An approach to lightweight static data race detection

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 a lightweight approach to static data race detection. It is based on the Lockset one, but it implements several simplifications that are aimed to reduce amount of false alarms. The approach is implemented on top of CPAchecker tool and its evaluation is in progress. The main target of our research and evaluation is operating system kernels but the approach can be applied to analysis of other programs as well.**

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

I. Introduction

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

At the moment 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, but they are able to find data races only at those paths that occurred during the actual execution of a program. As far as data race recquires two almost simultanious accesses to the same data, this fact reduces chance of its detection. Also it is known that a significant number of execution paths are difficult to reproduce in test environment. Examples of tools, implemented such method, are Eraser [3], RaceHound [4] and DataCollider [5].

Methods of static analysis have the other problems. The lightweight methods, e.g. method, implemented in the tool Locksmith [6], analyze source code superficially. Such methods allow to find simple errors and work very fast. But the number of false alarms is about 90% on device drivers and about 98% on some POSIX applications [7]. The idea of such methods can be improved to decrease number of false alarms. The another way is heavyweight analysis. It is much more precise, but requires a lot of time. In case of data race detection total number of places, where data race can occur, is too large. There are some experiments with verification of kernel modules source code, for example, DDVerify [8]. But the results showed that the heavyweight multithreaded analyis does not scale on such code. They got a combinatorial explosion of states, so, even for small modules the amount of recuired time and memory was huge.

In operating system kernels parallelism is more complex and less precise, because many kernel functions can be executed in parallel and it is difficult to define when the parallel execution can start. So finding data races in operating system kernel is much more difficult than in user-level programs.

So there is a need to create a lightweight method of static analysis which is easy to scale to large amounts of source code and allows to find most of error cases while keeps false alarms rate at reasonable level.

In this paper we suggest a new method of static analysis for data race detection, implemented on top of CPAchecker tool [9].

The rest of the paper is organized as follows. In the Section 2 required definitions are given. After that the idea of analysis is presented. The Section 4 describes the idea of configurable program analysis. After that implementation of our method is given. Then a need of annotations of source code is explained. The Section 7 talks about the solution architecture. After that the visualization of results is presented. In conclusion we briefly speak about the results, related work and future plans.

II. Definitions

In this article term thread is used to represent independent thread of execution in operating system kernel, for example, interrupt handlers and system calls executed on behalf of user space threads.

A **lock** is an object used for concurrent memory access exclusion. If lock is acquired from one thread the another thread trying to acquire the same lock can not continue its execution before the lock is released. For example, mutexes and spin locks are typical examples of such locks. We consider the kernel specific syncronization mechanizms such as disabling of interrupts and scheduling as specific locks as well. The lock can be linked with an address. For example, function `mutex_lock(&mutex)` linked with the address `&mutex`.

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 condition is a situation when there are two concurrent access to the same shared data and at least one of the access events is writing. Data race does not always lead to an error (for example, access to statistics counter), but it is a symptom of it.

III. Lightweight method of data race detection

Our method is based on algorithm Lockset [3]. This algorithm considers two usages of the same data as a data race if these usages occur with disjoint sets of locks. The algorithm Lockset stores locks for every thread and set `C` of potential candidates of locks for every usage of shared data. If a usage occurs it intersects `C` and set of acquired locks for current thread and obtain new set $\tilde{C}$. If the last one is empty this is potential data race.

As far as our goal is to develop a lightweight static analysis method to analyze large amount of source code we apply some simplifications.

Our first heuristics is a definition if shared data is the same. We do not analyze a memory model but consider memory locations as the same only by syntax rules. For variables the equality of memory locations follows only from the equality of variables. Pointers with equal names are always considered to be pointed to the same memory area.

The second simplification makes a try to decrease a number of false alarms. We generate warnings only for usages of data, protected by locks at least once. If there are two usages without locks at all we do not generate warning. This is rather strong limitation because many real errors can occur exactly without locks.

The last simplification is the model of threads. We consider that every kernel API function, specified in documentation, could be executed in parallel with other one, including itself. The actual interrelation between kernel API functions is more difficult.

