Minimizing the number of static verifier traces to reduce time for finding bugs in Linux kernel modules

Vitaly Mordan
Institute for System Programming
Russian Academy of Sciences
Moscow, Russia
Email: mordan@ispras.ru

Evgeny Novikov
Institute for System Programming
Russian Academy of Sciences
Moscow, Russia
Email: novikov@ispras.ru

***Abstract*— Static verifiers usually stop after they find a first bug in a program under analysis. This slows down the process of finding and fixing of bugs of the same kind in a given Linux kernel module. In order to solve this problem we used the static verifier CPAchecker with option to continue analysis after finding of a first bug. Besides we extended LDV Tools – a toolset for verification of Linux kernel modules – for finding several bugs in a given module against a specified rule specification. But first experiments revealed a new problem – the verifier produced too many similar traces. The given paper introduces a formal definition of equivalent traces and presents different comparison algorithms and a semi-automated approach, which makes possible to find several bugs in a given Linux kernel module against a specified rule specification at once.**

Keywords—Linux kernel module; correctness rule; static verifier trace; equivalence class.

I. INTRODUCTION

The Linux kernel is one of the most fast-paced software projects [1]. The number of its active developers is more than one thousand. A new release comes out in 2-3 months, it contains thousands of changes. At present the Linux kernel consists of more than 15 million lines of code. At the same time every bug in the Linux kernel is critical [2]. Researches revealed that most of bugs in the Linux kernel are located in its modules (modules contain approximately 7 times more bugs than the kernel itself) [3]. One of the approaches to find those bugs is using static verifiers. At the moment just LDV Tools [4] among all toolsets that allow to apply static verifiers for Linux kernel modules are under active development. This toolset has already helped to find 150 bugs in Linux kernel modules and corresponding patches were applied by the Linux kernel developers [5].

Each Linux kernel module can contain any number of bugs. At the same time most of static verifiers stop after they find a first bug. One can fix this bug and repeat verification until no bugs will be detected in a given module but this approach may take a lot of time and can not be automated.

Let us consider the following program (Fig. 1). There is function *getnchar* which works properly only with positive values of its parameter. A static verifier needs to check that all calls of this function is correct. A formal task for the static verifier is to check that label *ERROR* can not be reached.

```
1:  void getnchar(int n)
2:  {
3:    if (n <= 0)
4:      ERROR: goto ERROR;
5:    ...
6:  }
7:  int main()
8:  {
9:    getnchar(-1);
10:   getnchar(-2);
11:   return 0;
12: }
```

Fig. 1. Program with 2 bugs.

There are two bugs in this program: at lines 9 and 10 correspondingly. Most of static verifiers most likely will find the first one and will stop their analysis. After fixing of this bug the user may think the program is safe but it is not the case the program still has another bug.

For simple programs this problem is not so critical. But for such big projects like the Linux kernel, where checking of all modules against one rule specification requires more than 1 day and where fixing of bugs requires preparing and applying of corresponding patches, this problem greatly increases time for finding all bugs.

This paper presents an approach to reduce time for finding all bugs in Linux kernel modules. The approach based on Linux Driver Verification Tools (LDV Tools), which are an open source toolset for checking correctness of Linux kernel modules against rule specifications with help of different static verifiers [6]. The LDV Tools architecture is presented in Fig. 2.

LDV Tools provide an interface to verify a set of modules against a set of rule specifications. When verification is complete, verification results will be placed in an archive. After that, this archive can be uploaded to a database. LDV Analytics Center allows to analyze the verification results. It is integrated with Error Trace Visualizer that visualize traces from a static verifier to allow the user either to find actual bugs or to identify source of false alarms. Also LDV Analytics Center provides an

interface to Knowledge Base that keeps information on all already analyzed unsafes. Knowledge Base scripts mark new uploaded static verifier traces if they are equivalent to any of already analyzed traces by specified criteria.

