Electronic Communications of the EASST
Volume 079 (2020)

Interactive Workshop
on the Industrial Application of Verification and Testing
ETAPS 2020 Workshop
(InterAVT 2020)

Data Race Detection in the Linux Kernel with CPALockator

Pavel Andrianov and Vadim Mutilin

8 pages

Guest Editors: Stylianos Basagiannis, Goetz Botterweck, Anila Mjeda
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Data Race Detection in the Linux Kernel with CPALockator

Pavel Andrianov[1] **and Vadim Mutilin**[2]

[1,2]Ivannikov Institute for System Programming of the RAS
[2]Moscow Institute of Physics and Technology

Abstract: Most of the state-of-the-art verification tools do not scale well on complicated software. Our goal was to develop a tool, which becomes a golden mean between precise and slow software model checkers and fast and imprecise static analyzers. It allows verifying industrial software more efficiently. Our method is based on the Thread-Modular approach elaborating the idea of abstraction from precise thread interaction and considering every thread separately, but in a special environment, which models thread effects on each other. The approach was implemented in the CPAchecker framework and was evaluated on benchmarks based on Linux device drivers for data race detection. It demonstrated that predicate abstraction allows keeping a false alarms rate at a reasonable level of 52%. Moreover, it did not miss known real bugs found by analysis of commits in the Linux kernel repository thus confirming the soundness of the approach.

Keywords: Data race, Thread-Modular approach, Linux kernel.

1 Introduction

Multithreaded software is widely spread nowadays for efficient usage of multiple CPU cores. That is also correct for operating systems, which may contain a lot of parallel activities, for example, system calls, interrupt handlers and so on.

In addition to generic bugs, which are common for all kinds of software, multithreaded programs may contain specific ones, related particularly with a parallel execution: deadlocks and races.

Data races are an important subclass of race conditions and are defined as a situation when a simultaneous accesses to the same memory take place from different threads, where one of the accesses is a write. In general, a data race may not lead to failure directly, but it is a symptom of an error.

There were developed a lot of different approaches for data race detection. Usually, they are divided into static and dynamic ones. Further, we will concentrate mostly on static approaches, which may guarantee the correctness of the code under certain assumptions.

Precise approaches to model checking are based on considering all possible thread interleavings. Thus, they allow to be very precise and consider different complicated cases, like lock-free synchronization and weak memory models. As a drawback, they have spent a lot of resources (time and memory) per a verification task and thus can not be applied to large benchmarks.

An opposite case of static verification is a data flow-based analysis. The corresponding tools are very fast but not so precise as model checkers. Particularly, the simple static analysis can not perform a path sensitive analysis. As a result, there are produced a lot of false alarms and some of the tools apply unsound filters to remove a part of false alarms, and it may lead to a missed bug.

Our goal was to develop a tool that unites advantages both fast data flow analysis and precise model checking. One of the ideas to build such an approach is a sequential combination of two stages: the first stage is an imprecise analysis, which produces hints for the second stage, which is a traditional model checking. The other idea is a thread-modular approach, which considers each thread separately, but in a special environment [HJMQ03, GPR11].

We implemented a thread-modular approach extended with projections in the CPAchecker framework [1]. CPAchecker contains different kinds of analysis (CPAs), which may be combined. We extended several CPAs for support of the thread-modular approach with projections and implemented two specific CPAs: ThreadCPA for thread analysis and LockCPA for tracking synchronization primitives.

We successfully applied the CPALockator tool to a set of benchmarks that are based on the Linux device drivers. The final rate of false alarms is quite low – 52%. We performed an analysis of a set of existing fixes in Linux drivers, and the tool can detect 5 of 8 known bugs. The rest of them are missed due to timeout.

2 Basic Idea of the Thread-Modular Approach with Projections

Consider a simple program, which has only two active threads. Figure 1 presents a simple model example, which uses an ad-hoc synchronization. The first thread initializes global data (in this case, a global variable *g*) and then sets a flag, meaning the shared data are ready. The second thread is allowed to use the shared data only after setting the flag. Thus, there is no data race on a global variable *g*.

```
volatile int g = 0;
Thread1 {
1:  g = 1;
2:  d = 1;
3:  ...
}
```

```
volatile int d = 0;
Thread2 {
4:  if (d == 1) {
5:     g = 2;
6:  }
}
```

Figure 1: An example of a small program

A suggested approach is based on a thread-modular approach. The approach considers each thread separately but in a special environment, which is constructed also during the analysis. The environment computation is based on the analysis of all threads, as every thread is a part of the environment for other threads. For each thread, a set of its actions, which may affect other threads, is collected. The actions include modification of shared variables, acquiring synchronization primitives and so on. Environment precision strongly affects the precision of the whole analysis. However, there is the main question, how to compute and represent the environment efficiently.

