

Predicate Abstraction Based Configurable Method for Data Race Detection in Linux Kernel

Pavel Andrianov^(✉), Vadim Mutilin, and Alexey Khoroshilov

Institute for System Programming of the Russian Academy of Sciences,
Moscow, Russia
`andrianov@ispras.ru`

Abstract. The paper presents a configurable method for static data race detection. The method is based on a lightweight approach that implements Lockset algorithm with a simplified memory model. The paper contributes two heavyweight extensions which allow to adjust required precision of the analysis by choosing the balance between spent resources and a number of false alarms. The extensions are (1) counterexample guided refinement based on predicate abstraction and (2) thread analysis. The approach was implemented in the CPALockator tool and was applied to Linux kernel modules. Real races found by the tool have been approved and fixed by Linux kernel developers.

Keywords: Static analysis · Data race · Predicate abstraction

1 Introduction

Bugs which are related to a parallel execution of code are considered to be the most difficult for detecting and fixing. Race conditions are still remaining the most numerous class of such bugs [11]. Data race condition is a situation in which a shared memory location may be accessed from several threads simultaneously. Such bugs can lead to severe consequences even to failure of the whole software system [10].

There exist many tools for automatic detection of race conditions. We distinguish two basic classes of static analyzers: lightweight and heavyweight. The first ones require significantly less resources, but frequently based on an unsound approaches leading to inaccuracies of the analysis such as missed errors. Heavyweight analyzers allow to prove absence of errors, but consume a lot of resources, such as time and memory. It means that heavyweight approaches are still impossible to apply to real software systems.

Our main goal is to develop a lightweight static data race analyzer which can be applied to operating system kernels. The key issue is increasing of the accuracy of the analysis in a such way to make the tool useful for the verification of such complicated software systems as the Linux kernel. Our contribution is a

lightweight approach extended with heavyweight techniques which were usually applied for precise verification of small programs.

The paper is organized as follows. Key challenges are presented in Sect. 2. Then an overview of the method is given. The implementation of the approach is discussed in Sect. 4. Two extensions: refinement for data race conditions and thread analysis are described in Sects. 5 and 6. Section 7 presents the results of application to Linux kernel modules.

2 Key Challenges of Data Race Detection with Static Analysis

Let us remind key concepts of static analysis. A program is modeled with a control flow automaton (CFA) which is a set of states and transitions related to operators in the program. A state of a program is considered to be its memory status which includes a program counter determining a particular location in the program. The goal of static analysis is to determine a set of reachable states from a given state via transitions. If the set of all reachable states includes a special one (so called error state) the bug is considered to be found. As the program usually has a huge amount of states, an abstraction over states is used. The structure of the abstraction determines the accuracy of the analysis. For example, if the abstract state represents values of all variables in a program, the analysis is very accurate, but very slow. On the contrary, if the abstract state does not consider values of variables at all, the analysis is very fast and imprecise.

Now let us define a type of bugs which the analysis will detect. In software systems race conditions usually arise in case of simultaneous accesses to the same shared memory locations from different threads, one of the accesses is a write access. Such races are called data races.

Race conditions do not always lead to an incorrect state of a program. For example, simultaneous modification of a statistic counter leads only to an insignificant differences. Such situations are known as benign races.

The task of static data race detection is a complicated problem, as in general case to prove the correctness of all possible interleavings of several threads should be checked, and it is the problem of exponential complexity. Methods of static analysis with interleavings solve the problem for a very small programs [1, 7, 9].

Unlike to dynamic analyzers which always know what memory location is accessed, static analyzers do not know values of pointers. It means that it is very hard to precisely determine that two accesses are performed to the same memory location. There are different approaches to deal with the problem: a precise memory model, alias analysis and others. Alias analysis provides a map from pointers to a set of memory locations which they point to. While using a conservative approach which considers all possible aliases so-called may-aliases, the tool get an enormous size of the map that leads to high resource consumption. Such techniques are used only in heavyweight tools which are targeted at accuracy of the analysis. Lightweight tools which are valued for their speed, likely do not use such methods or use heuristics which may miss bugs in some cases.

