# Static Verification of Linux Kernel Configurations[1]

[1] *S.V. Kozin <kozyyy@yandex.ru>*
[2] *V.S. Mutilin <mutilin@ispras.ru>*
[1] *National Research University Higher School of Economics, 20 Myasnitskaya Ulitsa, Moscow, 101000, Russia*
[2] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*

**Abstract.** The Linux kernel is often used as a real world case study to demonstrate novel software product line engineering research methods. It is one of the most sophisticated programs nowadays. To provide the most safe experience of building of Linux product line variants it is necessary to analyse Kconfig file as well as source code. Ten of thousands of variable statements and options even by the standards of modern software development. Verification researchers offered lots of solutions for this problem. Standard procedures of code verification are not acceptable here due to time of execution and coverage of all configurations. We offer to check the operating system with special wrapper for tools analyzing built code and configuration file connected with coverage metric. Such a bundle is able to provide efficient tool for calculating all valid configurations for predetermined set of code and Kconfig. Metric can be used for improving existing analysis tools as well as decision of choice the right configuration. Our main goal is to contribute to a better understanding of possible defects and offer fast and safe solution to improve the validity of evaluations based on Linux. This solution will be described as a program with instruction for inner architecture implementation.

**Keywords:** Software Product Lines, Linux, Kconfig, Preprocessor

**DOI:** 10.15514/ISPRAS-2017-29(4)-14

**For citation:** Kozin S.V., Mutilin V.S. Static Verification of Linux Kernel Configurations. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 4, 2017, pp. 217-230. DOI: 10.15514/ISPRAS-2017-29(4)-14

## *1. Introduction*

Nowadays, software is used to solve increasingly important and complex tasks, due to this fact the complexity of software architectures is also constantly growing. With

[1] The work is supported by RFBR grant N16-01-00352





the increasing complexity of programs, the complexity of development, analysis and maintenance arises. There are many methods that allow you to reduce the costs of supporting the software life cycle. One such method is the creation of variable systems (or family of systems, software product families, software product lines). The superiority over the usual development method is that systems are manufactured with the condition of multiple elements used for several systems with a similar set of functions, taking into account a specific target audience of users. At the same time, a complex and widely known representation of variable software is the Linux operating system [1-5].

In the development of variable systems, a variability model and a variation mechanism play a fundamental role. The variability model specifies the space of possible variants of this family of systems. Usually it is determined by a set of features or configuration parameters, by the sets of their possible values and constraints on possible combinations of these values, each variant of the system corresponds to a certain set of values of all features. The variation mechanism provides the ability to build all possible system variants from a limited set of created and followed artifacts. In Linux, the variability model and its relationship to the variation mechanism is built on the basis of Kconfig files, Makefile files and additional scripts. Kconfig describes all possible features, as well as their relationship with each other in a special language. Then on the basis of Kconfig, the configuration file .config is defined, which describes the version of the system. It consists of a set of configuration variables described in Kconfig and values that satisfy the constraints of Kconfig. During kernel assembly, the values of variables specified in .config are passed to the code as constants for the preprocessor and to Makefiles, which will be used in the variation mechanism. Makefiles contain information about objects in kernel: what files are included and which mode of compilation will be used [5].

In the field of operating systems (hereinafter the operating system we mean the kernel and the underlying OS libraries, providing the interfaces for work with computing resources and hardware), a mechanism of conditional compilation of C / C ++ languages is widely used as a mechanism for variability of the mixed type (based on macros #ifdef, #if, else #else). Blocks that are surrounded with variability mechanism macros, are called variable blocks. It allows you to compose code at the build stage that combines various variables specified by a set of characteristic values that are conditional compilation parameters in this case (defined by the #define and #undef macros, as well as preprocessor setup parameters). The expression after macros is called block precondition, if configuration turns it into ‘true’, block gets compiled [5].