IV. Configurable static analysis

As far as our method implementation is based on Configurable Program Analysis (CPA) [9] let us briefly describe it.

The main idea of configurable static analysis is to combine advantages of data flow analysis and model checking methods and create a way to configure its interrelation. On the one hand, in data flow analysis algorithm works with control-flow graph (CFG) of program, disseminating information along the edges, on the other hand, model checking algorithms of heavyweight analysis unroll the tree of reachable states until one of the states will not be covered by another state.

In configurable static analysis proposed in [9] it is allowed to configure the analysis algorithm CPA choosing the merging operator and way to check the completion of the analysis. In addition, configurable static analysis can be configured of several algorithms CPA offering different types of analysis.

Configurable program analysis (`D, transfer, merge, stop`) consists of an abstract domain `D`, transfer relation `transfer`, merge operator `merge`, the stop operator `stop` as described below. These four components configure analysis algorithm and affect the accuracy and the resources required for analysis.

Abstract domain `D` specifies the set of abstract states. Every abstract state is matched to its abstract value, i.e. set of concrete states, which it represents. Concrete state of program is a mapping of program variables set into the 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 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 the data flow analysis merging always happens, when nodes refer to the same point in the program. In classical methods of model checking nodes are never merged.

Operator `stop` checks whether the current state is covered by set of given states (already completed state). It determines when consideration of a path is stoped in current node. In the data flow analysis stop occurs when there is no abstract state including new concrete states, i.e. a fixed point is reached. In the methods of model checking stop occurs when one set of concrete states corresponding an abstract state is a subset of states corresponding to some other abstract state.

Let us look at one example of CPAs configuration tree (Fig. 1).

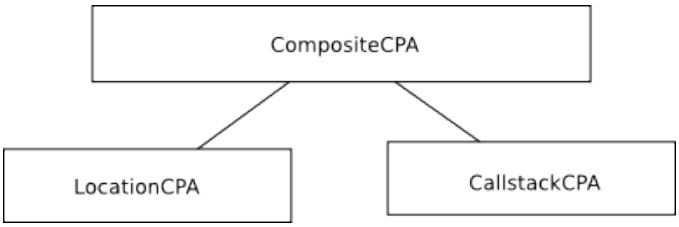

Fig. 1. Simple CPA configuration tree

We have three CPAs. The main is *CompositeCPA*. It includes *LocationCPA* and *CallstackCPA*.

State of *LocationCPA* contains only a line of source code. Hence its abstract domain is the set of possible line numbers.

`Transfer relation` changes the line number of current state to the line number of the edge successor.

`Merge` operator never merges states. `Stop` occures only if we have already analyzed the current state before.

State of *CallstackCPA* consists of function calls stack. If we call a new function we push its name at 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 previous ones.

The aim of *CompositeCPA* is to combine CPAs mentioned above. Its domain is a cartesian product of *LocationCPA* domain and *CallstackCPA* domains. Transfer relation of *CallstackCPA* calls the containing transfer relations. 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 containing ones. To merge two states of *CompositeCPA*, first, Location states are merged, then Callstack ones are merged and finally they are combined into a next Composite state. The `stop` operator works in the similar way: if any containing CPA consider to stop analysis the analysis is stoped.

Consider, how the composition of CPAs analyzes the simple code:

```
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. }
```

In figure 2 there is a path of an analysis of the program above.

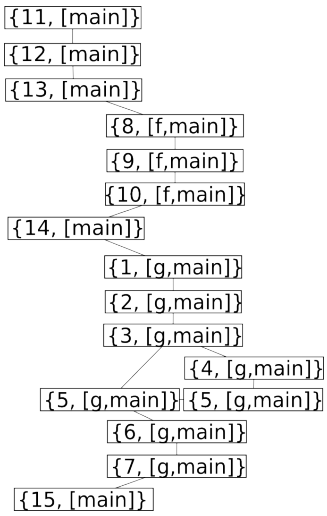

Fig. 2. Analysis path

First number in the braces is representing the state of *LocationCPA* (the line number) and after that follows the call stack of functions. Analysis starts from the main function, then it analyses the function `f`, then goes to `g`. In this function it meets if 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.