One of the ways for decreasing a set of variables which may be involved in a race condition is enabling an additional analysis of shared data. The analysis outputs a set of potential shared variables. If a variable can never be shared, it can not be involved in race condition. The main issue is conservatism of the analysis. On the one hand, non conservative analysis can not handle complicated cases, for example, related to pointer arithmetics. Thus, it can miss shared variables and consequently may miss a bug. On the other hand, if the analysis is very conservative, the result is a set of shared variables that does not differ from the set of all variables and it affects the speed.

The second important problem of static analysis is a significant percentage of false alarms in comparison to dynamic analysis. Let us consider the main causes.

The first cause of false alarms is an inconsistency of conditions in the path. In this case the path is called infeasible. Often the infeasibility of the path is related only to operations in one thread. If from the whole path of execution consisted of the paths in different threads one local infeasible path can be extracted, then local infeasibility takes place. Otherwise, there is global infeasibility.

The second cause of false alarms is the detection of all possible synchronization primitives which can be used for concurrent memory access exclusion. First, all synchronization primitives are also objects in memory which may be equal. The issue of pointers equality again leads us to precise memory model. Second, conditional actions, like *mutex_trylock*, exist. A program behavior depends on a possibility to acquire a lock, and if the lock is successfully acquired the function returns a special code (usually zero). To be able to handle such cases an analysis should somehow link the return value to the acquired lock.

The third important cause of false alarms is a thread dependency. Often a program performs some preparation actions (initialization) and then creates a number of auxiliary threads. The initialization may be done without any synchronization and there can not be race conditions, as the other threads will be created later. Thus, there is a need in a special kind of analysis, like thread analysis, to determine blocks of code which can be executed in parallel.

3 Method Overview

As it was already said the *program state* is its memory state including the program counter. For multithreaded program the state includes a set of program counters for every thread of execution. *Race condition* is a program state, where two accesses to the same memory location from the different threads is possible, one of the accesses is a write access.

Now let us define a *projection* of a program state on a thread. A projection on a particular thread includes memory state which is available only for the thread: its local variables and shared data. To abstract from details of interaction between threads, further we will consider *analysis on projections*. Such analysis considers every thread separately from the others which are an environment for the first one.

To define race conditions on projections we need the notion of *compatibility*. We define two projections as compatible, if there exists one program state which

can be projected to the given projections. For race condition we need two projections and they should be compatible, so that they correspond to one program state. We define a *race condition on projections* as a pair of projections which are compatible and corresponding program state is a race condition state.

The analysis operates with *abstract states* which include a set of program states. As we consider analysis on projections, hereinafter the term abstract state we will use as abstract projection-state.

The presented method is based on the algorithm *Lockset*. The algorithm was initially implemented in a dynamic analyzer [12]. Our analysis is constructed in a similar way, but is implemented in static analyzer. An abstract state of the analysis stores information about acquired locks for every thread. After construction the whole graph of abstract reachable states for every memory access there is an abstract state containing a set of acquired locks. Two states are considered compatible if the intersection of containing lock sets is empty. According to definition a warning about a potential race condition is reported if we have two compatible states, where two accesses to the same memory location from the different threads is possible, one of the accesses is a write access.

Calculation of lock sets during analysis stage is simple: if the analysis finds a function acquiring a lock the corresponding lock is added to the abstract state, and in the case of releasing a lock, the corresponding lock is removed from the abstract state.

We use a simple memory model. For every pointer a unique identifier based on its name and scope is constructed. If two identifiers are equal, the pointers are considered to point to the same memory location. Thus, one pointer always points to the same memory without a dependency on a program location. Moreover, while analyzing field access expressions we consider only field names, but not the base expression of the structure. Thus, the memory model in the analysis makes an assumption that memory locations which are pointed by two pointers $A \rightarrow a$ and $B \rightarrow a$ are equal for the structures of the same type. If the structures A and B have different types, the memory locations $A \rightarrow a$ and $B \rightarrow a$ will be considered different.