The complexity of variability models for modern operating systems is very high, for example, the Linux kernel version 2.6.32 has 6319 characteristics, more than 10,000 constraints that can be used up to 22 individual characteristics, with the majority of characteristics depending on at least 4 others, and the maximum depth of the dependency tree is 8 [6]. This complexity causes a large number of errors, primarily due to the difficulty of taking into account all the factors that a developer of a separate code element should do. To identify and cope with these errors, it is necessary to use

specialized techniques of analysis and verification. Complexity of analysis that is typical for systems with such a variability mechanism arise because of the huge size of the possible variants space (which makes it completely unrealistic to check them all). Due to using of conditional compilation, each fragment does not have to be a separate component with a certain behavior that can be analyzed separately from the rest of the code, usually such fragments are just insertions into the common code, and can only be checked in certain combinations with each other. The need to solve these problems imposes special requirements on the tools and analysis methods used for complex variable operating systems and system software in general. These requirements are specific for analysis and verification - the methods used to create such systems, by themselves, do not facilitate their analysis [7, 8]. The main goal of this work is to propose a method capable of coping with the verification of the Linux OS taking into account the variability, to give acceptable accuracy of verification and speed of execution comparable to the verification of common programs.

An errors analysis of such complex systems as Linux can be done with a lot of different tools. The most convenient of them are static analyzers and static verifiers. Static code analysis is the analysis of software produced (as opposed to dynamic analysis) without real execution of the programs under investigation. Existing static analyzers (such as Coverity [9] or Svace [10]) and static verifiers (such as BLAST, CPAchecker [11]) are only designed to work with the code already compiled. Accurate analysis requires pre-assembling of a specific configuration, and only after that start of the actual analysis. As a result, the total inspection time becomes unacceptably large. There is a class of tools that are focused on analysis of a set of possible configurations, they do not split the phase of building configurations and code analysis. As the example of such tools we can take a look at TypeChef and Undertaker. These tools are designed to solve special problems in the sphere of variability. Undertaker is looking for a "dead code" - such a problems when different configurations give the same product, besides it has a lot of built-in helper modules to provide main task, and one more function - assembling the minimal Linux kernels for individual use cases. TypeChef is looking for linking and compile errors with a variability-aware method. It is important to say, that TypeChef can not find difficult problems in code like complex static analysers. Suggested in this paper tool should analyze code deeply like BLAST or CPAchecker and, on the other hand, should do it in variability-aware way without all-configuration brute forcing. For this task we suggest to use CPAchecker due to its outstanding abilities in the error findings, despite CPAchecker's expenditure of time [11]. The maximum reduction in verification time should be achieved through the optimal selection of configurations that will be directed to the input of the static analyzer.

## 2. Configuration Set Selection

The problem of selecting configurations can be considered as a splitting of the configuration space into equivalence classes. Each of the selected configurations will belong to one of the classes. To split the configuration space into classes, it is





suggested to use the MC/DC test coverage criterion. The advantage of this choice is that it allows you to significantly reduce the number of classes, even for large software systems. In addition, methods of analysis/construction of configurations for preprocessor directives are similar to methods for test data generation for coverage of programs in programming language, where the MC/DC criterion has proven itself well. If we take a look at standard usage of MC/DC for code coverage, we can find out that conditionals in usual programming language has the same structure as conditional compilation directives, the difference is in compilation (conditional compiling directives will not compile code under the block if condition is not true, which is the same as not giving control to code inside usual conditionals.

There are 2 basic notions in MC/DC: *decision* and *condition*. *Decision* is a propositional formula which consists of conditions. If it is true, then control will be given to block with such decision (in the case of preprocessor - code will be compiled). Otherwise, control will not be given to this block (code will not be compiled). *Condition* is a logical part of decision which connects to other conditions with logical functions.

A set is considered to reach 100% coverage by this metric, if:

1. Each decision takes every possible outcome.
2. Each condition in a decision takes every possible outcome.
3. Each condition in a decision is shown to independently affect the outcome of the decision.