There are several examples of more than one bug for a given rule and a given Linux kernel module, which were found by LDV Tools [5]. For example, bug reports L0029[1] and L0034[2] are devoted to a lack of mutex release in different functions in the kernel of versions 2.6.38 and 2.6.39 correspondingly. Bug report L0030[3] deals with a lack of mutex release in a couple of places and so on. These examples show that sometimes it is possible to find similar bugs if a verification toolset is able to find one of them. In order to solve this problem it was decided to extend LDV Tools so that the toolset could find all bugs in a given module against a specified rule specification.

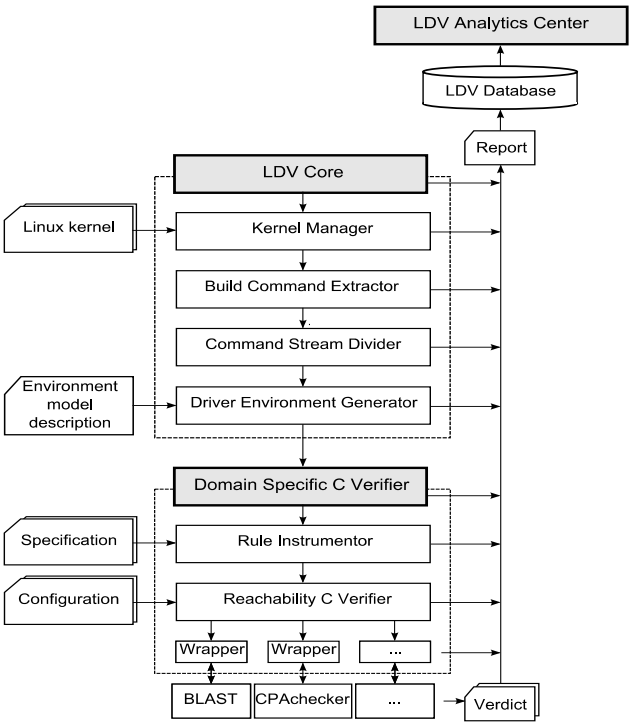

Fig. 2. LDV Tools architecture.

First, we need a static verifier that is able to continue analysis after a first bug is found. CPAchecker is a tool for configurable software verification [7]. It aims at easy integration of new verification components. With revision r8387[4] it has an option "find and report multiple specification violations". Also it already has been integrated with LDV Tools [6]. That is why it was decided to use CPAchecker as the static verifier for finding all bugs in a given Linux kernel module against a specified rule specification in LDV Tools.

1 http://linuxtesting.org/results/report?num=L0029
2 http://linuxtesting.org/results/report?num=L0034
3 http://linuxtesting.org/results/report?num=L0030
4 https://svn.sosy-lab.org/software/cpachecker/trunk/

Second, some components of LDV Tools were improved so that they can process multiple traces (formal definitions will be given in Section II) from CPAchecker. The CPAchecker wrapper was extended to get multiple traces from the static verifier and to send them to Reachability C Verifier. Reachability C Verifier was improved to put all traces into a final report. Finally, LDV Uploader, which uploads analysis results into a database, was extended for uploading several traces for a given module.

After that we conducted experiments. At first LDV Tools with CPAchecker were checked on known issues (L0029, L0034 and L0030). 2 bugs were found for L0029/L0034 and 3 bugs were found for L0030 – as expected. More thorough evaluation of LDV Tools with CPAchecker was made on known bugs in Linux kernel 3.12-rc1 (in total 15 modules and 4 rule specifications). Then a main issue of this approach was revealed – the total number of found traces was 1998 for only 15 modules. Manual examination revealed only 23 different bugs (see Table 1), a lot of traces were similar.

In order to find several bugs in Linux kernel modules in practice the number of found traces should be minimized, but we should pay much attention to keep those traces that correspond to different bugs.

Next section gives some formal definitions. Section III describes comparison algorithms for traces. In section IV semi-automated approach was suggested to further improve comparison algorithms. In Section V the results are presented.

II. FORMAL DEFINITIONS

Below some formal definitions are given.

