Lightweight Static Analysis for Data Race Detection in Operating System Kernels

Andrianov Pavel
Institute for System Programming
Russian Academy of Sciences,
Email: andrianov@ispras.ru

Khoroshilov Alexey
Institute for System Programming
Russian Academy of Sciences,
Email: khoroshilov@ispras.ru

Mutilin Vadim
Institute for System Programming
Russian Academy of Sciences,
Email: mutilin@ispras.ru

Abstract—**The paper presents an approach to lightweight static data race detection. It takes into account the specifics of operating system kernels, such as complex parallelism and kernel specifics synchronization mechanisms. The method is based on the Lockset one, but it implements two heuristics that are aimed to reduce amount of false alarms: a memory model and a model of parallelism. The main target of our research and evaluation is operating system kernels but the approach can be applied to the other programs as well.**

Keywords. static analysis, data race, operating system kernel, shared data

1. Introduction

Despite a great progress in the field of software verification, bugs associated with parallel execution remain among the most difficult ones to identify. Moreover, concurrency bugs are rather numerous, for example, on average they make up about 20% of all bugs across file systems in the Linux kernel [1]. The most common causes of bugs associated with parallel execution of operating system kernel are data races in which simultaneous access to shared data from multiple threads takes place. In particular, analysis of bug fixes for a year of Linux kernel development has shown that bugs associated with data races constitute the most numerous class and make up 17% of typical bugs [2].

There are two ways for finding data races automatically: dynamic analysis and static analysis. Dynamic analysis techniques allow to obtain a relatively small percentage of false alarms. Examples of tools, implementing dynamic methods, are Eraser [3], RaceHound [4] and DataCollider [5]. They are able to find potential data races only at those paths that occur during the actual execution of a program. A data race requires two almost simultaneous accesses to the same data which complicates its detection. Tools, which use the method of vector clocks, can deal with two accesses occurred at different times, but they are sensitive to order of operations. Also it is known that a significant number of execution paths are difficult to reproduce in a test environment.

Methods of static analysis have the other problems. The heavyweight static analysis is precise but requires a lot of time. In case of data race detection the total number of places, where data race can occur, is too large. There were some experiments with verification of kernel modules source code [8]. Their results showed that heavyweight analysis does not scale on such code. There is a combinatorial explosion of states, so even for small modules the amount of required time and memory was huge.

The methods of lightweight static analysis, e.g. method implemented in the Locksmith tool [6], operate very fast, but the number of false alarms is usually very high. For Locksmith the average number of false alarms was 73% on some POSIX applications and about 96% on several device drivers [7]. The existing methods do not take into consideration some specifics of operating system kernels described below, so the majority of drivers and especially the kernel itself are very difficult to analyze.

In operating system kernels parallelism is more complex, because they are event driven. Many kernel functions can be executed in parallel and it is difficult to define when parallel execution can start. Also in operating system kernels there are additional kernel specific synchronization mechanisms such as disabling of interrupts and scheduling. One more feature is active usage of pointer arithmetic. As a result finding data races in operating system kernels is more difficult than in user space.

In this paper we suggest a new method of lightweight static analysis for data race detection which will be easy to scale to large amounts of source code keeping false alarms rate at reasonable level and will take into consideration the specifics of operating system kernels.

The rest of the paper is organized as follows. In Section 2 required definitions are given. Section 3 describes the idea of the proposed method. After that the idea of Configurable Program Analysis (CPA) is given. The implementation of our method is discussed in Section 5. Section 6 talks about the integration of the implementation into LDV Tools [11]. In Section 7 we speak about the results, then in Section 8 - about related work. In the conclusion future plans are presented.

2. Definitions

In this paper a term **thread** is used to represent an independent flow of execution in operating system kernel, e.g. hardware interrupt handling and executing system calls on behalf of user space thread. If some system call can be interrupted by a hardware interruption, we consider them to be executed in parallel.

A **lock** is an object used for concurrent memory access exclusion. If some lock is acquired from one thread then another thread, trying to acquire the same lock, can not continue its execution until the lock is released. For example, mutexes and spinlocks are typical examples of such locks. We consider kernel specific synchronization mechanisms such as disabling of interrupts and scheduling as specific locks as well. Function `irq_disable()` disables interrupts and thus restricts any parallel execution, so we consider that imagine global lock `irq_disable` is acquired. Some locks can be acquired several times, in this case recursive acquiring takes place.

