP instruments the entry and exit points of each driver function, to identify concurrently executed driver functions at runtime.
- *Possible shared-variable accesses.* SDILP collects information about them for race detection at runtime.

3.3.2 Online Race Detection

In this phase, using the instrumented code, SDILP collects runtime information and performs dynamic lockset analysis to detect data races during driver execution.

Runtime monitoring. SDILP monitors possible shared-variable accesses and maintains locksets for these accesses. Moreover, SDILP monitors and identifies driver functions that are concurrently executed. Indeed, SDILP only performs lockset analysis of the variable accesses in these functions, to reduce false positives of data-race detection. As a function can be called from different call chains in different concurrency contexts, SDILP collects the call chains for concurrently-executed functions. Besides, SDILP specially monitors interrupt handling, as interrupt handlers always reuse the normal runtime call stack of the driver. In detail, these kinds of information are collected as follows:

Possible shared-variable accesses. SDILP monitors each possible shared-variable access, to record the memory address and type information (the data type, and the data structure type and field name, if any) of the accessed variable, the access type (read or write) and the access location. For device drivers, a challenge of dynamic analysis is that a variable may have the same memory address as another variable that has already been freed by other kernel modules invisible to dynamic analysis. To distinguish between two variables having the same memory address, SDILP uses their type information. For example, suppose that there are two write operations `dev->data = 5` and `skb->pos = 10`, where `dev->data` and `skb->pos` use the same heap-memory address at different times in the execution. The two variables' data types are both integers, but their data structure types and fields are different, and thus SDILP can identify them as different variables.

Locksets. SDILP records the calls to lock and unlock functions to collect locksets. For each such call, SDILP records its location, the function name and the related lock object's memory address.

Driver functions. SDILP instruments the entry and exit points of each driver function to collect the call chains of variable accesses and the call chains of concurrently executed functions. SDILP maintains a set that contains the names of the currently executed functions and the IDs of the currently running threads. For example, when a driver function *F1* begins to be executed, SDILP adds the function name and running thread ID to the front of the set. While *F1* is being executed, if another driver function *F2* begins

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3138735

to be executed, SDILP looks for the entry of *F1* in the set. If the running thread ID of *F1* is different from that of *F2*, then *F1* and *F2* are executed in different threads at the same time, and thus *F1* and *F2* are considered to be concurrently executed. In this case, SDILP records the call chains of *F1* and *F2* as a call-chain pair of concurrently executed functions. When *F1* returns, the corresponding entry is deleted from the set.

Interrupt handling. Hardware interrupts often occur during driver execution, and may cause runtime monitoring to record a call stack that mixes the interrupt context (at the top of the stack) with the driver context (lower in the stack) [22]. Specifically, when a hardware interrupt is raised while a driver function is executing, the driver suspends executing the current function F to execute the corresponding interrupt handler. In this case, a lock acquired in the function F may be mistakenly considered to be used in the interrupt handler, producing incorrect locksets. To solve this problem, SDILP calls the specific kernel interface `in_interrupt` in Linux to check whether the currently executed function is in interrupt handling. When the driver executes an interrupt handler, SDILP maintains a separate call chain and lockset.

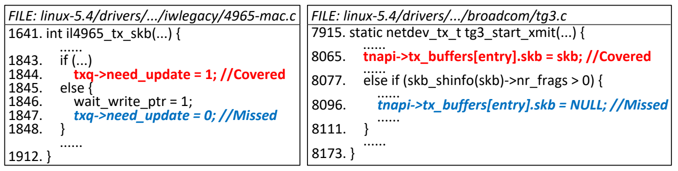
Race detection. Our previous approach DILP [1] collects runtime information in a log file during driver execution, and performs lockset analysis of the log file after driver execution. However, we observe that DILP introduces much runtime overhead caused by writing to the log file when many variable accesses occur at runtime, and thus it sometimes fails to run complex workloads normally. To solve this problem, SDILP collects runtime information in a memory buffer and performs an online lockset analysis, without writing to a log file. Specifically, when a driver function F ends its execution, given the collected runtime information, SDILP first identifies the functions that have finished their execution and are concurrently executed with the function F, and then performs a lockset analysis of these functions and F with their calling contexts. After all the functions that are concurrently executed with the function F have completed and been analyzed, SDILP drops the collected runtime information about F to save memory. Compared to DILP, SDILP has the extra runtime overhead of performing the online lockset analysis, but it eliminates the runtime overhead caused by writing to the log file. As a whole, in our tests, SDILP has lower runtime overhead than DILP.

Figure 5 shows the procedure of this lockset analysis for a function F and the set F_{set} containing the functions that are concurrently executed with F and finish before the end of the execution of F. First, SDILP identifies the accesses to possible raced variables in F, if an access is protected by at least one lock (lines 1-6). The information about these accesses is collected in *race_access_set*. Second, SDILP compares each possible raced-variable access in F and each possible shared-variable access in F_{set} (lines 7-20), and reports a data race if: 1) their accessed variables reference the same memory address and have the same type information; 2) the intersection between their locksets is empty; 3) at least one of them is a write.

Separate thread for dynamic analysis. For dynamic analysis, runtime monitoring and race detection require many extra operations, and thus executing them on the driver thread can introduce much runtime overhead. Moreover, many of these extra operations require synchronization, which can heavily affect driver concurrency. To solve these problems, SDILP creates a separate thread to perform dynamic analysis. Specifically, the collected runtime information is wrapped as a message, which is passed to the message queue of this separate thread. This thread fetches the messages from the message queue, to record runtime information and perform data-race detection.

```
Procedure: Detecting data races by analyzing the runtime information
Input: access_set – Possible shared-variable accesses in F
       con_access_set –Possible shared-variable accesses in F_set
 1: race_access_set := ø;
 2: foreach access in con_access_set do
 3:     if GetLockSet(access) != ø then
 4:         AddSet(access, race_access_set);
 5:     end if
 6: end foreach
 7: foreach access in race_access_set do
 8:     foreach con_access in con_access_set do
 9:         if (GetAccessVar(access) == GetAccessVar(con_access) and
10:             GetLockSet(access) ∩ GetLockSet(con_access) == ø ) then
11:             if (GetAccessType(access) == WRITE or
12:                 GetAccessType(con_access) == WRITE) then
13:                 ReportDataRace(access, con_access);
14:             end if
15:         end if
16:     end foreach
17: end foreach
```

Fig. 5. Data-race detection by analyzing the runtime information.


```
FILE: linux-5.4/drivers/.../iwlegacy/4965-mac.c
1641. int il4965_tx_skb(...) {
          ......
1843.     if (...)
1844.         txq->need_update = 1; //Covered
1845.     else {
1846.         wait_write_ptr = 1;
1847.         txq->need_update = 0; //Missed
1848.     }
          ......
1912. }
```
(a) Two data races in the *iwl4965* driver

```
FILE: linux-5.4/drivers/.../broadcom/tg3.c
7915. static netdev_tx_t tg3_start_xmit(...) {
          ......
8065.     tnapi->tx_buffers[entry].skb = skb; //Covered
          ......
8077.     else if (skb_shinfo(skb)->nr_frags > 0) {
              ......
8096.         tnapi->tx_buffers[entry].skb = NULL; //Missed
              ......
8111.     }
          ......
8173. }
```
(b) Two data races in the *tg3* driver

Fig. 6. Example of a data race missed by dynamic analysis.