Static verifier trace – is a sequence of operations (assignments, function calls, assumptions, etc.) in program source files that leads from a specific entry point to a specific label. By default entry point *main* and label *ERROR* are used. Different static verifiers use different formats for their traces, so in LDV Tools all such traces are converted into the common format [8]. There are 4 types of nodes which can be used in traces in the common format:

- *CALL* – call of a specified function;
- *RETURN* – return from the last called but not yet returned function;
- *ASSUME* – choosing a specified branch in conditional clauses;
- *BLOCK* – contains assignments and some auxiliary operators (like *goto)*.

There is also information on a line number, on a source file and formal arguments (for function calls) for each operation. For example, the static verifier trace in the common format for the program in Fig. 1:

```
8  "test.c" CALL      : main()
10 "test.c" CALL 'n' : getnchar(-1)
3  "test.c" ASSUME    : (n < 0)
4  "test.c" BLOCK     : goto ERROR
```

Function call tree – a tree corresponding to function calls in a static verifier trace. Hereafter we suppose that all function

calls in function call trees are ordered. It can be obtained by removing *ASSUME* and *BLOCK* nodes from static verifier trace.

LDV model functions – functions in rule specifications, which contains the main logic of correctness rules [9].

Bug – a reason which causes a violation of a specified correctness rule. For example, it is "lack of mutex release in probe function". Fixing of a bug is usually represented as a patch.

Two or more static verifier traces are called **equivalent** if they correspond to the same bug. In other words, for all equivalent static verifier traces a reason of a correctness rule violation is the same.

Therefore all static verifier traces can be divided into equivalence classes. Each equivalence class contains only equivalent static verifier traces, i.e. corresponding to the same bug. After such dividing it will be necessary to manually analyze only one static verifier trace from each equivalence class. In this case only one static verifier trace for each bug will be in the final report and their number can be reduced without missing any bugs.

III. STATIC VERIFIER TRACE COMPARISON ALGORITHMS

In order to divide all static verifier traces into equivalence classes we propose to use special algorithms for their comparison. Ideally results of theses algorithms should satisfy the definition of equivalent traces. But in practice it is often impossible to automatically understand where is a reason of a correctness rule violation in a static verifier trace.

Static verifier trace comparison algorithms should somehow reduce the number of static verifier traces, but do not remove traces that correspond to different bugs. They can take into account specifics of Linux kernel modules and correctness rules. For example, the main logic of correctness rules is in LDV model functions.

All suggested below static verifier trace comparison algorithms are not absolutely correct in terms of the given definition but in practice it is expected that the number of static verifier traces corresponding to different bugs, which were called equivalent by an algorithm, to be minimal (or even zero) because of specifics of Linux kernel modules and LDV Tools rule specifications.

A. Function call tree comparison algorithm

The first algorithm is aimed to not consider *BLOCKs* and *ASSUMEs* in static verifier traces while comparing these traces. *BLOCKs* usually are used for assignments which can not lead to the *ERROR* label unless they are inside a model function. If *ASSUME* adds some new branch in a static verifier trace it will be revealed in function calls in that branch if so. So only *CALLs* and *RETURNs* from static verifier traces are considered by this algorithm.

Function call tree comparison algorithm: static verifier traces are considered equivalent if their function call trees are equal.

For example, next two static verifier traces for the same program will be equivalent:

```
CALL   : function_1()
ASSUME : (x != 1)
BLOCK  : x = 0;
CALL   : function_2()
BLOCK  : y = 1;
RETURN
```

and

```
CALL : function_1()
ASSUME : (x == 1)
CALL : function_2()
BLOCK  : y = 1;
RETURN
```

This algorithm was implemented as a function in the Reachability C Verifier component. The CPAchecker wrapper gets all found static verifier traces and sends them to Reachability C Verifier as before. But before adding them into the final report Reachability C Verifier will call the function implementing the given comparison algorithm, if the specific option was specified in launching LDV Tools. This function will filter received traces and will return only the first static verifier trace from each equivalence class. After that, those traces will be added into the final report.