V. Implementation

The implementation of the method is proposed to be into two stages. First of all, the shared data are identified, then for every usage of shared data the set of acquired locks is obtained. Figure 3 represents these stages in CPAchecker-Lockator. The

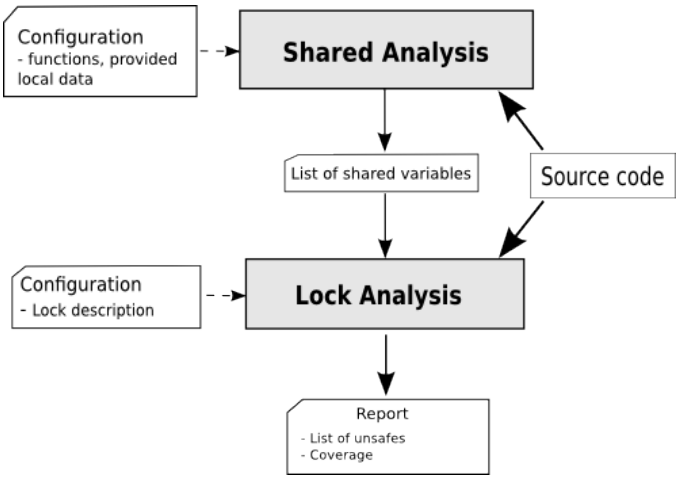

Fig. 3. Stages of analysis in CPAchecker-Lockator

configuration for *Shared analysis* consists of functions which introduce local data, for example, `calloc()`, `malloc()` and so on. We are sure that pointer returned by these functions points to local data and in current point of program it can not be shared. The configuration for *Lock analysis* includes locks descriptions and annotations which are described in section VI.

A. CPA configuration for Shared analysis

Shared analysis is used for collecting the list of shared variables in every point of a program, see Fig. 4.

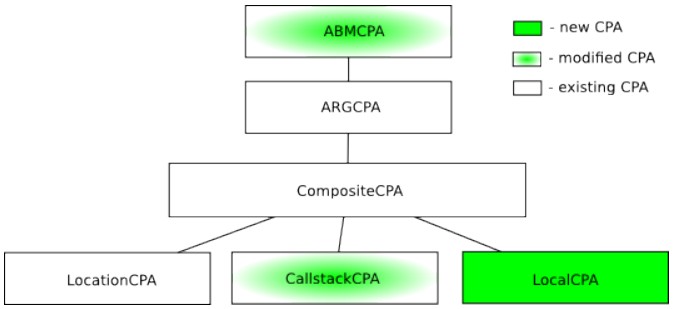

Fig. 4. Shared analysis configuration

BAMCPA (Block Abstraction Memorization) [10] — is responsible for modularity of analysis. If a function has been already analyzed with some state before the call and a set of resulting states on return from the function were already stored, the reanalysis of this function does not occur, the stored states are used instead.

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

CompositeCPA — provides the analysis where the state is a product of the containing CPAs and the transter is performed for all containg CPAs simultaneously.

LocationCPA — stores current line of source code in its state and is responsible for traversal of CFG.

CallstackCPA — stores a stack of called functions, so it is responsible for transfering by function calls and returns.

LocalCPA — is responsible for detecting locality of all variables accessible in current point of program. The data status can be *local*, *global* or *unknown*. The task of transfer operator is to spread the status variables for assignment operators and function calls. For example, if there is assignment `a = b` then status of variable `b` is transferred to the variable `a`. At merge points the analysis joins results on the branches, such that for each variable the maximal status is taken. Considering the following example:

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

At the merge point of the two branches of the if statement the status for `a` variable is taken as maximal status between then-branch and else-branch. Therefore, if `b` is *local* and `c` is *global*, then the result for `a` is *global*.