Let us consider an example of lock analysis.

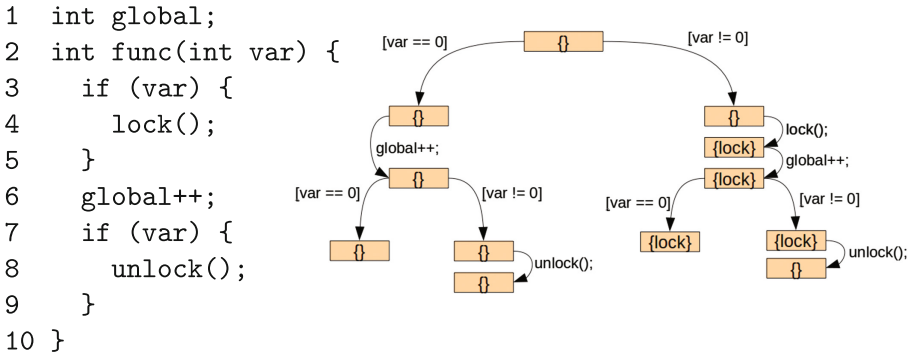


Fig. 1. An example of lock analysis

The Fig. 1 presents only abstract states which contain information only about acquired locks. Suppose that the function is called from several threads. A race condition will be found in the example: there exists a transition (line 6), performing an access to the global variable *global*, from the state which has no acquired locks {}, and also from the state which has a *lock* acquired. Intersection of the lock sets is empty, thus the potential race condition takes place.

It should be noted that the approach uses a heuristic to decrease an amount of false alarms. It is based on an assumption that situations when all the accesses to some shared data are never protected are rare. Usually the majority of accesses to one shared memory location use necessary lock protection and only a small number of accesses to the same memory are missing the required locks. If the analysis obtained an empty set of locks for all accesses to some variable then it likely means that we missed something, e.g. more complicated synchronization or thread dependencies. According to the heuristic race conditions will not be reported for the shared variables which are not protected at all, although the classic *Lockset* reports it as a potential race condition.

4 Implementation

The described method is implemented using a concept of CPA (Configurable Program Analysis) [4] and called CPALockator. The concept allows to launch several types of analysis together: sequentially or in parallel. Every analysis may choose a balance between precision and consumed resources itself.

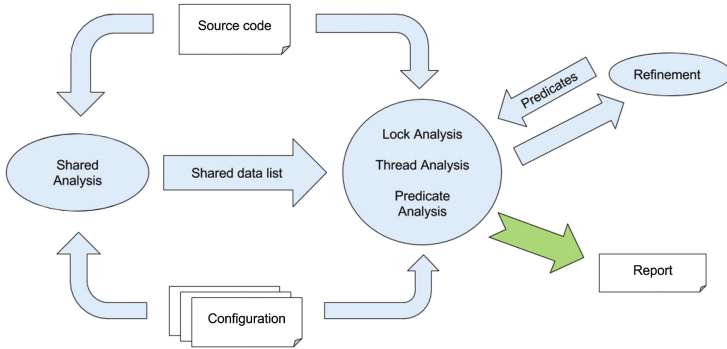


Fig. 2. A method scheme

Consider the analysis scheme presented in the Fig. 2. First, the shared analysis is launched. Its result is a set of unshared variables for every program location, mostly we interested in pointers which point to a local data. The other variables are treated as shared. Further race conditions are reported only for shared variables. Then a parallel composition of the three analysis is launched:

Lock Analysis, Thread Analysis and Predicate Analysis. At this stage lock sets are calculated for each access in a reached set by Lock Analysis, and potential parallel blocks of code are determined in Thread Analysis. After finishing the construction of the reached set, the set of potential race conditions is calculated using the notion of compatibility of states. For each potential race condition we have two paths leading to memory accesses and for them we start the next step – Refinement. During the refinement for every path a logical formula is constructed. Then logical satisfiability of the formula is checked. If one of the paths is infeasible, for example, a value of variable is not considered, the abstraction is rebuilt. The corresponding condition on the variable affecting infeasibility of the path is added as a new predicate to Predicate Analysis, and next iteration of the analysis is started. It will construct the new abstraction without the infeasible path. The process may continue until it will converge.