3.3.3 Offline Race Extension

In this phase, with the data races found by the dynamic analysis, SDILP uses static lockset analysis of the driver code after driver execution, to extend the race-detection results.

Figure 6(a) shows two real data races in the *iwl4965* driver. One of them is found by the dynamic analysis but the other one is missed. The dynamic analysis finds a data race involving the variable `txq->need_update` on line 1844, as the code on this line is covered at runtime. As shown in the code, this variable is also accessed on line 1847 with the same context (including the lockset and the caller) as the found data race, and thus there should be another data race on line 1847. However, the code on line 1847 is not covered at runtime, so dynamic analysis misses this data race. In this example, the extra race is near the race found by the dynamic analysis, but this is not always the case. For example, Figure 6(b) shows another case in the *tg3* driver. The dynamic analysis finds a data race on line 8065 at runtime, but misses another similar data race on line 8096 because the related code is not covered at runtime.

To detect such missed data races, according to the data races found by dynamic analysis, SDILP performs a static lockset analysis of the driver code, as shown in Figure 7. For a variable access of a given data race, namely *race_access*, SDILP first identifies the function that includes this access as *caller*. Then, SDILP performs intra-procedural and flow-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3138735, IEEE Transactions on Software Engineering

Procedure: Extending race-detection results using static analysis
Input: *race_access* – The variable access of a given data race *race*

```
caller := GetCaller(race_access);
access_set := GetAccessSet(caller);
foreach access in access_set do
    if (GetAccessVar(access) == GetAccessVar (race_access) and
        GetLockSet(access) == GetLockSet(race_access)) then
        pair_race_access := GetRacePairAccess(race_access, race);
        if (GetInstType(access) == WRITE or
            GetInstType(pair_race access) == WRITE) then
            ReportDataRace(access, pair_race_access);
        end if
    end if
end foreach
```

Fig. 7. Static analysis of the driver code to find data races.

TABLE 2
Tested device drivers

Class	Driver	Hardware devices	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	29.3K
	tg3	Broadcom BCM5721 Ethernet Controller	21.8K
Wireless	iwl4965	Intel 4965AGN Wireless Controller	28.8K
	b43	Broadcom BCM4322 Wireless Controller	56.6K
	ath9k	Atheros AR5418 Wireless Controller	88.2K
Sound	cmipci	C-Media CM8738 Sound Card	3.4K
	maestro3	ESS ES1988 Allegro-1 Sound Card	2.8K
	ens1371	Ensoniq ES1371 Sound Card	2.9K

TABLE 3
Workloads of testing drivers

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

sensitive analysis to collect the lockset of each variable access in *caller*, and produces a resulting set *access_set*. After that, SDILP checks each item *access* in *access_set*. If the accessed variable and lockset of *access* are identical to those of *race_access*, namely the two accesses involve the same variable and hold the same locks, then *access* may also cause a data race. In this case, SDILP gets the variable access paired with *race_access* in the given data race, namely *paired_race_access*, and reports a new data race of *access* and *paired_race_access*, if at least one of them is a write operation.

3.4 Improvements over DILP

Compared to our previous approach DILP [1], SDILP introduces static analysis and achieves two main benefits. First, DILP monitors all variable accesses in the driver, and thus it introduces much runtime overhead. To solve this problem, SDILP performs a static taint analysis to identify possible shared-variable accesses at compile time, and only monitors these variable accesses at runtime. Thus, SDILP can achieve better performance than DILP, which increases the possibility of finding more data races in the given testing time. Second, according to the data races found by dynamic analysis, SDILP performs a static lockset analysis of the driver code, to extend the results of data-race detection. Thus, SDILP can find some data races missed by DILP.

4 Evaluation

In this section, we evaluate SDILP on Linux device drivers and compare it to existing race-detection approaches.

4.1 Experimental Setup

To validate the effectiveness of SDILP, we evaluate it on real Linux device drivers. The tested device 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 2.2, which we have found to have many data races caused by inconsistent locking discipline; 3) they should run as kernel modules, so that we can enable SDILP by installing the SDILP 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 2 lists some information about the tested drivers of Linux 5.4.

The experiment runs on a common Lenovo PC with four Intel i7-3770@3.40G processors and 8GB physical memory. The driver source code is compiled using Clang 9.0 and GCC 6.5. We test each driver 5 times, due to the non-determinism of driver concurrency. During each test, we install a driver in the system, and run it with workloads on 4 threads, and finally remove it. The workloads are shown in Table 3, and each of them is executed with a 10-second timeout. We select these workloads because they are commonly used and related to the tested drivers [44], [45]. Note that the numbers used in the tests (such as 5 times and 4 threads) are randomly selected with no special purpose, and they can be changed as needed.

To validate whether SDILP can find known data races, we use it to test the 12 drivers in an old Linux version 3.3.1 (released in April 2012). To validate whether SDILP can find new data races, we use it to test the 12 drivers in a newer Linux version 5.4 (released in November 2019). To show the improvements of SDILP over our previous approach DILP [1], we evaluate both approaches on Linux 3.3.1 in Sections 4.2 and 4.4. In Section 4.6.1, we also experimentally compare SDILP to two state-of-the-art kernel race checkers, namely KernelStrider [33] and KCSAN [34]). As we focus on detecting data races not concurrency bugs, we consider that both benign and harmful races are real races, as done by many existing approaches [6], [14], [15], [46].