Shared data — an area of memory which is available from several threads. In C language shared data is presented by global variables and pointers to memory which is accessible from several threads via legal C constructions. It is important to note that sharedness is a characteristic of time. A local data can become shared at one point and return its local status later.

Usage of data — read or write data access.

Data race is a situation when there are two concurrent usages of the same shared data and at least one of access is write. Data race does not always lead to a bug (e.g. access to a statistics counter), but it is a symptom of it. These cases are called benign races.

3. Method of lightweight data race detection

Our method is based on Lockset one[3]. This method maintains set C(v) of potential locks for every shared data v. This set contains those locks that have protected v for the computation so far. Lock l is in C(v) if in the computation up to that point every thread that has accessed v was holding l at the moment of the access. C(v) is initialized by all possible locks. When the variable is accessed, C(v) is updated with the intersection of C(v) and the set of locks held by the current thread. If C(v) becomes empty it indicates a potential data race.

To describe the algorithm of finding data races we should answer following questions:

- When does parallel execution start?
- What is data?
- What data is considered to be the same?
- What are the locks and what are the rules of operations with them?
- What locks are considered to be the same?

Lockset uses points of threads creation to define when parallel execution starts. For operating system kernels it is more difficult to determine when parallel execution begins. We consider that every system call and interrupt handler could be executed in parallel with other ones including itself. The actual interrelation between them is more complex. The thread model in our method is represented by main function, which contains calls of all system calls and interrupt handlers.

Lockset is implemented in the dynamic tool, which operates with run-time memory locations. Our method considers variables and fields of structures as an unit of data by default. There are situations when the accesses to different structure fields should be guarded by locks. For example, we have structure type representing the shared linked list with fields `next` and `prev`. There are two accesses: to the field `next` of one instance of structure and to the field `prev` of another. All static methods which operate with memory locations will have problems in such case, because it is always difficult to understand that two different pointers may points to the same memory location. In our method we can annotate that accesses to these fields are the accesses to the whole list.

As far as Lockset operates with memory locations, the data is shared, if there are two accesses to the same address. For operating system kernels it is difficult to build the data flow graph because of pointer arithmetic and massive parallelism. So this method does not work as well as for user-space programs. In our method the equality of memory locations follows only from the syntax rules. A global pointer is always considered to be pointed to the same memory area. So does a local pointer in a given function. Two structure fields are considered to be pointed to the same memory location if the type of the structure is the same and the names of these fields are equal. It is important to note, that in the name of structure is not considered. So, if structure pointers `A` and `B` have the same type, accesses `A->x` and `B->x` will be considered to be pointed to the same memory location. If structures `A` and `B` are not related this suggestion leads to the false alarm. For this reason there are 18% of all false alarms.

Lockset considers the lock as an object, which can be acquired. This method supports only two operations with it: acquire and release. Our method allows to specify a lock: the acquiring and releasing functions (several ones are possible) and its arguments, recursiveness. The equality of locks follows from equality of object names (variables) in the both of methods.

It is important to note, that our method is interprocedural and explores each path separately as long as they result in different states.

4. Configurable Program Analysis

The suggested method implementation is based on configurable program analysis (CPA) [9]. It is briefly described below.

Configurable program analysis can be combined of several algorithms offering different types of analysis. In addition, it is allowed to configure the analysis algorithm choosing the merging operator and a way to check the termination of analysis.

Configurable program analysis (*`D`*, *`transfer`*, *`merge`*, *`stop`*) consists of abstract domain *`D`*, transfer relation *`transfer`*, merging operator *`merge`* and stop operator *`stop`*. These four components configure the analysis algorithm and affect an analysis accuracy and resources consumption.

Abstract domain *`D`* specifies a set of abstract states. Every abstract state corresponds to its abstract value, i.e. set of concrete states which it represents. A concrete state of a program is a mapping of program variables into values of these variables.

Transfer relation *`transfer`* determines for each abstract state **e** potential following abstract states {**e'**}, where each transition is marked by an edge of the control flow graph (CFG).

Operator `merge` allows to combine information from several paths of analysis. It determines, when two nodes of the reachability tree are merged into one and when they are analyzed individually. In classic lightweight analysis merging always happens, when nodes refer to the same point in the program. In traditional heavyweight analysis nodes are never merged.

Operator `stop` checks whether a current state is covered by a given set of states (already traversed state). It determines when consideration of a path is terminated in a current node. In classic lightweight analysis stop occurs when there is no abstract state including new concrete states, i.e. a fixed point is reached. In heavyweight analysis stop occurs when one set of concrete states corresponding an abstract state is a subset of states corresponding to some other abstract state.

