ISSN 0361-7688, Programming and Computer Software, 2015, Vol. 41, No. 1, pp. 49–64. © Pleiades Publishing, Ltd., 2015.
Original Russian Text © I.S. Zakharov, M.U. Mandrykin, V.S. Mutilin, E.M. Novikov, A.K. Petrenko, A.V. Khoroshilov, 2015, published in Proceedings of the Institute for System Programming of RAS, 2014, Vol. 26, No. 2.

Configurable Toolset for Static Verification of Operating Systems Kernel Modules

I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. K. Petrenko, and A. V. Khoroshilov

Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia
E-mail: ilja.zakharov@ispras.ru, mandrykin@ispras.ru, mutilin@ispras.ru, novikov@ispras.ru, petrenko@ispras.ru, khoroshilov@ispras.ru

Received September 29, 2014

Abstract—An operating system (OS) kernel is a critical software regarding to reliability and efficiency. Quality of modern OS kernels is already high enough. However, this is not the case for kernel modules, like, for example, device drivers that, due to various reasons, have a significantly lower level of quality. One of the most critical and widespread bugs in kernel modules are violations of rules for correct usage of a kernel API. One can find all such violations in modules or can prove their correctness using static verification tools that need contract specifications describing obligations of a kernel and modules relative to each other. This paper considers present methods and toolsets for static verification of kernel modules for different OSs. A new method for static verification of Linux kernel modules is proposed. This method allows one to configure the verification process at all its stages. It is shown how it can be adapted for checking kernel components of other OSs. An architecture of a configurable toolset for static verification of Linux kernel modules that implements the proposed method is described, and results of its practical application are presented. Directions for further development of the proposed method are discussed in conclusion.

Keywords: operating system kernel, kernel module, software quality, static verification, contract specification, environment model, specification of rule for correct usage of API.

DOI: 10.1134/S0361768815010065

1. INTRODUCTION

Reliability and efficiency of an operating system (OS) kernel are important characteristics of its quality, since a kernel is a heart of an OS and all user applications rely on it. In most modern OSs, kernels implement only primary functionality, like, for example, processor scheduling, memory management, and interprocess communication. Therefore, kernels, on the one hand, have a relatively small size and, on the other hand, their high quality has been formed over a long period of time in various use cases.

In most OSs, kernel functionality can be extended by dynamic loading of modules. A common example of a kernel module is a device driver that is required when a particular device is connected. Moreover, file systems, network protocols, audio codecs, etc. are often implemented as kernel modules. Amount of source code for modules delivered with an OS kernel may considerably exceed that of the kernel (for instance, by a factor of eight for the Linux kernel). Furthermore, due to various reasons many developers grant no access to source code of their modules, distributing them in the form of binary code. This considerably complicates application of certain quality assurance approaches. Not all kernel modules can be used intensively (for example, drivers of specific devices). All this lead us to the fact that kernel modules of various OSs have a considerably lower level of quality than kernels themselves. This fact is confirmed by investigations showing that just open source modules contain seven times more bugs than an OS kernel does [1–3].

High performance of OS kernel modules is due to the fact that most of them operate in the same address space and with the same privileges as a kernel. Thereby, bugs in modules may result in unstable operation of a kernel or an entire OS. Analysis of commits to Linux kernel modules showed that violations of rules for correct usage of the kernel API in modules are the source of about half of all bugs that are unrelated to violations of specifications for hardware, network protocols, audio codecs, etc. [4]. Similar statistic data for other OSs is unavailable to us; however, this type of bugs is of rather high interest for ongoing investigations [5].

1.1. Approaches to Finding Violations of Rules for Correct Usage of a Kernel API in Modules

There are different ways to find violations of rules for correct usage of a kernel API in modules. In prac-

tice, code review and testing are used most often. These approaches help to find and fix a reasonably large number of bugs in modules. They, however, fail to find all possible bugs [6]. Thorough code review requires great manual efforts; therefore, it is performed in full only for an OS kernel but not for kernel modules that are large, complex, and dynamically evolving programs. Testing is usually performed automatically to reduce efforts as compared to code review. This approach, however, requires preparation of a test environment, which is quite complicated in the case of device drivers. Moreover, only small programs can be carefully checked by testing.