4.2 Detecting Data Races

Table 4 shows the results of data-race detection. The column *"Access site"* shows the number of instrumented variable-access sites in the driver LLVM bytecode; the column *"Race"* shows the number of found data races. As SDILP uses both dynamic and static lockset analyses to detect data races, the column *"Race"* of SDILP also shows the number of data races found by the two analyses. From Table 4, we find that:

1) SDILP drops around 42% of the variable-access sites instrumented by DILP. Most of the dropped variable-access sites are only related to local variables that are unlikely to be shared by different threads, so they are useless for data-race detection. However, DILP instruments all variable accesses in the driver code, including these useless variable-access sites. Thus, SDILP can help reduce the runtime overhead of monitoring variable accesses during dynamic analysis.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3138735, IEEE Transactions on Software Engineering

TABLE 4
Race detection results.

Class	Driver	DILP (Linux 3.3.1)		SDILP (Linux 3.3.1)		SDILP (Linux 5.4)	
		Access site	*Race*	*Access site*	*Race (Dynamic+Static)*	*Access site*	*Race (Dynamic+Static)*
Ethernet	e100	3092	2	2093	4 (1+3)	2127	16 (3+13)
	dl2k	1826	2	1100	7 (4+3)	1174	8 (7+1)
	8139too	1857	3	1204	3 (3+0)	1263	1 (1+0)
	3c59x	3253	5	1831	16 (13+3)	1720	15 (12+3)
	e1000e	18607	1	10509	37 (33+4)	11050	26 (22+4)
	tg3	17013	0	10952	0 (0+0)	12310	11 (6+5)
Wireless	iwl4965	22436	0	12899	45 (37+8)	12300	27 (20+7)
	b43	26126	0	15117	2 (1+1)	16912	5 (4+1)
	ath9k	59903	0	32778	47 (24+23)	42706	4 (3+1)
Sound	cmipci	2744	0	1533	4 (2+2)	1570	4 (2+2)
	maestro3	1615	0	951	0 (0+0)	993	0 (0+0)
	ens1371	1871	0	1239	0 (0+0)	1273	0 (0+0)
Total		160343	13	92206	165 (118+47)	105398	117 (80+37)

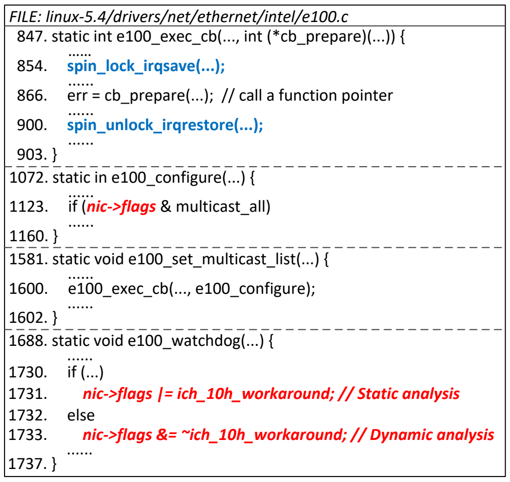

```
FILE: linux-5.4/drivers/net/ethernet/intel/e100.c
847. static int e100_exec_cb(..., int (*cb_prepare)(...)) {
       ......
854.   spin_lock_irqsave(...);
       ......
866.   err = cb_prepare(...);  // call a function pointer
       ......
900.   spin_unlock_irqrestore(...);
       ......
903. }
1072. static in e100_configure(...) {
       ......
1123.  if (nic->flags & multicast_all)
       ......
1160. }
1581. static void e100_set_multicast_list(...) {
       ......
1600.  e100_exec_cb(..., e100_configure);
       ......
1602. }
1688. static void e100_watchdog(...) {
       ......
1730.  if (...)
1731.     nic->flags |= ich_10h_workaround; // Static analysis
1732.  else
1733.     nic->flags &= ~ich_10h_workaround; // Dynamic analysis
       ......
1737. }
```

Fig. 8. Two found data races in the *e100* driver.

2) SDILP finds 165 and 117 data races in the tested drivers of Linux 3.3.1 and 5.4, respectively. We manually check these data races, and identify that all of them are real. We also find that 64 of the data races in Linux 3.3.1 have been fixed in Linux 5.4. Thus, SDILP can find known data races. We have reported the data races found in Linux 5.4 to the related driver developers, and 50 of them have been confirmed. We have not yet received a reply for the others. Thus, SDILP can find new data races.

3) Among the data races found by SDILP, 47 of 165 in Linux 3.3.1 and 37 of 117 in Linux 5.4 are found by the static lockset analysis, because the variable accesses of these data races are not covered by our test workloads at runtime but have the same context as the variable accesses of a data race found by dynamic analysis. These results show that the static lockset analysis of SDILP is effective in finding data races missed by dynamic analysis.

Figure 8 presents two data races found by SDILP in the Linux 5.4 *e100* Ethernet controller driver. The function `e100_set_multicast_list` calls `e100_exec_cb` with a function pointer argument referencing `e100_configure` on line 1600. 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 866. The function `e100_configure` reads the variable `nic->flags` in a branch condition on line 1123. During driver execution, the mentioned function call chain is concurrently executed with that of the function `e100_watchdog`, in which the variable `nic->flags` is written on line 1733 without lock protection. SDILP reports a data race here. After driver execution, SDILP performs static lockset analysis, and finds that `nic->flags` is also written on line 1731, while holding the same locks as the write operation on line 1733. Thus, SDILP also reports a data race on line 1731.

4) For the tested drivers of Linux 3.3.1, SDILP finds all the data races found by DILP, and it also finds 152 data races missed by DILP for two reasons. First, SDILP uses a static taint analysis to reduce the runtime overhead of dynamic analysis, and thus it can more efficiently execute the test workloads, which increases the probability of finding more data races in a given testing time. This reason helps SDILP find 106 data races missed by DILP. Second, SDILP uses a static lockset analysis to detect data races missed by the dynamic analysis. This reason helps SDILP find 46 data races missed by DILP.

5) We check the 13 data races found by DILP in Linux 3.3.1, and find that 4 of them (2 in dl2k and 2 in 8139too) have been fixed in Linux 5.4. Besides these 4 data races, SDILP also finds 60 data races fixed in Linux 5.4 but missed by DILP. Interestingly, according to the 4 fixed data races found by DILP, SDILP's static lockset analysis finds 3 additional data races (in dl2k) by extending these data races. These 3 data races are fixed in the same patch as those 4 data races in Linux 5.4.