In other words, in the full test, in accordance with the MC/DC coverage criterion, it should be demonstrated that every condition that can influence the resulting value of the decision that includes it actually changes its value regardless of the other conditions.

Example:

Some module of Linux has variable block inside. For example, consider a decision: A && B || C; where A, B, C are some boolean constraints of Kconfig.

The decision is applicable for variable block (Fig. 1).

```
#if (A && B || C)
  …
#endif
```

*Fig. 1. Example of variable block*

We can extract the conditions out of the decision: A, B, C.

That means that we have to build such table for this variable block (Table 1).

*Table 1. Truth table for variable block in Fig. 1*

| A | B | C | Decision |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Now we will find those pairs of conditions values where the change of one of them affects on decision. For each condition we have to get only one pair, and then choose minimal amount of them to cover all conditions. This will satisfy third MC/DC point. Pairs will be (Table 2)

*Table 2. MC/DC coverage table*

| A | B | C | Decision | A | B | C |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | | C1 |
| 0 | 0 | 1 | 1 | | | C1 |
| 0 | 1 | 0 | 0 | A1 | | C2 |
| 0 | 1 | 1 | 1 | | | C2 |
| 1 | 0 | 0 | 0 | | B1 | C3 |
| 1 | 0 | 1 | 1 | | | C3 |
| 1 | 1 | 0 | 1 | A1 | B1 | |
| 1 | 1 | 1 | 1 | | | |

According to table we will test only pairs that are marked as A1, B1 and C3 or C2; These 2 are minimal sets for MC/DC coverage.

We can notice that from 8 possible combinations, we can use only 4 to get full coverage according to MC/DC method (0-1-0, 0-1-1,1-0-0,1-1-0).

In general, the MC / DC metric allows 2n different situations to be used instead of $n^2$ condition combinations.[14]

## 3. Kernel Check Stages

For each of the found configurations, you should run a kernel verification. The program will scan the kernel in 6 stages: configuration, search for a "dead code", preprocessing, compiling, linking, searching for run-time errors. It is also worth noting that we will look for not only errors, but also configuration defects. Defects - is a broader concept, and it includes not only system errors, but also possible errors of the kernel without processing the interrupt.





Description of the stages:

A. Search for a “dead code”.
   A "dead" code is such a code in which control is not transferred under any circumstances. This code contributes to a common configuration error when two different configurations produce the same product at the output. This problem can be solved using the built-in tools of the Undertaker.

B. Configurations.
   At the time of build, Linux itself checks the configuration file, but it's worth checking it for recursive dependencies and non-existent variables in the code, but existing in Kconfig (and vice versa).

C. Preprocessing.
   Preprocessing is performed just before compilation and at this stage we will get the code that will be compiled into the final version of the program. Most of errors at this stage can find by a preprocessor. It remains for us to inform the user of a possible conflict of names if we see duplicate names of preprocessor variables and their redefinition, because they have one nominal space and the developer may not notice the problem of overriding.

D. Compilation.
   Compilation is complete on the compiler side. Here there are such errors as: detection of an undeclared variable / function, missing punctuation in the code, etc.

E. Linking.
   As well as compilation is completely redirected to the linker. The linker finds errors in missing libraries or files.

F. Execution Errors.
   The most difficult part of the test is using the LDV and CPAchecker tools. LDV assigns labels to the code of the program according to the preset rules, while CPAchecker searches for them and builds accessibility graphs to them so we can see how a label is achieved.

## 4. Configuration Set Selection

To solve this task we suppose the workflow for a program called OStap (Fig. 2). This program will find all variable blocks, get all the propositional formulas for entering each of the variable blocks. After that program will extract all conditions from those formulas and will use **MC/DC** metric to get all necessary configurations that will be checked with static verifier and dead code detector to perform verification stages A-F.

