ISSN 0361-7688, Programming and Computer Software, 2023, Vol. 49, No. 7, pp. 11–17. © Pleiades Publishing, Ltd., 2023.

Runtime Verification of Operating Systems Based on Abstract Models

D. V. Efremov[a,*] (ORCID: 0000-0002-9916-056X), **V. V. Kopach**[a,**] (ORCID: 0000-0003-2461-9986), **E. V. Kornykhin**[a,b,***] (ORCID: 0000-0001-9303-3132), **V. V. Kuliamin**[a,b,c,****] (ORCID: 0000-0003-3439-9534), **A. K. Petrenko**[a,b,c,*****] (ORCID: 0000-0001-7411-3831), **A. V. Khoroshilov**[a,b,c,d,******] (ORCID: 0000-0002-6512-4632), and **I. V. Shchepetkov**[a,c,*******] (ORCID: 0000-0002-5794-004X)

[a] *Ivannikov Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia*

[b] *Moscow State University, Moscow, 119991 Russia*

[c] *National Research University Higher School of Economics, ul. Myasnitskaya 20, Moscow, 101978 Russia*

[d] *Moscow Institute of Physics and Technology, Institutskii per. 9, Dolgoprudnyi, Moscow oblast, 141700 Russia*

**e-mail: efremov@ispras.ru*

***e-mail: vkopach@ispras.ru*

****e-mail: kornevgen@ispras.ru*

*****e-mail: kuliamin@ispras.ru*

******e-mail: petrenko@ispras.ru*

*******e-mail: khorochilov@ispras.ru*

********e-mail: shchepetkov@ispras.ru*

Received January 10, 2023; revised January 18, 2023; accepted February 13, 2023

AbstractHigh complexity of a modern operating system (OS) implies the use of complex models and high-level specification languages to describe even individual aspects of OS functionality, e.g., security functions. The problem of verifying the conformity between an OS and these models requires the establishment of a rather complex relationship between elements of OS implementation and elements of the model. In this paper, we present a method to establish and support this relationship, which can be effectively used in OS testing and runtime verification/monitoring. This method was successfully used in testing and monitoring the conformity between Linux kernel components and Event-B models.

Keywords: runtime verification, Linux kernel, LSM, IMA/EVM, Event-B, and ProB

DOI: 10.1134/S0361768823070034

1. INTRODUCTION

Complexity of a modern operating system (OS) implies the support of verification and validation methods used in the process of its development. A promising approach that can deal with this increasing complexity is the use of abstract models, which do not take into account low-level and OS-specific details of OS implementations. In addition, it is reasonable to separate individual aspects of OS functionality, security, performance, etc. into different levels or modules of models. When combined, these methods significantly facilitate the formulation and verification of requirements, as well as the application of various analysis and proof tools.

In practice, an accurate and complete description of various functional aspects of a modern industrial OS, e.g., access control, requires complex models developed using sufficiently expressive languages. We successfully used Event-B [1] in several projects to describe and formally verify access control models, including implementations of multilevel security mechanisms, for Linux-family OSs [2, 3]. The next step is to check the conformity between the developed models and the real-world behavior of the OS in order to identify possible discrepancies and inconsistencies.

Taking into account numerous difficulties associated with the effective use of deductive verification approaches when working with OS kernel code, it seems reasonable to employ testing and dynamic verification methods for this purpose [4]. This requires the development of a testing and monitoring method with the use of a relatively large Event-B model as an oracle. This paper considers the application of formal models to the monitoring and dynamic verification of OS behavior, as well as describes key aspects of the proposed method.

2. EVENT-B MODELS

Event-B [1] is a formal method based on set theory and predicate logic.

It has a simple notation and is supported by Rodin [5], a platform for model development and verification. A model in the Event-B language is a *discrete system of transitions* in which the components of a modeled system are described in terms of *states*, transitions between states (which are called *events*), and properties that must hold in each state (*invariants*). Models can be decomposed into individual modules (which are called levels) by *stepwise refinement* [6].

Rodin can automatically generate *verification conditions:* statements that must be proven to verify the correctness of a model. Rodin also provides tools for their automated and interactive proof. The ProB animator and model checker is available [7] as an extension for Rodin and as an independent tool.