In a sequential analysis, there is a successful technique, which allows reducing the number of considered program states – an abstraction. It allows to abstract from minor details of a

[1] https://cpachecker.sosy-lab.org/

program and considers general (abstract) state. Each abstract state may correspond to a set of real (concrete) program states. This idea allows to significantly increase the efficiency of the analysis.

The key idea of the suggested approach is an extension of abstraction not only to the program states but also to the operations of a thread. Adjusting the level of the abstraction, it is possible to choose a balance between speed and precision of the tool.

Figure 2 shows a part of the Abstract Reachability Graph (ARG) for the first and the second threads from figure 1. There is no interaction considered, so this is not a final step of the analysis.

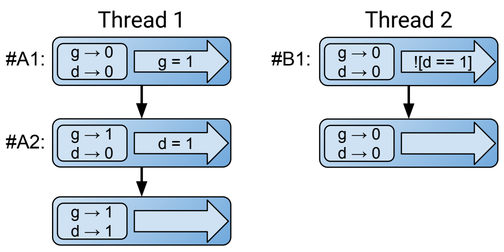

Figure 2: Abstract transitions for two threads without any interaction

The analysis in the example is based on a simple Value Analysis, which tracks only explicit values of variables. A transition contains an abstract state and an abstract operation. The first abstract state in both threads contains information that both global variables (d and g) are equal to zero ($g \rightarrow 0$). After performing an operation $g = 1$ (transition #A1) the value of g is updated into 1 ($g \rightarrow 1$) in the second abstract state (transition #A2).

After constructing an ARG for two threads separately, we need to consider the influence of threads to each other, i.e. to construct an environment. For every thread operation, we compute its *projection* – a representation of operation in a thread for other threads as an environment. For example, modification of local variables can not affect on other threads, so the corresponding projection is empty. Modification of a global variable may affect other threads, so the projection may be equal to the original transition or overapproximate it, for example, by abstraction from a precise assigned value. A projection may contain not only information about action but also a condition for performing this action, so-called *guard*. Consider a transition #A1. We may represent the corresponding projection in the following way: if the value of g equals zero, it may be changed to one. In other words, the projection consists of two parts: a *guard* ($[g == 0]$) and an *action* ($g \rightarrow 1$). The guard corresponds to a predecessor abstract state and an action corresponds to an operation.

The figure 3 presents computed projections for the first thread. There are two transitions, which may affect the global variables: #A1 and #A2. We compute the corresponding projections: #P1 and #P2. Then every projection has to affect the second thread, i.e. apply to all possible (according to the guard) transitions of the second thread. We apply the projection #P1 to the transition #B1, as the state is *compatible* with the guard of the projection. Only after that, we may apply the projection #P2 to the new transition #B2, which requires g to be equal to one. And only then the second thread may go through a new transition #B3, which discovers new paths. Note, the figure presents only projections for the first thread, in complete ARG there should be also projections for the second thread as well.

For data race detection we have to find two transitions, which modify the same variable. The