In theory one can imagine an example that requires an infinite number of refinements, however in practice analysis usually needs only a finite number of refinement iterations. For a large amount of source code one iteration of refinement stage may take dozens of seconds. Therefore usually a time limit is used, and when the time is expired all found race conditions are reported to a user, even if there may be spurious ones. Experiments show that in many cases spurious race conditions are removed with refinement quickly and the rest of time is spent to prove the non spuriousness of the others.

Shared analysis and lock analysis have been already presented in the paper [2]. Predicate analysis is a basic analysis which was implemented in CPA concept long ago [3]. Detail description of two contributions of the paper: Refinement process and Thread Analysis are presented in the next sections.

5 Refinement

The task of the refinement is to exclude spurious race conditions, related to infeasible paths. Note, that the refinement solves the task of local infeasibility and does not consider an interaction of threads with each other.

Consider the example on the Fig. 1. If the analysis does not consider possible values of variables, four paths will be analyzed. One of them, in which the lock is acquired (line 4) and is not released, affects the further analysis of the other code. The path is infeasible, as the condition expression is the same in both condition operators (lines 3 and 7). So, in a real execution only two paths may appear, each of them has no acquired lock at the end.

The refinement method which is used in our analysis, is based on the classic algorithm CEGAR – Counterexample Guided Abstraction Refinement [5]. First, let us describe the original approach. The basic idea is that the chosen property is proved not for the initial system which is rather complicated, but for a simplified model of the system which is called an abstraction. The abstraction may be very rough and miss large number of details, but it should be correct. In other words, all states reachable in initial system should have corresponding reachable states in the abstraction.

A number of states in abstraction is usually less than in the initial system that makes its analysis easier, but due to imprecise model there may appear spurious race conditions. In the case a counterexample is built, it is an example of error path, leading to an error. After that the counterexample is checked on the initial system. If it is feasible, the found error is considered to be a real bug. The other case means the counterexample is obtained due to an imprecise abstraction. Then the abstraction is refined, guided by the counterexample, and details affecting on the feasibility of the counterexample are added. After that the analysis continues on the refined abstraction. The iterations are repeated until the correctness is proven or the real bug is found.

Abstraction of the program may be done in different ways, but the most popular is predicate abstraction which is based on partition a set of program states (values of its variables) on subsets with equal value of chosen predicates.

To check a counterexample and abstraction refinement there is a need to represent a sequence of operators in initial program as logic formulas. There are many approaches, as pre- and postconditions or path formulas, based on SSA representation. To check satisfiability of the formulas the different SMT-solvers (Satisfiability Modulo Theories) are used. If the path formula is unsatisfiable, the conditions which then will be added to analysis, should be extracted. The conditions are represented by predicates. There are different ways of extracting predicates from a path formula, for example, syntactic methods or Craig interpolation [6, 8].

CEGAR approach (Fig. 3) was implemented in different static verification tools. It is successfully applied for solving reachability tasks, e.g. checking reachability of an error location in a program. If a path from an entry point to an error location is found the refinement is performed.