Unfortunately this algorithm is far from optimal since there is still a lot of equivalent static verifier traces from different equivalence classes. In experiments 1998 static verifier traces from 15 Linux kernel modules were divided into 1812 equivalence classes while the ideal result is 23.

In general case this algorithm can add static verifier traces, that correspond to different bugs, into the same equivalence class. For example, assignments and assumes outside function calls can lead to different bugs (Fig. 3).

```
1:  void func(int arg)
2:  {
3:    if (arg == 0)
4:      mutex_lock(&mutex);
5:    elsif (arg == 1)
6:      ...// no function calls <- 1st bug
7:    else
8:      ...// no function calls <- 2nd bug
9:    mutex_unlock(&mutex);
10: }
```

Fig. 3. Example of incorrect operation of the function call tree comparison algorithm.

There are two different static verifier traces:

```
CALL   : func
ASSUME : arg == 1
CALL   : mutex_unlock
```

and

```
CALL    : func
ASSUME  : arg != 0
ASSUME  : arg != 1
CALL    : mutex_unlock
```

which will belong to the same equivalence class.

B. Model functions comparison algorithm

Since the main logic of rule specifications is in LDV model functions, the previous algorithm was improved to shrink function call trees so that each their leaf is a model function call.

Model functions comparison algorithm: static verifier traces are considered equivalent if they have equal model function call trees, in which any subtree have a model function call.

Thus in order to compare two static verifier traces with this algorithm the following steps are required:

1) Find function call trees for both static verifier traces (like in the previous algorithm).
2) Determine all model functions, that were called in a module under analysis (all of them have specific comments "*LDV_COMMENT_MODEL_FUNCTION_DEFINITION*").
3) Delete from both trees all subtrees, that does not have any model functions calls at all.
4) Compare resulting model function trees.

For example, next two parts of static verifier traces will be equivalent (functions *f()*, *g()* and *h()* do not have any calls of model functions in their function call trees):

```
CALL    : f()
RETURN
CALL    : g()
CALL    : h()
RETURN
CALL    : ldv_func()
RETURN
RETURN
```

and

```
CALL    : g()
CALL    : ldv_func()
RETURN
RETURN
```

This algorithm was implemented as a function in Reachability C Verifier component similar to the function call tree comparison algorithm.

In practice this algorithm showed much better results – 1998 static verifier traces were divided into 482 equivalence classes (Table 1). For some modules (like *drivers/usb/misc/ftdi-elan.ko*) this algorithm made significant improvement – 947 static verifier traces were divided into 47 equivalence classes. But there are still cases (like *drivers/input/tablet/gtco.ko*), where this algorithm could not reduce the number of static verifier traces at all.

C. Probe comparison algorithm.

In the previous algorithms only types of nodes from static verifier traces and LDV model functions of specified rule specifications were used. But also there are some specific functions in Linux kernel modules, that also can be used in order to compare static verifier traces. Examples, where the previous algorithms failed, may provide information on what can be helpful.

More detailed analysis revealed that probe functions of modules (these functions initialize new attached devices for instances) can be called any number of times. So if this function has a bug, which will be revealed lately (e.g. memory leak), it can be called any number of times before this bug finally will be found (Fig. 4).

```
1: probe(); // probe failed
2: // ... any number of failed probe calls
3: probe(); // probe failed
4: ldv_check_final_state(); // bug will be
found
```

Fig. 4. One bug in the probe function can cause a lot of different static verifier traces.

Thus we have a lot of different static verifier traces that actually correspond to the same bug. This issue should be resolved in order to reduce the number of equivalent static verifier traces.

Probe comparison algorithm: static verifier traces are considered equivalent if they are equivalent by the model functions algorithm and all sequences of calls of a probe function with the same name are considered as one function call.

Theoretically there can be static verifier traces, that are not equivalent, but were called equivalent by this algorithm. But in practice, there were no such cases so far.