6) Among the 47 races found by static lockset analysis in Linux 3.3.1, 37 races are fixed in Linux 5.4. 25 of these 37 races are fixed in the same patches as 27 races found by dynamic lockset analysis, and 12 remaining races are fixed in other patches. Among the 37 races found by static lockset analysis in Linux 5.4, 24 have been confirmed by developers. 18 of the confirmed data races have been fixed. All these 18 fixed races are fixed in the same patches as 20 races found by dynamic lockset analysis. Indeed, for each tested driver, we reported all the data races found by SDILP to the developers, without describing which data races are found by dynamic or static lockest analysis. For this reason, the developers confirmed or fixed these data races together, without knowing how the data races were originally found.

Reviewing the data races found by SDILP, we find that:

1) Interrupt handling is a significant source of the found data races. Specifically, 117 data races in Linux 3.3.1 and 66 data races in Linux 5.4 occur in interrupt handlers. Indeed, developers may overlook the driver concurrency introduced by interrupt handling, causing these data races.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3138735

2) Many of the found data races involve function pointers in the concurrently-executed function-call chains. Specifically, 39 data races in Linux 3.3.1 and 21 data races in Linux 5.4 have this property. The two data races shown in Figure 8 are such examples. Without exact runtime information, it is often hard to correctly identify the set of functions referenced by a function-pointer call, making such data races hard to find by statically checking the driver code.

3) All the found data races involve data structure fields. An example is the raced variable `nic->flags` in Figure 8. To share variables in different driver functions executed on different threads, device drivers often wrap these variables in specific data structures, and pass pointers to these data structures as function arguments. Thus, device drivers often access shared variables via data structure fields.

4.3 Impact of the Found Data Races

We estimate the impact of the found data races, from two points of view. On the one hand, for control flow, if the raced variable is in the branch condition (such as "if" and "while"), the related data race is identified to be harmful, because it affects the control flow. On the other hand, for data flow, if the raced variable is an array index or pointer-access offset, the related data race is identified to be harmful, because it affects the access to array element or pointer at runtime, which can cause unexpected behaviors of the driver. Among the 165 data races found in Linux 3.3.1, we identify that 94 of them to be harmful, including the 64 data races fixed in Linux 5.4; among the 117 data races found in Linux 5.4, we identify that 65 of them can be harmful, including the 50 data races confirmed by driver developers.

For the 94 harmful data races in Linux 3.3.1, 62 can influence control flow and 32 can influence data flow; for the 65 harmful data races in Linux 5.4, 45 can influence control flow and 20 can influence data flow. In our experience, developers are more likely to fix such harmful data races, as they have high impact on data flow and control flow, which can cause reliability and security problems at runtime. As for benign data races, developers tend not to handle or fix benign data races, for two reasons. First, their impact on data flow and control flow is low, and thus race conditions are allowed at runtime. Second, adding locks to fix benign data races may also degrade program performance.

Figure 9 shows a harmful data race found by SDILP in the Linux 5.4 *cmipci* sound card driver. During our runtime testing, the functions `snd_cmipci_pcm_trigger` and `snd_cmipci_interrupt` are concurrently executed. SDILP detects that the write of `rec->running` on line 897 and the read of `cm->channel[0].running` on line 1454 access the same data, but the write is protected by the spinlock `cm->reg_lock` while the read is not, and thus a data race can occur. The write on line 897 is performed when the sound is being stopped, and the read on line 1454 is performed when the interrupt is handled. Thus, due to this data race, when the sound is being stopped and the interrupt is handled at the same time, the variable `cm->channel[0].running` can be non-zero, and thus the function `snd_pcm_period_elapsed` on line 1455 can be called to update the PCM (Pulse Code Modulation) status and sound card buffer. However, in fact, when the sound is not played, the PCM status and sound card buffer should not be updated. Otherwise, if the sound is started again, the PCM status and sound card buffer may be incorrect, which can cause the sound to be played abnormally. The related driver developer also confirmed the harmfulness of this race and said it could damage the sound card's functionality.

```
FILE: linux-5.4/sound/pci/cmipci.c
874. static int snd_cmipci_pcm_trigger {
        ......
885.    spin_lock(&cm->reg_lock);
        ......
896.    case SNDRV_PCM_TRIGGER_STOP:  // Stop the sound
897.       rec->running = 0;
        ......
920.    spin_unlock(&cm->reg_lock);
921.    return result;
922. }
-------------------------------------------------------------
1430. static irqreturn_t snd_cmipci_interrupt(...) {
        ......
1454.   if ((status & CM_CHINT0) && cm->channel[0].running)
1455.      snd_pcm_period_elapsed(...); // Update the PCM status
        ......
1460. }
```

Fig. 9. A harmful data race in the *cmipci* driver.

TABLE 5
Performance results.

Driver	Original		DILP		SDILP	
	Throughput	*CPU*	*Throughput*	*CPU*	*Throughput*	*CPU*
e100	94.1Mb/s	1.5%	7.5Mb/s	12.7%	45.5Mb/s	9.5%
dl2k	94.0Mb/s	2.2%	12.3Mb/s	10.4%	85.5Mb/s	7.8%
8139too	90.6Mb/s	1.4%	43.6Mb/s	8.5%	75.3Mb/s	3.9%
3c59x	94.1Mb/s	1.7%	16.5Mb/s	16.4%	36.1Mb/s	13.5%
e1000e	93.9Mb/s	1.3%	7.8Mb/s	10.9%	46.1Mb/s	8.6%
tg3	94.1Mb/s	1.3%	24.7Mb/s	11.5%	83.1Mb/s	11.1%
iwl4965	13.5Mb/s	1.7%	1.2Mb/s	12.1%	8.7Mb/s	10.3%
b43	12.6Mb/s	1.6%	2.2Mb/s	12.8%	8.1Mb/s	10.8%
ath9k	13.4Mb/s	1.7%	1.5Mb/s	12.7%	7.4Mb/s	9.5%
cmipci	-	0.5%	-	1.7%	-	1.0%
maestro3	-	0.7%	-	3.1%	-	1.9%
ens1371	-	0.7%	-	3.5%	-	1.8%

At present, we manually estimate the impact of the data races found by SDILP. Several existing approaches [47]–[49] can automatically classify data races by their impact, through analyzing the source code. These approaches are orthogonal to SDILP, and they can be used to reduce the manual work of identifying harmful races found by SDILP.