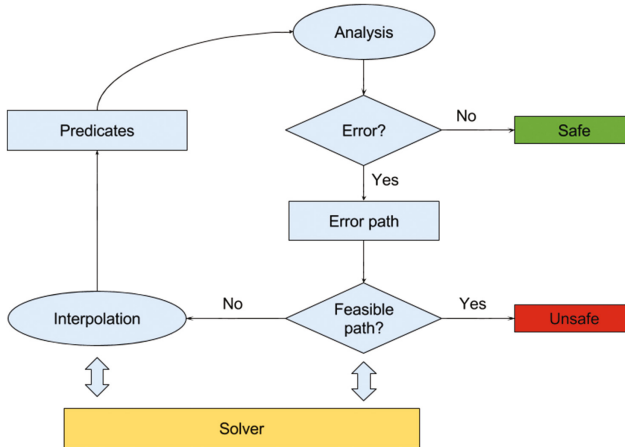


Fig. 3. A CEGAR approach

For checking race conditions we use a modified version of CEGAR. Instead of checking feasibility of the *Error path* we are checking spuriousness of race condition. As far as race condition includes at least two paths we should check feasibility of each path and compatibility of corresponding states. If we found two feasible paths then a race condition is reported. Otherwise, for infeasible paths we get new predicates and the abstraction is reconstructed.

In the example on Fig. 1 during refinement a logical path formula is constructed, containing conditions $[var == 1]$ and $[var == 0]$. The formula is passed to a special component, called Solver which returns a verdict that the formula is unsatisfiable. Together with the verdict the solver returns interpolants which in fact are a contradictory part of the formula. For our example interpolant may be, for example, $[var == 0]$. Further analysis will consider the value of the variable and the infeasible path will not appear in the new abstraction. The Fig. 4 presents a graph of abstract states for Predicate Analysis and Lock Analysis for the program. Braces contain an abstract state of Lock Analysis and abstract states of Predicate Analysis are in square brackets.

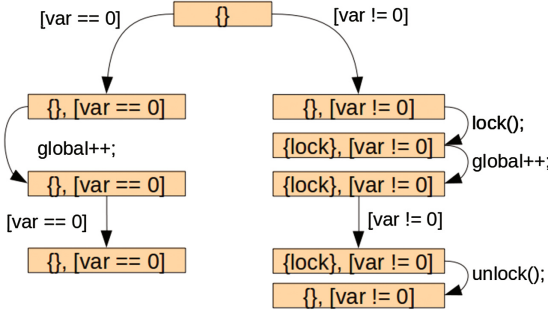


Fig. 4. An example of predicate analysis with lock analysis

Actually a program may contain not only one race condition. There are two ways in CEGAR approach to behave.

- (1) The *Analysis* continues until the first race condition is found. It means that a pair of compatible states with accesses to the same shared data is found. Then the refinement should check a path to each state for feasibility.
- (2) The potential race conditions are refined after the *Analysis* stage and if there are any spurious race conditions, the analysis is repeated on the new more precise abstraction. A disadvantage of the strategy is an amount of repeated work. For example, for similar paths the result of its refinement is equal, and there is no need to refine them all.

We made experiments with both strategies. Both of them show good results, but the second one is more flexible: the CEGAR loop may be stopped and the results reported to the user if the abstraction reaches some level of precision. The first strategy requires highly precise abstraction to be able to output results.

The described refinement method allows to exclude spurious race conditions caused by local infeasible paths. It requires a great amount of time, as to excluding all infeasible paths from the abstraction a high level of precision is required that means a great number of predicates. If the path contains a loop, the refinement should consider all its iterations. The results of our experiments show that majority of spurious race conditions are eliminated from abstraction quickly, and remaining time is spent for proving the correctness.

6 Thread Analysis

There are cases, when accesses to shared data are allowed without usage of explicit synchronization primitives. The main cause is that at the moment only one thread may be active. For example, at initialization stage other threads usually are not created. If the dependency between threads is not considered, many spurious race conditions appear. Consider the example: one thread creates another. The main function start performs some initialization actions, then creates a thread which works with a global variable. As the worker thread can not be executed before it is created, a simultaneous access to the global variable is impossible.

Abstract state of thread analysis contains a set of labels which describes a set of active threads at the moment. Compatibility of states is defined as existence of compatible labels which means corresponding threads can be active simultaneously. A set of labels is modified at points of thread creation or thread joining. The label is a pair: a unique name, related to the thread, and a binary flag. At thread creation point a label with zero flag is added to the set of labels of parent thread, and a label with flag one is added to the set of labels of child thread.