The main idea of this algorithm is the following. If there was a bug in some function, there is no need to call that function again and again, a first trace already corresponds to that bug. That is why it is enough to call such the function only once. And it turned out that probe functions were often the reason of such problems.

In practice this algorithm showed the best results – 1998 static verifier traces were divided into 64 equivalence classes (Table 1). For some modules (like *drivers/input-/tablet/gtco.ko, drivers/media/rc/imon.ko, drivers/staging/media/lirc/lirc_sasem.ko* etc.) the result was ideal. Only in one case there was still many static verifier traces – 35 for *drivers/usb/misc/ftdi-elan.ko*. Nevertheless it is much more easy to analyze 64 static verifier traces than 482.

It is unlikely that this algorithm will be used in practice together with the previous algorithms in the future, because it is rather specific. But for a given research it has helped to significantly reduce the number of equivalent static verifier traces and has made possible to analyze them. This revealed more common problems that can not be solved simply with comparison algorithms.

IV. SEMI-AUTOMATED APPROACH

In order to find a better solution to minimize the number of static verifier traces more thorough analysis of probe comparison algorithm results was made.

A. Probe comparison algorithm results analysis.

For 9 modules eventually results are ideal (see Table 1). 2 modules among them contains more than one bug (*drivers/media/rc/imon.ko* and *drivers/media/rc/imon.ko*). For *drivers/media/rc/imon.ko* there were 3 equivalence classes. There are 3 paths in function *imon_probe* that lead to missing of put after get. For *drivers/net/wireless/ath/carl9170/carl9170.ko* there were 7 equivalence classes which correspond to 7 different bugs (*rcu* calls inside critical section).

For *drivers/staging/media/as102/dvb-as102.ko* additionally found static verifier trace is actually a false alarm because of an incorrect environment model (release was called without probe). So this problem is beyond static verifier trace comparison algorithms.

Module *drivers/usb/misc/ftdi-elan.ko* introduces a new problem. There is one bug, that was revealed later in different places which becomes different static verifiers traces. The same problem is in module *drivers/media/usb/pvrusb2/pvrusb2.ko*. This problem is the general case of an issue with probe functions considered earlier.

Module *drivers/net/tun.ko* contains 1 bug – *hlist_add_head_rcu* call inside critical section. But it also contains 4 different traces to the given bug. The same problem (different paths to the bug) is in 2 other modules. In module *net/rxrpc/af-rxrpc.ko* an additional path with lock-unlock was found. In both cases bugs were the same. In module *drivers/net/wireless/rtl818x/rtl8187/rtl8187.ko* a second trace consists of a correct and incorrect calls of function probe. Sometimes other paths to a bug can be helpful for understanding and for fixing the bug, but we consider them excessive.

B. Revealed problems.

The distribution of reasons of diversities from ideal results on 15 Linux kernel modules for probe comparison algorithm is presented in Fig. 5.

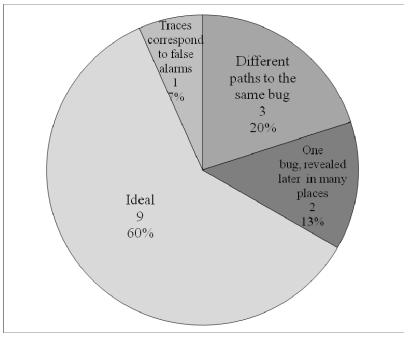

Fig. 5. The distribution of reasons of diversities from ideal results on 15 Linux kernel modules.

Static verifier trace comparison algorithms do not aim to deal with false alarms, so we are not going to consider this problem in the given paper. But other problems ('one bug revealed later in different places' and 'different paths to the same bug') should be resolved. The probe algorithm has solved the problem when one bug was revealed later in many places for one special case. But in the general case this problem can not be solved so easily.