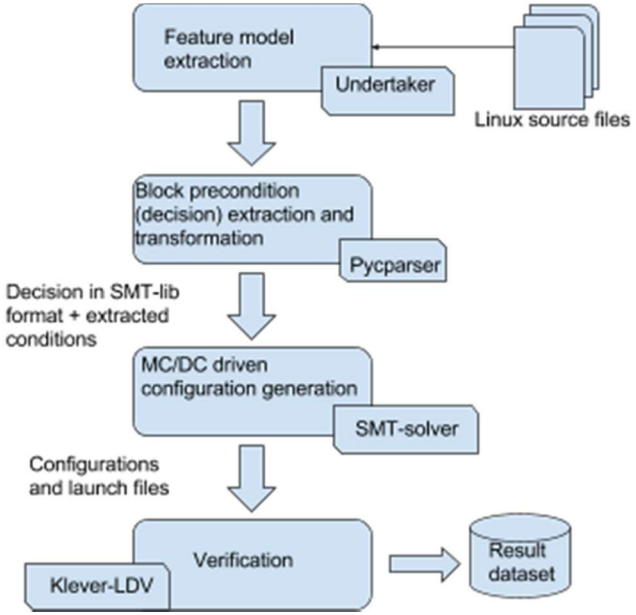


*Fig. 2. OStap workflow.*

## 4.1 Feature model extraction

First of all, we transform Kconfig files using Kconfigdump module of **Undertaker** to get feature model of Linux kernel. Feature model is splitted by architectures sets of formulas. All dependencies implemented in Kconfig file are represented in a dump as a set of logical formulas. So it is possible to get full precondition for each feature just looking at line with it's name in model file. For example: if we have to turn on option A to turn on option B there will be …&&A addition to any decision where config B is have to be turned on.

## 4.2 Block precondition extraction

### 4.2.1. Coarse block precondition extraction

Secondary, we need to get all variable blocks in the given code, which is a module or a set of modules that consists of different .c files. These modules can be compiled according to Makefiles, where modules are marked as compiled, compiled as LKM, non-compiled. This mechanism of variability (Kbuild variability) in Linux is made for modules and helps to organize builds. Due to the fact, that another variability mechanism in Linux uses preprocessors we can find all #ifdef, #if, #else, #ifndef blocks to be sure that we have found all the variable blocks in the given code. This





may be done using standard tools of most of popular operating systems or using programming language tools for work with file system.

When we have all necessary blocks and their positions in the files we can get propositional formulas using **Undertaker**[13] tool for them. As a result of the undertaker infrastructure we can easily calculate preconditions for preprocessor blocks in a file, just by specifying the file and line number. If a configuration model is loaded, it will also fetch all interesting items from the model. Say you want to have the **block precondition** for line 359 and line 370 in init/main.c.

Then for each block we use **Undertaker's** option -blockpc to get blocks precondition based on dumped feature model and code. This precondition is a decision in **MC/DC** metric theory. It is also necessary to say that later we will use **SMT-solver** to work with **MC/DC** metrics, so we patched Undertaker for providing output in a SMT-lib way (Polish notation). C expressions with operands, are not able to be converted due to architecture of undertaker, such code may be marked with special symbols to be changed in the future.

```
$ undertaker -j blockpc init/main.c:359
init/main.c:370
I: Block B20 | Defect: no | Global: 0
B20
&&
( B18 <->  ! CONFIG_SMP )
&& ( B20 <->  ( B18 )  && CONFIG_X86_LOCAL_APIC )
&& ( B22 <->  ( B18 )  && ( ! (B20) )  )
&& ( B25 <-> ( ! (B18) )  )

I: Block B25 | Defect: no | Global: 0
B25
&&
( B18 <->  ! CONFIG_SMP )
&& ( B20 <->  ( B18 )  && CONFIG_X86_LOCAL_APIC )
&& ( B22 <->  ( B18 )  && ( ! (B20) )  )
&& ( B25 <-> ( ! (B18) )  )
```