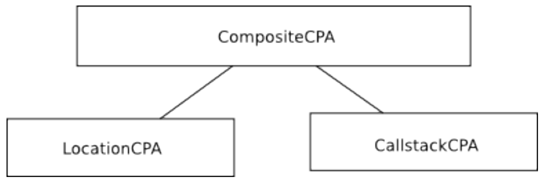


Fig. 1. Simple CPA configuration tree

Let us look at one example of the CPA configuration tree (Fig. 1). We have three CPAs. The main is *CompositeCPA*. It includes *LocationCPA* and *CallstackCPA*.

State of *LocationCPA* contains a node of CFG, e.g. a line number of source code. Hence, its abstract domain is a set of possible nodes. `Transfer relation` changes a line number of current state to a line number of an edge successor. In this CPA `merge` operator never merges states. `Stop` occures only if the CPA has already analyzed current state before.

State of *CallstackCPA* consists of a function callstack. If we call a new function we push its name at the top of the stack. If we return back we pull its name from the stack. It is the work of transfer relation. `Stop` and `merge` are the same as the previous ones.

The aim of *CompositeCPA* is to combine CPAs considered above. Its abstract domain is a cartesian product of *LocationCPA* and *CallstackCPA* domains. The transfer relation of *CallstackCPA* calls the transfer relations of wrapper CPAs. First, it obtains a new state of *LocationCPA*, then a new state of *CallstackCPA*, and combines them together, thus we get the new state of *CompositeCPA*.

`Merge` and `stop` operators are also a combination of the wrapper ones. To merge two states of *CompositeCPA*, first, *LocationCPA* states are merged, then *CallstackCPA* ones are merged and finally they are combined into a next *CompositeCPA* state. The `stop` operator works in the similar way: if all wrapper CPAs stop the *CompositeCPA* stops as well.

Let us consider how the composition of CPAs analyzes the following simple program (Fig. 2). In Fig. 3 there is a graph of an analysis of the program.

```
1  int g(int a) {
2    int b = 0;
3    if (a == 0) {
4      b++;
5    }
6    return b;
7  }
8  int f() {
9    return 0;
10 }
11 int main() {
12   int t;
13   t = f();
14   g(t);
15 }
```

Fig. 2. Example of program

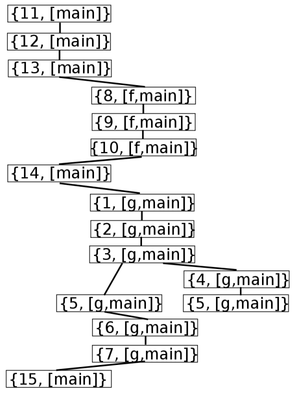


Fig. 3. Analysis graph. First number in the braces represents a state of *LocationCPA* (a line number) and after that follows a callstack of functions.

The tool starts from the main function, then it analyses function `f`, then goes to `g`. In this function it meets the condition at line 3. It analyses two branches and gets the same resulting states at line 5. It means that one state is covered by another, so it continues the analysis with the only state.

5. Implementation

The implementation of the method contains two stages. First of all, shared data is identified, then for every usage of shared data a set of acquired locks is obtained. Fig. 4 represents these stages. A CPA configuration for *Shared Data Analyzer* consists of functions which produce local data, e.g. `calloc()`, `malloc()` and so on. We assume that pointers returned by these functions point to local data, which can not be shared in corresponding points of a program . A configuration for *Lock Analyzer* includes description of locks and annotations which are described in Section 5-C.

A. CPA configuration for Shared Data Analyzer

Shared Data Analyzer is used for collecting a list of shared variables in every point of a program, see Fig. 5.

BAMCPA (Block Abstraction Memorization) [10] is responsible for modularity of the analysis. If a function has been already analyzed with some state before the call and a set of