Thread join is handled in a more complicated way. In general case the algorithm of labels flow becomes very difficult, so we consider a simplified case assuming a thread is joined in that thread which created it. In that case after joining the corresponding thread is removed from the set of labels of parent thread. With assumption made in the parent set of labels must have a label with zero flag.

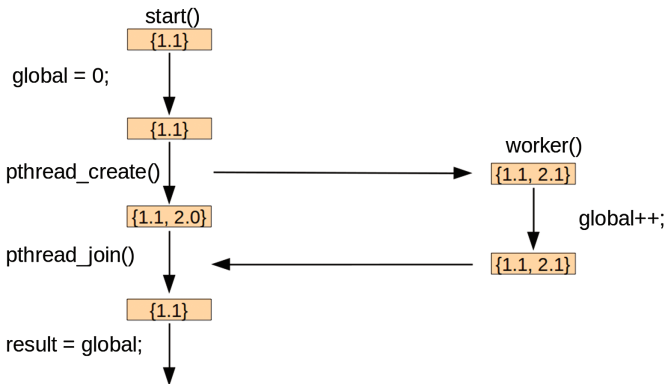
For the example on Fig. 5 the analysis calculates a set $\{1.0\}$ for the first access to a global variable which means only one thread is working (Fig. 6). For the second access to the same memory from the thread worker the set is $\{1.0, 2.1\}$. These two sets are not compatible, as they have no labels with different flags (the first label has the same flag in the both sets). Thus, the race condition is not reported.

```

1  int global;
2
3  int worker(void* arg) {
4      ...
5      global++;
6      ...
7  }

8  int start() {
9      ...
10     global = 0;
11     ...
12     pthread_create(&thread,
13         ..., worker, ..);
14     ...
15     pthread_join(&thread);
16     result = global;
17     ...
18 }

```

Fig. 5. Example of multithreaded code**Fig. 6.** An example of thread analysis

The described method does not support unbounded number of thread creation, this is a direction for a future work. The idea is to use abstract *self-parallel* threads instead of number of similar thread creations. Note, the described method is applied to the simple cases when a thread is joined in the same thread, where it was created. In theory the approach allows to perform the analysis of such cases, but the implementation requires more deep modification of the tool. However, the complicated cases hardly ever appear in real software, so the assumption is rather natural.

7 Experiments

Experiments were performed with a benchmark set which is based on modules of Linux kernel 4.5-rc1, subsystem drivers/net/wireless. Preparation of the benchmarks was performed with LDV Tools infrastructure [14] which forms a driver environment [13]. There were four launches: with one of the extensions (refinement and thread analysis), with both of them and without them.

113 modules of the subsystem were successfully analyzed (Table 1). There were several main causes, leading to unknown verdicts in the other modules, for example, time and memory limits, but the majority of them (54 cases) are related to failures of components of LDV infrastructure.

Table 1. Results of the launch on drivers/net/wireless

	Warnings	Unknowns	Safes	Time, h	Memory, Gb
Thread analysis, Refinement	5	61	51	3.2	8.1
Refinement	6	67	44	4.1	4.0
Thread analysis	27	57	49	2.3	8.2
Default analysis	186	54	43	2.1	3.5

Results show that the thread analysis does not hardly increase the time of analysis, but requires a lot of memory. Refinement behaves contrary. Explanation is simple: thread analysis requires an own state space, and refinement only checks the reachability of the existing states. For the launch with two extensions warnings about race conditions were investigated. All of the five warnings are related to the cases, when data is extracted from a shared list under protection, but the other work is performed without locks. The cases are false alarms.

After that modules of the whole drivers/directory were analyzed with both of the two extensions. There were 2219 warnings, which correspond to 405 modules. Usually, the cause of warnings per module is the same, thus only one warning for every module was analyzed. The most important cause of false alarms is an imperfection of environment model which leads to more than 50% of false warnings. For example, some driver handlers are called with lock protection, and the environment model missed that fact. Moreover, there are dependencies between handlers of different structures which do not allow them to be executed in parallel that is also not considered in the environment model. Note, the environment model is related to the stage of preparation, and not to the analysis.