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

B. CPA configuration for Lock analysis

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

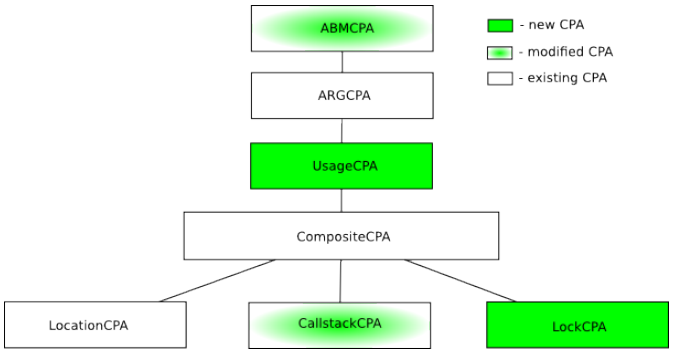

Fig. 5. Lock analysis configuration

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

UsageCPA collects statistics of data usage. Transfer relation of *UsageCPA* identifies variables used in the expressions for read/write access and keeps the call stack for the usage, as well as a set of acquired locks.

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

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

The *UsageCPA* is also used to establishing equality of variables, so they can be regarded as the same data for the analysis. This is required, for example, for lists, where the elements of the list usually have equal variable names like *next* given by the field name of the list structure. If we do not distinguish elements of different lists we get many false alarms, because the usages of different lists may be pretected by different lock sets. That is why we want to bind the variable representing the elements to the list variable name to distinguish between the other lists. For this purpose the configurations contains functions which are used to work with the list. For example, the expression `e = getElement(list)` binds the variable *e* to the variable *list* passed as a parameter. In transfer relation upon detection of an annotated function the binding relation is changed accordingly.

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

- Name of the lock;
- Recursive counter of acquires;
- Stack of function calls for every acquire.

Transfer relation changes the state of a plurality of acquired locks. When a lock acquisition function 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.

VI. Annotations

Let us consider the following chunk of code:

```
if (!isLOCKED) {
   lock();
}
global_var++;
if (!isLOCKED) {
   unlock();
}
```

In this example the increase of the global counter always occurs under the lock. Either someone acquires it earlier, or this function acquires it and releases it after increasing counter. But the analysis considers four paths because it can take if or else branch in both if statements. Two of these paths are 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 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 output of the function under lock affects all further paths of analysis. The annotation of functions are used to deal with such cases. It is a way to tell analysis that a function is always releases or acquires the lock.

Annotation describes function 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 — function always acquires a lock.
- Releasing lock — function always releases a lock.
- Reseting a lock — if the lock can be acquired several times recursively, the function totally releases it.
- Restoring a lock — function does not modify set of locks, all changes should be forgotten.

VII. Solution architecture

For race detection we reuse the Linux Driver Verification (LDV [11]) architecture developed within ISPRAS project for the verification of Linux operating system device drivers (see Fig. 6).

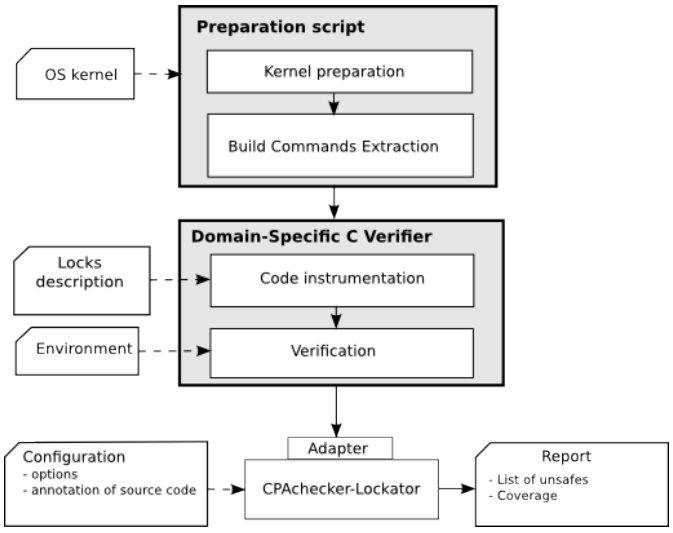

Fig. 6. Solution architecture

First kernel of operating system is prepared. During this stage compiler calls are replaced by our command extractor calls. Also other modules are prepared for building at this stage. Then build command stream is extracted by special scripts. Obtained command stream is transmitted to *Domain-Specific C Verifier* component. It instruments source code, using locks description. For example, it replaces macros used for acquisition and releasing locks by model functions, annotated in the configuration, because macros can be expanded to very difficult command sequence, while model function is easier to analyze.