4.4 Runtime Overhead

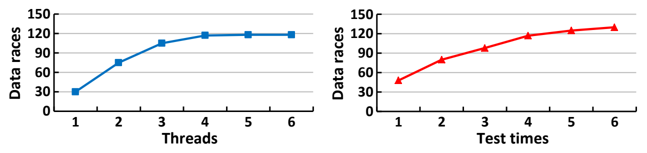
We measure the runtime overhead introduced by SDILP, to check whether it can significantly impact driver execution. To quantify the runtime overhead, we use common benchmarks to measure the performance of the original drivers and the drivers instrumented by SDILP. Moreover, to quantify the value of SDILP's static analysis used for reducing runtime overhead, we also measure the performance of the drivers instrumented by DILP.

For the Ethernet controller drivers and wireless controller drivers, we use *netperf* [50] to measure the network throughput and CPU utilization when sending 128-byte TCP bulk data blocks. For sound card drivers, we measure the CPU utilization when playing and recording a wave file for thirty seconds. We test each device driver 5 times, and calculate the average value of the network throughput and CPU utilization. Table 5 shows the results. We find that:

1) The runtime overhead introduced by SDILP is 3.7x on average. Specifically, the network throughput of the Ethernet controller and wireless controller drivers is decreased by 1/1.7 (the overhead is 1.7x), and the CPU utilization of all

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3138735, IEEE Transactions on Software Engineering



(a) Changing threads (b) Changing test times

Fig. 10. Result variation for the numbers of workload threads and tests.

tested drivers is increased by 5.3x. This runtime overhead is lower than many previous dynamic-analysis approaches to detecting data races in kernel-level programs, such as Intel's Thread Checker [51] that introduces 200x runtime overhead and Eraser [32] that introduces 10x-30x runtime overhead.

2) SDILP achieves lower runtime overhead than DILP. Specifically, for DILP, the network throughput of the Ethernet controller and wireless controller drivers is decreased by 1/7.5 (the overhead is 7.5x), and the CPU utilization of all tested drivers is increased by 6.8x. The runtime overhead introduced by DILP is 7.2x on average, which is about double the runtime overhead introduced by SDILP. During driver execution, DILP monitors all variable accesses at runtime, while SDILP only monitors possible shared-variable accesses identified by static analysis as involving global variables or pointer-typed function arguments. Thus, SDILP monitors much fewer variable accesses than DILP, which greatly reduces the runtime overhead.

4.5 Result Variation

As described in Section 4.1, we test each driver 5 times and run each workload with 4 threads. To check how these numbers affect race-detection results of SDILP, we perform two experiments. First, we test each driver 1-6 times, and still run each workload with 4 threads. Second, we still test each driver 5 times, and run each workload with 1-6 threads. Figure 10 shows the results for the 12 drivers of Linux 5.4.

We find that increasing the numbers of workload threads or tests enables finding more data races, as more thread interleavings can be covered. However, such benefit becomes smaller as the numbers of workload threads or tests increase, because fewer and fewer new thread interleavings are covered. For this reason, blindly increasing the numbers of workload threads and tests cannot always obviously benefit data-race detection in real-world experiments. Besides, we note that SDILP still finds 30 data races when each workload is run with just one thread. Indeed, even a single-thread workload can involve multiple kernel threads and trigger driver interrupt handling, which can in turn cause data races in drivers.

4.6 Comparison to Existing Approaches

In this subsection, we focus on comparing SDILP to existing race-detection approaches that can test Linux device drivers.

4.6.1 KernelStrider and KCSAN

KernelStrider [33] is an open-source data-race checker used for Linux kernel modules. Its race-detection strategy is similar to that of SDILP, namely first intercepting variable accesses to collect runtime information during driver execution and then detecting data races according to the collected information with a lockset analysis. Note that different from SDILP, KernelStrider only performs lockset analysis after program execution. In design, SDILP has four important advantages compared to KernelStrider:

TABLE 6
Comparison results of data-race detection

Driver	Linux 3.3.1		Linux 5.4	
	KernelStrider (fixed/real/all)	*SDILP (fixed/real/all)*	*KCSAN (confirmed/real/all)*	*SDILP (confirmed/real/all)*
e100	0/0/6	0/4/4	1/1/1	16/16/16
dl2k	1/1/15	7/7/7	2/2/2	8/8/8
8139too	0/0/4	3/3/3	0/0/0	1/1/1
3c59x	0/1/16	3/16/16	0/0/0	0/15/15
e1000e	3/3/85	12/37/37	1/1/1	7/26/26
tg3	0/0/47	0/0/0	1/1/1	5/11/11
Total	4/5/173	25/67/67	5/5/5	37/77/77

TABLE 7
Comparison results of performance.

Driver	Original		KernelStrider		KCSAN		SDILP	
	Throughput	*CPU*	*Throughput*	*CPU*	*Throughput*	*CPU*	*Throughput*	*CPU*
e100	94.1Mb/s	1.5%	94.0Mb/s	3.5%	80.3Mb/s	6.3%	45.5Mb/s	9.5%
dl2k	94.0Mb/s	2.2%	94.0Mb/s	5.3%	90.5Mb/s	5.8%	85.5Mb/s	7.8%
8139too	90.6Mb/s	1.4%	90.5Mb/s	2.8%	79.6Mb/s	3.2%	75.3Mb/s	3.9%
3c59x	94.1Mb/s	1.7%	94.1Mb/s	5.2%	75.2Mb/s	6.3%	36.1Mb/s	13.5%
e1000e	93.9Mb/s	1.3%	93.9Mb/s	3.5%	77.5Mb/s	4.6%	46.1Mb/s	8.6%
tg3	94.1Mb/s	1.3%	94.0Mb/s	3.8%	85.5Mb/s	5.2%	83.1Mb/s	11.1%

1) KernelStrider relies on static information about kernel interfaces, such as the function type and the function name, that is hard-coded in the implementation of KernelStrider. However, this information is specific to each kernel version. The most recent Linux kernel version that KernelStrider supports is Linux 4.5.[2] We have tried to run KernelStrider on the drivers we have tested in Linux 5.4, but it fails because the static information of many kernel interfaces is outdated. SDILP does not rely on hard-coded information. Instead, it automatically analyzes the driver code to perform code instrumentation. Thus, SDILP can conveniently test device drivers of different kernel versions.

2) KernelStrider does not identify which driver functions are concurrently executed. Thus, it may report false data races when the involved driver functions are never concurrently executed. SDILP monitors the execution of driver functions, and identifies concurrently executed functions to reduce false positives in race detection.

3) KernelStrider uses the interrupt status to maintain function call chains, but does not use this status to maintain locksets. Thus, the locksets maintained by KernelStrider may be incorrect when interrupts occur. SDILP uses the interrupt status to maintain both function call chains and locksets, which can reduce the possibility of reporting false data races involving interrupt handling.