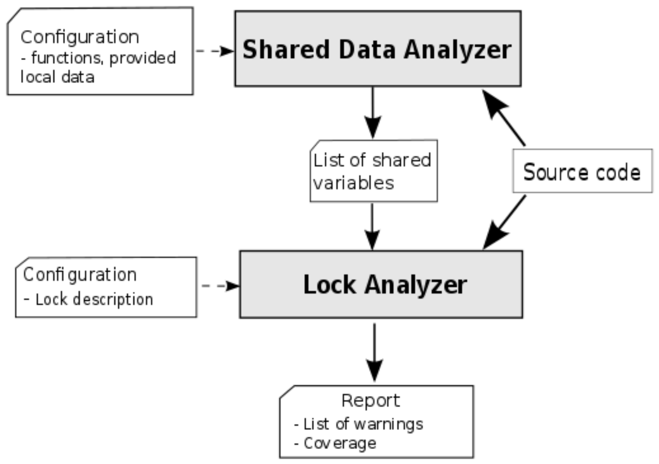


Fig. 4. Stages of analysis

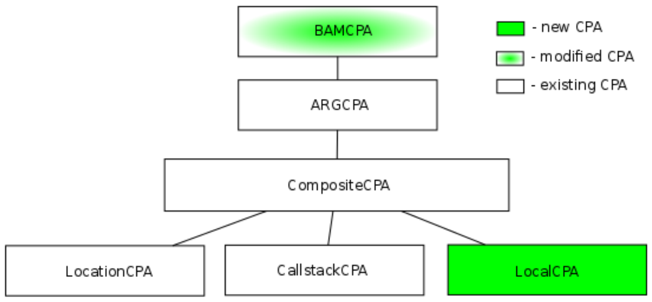


Fig. 5. Shared Data Analyser configuration

resulting states on return from the function were already stored, the reanalysis of this function does not occur, the stored states are used instead. We added to origin BAMCPA a possibility of recursion handling and ways of interaction between BAMCPA and our new CPAs.

ARGCPA (Abstract Reachability Graph) is responsible for restoration of a path from any state to the initial one. It stores parents and children for every state, so it can traverse all reached states and reestablish the path.

CompositeCPA, *LocationCPA* and *CallstackCPA* have already been described.

LocalCPA is responsible for detecting locality of all variables accessible in a current point of a program. The transfer operator should spread the sharedness of variables for assignment operators and function calls. For example, if the pointer `b` points to the shared memory and there is an assignment `a = b` then the sharedness of memory `*b` is transferred to the memory `*a`. After this assignement it is considered that `a` also points to the shared memory. At merge points *`merge`* operator joins results. In case of uncertainty the *shared* status is chosen. Let us consider the following example:

```
if(condition) {
  a = b;
} else {
  a = c;
}
```

If `b` is *local* and `c` is *shared*, then the resulting status for `a` is *shared.*

The result of this stage is a list of shared data at every program location. If we do not exactly know the sharedness we include corresponding data into the list, thus considering it as shared.

B. CPA configuration for Lock Analyzer

Lock Analyzer is used for collecting a set of acquired locks for every usage of shared data, provided by previous stage of analysis.

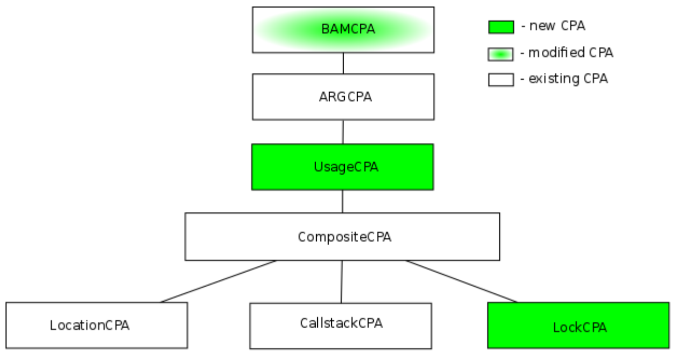


Fig. 6. Lock Analyzer configuration

BAMCPA, *ARGCPA*, *LocationCPA* and *CallstackCPA* are the same.

UsageCPA collects statistics of data usage. Transfer relation of *UsageCPA* identifies variables used in expressions for read/write access, keeps a callstack and a set of acquired locks for each usage.

At the end of analysis we obtain information about all usages for every shared data. Each usage consists of:

- a set of acquired locks;
- a stack of function calls;
- a line number;
- a CFG edge type (a function call expression, etc.);
- a type of access (READ, WRITE).

LockCPA maintains a set of acquired locks. Its state holds a set of locks acquired during program execution. Each lock contains information about:

- name of the lock;
- counter of recursive acquires;
- function callstack for every acquire.

The transfer relation changes the state, which consists of a set of acquired locks. When a function of acquiring a lock is called, the corresponding lock is added to the lock set or the counter is incremented. When a releasing function is called, the the counter is decremented and if it becomes zero the lock is removed from the set.

States of all CPAs are never merged. Analysis stops if a state has been already analyzed.