Then model of environment is included. It is presented by *main* function containing system calls which can be executed in parallel according to documentation and interrupt handlers calls. So we consider that all functions called from *main* are executed simultaneously.

After all preparations the source code is analyzed by CPAchecker-Lockator. It generates report containing a list of unsafe cases with detail information about each of them.

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 general report for launch on Linux driver `floppy.ko`

VIII. Visualization of results

To visualize the potential cases of data races 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 true error. Therefore it is necessary to present visualization of the unsafe error trace and its association with the context - source code under analysis. *Error Trace Visualizer* interprets the data received from the verifier, converts them and associates it with the 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, producing unsafes. The pointer variable means the access by pointer and simple one — the access to varible itself. Also the report lists all found locks. After that there is a list of all unsafes, that could potentially be a data race. For each unsafe the report contains a pair of usages with disjoint sets of locks.

Also there is an option to generate source code coverage. It shows the code which has been analyzed by the verifier and its relation to the whole kernel code.

An example of source code presentation is shown in Fig. 7.

Here we can see two functions called from the entry point (*main* function). The function `print` prints information about `global` variable, and `increase` increments its value with lock protection. There is a data race, because function `increase` can write to the variable simulteneously with the check in the 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. 8.

On the left side we can see the error call stack with points of acquiring locks and points of calling functions. Every point links to corresponding line on source code (see Fig. 7).

IX. Results

The method was applied to a real time operating system kernel. It has been already tested and was worked several years in production. The amount of analyzed code was about 50 000 lines. We found about 20 new data races, acknowledged by developers. Total amount of warnings was 139. It takes about 3 minutes and 6 Gb of RAM for the analysis. Also there was a test launch of the tool on the Linux kernel 3.8, on `drivers/` directory. The amount of analyzed modules was about 3500.

Source code

Race_example.c

```
1 #line 1 "../cil-files/Race_example.c"
2 int global;
3
4 int print() {
5    if (global % 2 == 0) {
6           printf("global is even: %d", global);
7     } else {
8           printf("global is odd: %d", global);
9    }
10 }
11
12 int increase() {
13     lock();
14     global++;
15     unlock();
16 }
17
18 int main() {
19     switch(undef_int()) {
20             case 0:
21                     print();
22                     break;
23
24             case 1:
25                     increase();
26                     break;
27     }
28 }
```

Fig. 7. Example of source code

Error trace

Function bodies | Blocks | Others...

```
    /*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. 8. Example of unsafe

The tool generated about 900 unsafe cases. Several of them were analyzed and one actual bug was found, but it had been already fixed in the newest version of Linux kernel.

X. Related work

In our method we are performing static analysis in contrast to dynamic analysis which has its own benefits. We are considering only methods for the analysis of C code, excluding for example Java analyzers, like [12]. The method of Locksmith [6] is the most similar. It is also based on Lockset algorithm, but has different approaches for the analysis of locks and shared data. Basically it performs intra-procedural analysis with propagation of constraints which gives it context-sensitivity, but it does not take into account path conditions. Our method is inter-procedural and explores each path separately as long as they result in different states. As far as it is based on CPAchecker it allows to extend the method with existing analysis, like explicit or predicate.

XI. Future work

The main problem of all methods of static analysis is a great amount of false alarms. Some of them can be discarded with help of shared analysis. But at the moment the large part of false alarms are caused by inaccuracies in analysis of expressions. For instance, now the analysis does not properly consider conditions in if statements.There is an existing method, called CEGAR (Counterexample Guided Abstraction Refinement) [14], which takes into account conditions by means of predicated abstraction, but it is used for checking reachability properties. In case of data races CEGAR algorithm should be modified to take into account that two threads should be considered instead of one. In particular, it means that it should be able to refine many error paths together.

Another issue is a full launch on Linux kernel with analysis of results.

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] Khoroshilov A.V., Mandrykin M.U., Mutilin V.S. *Introduction to CEGAR — Counter-Example Guided Abstraction Refinement* pp. 219-292. (in Russian)