4) KernelStrider relies only on dynamic analysis. Besides dynamic analysis, SDILP also uses a static lockset analysis to extend the race-detection results of dynamic analysis.

With some manual configuration, we have successfully run KernelStrider to test the six Ethernet controller drivers of Linux 3.3.1 that are tested by SDILP. The remaining six drivers are not Ethernet controller drivers, and supporting them would require extra manual work. These six drivers are tested with the same workloads (shown in Table 3) as for SDILP. Table 6 shows the results of data-race detection.

2. https://github.com/euspectre/kernel-strider/issues/6

We find that most of the data races found by KernelStrider are false, either because the related driver functions are never concurrently executed, or because the maintained locksets are incorrect when interrupts occur. On the one hand, among the 5 real data races found by KernelStrider, one is also found by SDILP. The other 4 data races are missed by SDILP, because the involved variable accesses are not protected by any lock, namely they are not caused by inconsistent locking discipline. On the other hand, 66 real data races found by SDILP are missed by KernelStrider. We infer that KernelStrider misses these races because it maintains incorrect locksets at runtime, especially when analyzing interrupt handling functions. In summary, SDILP achieves a lower false positive rate than KernelStrider, and finds many data races missed by KernelStrider.

Among the 5 real data races found by KernelStrider, 4 have been fixed in Linux 5.4, including one found by SDILP; among the 67 real data races found by SDILP, 25 have been fixed in Linux 5.4, including 24 missed by KernelStrider.

KCSAN [34] is an open-source data-race checker that has been integrated in the Linux kernel since October 2019. While SDILP uses lockset analysis, KCSAN uses code breakpoints and a sampling strategy to monitor memory accesses and catch data races at runtime. It is similar to DataCollider [30]. We have successfully run KCSAN to test the six Ethernet controller drivers in Table 6 in Linux 5.4, as KCSAN does not support old kernel versions like Linux 3.3.1. These six drivers are tested with the same workloads (shown in Table 3) as for SDILP.

We find that the 5 real data races found by KCSAN are all found by SDILP, and SDILP finds 72 real data races missed by KCSAN. Indeed, KCSAN uses hardware breakpoints to inspect memory-access instruction pairs, but catching two concurrently-executed instructions that have race conditions is difficult in practice, due to the non-determinism of thread interleavings. By using dynamic and static lockset analysis, SDILP extends the race-detection scope to all the memory accesses in concurrently-executed functions, which can find many data races missed by KCSAN.

Besides, the 5 real data races found by KCSAN are all confirmed by driver developers; among the 77 real data races found by SDILP, 37 of them have been fixed in Linux 5.4, including the 5 found by KCSAN.

Runtime overhead. Identical to Section 4.4, we measure the network throughput and CPU utilization when sending 128-byte TCP bulk data blocks, to calculate runtime overhead. Table 7 shows the detailed results.

We find that SDILP introduces higher runtime overhead than KernelStrider and KCSAN. Specifically, KernelStrider hardly decreases the network throughput, and increases the CPU utilization by 2.6x, so its average runtime overhead is 1.3x; KCSAN decreases the network throughput by 1.2x, and increases the CPU utilization by 3.4x, so its average runtime overhead is 2.3x; SDILP decreases network throughput by 1.7x, and increases the CPU utilization by 5.9x, so its average runtime overhead is 3.8x. Indeed, compared to KernelStrider and KCSAN, SDILP performs more operations during execution, such as collecting concurrently-executed functions, maintaining interrupt status and performing online lockset analysis.

Usability analysis. KernelStrider relies on static information about kernel interfaces, which is specific to each kernel version. Namely, the user needs to manually provide such information in KernelStrider when testing drivers in a new kernel version. By default, KCSAN is disabled in the Linux kernel, and thus the user needs to manually re-compile the kernel source code and open the kernel debugging option. Different from KernelStrider and KCSAN, SDILP can automatically run in different kernel versions without the need to re-compile the kernel source code. Thus, we believe that SDILP should be easier to use than KernelStrider and KCSAN in actual driver testing.

4.6.2 Other Approaches

Similar to SDILP, several existing approaches [37], [38] also integrate static analysis and dynamic analysis for data-race detection. Specifically, these approaches first perform static analysis to identify and instrument possible raced instruction pairs in the code, and then perform dynamic analysis to detect data races by monitoring these possible raced instruction pairs during execution. SDILP has two main differences from these approaches:

1) Due to the inaccuracy of alias analysis, their static analysis can miss real raced instruction pairs involving complex alias relationships, causing related data races to be missed by dynamic analysis. By contrast, the static taint analysis of SDILP over-approximates the set of possible shared-variable accesses, without taking into account alias relationships, and thus we believe that SDILP could find data races missed by these approaches. However, due to the differences in the static analyses, SDILP instruments and monitors more variable accesses than these approaches, causing SDILP to introduce more runtime overhead in dynamic analysis.

2) SDILP also uses a static lockset analysis to extend the race-detection results of dynamic analysis, which is new for these approaches.

As the approach of Choi et al. [37] is not open-source, we cannot experimentally compare to it. As for Razzer [38], we successfully run it with its source code [52] to test the six drivers of Linux 5.4 in Table 7 for 24 hours. Razzer only finds memory bugs and warnings caused by race conditions, and it has no race checker to find data races. In the experiments, Razzer finds no race bugs in these drivers.

4.6.3 False Negative Analysis

SDILP can still miss some real data races found by several existing race-detection approaches, for two reasons. First, SDILP only detects data races caused by inconsistent locking discipline, and thus if the raced variable accesses are not protected by any lock, SDILP cannot find related data races. This is why SDILP misses four real data races found by KernelStrider [33], as described in Section 4.6.1. Second, SDILP can only find data races whose caller functions are concurrently executed at runtime. In comparison, Eraser [32] does not have this requirement, and thus it can find data races whose caller functions are not observed to be concurrently executed at runtime. Eraser, however, has many more false positives than SDILP due to no concurrency checking of variable accesses.

5 DISCUSSION

In this section, we discuss the threats to validity, race-detection coveage and generality of SDILP.