Recently, application of methods and tools for static code analysis, i.e., analysis without actual execution (usually source code is analyzed), became a main trend in checking software. In practice, the so-called lightweight approaches are widely used. They use a lot of heuristics to check large programs in a time comparable to the compilation time. A drawback of heuristic-based tools for static code analysis is that they, on the one hand, miss certain bugs and, on the other hand, generate a great number of false alarms. These tools are mainly used to find violations of general safe programming rules, like, for example, null pointer dereference and buffer overflow [7, 8]. Certain tools are used to check rules for correct usage of the Linux kernel API in modules [9].

Static verification (thorough static code analysis) allows one to find all bugs of a given type in programs or to prove their correctness. Modern static verification tools (which implement, for example, the counterexample-guided abstraction refinement method [10]) already can prove feasibility of specified properties for medium-scale programs in a reasonable time. In particular, these tools can be used to verify OS kernel components, like, for example, kernel modules (for the Linux kernel, the size of most modules is several thousand lines of code).

Static verification tools themselves cannot find violations of rules for correct usage of a kernel API in modules, but they can solve the *reachability problem*. These tools determine reachability of a certain operator marked with a certain label from a given entry point. Thus, the problem of rule violation finding should someway be transformed to the reachability problem. To this end, we need to develop *specifications of rules for correct usage of a kernel API* relating violations of rules and reachability of an operator marked with a given label. Moreover, investigations showed that, in order to obtain acceptable results of static verification of kernel modules (to find bugs of a given type with a moderate number of false alarms), these tools need a quite accurate *environment model* describing scenarios of interaction between a kernel and modules that occur in a real environment [11].

Thus, to find violations of rules for correct usage of a kernel API in modules, static verification tools need *contract specifications* describing formal obligations of a kernel and modules relative to each other. From the side of a kernel, the contract specifications should define correctly and fully a set of possible scenarios of interaction with modules and should provide a model of a kernel API used by modules. From the side of modules, the contract specifications should define which requests of modules to a kernel are considered to be correct.

Static verification tools are now being developed in universities and research institutes all over the world. Every year new tools become available [12–14]. Static verification tools implement various methods to prove feasibility of various specified properties. There is no single leader in this race, since these tools use different techniques and are designed for different classes of bugs. Therefore, in the long term, it is important to have an opportunity to use different static verification tools.

In this paper, we consider those static verification tools that can be used to check software written in the C programming language, since kernel modules for most of OSs are developed using C.

1.2. Characteristics of an OS Kernels Development Process

When developing contract specifications and performing static verification of modules, characteristics of an OS kernels development process should be taken into account.

The Microsoft Windows kernel is developed on a centralized basis with vast resources being allocated to develop and apply new technologies, in particular, to ensure quality by means of static verification. A great number of Windows kernel modules are developed and supported by hardware manufacturers only. This means, among other things, that only the module developers have access to their source code that is required to perform static verification. Therefore, often, the developers themselves perform static verification, analyze results, and update contract specifications when needed. For the Microsoft Windows OS, the API between the kernel and modules is stable,[1] thereby the same contract specifications can be used for different versions of the kernel.

The Linux kernel is an open source software and it is delivered with a great number of modules (about 4000 in the latest versions). All new versions of the Linux kernel are developed by more than 1000 developers from more than 200 organizations all around the world [15]. The developers do not write contract specifications by themselves. For the Linux OS, the API between the kernel and modules is not stable.[2] This

[1] Documentation on Windows Driver Frameworks is available on http://msdn.microsoft.com/en-us/Library/Windows/Hardware/ff557565%28v=vs.85%29.aspx.

[2] More information is available on http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt.

complicates development and maintainance of contract specifications, since, when new versions of the kernel become available, these contract specifications may require a refinement.

In this paper, we did not consider characteristics of other OS kernels development process, since we have no information about application of static verification tools to them.