C. Annotations

Annotations are used to configure the method to the specific code. There are three classes of annotations:

- Annotations of influence functions on synchronization primitives.
- Annotations of influence functions on shared data.
- Annotations of target data.

Let us consider the following chunk of code to explain the first type of annotations:

```
int f() {
   ...
   if (isGlobalPointer) {
      lock();
   }
   (*pointer)++;
   if (isGlobalPointer) {
      unlock();
   }
}
```

In this example the increase of the shared counter always occurs under the lock. If it is a local pointer we do not need a synchronization. The analysis considers four paths because it takes if-then-else branches in both if-statements. Two of these paths end with acquired lock, the other two end with empty set of locks. The first pair is infeasible, because the conditions in the if-statements are the same. So at the end the analysis has two states of acquired lock sets: $\{\text{lock}\}$ and $\{\varnothing\}$, where the first one is not reachable in the real execution.

Such situations do not often occur, but each of them offers a significant number of false alarms, since the final state of the function with acquired lock affects all further paths of analysis. Annotations of functions are used to deal with such cases. It is a way to tell analysis that a function is always releases or acquires some lock.

In the given example it is enough to add annotation that function `f` always release the lock.

Annotations describe functions in terms of *LockCPA* states. After the function has been analyzed, the state is adjusted in accordance with the annotation.

Currently 4 types of specifications are supported:

- Acquiring a lock — a function always acquires a lock.
- Releasing a lock — a function always releases a lock.
- Resetting a lock — if a lock can be acquired several times recursively, a function finally releases it.
- Restoring a lock — a function can modify a set of locks, but all changes should be forgotten at the end of the function.

To this type of annotations one can also add the configuration of locks. It is possible to specify acquiring, releasing and resetting functions, a depth of recursive acquiring. These annotations are handled by *LockCPA*.

The next type of annotations describes the influence on shared data. A function can return a local data or initialized a pointer transferred as argument by local data. Also data can become shared after function call. All these cases should be specified by annotations to precise the analysis. Now only functions provided local data are supported. These annotations are handled by *LocalCPA*.

The third type of annotations is used to establish equality of variables, so they can be regarded as the same data. This is required, for example, for lists, where elements of a list usually have equal variable names like *next*. If we do not distinguish elements of different lists we get many false alarms, because usages of different lists may be protected by different lock sets. That is why we want to bind the variables representing the elements of a list to the list. For this purpose configuration contains functions which are used to work with a list. For example, expression `e = getElement(list)` binds variable *e* to variable *list* passed as a parameter. These annotations are handled by *UsageCPA*.

The total number of annotations is not very big. There are about 50 annotations of the first type and 20 annotations each of the second and the third types.

6. Tool integration

We integrated the proposed method into Linux Driver Verification (LDV Tools) developed within project for the verification of Linux operating system device drivers (see Fig. 7) [11].

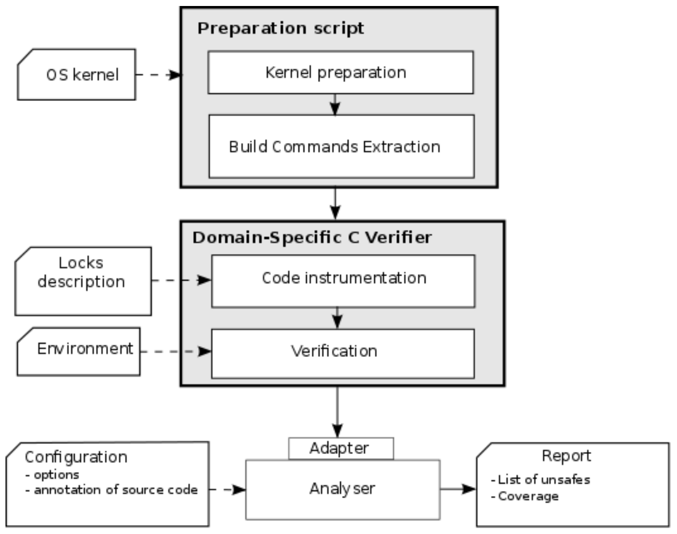


Fig. 7. Integration in the LDV architecture

First, a given kernel of an operating system is prepared. During this stage compiler calls are replaced by our command extractor calls. Then a build command stream is extracted by special scripts. This stream is transferred to *Domain-Specific C Verifier* component. It instruments source code using locks description. For example, it replaces macros used for acquiring and releasing locks by calls of model functions, annotated in the configuration, because macros can be expanded to very

Statistics	General	Unsafe
Global variables:	195	29
Simple:	122	23
Pointer:	73	6
Local variables:	3	0
Simple:	0	0
Pointer:	3	0
Structure fields:	118	24
Simple:	105	24
Pointer:	13	0
Total:	316	53

TABLE I. Example of one report for Linux driver `floppy.ko`

difficult command sequences, while model functions are easier to analyze.