*Fig. 3. Example of usual Undertaker output without model.*

```
$ undertaker -j blockpc -m
models/x86.model  init/main.c:370
I: loaded rsf model for x86
I: Using x86 as primary model
I: Block B25 | Defect: no | Global: 0
B25
&&
( B18 <->  ! CONFIG_SMP )
&& ( B20 <->  ( B18 )  && CONFIG_X86_LOCAL_APIC )
```

```
&& ( B22 <->  ( B18 )  && ( ! (B20) )  )
&& ( B25 <-> ( ! (B18) )  )

&&
(CONFIG_X86_32 -> ((!CONFIG_64BIT)))
&&
(CONFIG_X86_32_NON_STANDARD -> ((CONFIG_X86_32 &&
CONFIG_SMP && CONFIG_X86_EXTENDED_PLATFORM)))
&&
(CONFIG_X86_64 -> ((CONFIG_64BIT)))
&&
(CONFIG_X86_EXTENDED_PLATFORM -> ((CONFIG_X86_64)
&& (CONFIG_X86_32)))
&&
(CONFIG_X86_LOCAL_APIC -> ((CONFIG_X86_64 ||
CONFIG_SMP || CONFIG_X86_32_NON_STANDARD ||
CONFIG_X86_UP_APIC) && (CONFIG_X86_64 ||
CONFIG_SMP || CONFIG_X86_32_NON_STANDARD ||
CONFIG_X86_UP_APIC)))
&&
(CONFIG_X86_UP_APIC -> ((CONFIG_X86_32 &&
!CONFIG_SMP && !CONFIG_X86_32_NON_STANDARD)))
```

*Fig. 4. Example of usual Undertaker output with model.*

### 4.2.2 Detailed block precondition extraction

When we have formula by **Undertaker**, we have to change all the marked code to prefix view, in other words: to represent decision in SMT-lib way. Due to the fact that such code will be in a usual C representation, we can use any **C parser** that is able to build expression tree to rebuild the string. For example we can use **Pycparser** [15]. **Pycparser** can parse C code and represent it as ast-tree. Ast-tree is an expression tree with all operators of C language. Running through the tree, we can rewrite any C expression from infix to prefix view. To do this thing we need to apply this algorithm:

```
Algorithm prefix (tree)
    if (tree not empty)
          print (tree token)
          prefix (tree left subtree)
          prefix (tree right subtree)
    end if
end prefix
```





### 4.3 MC/DC driven configuration generation

Now we have ready-to-use prefix formula that is the same as 'decision' term in **MC/DC** metrics so it is time to start extracting conditions from a decision. To pass this stage we need to get all the single variables inside unary operator or without unary operator. Those variables will be the same as conditions.

When we have all conditions and decision for a variable block, we need to build a truth table for conditions. For example, we extracted 3 conditions: a > 0, b == true, a < 0, and out decision is (a > 0) && (b == true) || (a < 0). We look over all of their combinations that we got from truth table like it was described in II chapter (Table 3).

*Table 3. Truth table without decision for (a > 0) && (b == true) || (a < 0).*

| Line | A > 0 | B = true | A < 0 | decision |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | |
| 2 | 0 | 0 | 1 | |
| 3 | 0 | 1 | 0 | |
| 4 | 0 | 1 | 1 | |
| 5 | 1 | 0 | 0 | |
| 6 | 1 | 0 | 1 | |
| 7 | 1 | 1 | 0 | |
| 8 | 1 | 1 | 1 | |

Each line of the table reflects set of equalities related to column and value. For our example line 1 means: a > 0 = false, b == true = false, a < 0 == false, whereas line 6 means: a > 0 = true, b == true = false, a < 0 == true;

To calculate decision column we need to put such equalities and full decision formula into **SMT-solver**, so it calculates possible values of variables through equalities and then put it into decision formula to find solution. In our example in case of a > 0 = true and a < 0 = true **SMT-solver** will return error, so these sets of variables will not be used in future (Table 4).