We use Event-B to develop access control models for the Linux kernel. These models can be rather abstract (in that case, they are called *security policy* models) or they can be more detailed and closer to OS implementation (these models are called *functional specifications*). We usually construct models of both types for each OS we work with. This facilitates the analysis of security properties and the proof of their consistency while retaining the possibility of demonstrating the conformity between models and code (implementation). In [8], we described the process of working with several models of different levels of abstraction, including the proof of their conformity.

Functional specifications are low-level ones and describe *system calls* of the Linux kernel. We propose to use formal specifications as oracles for testing in order to verify that these system calls do not violate security policies.

3. ACCESS CONTROL IN LINUX

The Linux kernel, beginning with version 2.5 (2002) [9], includes the Linux Security Modules (LSM) framework. This subsystem is a set of modules that enhance the security of the kernel and implement access control management. The interaction between the modules and the kernel is organized using the LSM API.

This API makes it possible to implement additional access checks that improve existing ones. It is not designed to change the state of the kernel, outside of the fields specifically reserved for this API in kernel data structures.

Thus, the side effect of the security modules that correctly use this API from an architectural perspective is reduced only to the early termination of processing of some event in the kernel. The refinement mechanism in Event-B works in exactly the same way.

Currently (Linux kernel v. 5.15), the API is a list of 238 possible types of kernel event handlers (see LSM_HOOK in include/linux/lsm_hook_defs.h). A security module explicitly registers its handlers required to implement security policies built in it during module initialization. Under certain conditions, the Linux kernel calls registered handlers either to notify about a certain event, e.g., the allocation of a new inode in the file subsystem, or to control an operation from the side of security modules. In the latter case, depending on the result returned by a security module, the processing of the operation can either be continued by the kernel or terminated with an “access denied” code.

The LSM API is constantly changed during kernel development: new control and notification functions are added, whereas unused and redundant operations are removed (see Fig. 1).

At first, several LSMs could not work together. Later, kernel developers modified the interface [10] in such a way that each LSM handler had a corresponding list in the kernel. In this case, the kernel calls all registered LSM handlers according to the list, based on the priority of registering their security modules in the kernel. This makes it possible to include one major security module, which attaches its own data and labels to kernel structures, and several minor modules, which do not require storing additional data to implement security policies. With time, kernel developers refined the API [11] and kernel structures so as to separate data stored by security modules, which allows one to use several major modules simultaneously. The priority of calling these modules is preserved.

The integrity measurement architecture and extended verification module (IMA/EVM) for Linux is also an LSM; however, its checks are always called last. Historically, this was due to the need to provide simultaneous operation of the integrity control module with one of the mandatory control modules (SELinux, Apparmor, Smack, or Tomoyo) before the LSM API enabled simultaneous use of several modules. Currently, this also remains relevant, because integrity checks (IMA) require matching the checksum recorded in the extended attributes of a file with the checksum of the file content. Checksum calculation is a resource-intensive operation, and it is reasonable to postpone it until all other checks are carried out. The module allows one to control not only the integrity of the content of a file (IMA) but also the integrity of its metadata (EVM). In this case, metadata mean file

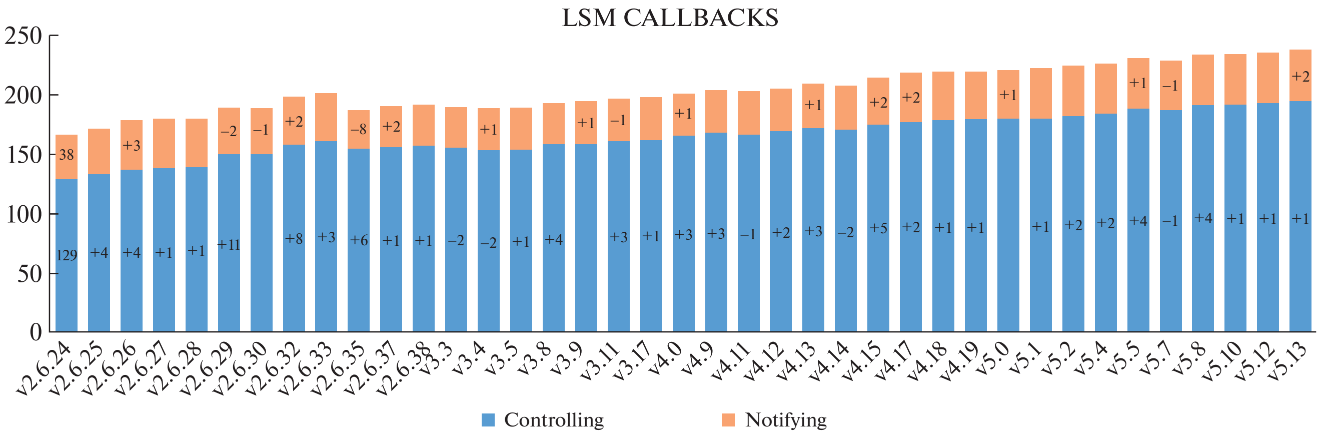

Fig. 1. LSM interface changes in the Linux kernel.