Then a model of environment is included. It is presented by *main* function containing system calls and interrupt handlers calls, which are supposed to be executed in parallel.

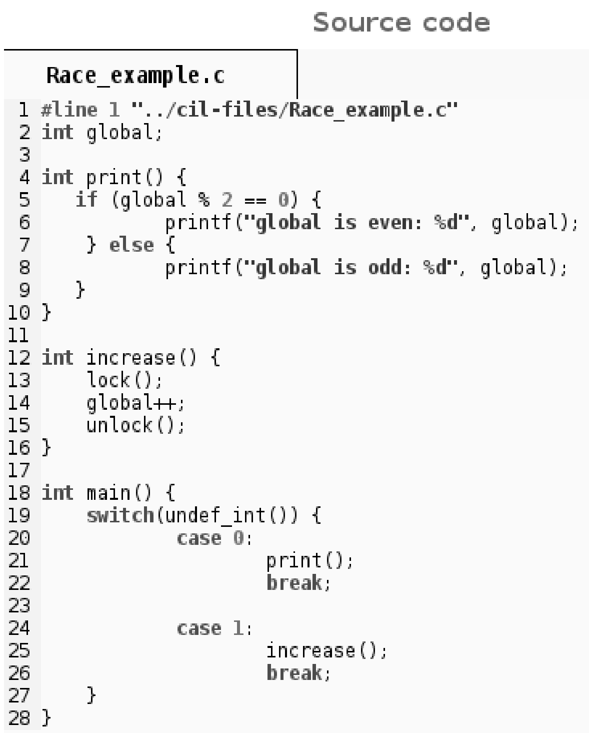
After all preparations a source code is analyzed by our data race analyzer. It generates report containing a list of warnings with detail information about each of them.

To visualize the warnings another component of LDV Tools called *Error Trace Visualizer* is reused. When the tool generates a warning about the data race, it must be shown to the user. Moreover, the user should check if it is a false alarm or a bug. Therefore, it is necessary to present visualization of the error trace and its association with the source code. The data race is represented by at least two usages with disjoint sets of locks. Its error trace contains the function callstacks for two usages with points of lock acquiring. *Error Trace Visualizer* interprets data received from the verifier, converts it and associates it with corresponding source code. To represent the results the HTML-report is generated. The main page of a report contains general statistics (Tab. I). There are total numbers of variables of each of three categories: global, local and structure fields and number of variables for which warnings are generated. The pointer variable means the access by pointer and simple one — the access to variable itself. Also the report lists all found locks. After that there is a list of all warnings. For each unsafe the report contains a pair of usages with disjoint sets of locks.

An example of source code representation is shown in Fig. 8. We emphasize again, that there is a model of parallelism and two functions called from the *main* function are considered to be executed in parallel.

Function `print` prints information about variable `global`, and `increase` increments its value with lock protection. There is a data race, because function `increase` can write to the variable simultaneously with the check in function `print`. So as a consequence of the race the printed output message may be wrong. Our tool generates a warning for variable `global` with error trace shown in Fig. 9. In the first line the total number of usages is printed. Only two of them are shown. The first usage means that the access to `global` is in function call. The second one occurs in assignment after acquiring the `example_lock`. The name `example_lock` is chosen because the acquiring function `lock()` has no arguments.

Also there is an option to generate source code coverage.



```
#line 1 "../cil-files/Race_example.c"
int global;

int print() {
   if (global % 2 == 0) {
           printf("global is even: %d", global);
    } else {
           printf("global is odd: %d", global);
   }
}

int increase() {
   lock();
   global++;
   unlock();
}

int main() {
   switch(undef_int()) {
           case 0:
                   print();
                   break;

           case 1:
                   increase();
                   break;
   }
}
```

Fig. 8. Example of source code. Parallel execution of `print()` and `increase()` can lead to a data race.

It shows the code which has been analyzed by the verifier and its relation to the whole kernel code.