*Table 4. Truth table with decision for (a > 0) && (b == true) || (a < 0).*

| Line | A > 0 | B = true | A < 0 | decision |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 1 |
| 3 | 0 | 1 | 0 | 0 |
| 4 | 0 | 1 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 |
| 6 | 1 | 0 | 1 | Error |
| 7 | 1 | 1 | 0 | 1 |
| 8 | 1 | 1 | 1 | Error |

In classical MC/DC theory we have to say that this example can not be covered by MC/DC due to the fact that 6 and 8 line gave us impossible conditions to resolve, but in a real case adaptation we can say that we covered all of possible conditions [14]. When the table is ready, we can find influencing variables, like it was described in chapter II (Table 5).

*Table 5. MC/DC coverage table for (a > 0) && (b == true) || (a < 0).*

| Line | A > 0 | B = true | A < 0 | decision | A > 0 | B ==true | A < 0 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | | | 1 |
| 2 | 0 | 0 | 1 | 1 | | | 1 |
| 3 | 0 | 1 | 0 | 0 | | | 2 |
| 4 | 0 | 1 | 1 | 1 | | | 2 |
| 5 | 1 | 0 | 0 | 0 | | 3 | |
| 6 | 1 | 0 | 1 | Error | | | |
| 7 | 1 | 1 | 0 | 1 | | 3 | |
| 8 | 1 | 1 | 1 | Error | | | |

For MC/DC coverage we need A/B and C pairs to check. That means that we have to extract a and b values related to those lines. These values form configurations that we will check with help of verifier. Each line for each configuration.

### 4.4 Verification

Finally, we have got necessary configurations to get 100% coverage using **MC/DC** metrics. Now we will push these configurations to static verifier (for example, **CPAchecker**). Also we will check code with **Undertaker** tool to find dead code blocks (Stage A of kernel check), which is also can be declared as configuration error.

Next step for pushing configurations is to create launch file for **LDV-Klever** tool and launch it. Launch file is filled with modules-to-check and rules for finding unsafe modules [16]. Before the launch, code will be compiled and stages B-E will be passed with compiler default methods. After that **LDV-Klever** main activity will be invoked to check code for a runtime defects (stage F). Output is a module with result: safe, unsafe or unknown. If module is unsafe, verifier shows error trace for this module.

During verifier launches we push results into the dataset where the structure unite modules, that we are checking, each module is splitted with configurations and results of **LDV-Klever**, **LDV-Klever** results are splitted with verdict and unsafe traces.

## *5. Conclusion*

This article describes the method, which allows you to check the Linux operating system for errors without being depended on the configuration. This approach provides high code coverage and an improved speed of verification comparing to





brute force method. The prototype of this program is already implemented. It will be finished and fixed in a nearest time.

In future work MC/DC will be better adapted to current aim(Linux kernel), also project will be tested in a real work during ISP RAN researches. There will be also replacement in used tools, probably or rewriting them.

This software product can be used in the production of distributions, as well as for verification of existing ones. As a result we will get linear depended amount of configurations for full testing of Linux systems.

## References


[1]. Jacobson I., Griss M., Jonsson P. Software Reuse, Architecture, Process and Organization for Business Success. Addison-Wesley, 1997.

[2]. Bosch J. Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach. Pearson Education, 2000.

[3]. Clements P., Northrop L. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering, Addison-Wesley, 2001.

[4]. Pohl K., Böckle G., van der Linden F. J. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, 2005. DOI: 10.1007/3-540-28901-1.

[5]. Kuliamin V.V., Lavrischeva E.M., Mutilin V.S., Petrenko A.K. Verification and analysis of variable operating systems. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 3, 2016, pp. 189-208 (in Russian). DOI: 10.15514/ISPRAS-2016-28(3)-12

[6]. Lotufo R., She S., Berger T., Czarnecki K., Wąsowski A. Evolution of the Linux kernel variability model. Proc. of SPLC'10, LNCS 6287:136-150, Springer, 2010. DOI: 10.1007/978-3-642-15579-6_10.