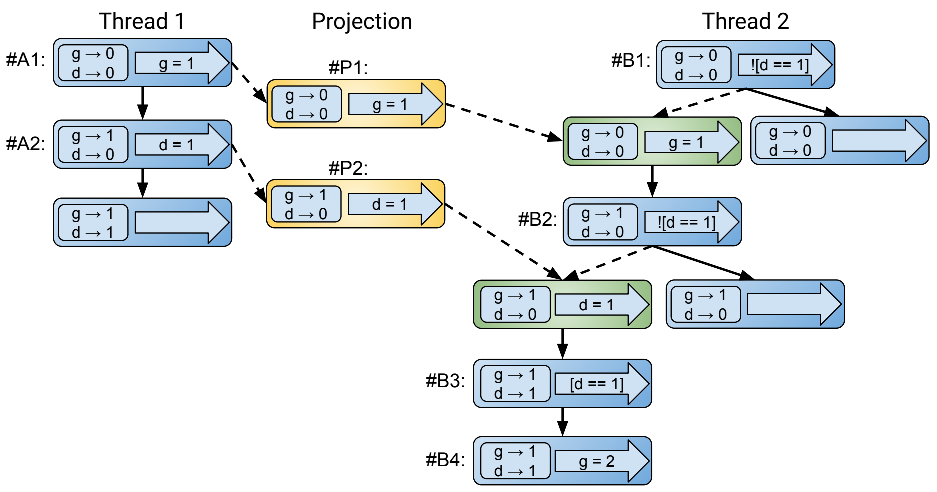

Figure 3: Application of projections to the second thread

example has potential candidates: #A1 and #B4. Now, we should check, if the two transitions may be executed in parallel. That means, that the corresponding abstract states must be a part of one global state, i.e. they must be *compatible*. In this case, the partial states are contradicting each other, as one has $g \rightarrow 0$ and the other $g \rightarrow 1$. So, the corresponding transitions can not be executed simultaneously. Thus, we conclude there is no data race for g. Note, there is a data race for d (transitions #A2 and #B2), but it may be considered as lock-free synchronization, and a potential race is a part of its implementation.

The suggested approach provides a lot of possible options and configurations for targeting to a particular task. A projection may be represented by more or less precise abstraction. Several projections may be joined altogether or considered separately. An example of more abstract transitions is presented in figure 4.

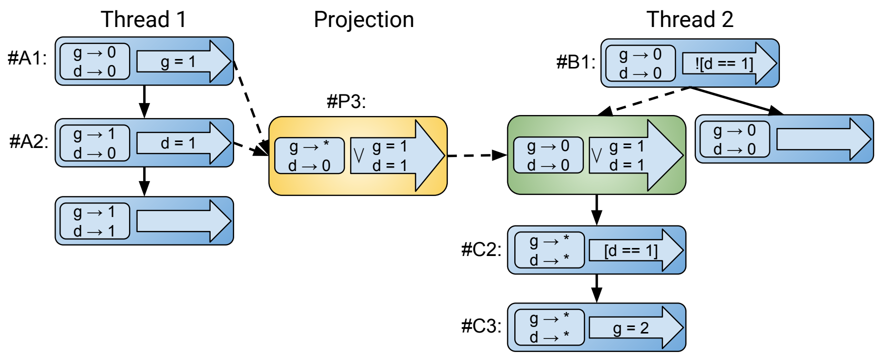

Figure 4: Application of projections to the second thread

The two initial projections (#P1 and #P2) are joined into a single one (#P3). Usually, it leads to losing some information, for example, here we lost a precise value of variable g. The action of projection also becomes more complicated. The second thread can not identify a precise value of the variables, as both of the variables are now equal to zero or one. The simple kind of analysis operates only with single explicit values of variables and both variables are considered to be equal to any random value. Now, transitions #A1 and #C3 become compatible, which means, that the race is reported on variable g.

The level of abstraction strongly affects the precision of the analysis and its speed, but the analysis always remains sound.

3 Data-Race Detection Algorithm

Data race detection algorithms may be divided into two stages: reachable states analysis and searching of pairs, which can form a race. In our case, the first subtask is related to ARG construction and may be solved with different configurations.

As we have already discussed, an ARG from abstract transitions is constructed with the chosen configuration of the CPALockator. Note, the abstract transitions in the graph are reachable with a certain abstraction level, so, they may not be reachable in a real execution. For refinement of the abstraction, we use a CEGAR algorithm (Counterexample Guided Abstraction Refinement). Note, a CEGAR algorithm was reused without significant modifications. However, it allows refining only path in a single thread, i.e. it can not identify contradictions between operations in different threads. Anyway, it is not a fundamental limitation of the approach, and a possible extension of the CEGAR algorithm for the thread-modular approach will allow obtaining more precise results.

After computation of the ARG we should find those transitions, which could form a data race. Usually, a race condition is defined as a situation, where simultaneous access to shared memory from different threads takes place. Two main questions, which appear for static data race detection, are the following: how we can determine, that the accesses are performed to the same memory and how to determine if the accesses are simultaneous. Further, we will discuss both of the features of the approach.