For example, there is one bug in function probe (a lock is acquired but not released), that could be revealed in different places later (Fig. 6). As one can see there is no sequences of calls of failing function probe – they were assigned to the same equivalence class by the probe comparison algorithm. In examples below *lock* acquires locks and *unlock* releases locks.

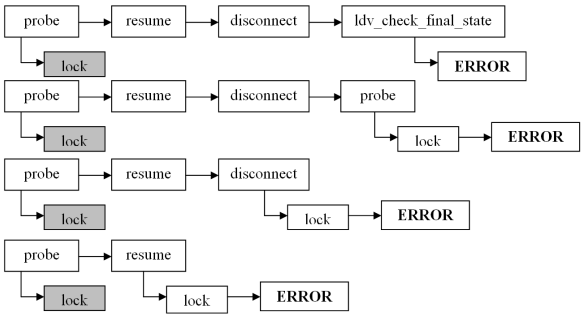

Fig. 6. 4 static verifier traces that correspond to the same bug.

At the same time there could be a situation with different bugs and different static verifier traces. For example, there are 2 bugs (release of an unacquired lock) in 2 different functions (Fig. 7).

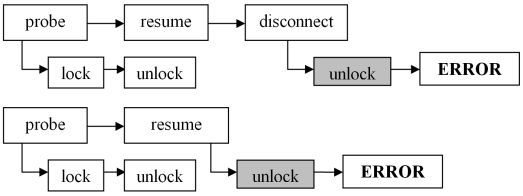

Fig. 7. 2 static verifier traces that correspond to different bugs.

C. Suggested approach.

To solve the problems presented above we suggest a semi-automated approach:

1) When all static verifier traces are added into the final report after some comparison algorithm was applied (for example, the probe comparison algorithm)

2) Then an expert marks up a some bug in LDV Analytics Center.

3) After that a special script finds all static verifier traces, that are equivalent to the trace corresponding to the bug, and excludes them from further analysis.

This approach was implemented as a script for LDV Tools Knowledge Base. The expert can mark up specified place in the static verifier trace (for example, function with a bug) and

use this script. The script will automatically mark all static verifier traces, that contains the same bug, among all traces relevant a given module. After that the expert can see in LDV Analytics Center which static verifier traces were considered to have the same bug and ignore them.

This approach has helped to resolve both problems. For 4 modules (*net/rxrpc/af-rxrpc.ko*, *drivers/media/usb/pvrusb2/pvrusb2.ko*, *drivers/net/wireless/rtl818x/rtl8187/rtl8187.ko* and *drivers/net/tun.ko*) only one bug was found as in ideal results (Table 1). For module *drivers/usb/misc/ftdi-elan.ko* 31 static verifier traces were marked as corresponding to the same bug, one trace is appeared to be a false alarm because of an incorrect environment model (same as above) and 3 traces are appeared to be false alarm because of pointer analysis in CPAchecker 1.2, which was used for conducting experiments (this problem was resolved in CPAchecker 1.3.4).

So, the number of static verifier traces was reduced to ideal results (if not consider false alarms) by means of the probe comparison algorithm and the semi-automated approach.

V. Results

The results of all experiments are presented in Table 1. In experiments LDV Tools rule specifications 39_7a[5], 106_1a[6], 132_1a[7] and 147_1a[8] were used.