attributes, extended attributes, rights, owner, and groups. The module also supports verification of cryptographic signatures of checksums.

The IMA/EVM has several modes of operation. Only two of them are important for understanding its operating principles. In the first mode, checksums are calculated for files; then, they are written to extended attributes. In this case, no access checks are carried out. In the second mode, checksums are checked when accessing a file for open, write, and execution. If they do not match, then access is denied. These modes do not switch dynamically: the OS kernel must boot in one of them.

An important feature of the LSM API is that the kernel calls registered handlers only after basic parameter checks and after discretionary access checks. The discretionary access policy cannot be canceled; it is implemented by the kernel itself rather than in a separate security module.

Thus, access control in the Linux kernel when processing a system call is currently implemented as follows. First, the basic check of system call arguments for correctness is carried out. Next, the kernel itself checks capabilities, if implied by the system call and its arguments (e.g., raw sockets require the CAP_NET_RAW capability of a process). Then, the kernel performs a discretionary check of rights, after which LSMs are called by the kernel in the order of their registration priority. Once access is granted by all LSMs, the kernel processes the system call and returns the result. If access is denied at any step, then the kernel terminates the system call and returns an error code (e.g., -EPERM or -EACCES) to the process that initiated it. It can be seen that the implementation of access control in the kernel is not monolithic: it consists of a basic mandatory part and additional modules, which can only introduce new checks to restrict access without affecting the results of previous checks. This separation fits well with Event-B models and the refinement mechanism.

The security modules explicitly rely on the kernel to correctly call LSM API functions during event processing. The security policies implemented by these modules imply that the LSM API is sufficient (contains the necessary control handlers) for their implementation as security modules. During runtime verification, we check not only whether a security module correctly implements a security policy but also whether the kernel calls a security module when processing events, as well as check whether the LSM API itself is sufficient to implement a required security policy.

To detect errors in implementation of security policies, the top-priority discrepancies are those of the form “model denies” <==> “OS kernel allows.” In practice, discrepancies due to lack of hardware resources are extremely rare, with the majority of discrepancies being due to the fact that a model does not fully reflect the behavior of a real-world system. Many corner cases are not described in the official documentation for system calls and manifest themselves in practice in the form of discrepancies.

4. DYNAMIC VERIFICATION

Hereinafter, the model is understood as a functional specification in the Event-B language that describes the behavior of system calls in the OS kernel.

Generally, the scheme of dynamic verification is represented as follows.

1. The initial environment for a test (including files, directories, links, owners and groups, their access rights, extended attributes of the mandatory access control module and integrity control module, and other data necessary to run the test) is prepared.

2. The data describing the initial environment, as well as the data of the test file, are written to the database.

3. A controlled test of the OS kernel is carried out in an isolated environment.

4. In parallel with the previous step, results of system call processing by the OS kernel with loaded security modules are written to the database.

5. The collected data on the initial environment of the test and the results of processing system calls by the OS kernel are converted into a trace suitable for reproduction on a model.

6. The animation of the model is initiated using the ProB tool.

7. The state of the model is filled with the data extracted from the initial environment of the test at step 2 and converted at step 5.

8. The system calls with their arguments and results extracted at step 4 and converted at step 5 are animated on the model. In the case of a discrepancy between results of real operations and those of model operations, the trace simulation stops and the cause of the discrepancy is recorded in the test log. In some cases, the discrepancy is not critical. Then, the animation is rolled back to the state of the previous system call and is continued while skipping the system call that caused the discrepancy. The animation process terminates successfully when all system calls are reproduced on the model and the results of their processing by the OS kernel correspond to the model results.

9. The process is repeated for each test.

Steps 1–4 and steps 5–8 can be performed separately at different times and on different machines.

At step 8, the results of syscall processing either match or not. In the former case, the animation continues.