5.1 Threats to Validity

The main threats to internal validity are about the lock synchronization mechanism and concurrency analysis of SDILP. Firstly, because SDILP is a lockset analysis approach, it cannot detect data races involving other synchronization mechanisms, such as memory barriers. Secondly, SDILP requires an accurate lockset analysis to check whether each pair of instructions in the two functions can be concurrently executed. Thus, if any lock primitive is missed or the lockset analysis is inaccurate, SDILP may find false data races. Thirdly, SDILP only detects data races caused by inconsistent locking discipline, and it cannot detect other patterns of data races, such as those whose involved variable accesses are not protected by any lock.

The main threats to external validity are about runtime overhead and code coverage. Firstly, though SDILP uses some techniques to reduce runtime overhead, it still introduces 3.7x runtime overhead on average in our evaluation. This runtime overhead may affect driver concurrency during testing, and thus SDILP may miss some real data races. Secondly, because SDILP needs to actually run the drivers, its code coverage heavily relies on the tested workloads and driver configuration. In the experiment, we only use common workloads and specific configurations to test drivers in normal cases. Thus, only some normal-execution code paths are covered. However, many infrequently-executed code paths (such as error handling code paths) are not covered, and thus some real data races may be missed, though SDILP uses a static lockset analysis to extend race-detection results.

5.2 Race-Detection Coverage

Despite some use of static analysis, SDILP is in essence a dynamic analysis approach, so it requires associated hardware devices to actually run the tested drivers and requires sufficient test workloads to cover concurrently executed code. These requirements are often hard to satisfy in practice. Moreover, the static lockset analysis of SDILP is intra-procedural, and it just extends the race-detection results of dynamic analysis, without analyzing the whole driver code, and thus real data races can be still missed. We believe that exploiting software fault injection (SFI) [53] and fuzzing [54] to test drivers with virtual devices [55] can help to find more data races without the associated hardware devices.

Besides, we believe that SDILP can detect general data races, if we drop the constraint that one of the two variable accesses is protected by at least one lock. This constraint, however, increases the confidence in the results, as it can reflect a strong evidence that developers understand that a concurrent access is possible. If this constraint is dropped, SDILP may report many false positives.

5.3 Generality

In fact, besides device drivers, we find that many of reported data races in other programs are also caused by inconsistent locking discipline. For example, we select the accepted race-fixing patches of three common programs (FFmpeg libraries [56], QEMU emulator [57] and Linux NFS filesystem [58]) in the Patchwork project from April 2015 to April 2018, by searching the patch titles. We get 80 such patches, and we find that 27 of these patches involve inconsistent locking discipline, resulting in a percentage of 34%. This percentage is close to that of device drivers (39% shown in Section 2.2). Thus, besides device drivers, we believe that SDILP is applicable to other multi-threaded programs, because they also have many data races caused by inconsistent locking discipline.

6 RELATED WORK

In this section, we introduce the related work and discuss the differences between SDILP and existing approaches.

6.1 Dynamic Analysis

Many approaches use dynamic analysis to detect data races with runtime information. They are based on the happens-before relation [20], sampling [59] or a lockset analysis [32].

Happens-before-based approaches [24], [60]–[62] track memory addresses and synchronization events to infer the temporal happens-before relation between two events. When two conflicting memory accesses α and β involve the same memory location, and neither α happens before β nor β happens before α, a data race may occur. Djit^{+} [24] uses 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 inferring the happens-before relation at runtime. To the best of our best knowledge, no happens-before-based approach has been used to test device drivers.

Sampling-based approaches [6], [8], [15], [30], [31], [34] monitor variable accesses at intervals instead of tracking all variable accesses, and thus they can achieve better performance than happens-before-based approaches. LiteRace [15] 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 [30] 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 same memory addresses, it delays the current thread for a short time. It uses hardware breakpoints to set data breakpoints at the access location to trap any second access during the delay; such a second access indicates a real race. However, these approaches may miss many real races when the sampling frequency is low and have much runtime overhead when the sampling frequency is high.

Lockset analysis approaches [13], [16], [32], [33], [63]–[65] 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 running thread. However, previous lockset analysis approaches often report many false positives, because they cannot ensure that the shared variables involved in the reported data races are actually concurrently accessed at runtime.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3138735, IEEE Transactions on Software Engineering

Our approach SDILP is based on lockset analysis, but it includes several novel features to reduce false positives. First, SDILP identifies the functions that are concurrently executed, and only performs lockset analysis of the memory accesses occurring in these functions. Second, specific to device drivers, SDILP considers interrupt handling, to reduce the possibility of reporting false races related to interrupts. Besides, SDILP introduces static analysis to reduce the runtime overhead of monitoring variable accesses and to detect more data races according to the dynamic-analysis results.

6.2 Static Analysis

Many approaches use static analysis to detect concurrency bugs without actually running the program. They are based on flow-insensitive type-based analysis (such as [66]–[68]) or flow-sensitive lockset analysis (such as [25], [35], [69]). For example, Flanagan et al. [66] propose a type-based analysis approach for Java programs. This approach introduces type annotations to capture synchronization patterns, and detects data races according to these patterns. RacerX [25] is a well-known static lockset analysis approach that detects data races and deadlocks in OS kernel code. It 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. DCUAF [23] is a static lockset analysis approach to detecting concurrency use-after-free bugs in Linux device drivers. It first analyzes each driver's code, and then statistically analyzes the results of all drivers to statically extract the pairs of driver interface functions that may be concurrently executed. With these function pairs, it performs a static summary-based lockset analysis to detect concurrency use-after-free bugs. Besides, some static approaches (such as RacerD [70] and SharC [71]) rely on the user's annotations to indicate concurrent code points and shared variables for data-race detection.

Static analysis can conveniently detect data races without running the tested programs, but it often reports many false positives due to lacking exact runtime information about concurrent execution. Even so, static analysis can help dynamic analysis to reduce runtime overhead and extend the race-detection results. SDILP uses this idea and has shown promising results.

6.3 Hybrid Static-Dynamic Analysis

Similar to SDILP, several approaches [37], [38] also integrate static analysis and dynamic analysis for data-race detection. They first perform static analysis to identify and instrument possible raced instruction pairs in the code, and then perform dynamic analysis to detect data races by monitoring these possible raced instruction pairs. The biggest difference between SDILP and existing hybrid approaches is that SDILP uses a static lockset analysis to extend the race-detection results of dynamic analysis, which is new for existing hybrid approaches. Moreover, SDILP and existing hybrid approaches use different static analyses to reduce the set of instructions monitored during dynamic analysis. Existing hybrid approaches perform static analysis to identify and instrument possible raced instruction pairs, while SDILP performs taint analysis to identify and instrument possible shared-variable accesses. The static analysis of existing hybrid approaches can identify fewer instructions as relevant than SDILP's taint analysis, and thus existing hybrid approaches can achieve less runtime overhead for dynamic analysis. However, alias analysis of concurrent code is much more complex and less accurate than taint analysis of sequential code [19], and thus existing hybrid approaches may miss some real raced instruction pairs involving complex alias relationships, causing related data races to be missed by dynamic analysis. Because the goal of SDILP's taint analysis is to identify shared-variable accesses from sequential code, not to identify possible raced instruction pairs from concurrent code, using SDILP's taint analysis in existing hybrid approaches may not be suitable.