Linux kernel module	RS	FE	Tree	MF	Probe	SAA	Manual
net/rxrpc/af-rxrpc.ko	39_7a	6	5	2	2	1	1
drivers/media/usb/pvrusb2/pvrusb2.ko	106_1a	3	3	2	2	1	1
drivers/input/tablet/gtco.ko	132_1a	65	65	65	1	1	1
drivers/isdn/gigaset/bas_gigaset.ko	132_1a	17	17	2	1	1	1
drivers/media/rc/imon.ko	132_1a	136	76	15	3	3	3
drivers/staging/gdm724x/gdmulte.ko	132_1a	4	4	4	1	1	1
drivers/staging/gdm72xx/gdmwm.ko	132_1a	6	6	5	1	1	1
drivers/staging/media/as102/dvb-as102.ko	132_1a	2	2	2	2	2	1
drivers/staging/media/lirc/lirc_imon.ko	132_1a	394	278	15	1	1	1
drivers/staging/media/lirc/lirc_sasem.ko	132_1a	234	234	234	1	1	1
drivers/net/wireless/rtl818x/rtl8187/rtl8187.ko	132_1a	2	2	2	2	1	1
drivers/usb/misc/ftdi-elan.ko	132_1a	947	938	47	35	5	1
drivers/usb/wusbcore/wusb-cbaf.ko	132_1a	154	154	76	1	1	1
drivers/net/tun.ko	147_1a	17	17	4	4	1	1
drivers/net/wireless/ath/carl9170/carl9170.ko	147_1a	11	11	7	7	7	7
Total – 15 modules	4	1998	1812	482	64	28	23

Table 1. Summary table
('RS' – rule specification, 'FE' – first experiment,
'Tree' – function call tree comparison algorithm,
'MF' – model function comparison algorithm,
'Probe' – probe comparison algorithm,
'SAA' – semi-automated approach,
'Manual' – manual examination).

The suggested approach has helped to find 8 new bugs in Linux kernel 3.12-rc1 modules *drivers/media/rc/imon.ko* and *drivers/media/rc/imon.ko.*

5 http://forge.ispras.ru/issues/867
6 http://forge.ispras.ru/issues/2742
7 http://forge.ispras.ru/issues/3306
8 http://forge.ispras.ru/issues/3832

VI. Conclusion

The suggested approach provides means to reduce the number of static verifier traces by dividing them into equivalence classes. Proposed algorithms for static verifier trace comparison show acceptable results and they allow to reach ideal results with not big efforts of the experts. This approach makes possible to analyze all bugs in a given Linux kernel module against a specified rule specification found by LDV Tools and CPAchecker static verifier.

The suggested approach should also be tested on known false positives to make sure that it also works as expected. In future this approach will be extended to check several rule specifications at once, that is very promising area of research. This will further reduce time for finding bugs in Linux kernel modules.

The suggested approach can be applied in other areas outside of Linux kernel modules and LDV Tools.

References

[1] Corbet J., Kroah-Hartman G., McPherson A. *Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*. http://go.linuxfoundation.org/who-writes-linux-2012, 2012.

[2] Beyer D. Petrenko A. *Linux Driver Verification.* In Proc. Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies, LNCS, vol. 7610, pp. 1-6, 2012. doi: 10.1007/s10009-007-0044-z.

[3] Chou A., Yang J., Chelf B., Hallem S., Engler D. *An Empirical Study of Operating System Errors*. In Proc. 18th ACM Symposium on Operating Systems Principles (SOSP), pp. 73-88, 2001. doi: 10.1145/502034.502042.

[4] 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).

[5] Bugs found in Linux kernel modules with help of LDV Tools. http://linuxtesting.org/results/ldv.

[6] Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. *Towards an Open Framework for C Verification Tools Benchmarking*. In Proc. Perspectives of Systems Informatics (PSI), LNCS, vol 7162, pp. 82-91, 2012. doi: 10.1007/978-3-642-29709-0_17.

[7] Beyer D., Keremoglu M.E. *CPAchecker: A Tool for Configurable Software Verification*. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6806, pp. 184–190, 2011. 10.1007/978-3-642-22110-1_16.

[8] Novikov E.M. *Uproshhenie analiza trass oshibok instrumentov staticheskogo analiza koda*. [*Simplification of static verifier traces analysis*]. APSPI, 2011. (in Russian).

[9] Novikov E.M. *Razvitie metoda kontraktnykh spetsifikatsij dlya verifikatsii modulej yadra operatsionnoj sistemy Linux* [*Development of a contract specification method for the verification of Linux kernel modules*]. Dissertatsiya na soiskanie uchenoj stepeni k.f.-m.n. [PhD thesis], 2013 (in Russian).