A warning is recorded in the test log only when the model denies access and the OS kernel denies access with an error code unrelated to its control (e.g., not enough memory code -ENOMEM). When the model denies access and the OS kernel allows it, the animation terminates and the corresponding error, model state, and condition violated in the model are recorded in the test log.

The opposite case (the model allows access while the OS kernel denies it) can also be due to an error in the implementation of a security policy. However, it is less critical from a security perspective, which is why, in this case, the animation process continues while skipping the system call on which the discrepancy occurred. If, in a real-world system, access is denied due to lack of resources, e.g., error code -ENOMEM, then the event is simply skipped during animation. If the error code is associated with access control (e.g., -EPERM or -EINVAL), then it may indicate a non-critical error in the implementation, causing a denial of access unnecessary from a security perspective. The information about this discrepancy is recorded in the test log, and the animation continues.

4.1. Implementation

To implement runtime verification, we developed components for monitoring system calls in the OS kernel, collecting information about the initial environment of the test, trace translation, and trace simulation on the model.

The component for collecting information about the initial environment of the test gathers data about system users and their groups, as well as information about the test files passed to it as its arguments. Information about discrete rights, owner, group, inode, and mount volume, as well as complete information about files and directories up to the root partition, is extracted from the files and test itself.

The syscall monitor is implemented as a Linux kernel module and a user-space program. The monitoring module uses Linux kernel mechanisms kprobes and kretprobes to intercept system call arguments and their processing results; it also uses netlink sockets to pass messages with system call events to the user-space program. The latter controls the monitoring process and writes its results to the database.

The trace translation component converts system call traces from the real-world system into a model trace. As arguments, it uses data about the initial environment of the test, as well as the trace previously collected during test monitoring. When translating a trace, all real-world entities are translated into enumerated elements of the model state. Additional events, which correspond to the steps of preparing the initial environment from the tests, are also included in the model trace. For instance, the init event is added, which updates the model state to match the initial environment of the test. Without it, it would be necessary to reconstruct the sequence of system calls required to reach this model state. Also, the login pseudo-event is added to the Event-B specification, which allows the preparatory manipulations with the model to be separated (in the trace) from the initiation of a test run.

The animation component replays traces on the model and checks the access results obtained on the real-world system for conformity with the model. At the final step, it displays the coverage statistics for the model elements and the test result (including the conditions violated in the model, if any). The component is implemented based on the ProB de.prob2.kernel library.

4.2. Tests

Below, we describe some tests used for dynamic verification. A test is a program that performs a test action on a tested OS. A test action is one of the Linux kernel syscalls included in a functional specification. A test is associated with the description of an OS state before a test action (i.e., existing files, directories, users, groups, labels, etc.). This description, together with a target system call, is stored in the database and is used to prepare the initial environment.

A test set is constructed based on the functional requirements of system calls represented in a model. An alternative would be to use a model itself (its structure) to build a test set. The requirements allow one, using the test set, to check not only the OS implementation but also the model.

The approach was tested on a model of system calls that interact with the file system. The list of these system calls is as follows: open, creat, mkdir, getdents, getxattr, setxattr, link, symlink, chmod, chown, execve, and unlink.

The functional requirements were divided into three categories: discrete access control (DAC) properties, mandatory access control (MAC) properties, and integrity (IMA/EVM) properties [12]. Each category has its own subcategories, e.g., object access, path traversal, etc. The tests cover an access for each category for each system call for each object type (file, directory, link) for each user type (privileged, unprivileged, with preferences) for each system call (if applicable).

The tests are minimalistic, independent, and reproducible: they execute only one system call under monitoring. This approach makes it possible to simplify the OS kernel debugging process and access control system, as well as to identify inaccuracies in or incompleteness of functional specifications. In addition, the approach facilitates the translation of a test trace into a model trace, as well as the process of trace simulation on the model.

To generate tests, we use the combinatorial approach. DAC tests check access to an object (file or directory), traversal of the path to an object, and directories with SetGID and Sticky bits. The tests vary the object type (file/directory), system call, object owner, user and group of an access subject (process), ACLs, additional groups of the subject, path length to an object, and access rights for directories along the path.

MAC tests check mandatory rules of access to an object and to its parent directory, as well as access to an object with special attributes. These tests vary the object type (file/directory), mandatory label of an object and subject (process), system call, object owner, subject owner, path length to an object, mandatory label of directories along the path to an object, and special attributes.