6.4 Symbolic Execution

Several approaches [27], [46], [72], [73] use symbolic execution to detect data races in device drivers. 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. WHOOP [73] is a symbolic-execution based approach to detecting data races in 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 can achieve high code coverage and accurately detect data races. However, solving path constraints and exploring paths are often time-consuming when analyzing large and complex driver code.

7 Conclusion

In device drivers, many data races are caused by a common pattern that we call inconsistent locking discipline. To detect such data races, we propose a hybrid static-dynamic analysis approach, named SDILP. It uses dynamic analysis to monitor driver execution and detect data races, and uses static analysis to reduce runtime overhead and extend the race-detection results of dynamic analysis. We evaluated SDILP on 12 Linux device drivers and found 117 new real data races. 50 of them have been confirmed by driver developers. Compared to our previous approach DILP, SDILP achieves lower runtime overhead and finds more data races. Our results show that static analysis can indeed enhance dynamic analysis in data-race detection.

SDILP can be improved in some aspects. First, because SDILP needs to actually run the drivers, it may miss some driver code and related data races that are not covered by the test suite. To address this limitation, we plan to introduce software fault injection and fuzzing techniques to improve code coverage and detect more data races. Second, besides data races caused by inconsistent locking discipline, we plan to improve SDILP to detect other kinds of concurrency problems, such as atomicity violations and concurrency use-after-free bugs. Finally, besides Linux device drivers, we plan to port SDILP to other operating systems to test their drivers, and to apply SDILP to user-level applications.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful advice on the paper. We also thank the Linux driver developers who gave helpful feedback on our race reports. This work was supported by the National Natural Science Foundation of China under Project 62002195.

REFERENCES

[1] Q.-L. Chen, J.-J. Bai, Z.-M. Jiang, J. Lawall, and S.-M. Hu, "Detecting data races caused by inconsistent lock protection in device drivers," in *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 366–376.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] P. Zhou, R. Teodorescu, and Y. Zhou, "HARD: hardware-assisted lockset-based race detection," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007, pp. 121–132.

[17] "Helgrind: a thread error detector," https://valgrind.org/docs/manual/hg-manual.html.

[18] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Scalable and precise dynamic datarace detection for structured parallelism," in *Proceedings of the 33rd International Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 531–542.

[19] Y. Sui, P. Di, and J. Xue, "Sparse flow-sensitive pointer analysis for multithreaded programs," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO)*, 2016, pp. 160–170.

[20] 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.

[21] L. Ryzhyk, Y. Zhu, and G. Heiser, "The case for active device drivers," in *Proceedings of the 1st Asia-Pacific Workshop on Systems (APSys)*, 2010, pp. 25–30.

[22] J.-J. Bai, Y.-P. Wang, and S.-M. Hu, "AutoPA: automatically generating active driver from original passive driver code," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*, 2018, pp. 288–299.

[23] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency use-after-free bugs in Linux device drivers," in *Proceedings of the 2019 USENIX Annual Technical Conference*, 2019, pp. 255–268.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] "KernelStrider: detecting data races in Linux kernel modules," https://github.com/euspectre/kernel-strider.

[34] "KCSAN: concurrency sanitizer for the Linux kernel," https://github.com/google/ktsan/wiki/KCSAN.

[35] 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.

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

[37] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," in *Proceedings of the 23rd International Conference on Programming Language Design and Implementation (PLDI)*, 2002, pp. 258–269.

[38] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: finding kernel race bugs through fuzzing," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, 2019, pp. 754–768.

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

[40] 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3138735, IEEE Transactions on Software Engineering

[41] A. Lochmann, H. Schirmeier, H. Borghorst, and O. Spinczyk, "LockDoc: trace-based analysis of locking in the Linux kernel," in *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, 2019, pp. 1–15.

[42] H. Wang, J. Zhai, X. Tang, B. Yu, X. Ma, and W. Chen, "Spindle: informed memory access monitoring," in *Proceedings of the 2018 USENIX Annual Technical Conference*, 2018, pp. 561–574.

[43] "Linux kernel documents about locking," https://docs.kernel.org/locking/.

[44] K. Cong, L. Lei, Z. Yang, and F. Xie, "Automatic fault injection for driver robustness testing," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 361–372.

[45] J.-J. Bai, Y.-P. Wang, and S.-M. Hu, "Automated and reliable resource release in device drivers based on dynamic analysis," *Journal of Systems and Software (JSS)*, vol. 137, pp. 463–479, 2018.

[46] 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.

[47] L. Zhang and C. Wang, "RClassify: classifying race conditions in web applications via deterministic replay," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 278–288.

[48] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," in *Proceedings of the 28th International Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 22–31.

[49] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. data race bugs: telling the difference with Portend," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 185–198.

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

[51] 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.

[52] "Razzer repository," https://github.com/compsec-snu/razzer.

[53] H. A. Rosenberg and K. G. Shin, "Software fault injection and its application in distributed systems," in *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS)*, 1993, pp. 208–217.

[54] "Fuzzing," https://en.wikipedia.org/wiki/Fuzzing.

[55] T. Yu, X. Qu, and M. B. Cohen, "VDTest: an automated framework to support testing for virtual devices," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 583–594.

[56] "Patchwork for FFmpeg," https://patchwork.ffmpeg.org/project/ffmpeg/list/.

[57] "Patchwork for QEMU," https://patchwork.ozlabs.org/project/qemu-devel/list/.

[58] "Patchwork for Linux NFS filesystem," https://patchwork.kernel.org/project/linux-nfs/list/.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "RacerD: compositional static race detection," in *Proceedings of the 33rd International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2018, pp. 1–28.

[71] Z. Anderson, D. Gay, R. Ennals, and E. Brewer, "SharC: checking data sharing strategies for multithreaded C," in *Proceedings of the 29th International Conference on Programming Language Design and Implementation (PLDI)*, 2008, pp. 149–158.

[72] 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.

[73] 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.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.