In real software, there are a lot of different operations with pointers, structures, and more complicated data. The variety of pointers breaks any kind of alias analysis. Moreover, most of the approaches require the pointers to be correctly initialized, which is difficult to reach while analyzing libraries, modules and so on. Thus, we use a BnB memory model [AFM+17], which divides all memory into a disjoint set of regions. Every region corresponds to a specific data type or a structure field (if an address-of expression is not used for it). According to the memory model, we consider access to the same region as access to the same memory. The BnB memory model has certain limitations. First of all, casting and address arithmetic may lead to a missed bug, because the two accesses will not be placed in the same region. Also, the memory model may lead to false alarms in cases, when two pointers of the same type never point to the same memory but are placed in the same region.

If we have two accesses to the same region, we have to check, if two accesses are taking place simultaneously. We use the already mentioned notion of *compatibility* of the corresponding transitions. Two partial states are compatible, if they may be parts of a single global state. If the partial states are not compatible, that means, the corresponding transitions can not be executed in parallel. Thus, this approach is an extension of a Lockset algorithm, which defines a data race, as a pair of accesses with a disjoint set of locks. A compatibility check uses different kinds of analysis, including analysis of synchronization primitives, predicate analysis, thread analysis, hence the approach is more precise than the default Lockset algorithm.

4 Evaluation on the Linux Device Drivers

CPALockator[2] tool was evaluated on a benchmark set, which is based on device drivers of the subsystem *drivers/net* of the Linux kernel v4.2.6[3]. For each kernel module, a verification task was prepared with the help of the Klever framework [NZ18]. For 473 modules it prepared 425 tasks for CPALockator. Detailed information about its verdicts is presented in the table 1.

The table 2 presents the results of the analysis of warnings. We will discuss them further.

Tasks	Description
261	Safe (no bugs)
22	Unsafe (races found)
142	Unknow (timeout)

Table 1: Overall verdicts

Number	Percent	Description
41	48%	True (bugs)
25	29%	Imprecise scenario model
8	9%	Imprecise memory model
7	8%	Specifics of interrupts
4	5%	Other imprecisions in the analysis

Table 2: Analysis of warnings in Linux kernel modules

41 warnings of 85 (48%) correspond to real bugs. 10 of them are related to so-called "benign" races. It is, for example, a race on statistics counter or during printing debug information. The main part of bugs corresponds to a situation when a new thread is created while an initialization process is still uncompleted. Thus, the new thread may access uninitialized memory or not allocated memory. The situation usually occurs, when registration of driver handlers is performed before the complete initialization of driver data. Note, several warnings may be outputted per one module and these 41 warnings correspond to 15 modules.

44 of 85 warnings (52%) are false alarms. The main part of them (25 warnings) is related to the imprecise communicating scenario model, which defines scenarios of driver activities. The issue is not related to the CPALockator tool and is caused by an internal part of the Klever framework. So, the corresponding warning is correct for the verification tool, but the data race is impossible in a real execution. For example, an imprecise communicating scenario model may consider some driver activities as parallel, but in a real execution, the corresponding scenario is impossible. Other examples – absent models of external functions, which are important for analysis, or incorrect driver input data are also a responsibility of Klever task generator.

The main part of false alarms, related to the CPALockator tool, is due to imprecise memory model – 8 warnings. To be conservative CPALockator may consider different memory as the same (BnB model) and produce a warning about potential simultaneous access, although the accesses are not to the same memory. This is mostly happening for structure fields.

One more important reason of false alarms is low-level constructions in the Linux kernel. Currently, an interrupt handler is considered to be executed like an ordinary thread, but actually, it has certain specifics. For example, the execution of the interrupt handler may be interrupted only in special cases and only by another interrupt. It means, that the interrupt handler is executed atomically in parallel with an ordinary thread. Moreover, the registration of an interrupt handler may not mean that it becomes active. In some cases interrupts should be activated in the device.

[2] CPAchecker-theory-with-races@32609

[3] https://gitlab.com/sosy-lab/software/ldv-benchmarks.git , directory *linux-4.2.6-races*

Small classes of false alarms are dedicated to the analysis imprecision: shared analysis, problems with complex data structures and model variables.

5 Evaluation on the Set of Known Bugs

Analysis of causes of false alarms was performed on a set of benchmarks, which are based on fixes of existing bugs in stable Linux kernel versions in 2014[4]. 795 commits are extracted using keywords[5] from the whole set of 4047 commits. Then we performed a manual analysis of commits and filtered only those commits, which are related to data races and contain only lock-based synchronization primitives and are in kernel modules, see details in the table 3. We have got 13 modules after that and launched the Klever framework. It failed to prepare verification tasks for 5 modules, hence the resulting set contains 8 verification tasks[6].