In IMA/EVM tests, the following parameters are varied:

• the subjection of an object to integrity control;

• the immutable bit of an object;

• the checksum of a file, stored in the extended attributes of the file.

To minimize the test set, we do not generate all pairs of mandatory labels for subjects and objects. Instead, we use only sets of pairs of labels in different important relationships (the labels are equivalent, the first one dominates, the second one dominates, and the labels are incomparable).

Based on the results of trace simulation on the model, the statistics of model element coverage is printed. This information can be used to supplement the test set for more complete coverage of model elements and parts of conditions in the model.

5. RESULTS

As part of this study, a prototype of the runtime verification system was implemented. It checks the conformity between the behavior of the Linux kernel (together with the closed source mandatory access control LSM and the IMA/EVM) and the abstract models of system behavior in the Event-B language that formalize security policies and access control requirements for the whole OS. The prototype was tested on a set of Linux syscalls related to file system access.

This made it possible to detect some errors, including the error in the setxattr syscall model when only the XATTR_CREATE and XATTR_REPLACE flags were allowed and it was not taken into account that the flag variable may not have a special value and be equal to zero, as well as the modeling error with excessive requirements for the read and execute rights on an executable file in case of the execve syscall. The Linux kernel does not check for read rights. The modeling error with incorrect discretionary access rights checks for writing to a parent directory with a created link to a file, which leads to denied access in the real-world system versus granted access in the model.

An error was found in the utility for editing MAC labels, which is supplied with the MAC module; it manifested itself in incorrect assignment of access rights to a file when using the flag that disables some checks.

A logical error in the implementation of the MAC module was detected: when creating links, the access right to the parental directory of the file to which the link is created was not checked. Thus, the link was successfully created in contradiction with the security policy. In addition, an error in the implementation of the EVM module was detected: MAC labels were not measured (the checksum was not calculated). Changing a mandatory label should indicate an integrity violation of file metadata, which did not happen in the implementation.

6. RELATED WORKS

In [13], the correctness of LSM API function calls by the kernel (whether there are any paths in code where these calls are skipped) was considered. The authors used the fact that, in most cases, LSM func-

Table 1. Number of tests

syscall	DAC	MAC	IMA/EVM	all
open	141	14084	45	14270
Creat	57	9268	15	9340
mkdir	57	9268	15	9340
getdents	0	26	0	26
getxattr	110	9372	30	9512
setxattr	110	9372	30	9512
link	57	9268	15	9340
symlink	55	9268	15	9338
chmod	110	9372	30	9512
chown	110	9372	30	9512
execve	55	4712	15	4782
unlink	3015	27804	15	30834
all	3877	121186	255	125318

tion calls in the kernel are placed correctly. They collected traces of kernel system calls, their arguments, and results of LSM functions and functions used to access main kernel data structures. Based on the collected information, anomalies were detected when, in most cases, during a system call, the LSM API controlled access to kernel data, but there were paths in kernel code where the required checks were missing.

In the continuation [14] of their previous work, the authors checked not only the necessity of LSM calls on certain execution paths in the Linux kernel but also their sufficiency. In the previous work, the authors assumed that, in the most part of the kernel, necessary LSM calls were already placed correctly. In [14], the authors modeled necessary LSM calls for certain types of access to kernel data structures. Additional errors were detected when there was an LSM call but some data were not controlled and checked by it, which could lead to modification of critical data structures without appropriate checks, as well as to their complete skipping in the future because of unauthorized privilege escalation. The authors used the same approach with dynamic collection of execution traces from the Linux kernel and their subsequent static analysis.

In [15], dynamic verification was used to analyze Linux kernel modules, some approaches to instrumentation of loaded Linux kernel modules were discussed, and a KEDR dynamic verification tool was presented. This tool makes it possible to intercept function execution, track memory leaks, and simulate (fault injection) error codes for some kernel functions to check their processing paths in modules.

In [16], automata were employed to analyze states of kernel threads in real-time Linux with PRE-EMPT_RT. An important difference from other works is that complete analysis was performed in the kernel itself, since the frequency of intercepted events was so high that it was impossible to record them in the event log, extract it from the kernel into a file, and then analyze it. The entire analysis was carried out immediately upon occurrence of kernel events. The authors described several small automaton models of kernel states, translated them into C code, combined them into kernel modules, and associated transitions from one automaton state to another with kernel function calls by using Linux tracing mechanisms. In the considered automaton models, the very fact of calling certain functions without analyzing call arguments and call context was sufficient.