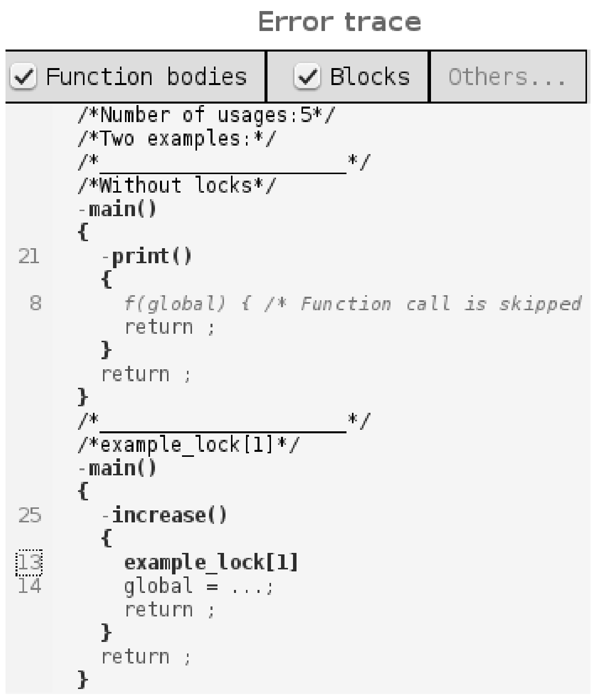
7. Results

The tool was applied to a real time operating system kernel, which has been already tested and has been used several years in production. The amount of code was about 200 000 lines of code, but only about 50 000 lines were analyzed. This is due to many functions was not included into *main* function. We found 20 new data races acknowledged by the developers. The number of warnings was 139. Without the Shared Data Analyzer the number of warnings would be 378. So this stage is very important. At the moment the large part of false alarms are caused by inaccuracies in the analysis of expressions. For instance, now the analysis does not properly consider conditions in if-statements.

It takes about 3 minutes and 6 Gb of RAM for the run. Also there was a pilot launch of the tool on Linux kernel 3.8 on the `drivers` directory. The number of analyzed modules was about 3500. The tool generated about 900 warnings. Several of them were analyzed and one actual bug was found, but it had been already fixed.

8. Related work

In our method we perform static analysis in contrast to dynamic analysis which has its own benefits. We are considering only methods for analysis of C code, excluding, for example, Java analyzers like [12]. The Locksmith method [6] is the most similar. It is also based on the Lockset algorithm, but has different approaches for the analysis of locks and shared data. Basically it performs intraprocedural analysis with propagation



```
     /*Number of usages:5*/
     /*Two examples:*/
     /*____________________*/
     /*Without locks*/
     -main()
     {
21     -print()
       {
8        f(global) { /* Function call is skipped
         return ;
       }
       return ;
     }
     /*____________________*/
     /*example_lock[1]*/
     -main()
     {
25     -increase()
       {
13       example_lock[1]
14       global = ...;
         return ;
       }
       return ;
     }
```

Fig. 9. Example of warning. Error trace with points of acquiring locks and points of calling functions. Every point links to corresponding line on source code (see Fig. 8)

of constraints which gives it context-sensitivity, but it does not take into account path conditions. Locksmith uses the points of thread creation to define when the parallel execution starts. It operates with run-time memory locations (addresses) and to establish the equality of data it builds data flow graph. This way is more precise for basic data types, but it has problems with casting, `void*` pointers and pointer arithmetic. Our method operates with variable names and it works better on kernel specific code, but generates more false positives for user-level programs. The way of calculation of shared data also has a difference. In Locksmith shared memory locations are defined for all points in program once, but there are cases, when the local data becomes shared after some actions. For example, the memory is allocated by `malloc()` and then it is assigned to a global pointer. Locksmith considers this memory as always being shared even before assignment. Our method allows to distinguish these cases. The Locksmith supports only standard locks, as `pthread_mutex_lock`. It also builds a data flow graph for arguments of functions, acquired a lock. It means that the way of establishing the equality of locks is more precise in case of simple programs.

Another tool which is based on the Lockset algorithm is Relay [13]. It describes the changes in the locksets and accesses to the memory locations relative to the function entry point. The thread model is similar to our approach - all functions are considered to be executed in parallel. But after the analysis heuristics can be applied, which remove warnings related to the inaccurate model. The data is symbolic L-Values, which are defined as follows: a variable, a pointer to variable, a field of a structure or pointer to another L-Value. So, the definition of data is very similar to ours, but the sharedness is different. For each program location every L-Value is mapped to R-Value, which may be $\perp$ (unassigned), $\top$ (unknown), integers, the incoming (initial) value of some L-Value, and a may-points-to set of L-Values. The data is the same (shared) if the L-Values are equal or the other L-Value is included in may-points-to set of current L-Value. Locks are objects, which can be acquired and released by specified functions. For every function the relative lockset is computed. It is a pair of definitely acquired locks and possibly released locks. An important note is that the locks are also data, relative to function entry point. So, if a lock is an argument of function call, it will be updated in terms of function caller variables. Finally, for every function there is a set of accesses with guarded locks and the effect of function, including the relative lockset.