There were cases, when structures of the same type are used for different purposes with different types of protection. Therefore, the simple memory model leads to about 10% of false warnings.

10% more of false alarms are related to cases when a data is extracted and removed from a shared set under protection and the next work is performed without locks. And about 10% are related to the inaccuracies in the analysis: function pointers, missed locks and shared data detection.

About 15% of warnings are true. Note, that one module with a race condition can produce more than 10 warnings for different variables. Thus, 290 found true warnings correspond to 32 race conditions. These errors are reported to kernel developers. The bugs were partly accepted, and several ones were fixed. The list of fixed bugs are placed at <http://linuxtesting.org/results/ldv>, data race category.

8 Conclusion

In the paper we described two heavyweight extensions of lightweight approach for data race detection, which is implemented on top of the CPAChecker tool. In fact it is a lightweight one, but it allows a flexible adjustment of the balance between resources and accuracy. Our method considers the specifics of operating system kernels, such as complex parallelism and synchronization primitives, and active usage of pointer arithmetic. One more feature is an ability to scale on large amounts of source code.

The described approach of static analysis for race detection shows good results. Analysis of causes of false alarms indicates that to a practical application of the tool to Linux kernel modules there is a need to improve the environment model. Development of the analysis may be continued in several directions: increasing the accuracy of the internal analyzes and a memory model, supporting new synchronization primitives, for example, RCU (read-copy-update) and development of approaches to speed up the analysis. Moreover, a separate task is to investigate the possibility of practical application of the tool to the other classes of tasks.

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_9
2. Andrianov, P., Khoroshilov, A., Mutilin, V.: Lightweight static analysis for data race detection in operating system kernels. In: Proceedings of TMPA-2014, pp. 128–135 (2014)
3. Beyer, D., Keremoglu, M., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Formal Methods in Computer-Aided Design, FMCAD 2010 (2010)
4. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_51
5. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
6. Craig, W.: Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symbolic Logic* **22**(3), 269–285 (1957), <https://www.cambridge.org/core/article/three-uses-of-the-herbrand-gentzen-theorem-in-relating-model-theory-and-proof-theory/7674DE501824D8FC294FB396CD5617DB>
7. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: a constraint-based verifier for multi-threaded programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 412–417. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_32

8. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. *SIGPLAN Not.* **39**(1), 232–244 (2004)
9. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded model checking of multi-threaded C programs via lazy sequentialization. In: Biere, A., Bloem, R. (eds.) *CAV 2014*. LNCS, vol. 8559, pp. 585–602. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_39
10. Levenson, N.: *Safeware: system safety and computers* (1995)
11. Mutilin, V., Novikov, E., Khoroshilov, A.: Analysis of typical faults in Linux operating system drivers (in Russian). *Proc. Inst. Syst. Program. RAS* **22**, 349–374 (2012)
12. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multi-threaded programs. *SIGOPS Oper. Syst. Rev.* **31**(5), 27–37 (1997)
13. Zakharov, I.S., Mutilin, V.S., Khoroshilov, A.V.: Pattern-based environment modeling for static verification of Linux kernel modules. *Program. Comput. Softw.* **41**(3), 183–195 (2015)
14. Zakharov, I., Mandrykin, M., Mutilin, V., Novikov, E., Petrenko, A., Khoroshilov, A.: Configurable toolset for static verification of operating systems kernel modules. *Program. Comput. Softw.* **41**(1), 49–64 (2015), <http://dx.doi.org/10.1134/S0361768815010065>

Tools and Methods of Program Analysis

4th International Conference, TMPA 2017, Moscow,

Russia, March 3-4, 2017, Revised Selected Papers

Itsykson, V.; Scedrov, A.; Zakharov, V. (Eds.)

2018, XVIII, 209 p. 71 illus., Softcover

ISBN: 978-3-319-71733-3