[7]. Lavrischeva E.M., Koval G.I., Slabospickaya O.O., Kolesnik A.L. Features of management processes when creating families of software systems [Osobennosti processov upravleniya pri sozdanii semejstv programmnyh system]. Problems of programming [Problemy programmirovaniya], 3:40-49, 2009 (in Russian).

[8]. Lavrischeva E.M., Koval G.I., Slabospickaya O.O., Kolesnik A.L. Teoreticheskie aspekty upravleniya variabel'nost'yu v semejstvah programmnyh sistem. Bulletin of KSU, a series of physics and mathematics [Vesnik KNU, seriya fiz.–mat. nauk], 1:151-158, 2011 (in Russian).

[9]. Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, Dawson Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World", Communications of the ACM, Vol. 53 No. 2, pp. 66-75

[10]. Borodin A.E., Belevancev A.A, A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels, Trudy ISP RAN/Proc. ISP RAS, vol 27, issue. 6, 2015, pp. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8 (in Russian).

[11]. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, "The software model checker BLAST", Int J Softw Tools Technol Transfer (2007) 9:505–525, Springer-Verlag 2007

[12]. Andy Kenner, Christian Kastner, SteffenHaase, Thomas Leich, "TypeChef: toward type checking #ifdef variability in C". Proceeding FOSD '10 Proceedings of the 2nd Internationsal Workchop on Feature-Oriented Software Development, pp. 25-32, Eindhoven, The Netherlands, Oct. 10, 2010.

[13]. Stephan Henglein. Vampyr configurability aware compile testing of source files. Linux Plumber Conference, Oct 15-17, 2014, Dusseldorf, Germany. Available at:

http://www.linuxplumbersconf.net/2014/ocw//system/presentations/2313/original/hengelein.pdf, accessed 12.01.2017.
[14]. Kulyamin V., Model-based testing [Testirovanie na osnove modeley]. (online publication). Available at: http://mbt-course.narod.ru/Lecture03.pdf, accessed 12.02.2017 (in Russian).
[15]. Alber Zever. Pycparcer wiki. (Onine publication). Available at: https://pypi.python.org/pypi/pycparser/2.14. accessed 7.05.2017.
[16]. I.S. Zaharov, M.U. Mandrykin, V.S. Mutilin, E.M. Novikov, A.K. Petrenko, A.V. Khoroshilov. Configurable Toolset for Static Verification of Operating Systems Kernel Modules. Trudy ISP RAN/Proc. ISP RAS, vol 26, issue 2, 2014, pp. 5-42 (in Russian). DOI: 10.15514/ISPRAS-2014-26(2)-1.

## Статическая верификация конфигураций ядра Linux[2]

*[1] С.В. Козин <kozyyy@yandex.ru>*
*[2] В.С. Мутилин <mutilin@ispras.ru>*
*[1] Национальный исследовательский университет Высшая Школа Экономики, 101000, Россия, г. Москва, ул. Мясницкая, д. 20.*
*[2] Институт системного программирования РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*

**Аннотация.** Ядро операционной системы Linux – это частый пример современных инженерных решений в области создания продуктовых линеек программного обеспечения. Сегодня это одна из наиболее сложных программных систем. Для того, чтобы обеспечить наиболее безопасное построение вариантов продуктовой линейки, необходимо анализировать конфигурационный файл Kconfig помимо исходного кода. Ядро содержит десять тысяч вариабельных переменных несмотря на современную инженерию. Исследователи в области верификации предлагают большое количество решения проблемы анализа. Стандартные процедуры верификации здесь не могут быть применены из-за времени проверки покрытия всех конфигураций. Мы предлагаем инструмент, который базируется на связи уже существующих программах для проверки кода и конфигурационного файла с метрикой покрытия. Такой пакет – это эффективный инструмент для расчета всех допустимых конфигураций для предопределенного набора кода и Kconfig. Предложенные методы могут быть использованы для улучшения существующих инструментов анализа, а также для выбора правильной конфигурации. Наша основная цель – лучше разобраться в возможных дефектах и предложить быстрое и безопасное решение для проверки ядра Linux. Это решение будет описано как программа с инструкцией по реализации внутренней архитектуры.