The method, proposed in [14] is focused on function pointer analysis and a fast computation of must-aliases. There are three thread models. The first one is similar to our model - every function can be executed in parallel with other ones. The second model is based on `fork` and `join` functions. The parallelism starts at fork operation and finishes at join point. The third model differs from previous one by absence of join points and limitation of number of tasks at one thread. The data are memory locations and the same locations are defined as set of must aliases. The locks are also a set of must-aliases.

We do not consider methods of model checking in detail, although many of them are used for data race detection([16], [17], [18]). The main problem is its scalability. Most of them are useful for applications with hundred of program locations. The another strong limitation of such methods is suggestion that threads interact with each other only by global variables.

9. Conclusion

In the paper we describe a new lightweight approach for data race detection, which is implemented on top of the CPAchecker tool. 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 main distinctions of suggested method are a memory model, a model of parallelism and a way of shared data determination.

The main problem of the method is a great amount of false alarms. To deal with them we want to try an existing method, called CEGAR (Counterexample Guided Abstraction Refinement) [15] which takes into account conditions by means of predicated abstraction. In case of data races CEGAR algorithm should be modified to take into account that two threads should be considered instead of one.

Also we want to apply the suggested method to the Linux kernel.

References

[1] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu, *A Study of Linux File System Evolution*, 11th USENIX Conference on File and Storage Technologies (FAST '13)

[2] Mutilin V.S., Novikov E.M., Khoroshilov A.V. *Analysis of typical faults in Linux operating system drivers.* Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, pp. 349–374, 2012 (in Russian).

[3] Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, Thomas Anderson *Eraser: A Dynamic Data Race Detector for Multi-threaded Programs* ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[4] Gerlits E.A., Kuliamin V.V., Maksimov A.V., Petrenko A.K., Khoroshilov A.V., Tsyvarev A.V. *Testing of Operating Systems*. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, pp. 73–107, 2014 (in Russian).

[5] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk *Effective Data-Race Detection for the Kernel* Operating System Design and Implementation (OSDI'10), 2010, USENIX.

[6] Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. *Locksmith: Practical Static Race Detection for C*, ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 33(1):Article 3, January 2011.

[7] Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. *Locksmith: Context-Sensitive Correlation Analysis for Race Detection*, Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pp. 320 - 331, ACM New York, 2006

[8] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, Georg Weissenbacher, *Model Checking Concurrent Linux Device Drivers*, ASE'07, November 4–9, 2007

[9] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz, *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis*, ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[10] Daniel Wonisch, *Block Abstract Memorization for CPAchecker*, TACAS 2012, LNCS 7214, pp. 531-533.

[11] Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. *Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]*. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).

[12] Mayur Naik, Alex Aiken, John Whaley *Effective static race detection for Java*. PLDI '06 Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, 41, 6, pp. 308 - 319, 2006

[13] Jan Wen Voung, Ranjit Jhala, Sorin Lerner, *RELAY: Static Race Detection on Millions of Lines of Code*. ESEC/FSE'07, 2007

[14] Kahlon V., Yang Y., Sankaranarayanan S., Gupta A.: *Fast and accurate static data-race detection for concurrent programs*. In: CAV'07. LNCS, vol. 4590, pp. 226-239. Springer (2007)

[15] Khoroshilov A.V., Mandrykin M.U., Mutilin V.S. *Introduction to CEGAR — Counter-Example Guided Abstraction Refinement* Trudy ISP RAN [The Proceedings of ISP RAS], vol. 24, pp. 219-292, 2013 (in Russian)

[16] C. Popeea, A. Rybalchenko. *Threader: a verifier for multi-threaded programs* In Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013), LNCS, vol. 7795, pp. 633-636, 2013.

[17] A. Gupta, C. Popeea, A. Rybalchenko. *Predicate abstraction and refinement for verifying multi-threaded programs* In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), pp. 331-344, 2011.

[18] A. Gupta, C. Popeea, A. Rybalchenko. *Threader: a constraint-based verifier for multi-threaded programs* In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), LNCS, vol. 6806, pp. 412-417, 2011.