7. CONCLUSIONS

In this paper, we have described a method for testing and monitoring OS kernel components that is based on functional specifications represented as formal models in the Event-B language. This method makes it possible to efficiently check OS implementations for conformity with rather complex formal models written in languages different from implementation languages by establishing the relationship between model events and system calls, as well as model variables and specific data obtained by tracing syscall parameters. A feature of the method is a rather complex initial OS configuration, within which it becomes possible to implement the necessary variety of test cases that cover various conditions from event specifications in the formal model.

The method was successfully used to test several different implementations of the LSM, which implements additional access controls for the Linux kernel. The tested LSM components implemented multi-level or mandatory access control rules.

FUNDING

This work was supported by the Russian Foundation for Basic Research, grant no. 20-01-00568.

CONFLICT OF INTEREST

The authors declare that they have no conflicts of interest.

REFERENCES

1. Abrial, J.-R., *Modeling in Event-B: System and Software Engineering,* Cambridge University Press, 2010.
2. Devyanin, P.N., Khoroshilov, A.V., et al., Using refinement in formal development of OS security model, *Lect. Notes Comput. Sci.,* 2016, vol. 9609, pp. 107–115.
3. Kuliamin, V., Khoroshilov, A., and Medveded, D., Formal modeling of multi-level security and integrity control implemented with SELinux, *Proc. Int. Conf. Actual Problems of Systems and Software Engineering (APSSE),* 2019, pp. 131–136.

4. Petrenko, A.K., Efremov, D.V., et al., Monitoring and testing based on multi-level program specifications, *Tr. Inst. Sist. Program. Ross. Akad. Nauk* ~~(Proc. Inst. Syst. Program. Russ. Acad. Sci.)~~, 2020, vol. 32, no. 6, pp. 7–18. https://doi.org/10.15514/ISPRAS–2020–32(6)–1
5. Abrial, J.-R., Butler, M., et al., Rodin: An open toolset for modelling and reasoning in Event-B, *Int. J. Software Tools Technol. Transfer,* 2010, vol. 12, no. 6, pp. 447–466.
6. Wirth, N., Program development by stepwise refinement, *Commun. ACM,* 1971, vol. 14, no. 4, pp. 221–227.
7. Leuschel, M. and Butler, M., ProB: A model checker for B, *Lect. Notes Comput. Sci.,* 2003, vol. 2805, pp. 855–874.
8. Khoroshilov, A., Kuliamin, V., et al., A state-based refinement technique for Event-B, *Proc. Ivannikov Memorial Workshop (IVMEM),* 2020, pp. 49–54.
9. Kroah-Hartman, G., LSM: Add all of the new security/* files for basic task control. https://git.kernel.org/pub/scm/linux/kernel/git/tglx/history.git/commit?id=2b15fe6334aebd7d3340f8b826acb79b138afa74 . Accessed January 15, 2022.
10. Edge, J., Progress in security module stacking. https://lwn.net/Articles/635771. Accessed January 15, 2022.
11. Edge, J., LSM stacking and the future. https://lwn.net/Articles/804906. Accessed January 15, 2022.
12. Linux Integrity Subsystem. http://linux-ima.sourceforge.net. Accessed January 15, 2022.
13. Edwards, A., Jaeger, T., and Zhang, X., Runtime verification of authorization hook placement for the Linux security modules framework, *Proc. 9th ACM Conf. Computer and Communications Security,* 2002, pp. 225–234.
14. Jaeger, T., Edwards, A., and Zhang, X., Consistency analysis of authorization hook placement in the Linux security modules framework, *ACM Trans. Inf. Syst. Secur.,* 2004, vol. 7, no. 2, pp. 175–205.
15. Rubanov, V.V. and Shatokhin, E.A., Runtime verification of Linux kernel modules based on call interception, *Proc. 4th IEEE Int. Conf. Software Testing, Verification, and Validation,* 2011, pp. 180–189.
16. de Oliveira, D.B., Cucinotta, T., and de Oliveira, R.S., Efficient formal verification for the Linux kernel, *Lect. Notes Comput. Sci.,* 2019, vol. 11724, pp. 315–332.

Translated by Yu. Kornienko

SPELL; OK