**Ключевые слова:** линейка программных продуктов, Linux, Kconfig, препроцессор

**DOI:** 10.15514/ISPRAS-2017-29(4)-14

**Для цитирования:** Козин С.В., Мутилин В.С. Статическая верификация конфигураций ядра Linux. *Труды ИСП РАН*, том 29, вып. 4, 2017 г., стр. 217-230 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(4)-14

[2] Работа поддержана грантом Российского фонда фундаментальных исследований №16-01-00352





## Список литературы

[1]. Jacobson I., Griss M., Jonsson P. Software Reuse, Architecture, Process and Organization for Business Success. Addison-Wesley, 1997.
[2]. Bosch J. Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach. Pearson Education, 2000.
[3]. Clements P., Northrop L. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering, Addison-Wesley, 2001.
[4]. Pohl K., Böckle G., van der Linden F. J. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, 2005. DOI: 10.1007/3-540-28901-1.
[5]. В.В. Кулямин, Е.М. Лаврищева, В.С. Мутилин, А.К. Петренко. “Верификация и анализ вариабельных операционных систем” Труды ИСП РАН, том 28, вып. 3, 2016, стр. 189-208. DOI: 10.15514/ISPRAS-2016-28(3)-12
[6]. Lotufo R., She S., Berger T., Czarnecki K., Wąsowski A. Evolution of the Linux kernel variability model. Proc. of SPLC’10, LNCS 6287:136-150, Springer, 2010. DOI: 10.1007/978-3-642-15579-6_10.
[7]. Лаврищева К.М., Коваль Г.И., Слабоспицкая О.О., Колесник А.Л. Особенности процессов управления при создании семейств программных систем. Проблемы программирования, 3:40-49, 2009.
[8]. Лаврищева К.М., Слабоспицкий А.А., Коваль Г.И., Колесник А.А. Теоретические аспекты управления вариабельностью в семействах программных систем. Вестник КНУ, серия физ.–мат. наук, 1:151-158, 2011.
[9]. Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, Dawson Engler, “A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World”, Communications of the ACM, Vol. 53 No. 2, pp. 66-75
[10]. Бородин А.Е., Белеванцев А.А., “Статический анализатор Svace как коллекция анализаторов разных уровней сложности” Труды ИСП РАН, том 27, вып. 6, 2015, стр. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8
[11]. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, “The software model checker BLAST”, Int J Softw Tools Technol Transfer (2007) 9:505–525, Springer-Verlag 2007
[12]. Andy Kenner, Christian Kastner, SteffenHaase, Thomas Leich, “TypeChef: toward type checking #ifdef variability in C”. Proceeding FOSD ’10 Proceedings of the 2nd Internationsal Workchop on Feature-Oriented Software Development, pp. 25-32, Eindhoven, The Netherlands, Oct. 10, 2010.
[13]. Stephan Henglein. Vampyr configurability aware compile testing of source files. Linux Plumber Conference, Oct 15-17, 2014, Dusseldorf, Germany. Available at: http://www.linuxplumbersconf.net/2014/ocw//system/presentations/2313/original/hengelein.pdf, дата обращения 12.01.2017.
[14]. Кулямин В., Тестирование на основе моделей. (online publication). http://mbt-course.narod.ru/Lecture03.pdf, дата обращения 12.02.2017.
[15]. Alber Zever. Pycparcer wiki. (Onine publication). Available at: https://pypi.python.org/pypi/pycparser/2.14. дата обращения 7.05.2017.
[16]. И.С. Захаров, М.У. Мандрыкин, В.С. Мутилин, Е.М. Новиков, А.К. Петренко, А.В. Хорошилов. Конфигурируемая система статической верификации модулей ядра операционных систем. Труды ИСП РАН, том 26, вып. 2, 2014, стр. 5-42. DOI: 10.15514/ISPRAS-2014-26(2)-1.