Table 3: Constructing a set of commits with known bugs for evaluation

Total amount	Description
4047	All commits
795	Extracted using keywords
43	Related only to data races
28	Use lock-based synchronization
13	In kernel modules
8	Klever prepared corresponding tasks

The results of launching CPALockator on 8 verification tasks with known bugs are presented in the table 4. The bugs are marked by commit identifiers. We run CPALockator on the tasks prepared by Klever framework for the repository before corresponding commit. For checking that the bug is not found after the fix we also launched CPALockator on a verification task prepared for the repository after corresponding commit.

Table 4: Evaluation on known bugs in stable versions of Linux kernel

Commit	Module	Result	Comment
0e2400e	drivers/char/virtio_console.ko	$+$	
7357404	fs/hfsplus/hfsplus.ko	$\pm$	Found with limited CEGAR iterations
f1a8a3f	drivers/net/bonding/bonding.ko	$\mp$	Found a nontarget bug with limited CEGAR iterations
1a81087	net/ipv4/tcp_illinois.ko	$\pm$	Nontarget bug was found
f0c626f	drivers/target/iscsi/iscsi_target_mod.ko	$\mp$	Found a nontarget bug with limited CEGAR iterations
aea9dd5	fs/btrfs/btrfs.ko	$-$	
10ef175	sound/soc/snd-soc-core.ko	$-$	Timeout
4036523	drivers/gpu/drm/i915/i915.ko	$-$	

- CPALockator found two bugs fixed in commits 0e2400e and 7357404. For the tasks prepared after the fix, the corresponding warnings are not found. It means that the bugs are detected correctly. The bug in 7357404 was found only if the CEGAR iterations are limited, in another case, there will be a timeout.

[4] https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
[5] fix, error, race, bug, failure, crash, etc
[6] https://gitlab.com/sosy-lab/software/ldv-benchmarks.git , directory *ldv-commit-races*

- For three more modules, CPALockator outputted warnings for the same memory location mentioned in the bug but for a different access location in the code. We call them nontarget bugs. For f1a8a3f and f0c626f the CEGAR iterations also must be limited to avoid the timeout.
- Three modules are too large and too complex and exceed 15 minutes timelimit. Note, the timeout is achieved in the first iteration of CEGAR.

Thus, 5 of 8 bugs were found, and for the rest 3 modules, there were unknown verdicts. The results confirm the soundness of the approach.

6 Conclusion

One of the main limitations of the approach is a size of verified code and its complexity. We do not discuss it much, as it is well known for any static verifier tool. However, CPALockator is able to solve real-world tasks, which are based on Linux device drivers, which shows practical benefit of the tool.

We also mentioned BnB memory model, which also affects on the verification process. This is a trade off between speed of the analysis and its precision.

One more limitation of the approach is support only lock-based synchronization primitives. Different atomic constructions, barriers are skipped for now.

We do not speak about function pointers, C constructions and so on, as it is not related to the approach itself. CPAchecker framework contain different kinds of analysis, for example, function pointer analysis, and they may be included into CPALockator if it is necessary.

The results on Linux device drivers are quite good, as the tool demonstrates a false alarms rate of 52%. Moreover, it does not miss real bugs on our benchmark set, that confirms the soundness of the approach. Thus, we may conclude that the tool is practically valuable for data race detection.

Future plans include further improvements in the overall approach: support new synchronization primitives, investigate different combination with other analyzes.

Bibliography

[AFM$^+$17] P. Andrianov, K. Friedberger, M. Mandrykin, V. Mutilin, A. Volkov. CPA-BAM-BnB: Block-Abstraction Memoization and Region-Based Memory Models for Predicate Abstractions. In *Proceedings of TACAS*. Pp. 355–359. 2017.

[GPR11] A. Gupta, C. Popeea, A. Rybalchenko. Threader: A Constraint-based Verifier for Multi-threaded Programs. In *Proceedings of CAV*. Pp. 412–417. Springer, 2011.

[HJMQ03] T. A. Henzinger, R. Jhala, R. Majumdar, S. Qadeer. Thread-Modular Abstraction Refinement. In *Proceedings of CAV*. Pp. 262–274. Springer, 2003.

[NZ18] E. Novikov, I. Zakharov. Verification of Operating System Monolithic Kernels Without Extensions. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*. Pp. 230–248. Springer, 2018.