1.3. Present Toolsets for Static Verification of Kernel Modules

In order to automate static verification of kernel modules for various OSs, *static verification toolsets* are developed. Thus far, only one static verification toolset has risen to the level of industrial application. This is Static Driver Verifier (SDV) [16], which was developed by Microsoft Corporation. This toolset can be used to check Microsoft Windows kernel modules for compliance to rules for correct usage of the kernel API by means of the SLAM static verification tool [17].

SDV automatically obtains information about source files and build options of modules using scripts describing a build process. In this static verification toolset, the part of the environment model that describes possible interaction scenarios between the kernel and modules is generated automatically based on specifications developed by the SDV developers for all types of modules and on annotations for modules that are developed manually. The model of the Windows kernel API used by modules is included into SDV. Specifications of rules for correct usage of the kernel API are written using a specification language for interface checking (SLIC) [18]. Today, SDV is delivered with a set of about 200 rule specifications. A research version of SDV [19] allows one to add rule specifications and to use the Yogi static verification tool. The developers of SDV paid great attention to automate launching of the process of kernel modules static verification and to support analysis of static verification results. The users obtain both overall results of static verification for analyzed modules over all rules being checked and visualized *error traces* (paths in source code where rules can be violated). As of the year 2010, SDV helped to find 270 bugs in kernel modules that are delivered with the Microsoft Windows OS.

For Linux kernel modules, two static verification toolsets were developed: DDVerify (Carnegie Mellon University, Pittsburgh, USA) [20] and Avinux (Eberhard Karls University, Tubingen, Germany) [21].

DDVerify uses its own scripts to extract information about source files and build options of analyzed modules. In this static verification toolset, contract specifications are developed completely manually using C. The DDVerify developers prepared an environment model for four types of modules, as well as for hardware interrupt handlers, timers, etc. Moreover, they included eight specifications of rules for correct usage of synchronization primitives and rules for correct initialization of variables before use in the environment model. DDVerify can be used to check modules using two static verification tools: CBMC [22] and SATABS [23]. To facilitate error traces analysis, the static verification toolset includes a plugin for the Eclipse IDE.

The Avinux static verification toolset automatically checks single preprocessed source files of kernel modules by modifying original scripts describing a kernel build process. In Avinux, the environment model should be developed almost completely by hand (only functions exported by modules for parameters of which initialization code is generated are called automatically). Specifications of rules for correct usage of the kernel API are written using an extended version of SLIC. Avinux checks rules for correct usage of synchronization primitives and rules for correct memory operation. This static verification toolset was integrated just with static verification tool CBMC. Avinux is implemented as an Eclipse plugin that helps to launch the toolset automatically. The users obtain error traces in the CBMC format, which considerably complicates their analysis.

None of the static verification toolsets described above takes into account all characteristics of the Linux kernel development process. SDV is intended only for static verification of kernel modules for the Microsoft Windows OS whose kernel API is stable. DDVerify requires adaptation of its build process, kernel header files, and contract specifications to each new version of the kernel. Moreover, this static verification toolset implements the environment model only for four types of modules out of several hundreds. Avinux does not support automatic check of modules consisting of several source files, requires manual development of the most part of the environment model, and manual maintenance of rule specifications. Both toolsets for static verification of Linux kernel modules showed no prominent results in practice and, now, they are not developed any more.

None of the above mentioned toolsets supports integration of third-party static verification tools. It is especially important when checking Linux kernel modules, since, on the one hand, there is a great number of static verification tools implementing completely different approaches and, on the other hand, the developers of static verification toolsets, in contrast to Microsoft Corporation, have limited resources to support a particular static verification tool.

Present static verification toolsets provide no means for comparative analysis of verification results and for automatic marking of bug reports; this is important, since there are quite many Linux kernel modules, rule specifications, static verification tools, and their configurations that, in addition, evolve with time.

Therefore, it is important to develop a new method and toolset for static verification of Linux kernel modules.

1.4. Outline of the Paper

The new method for static verification of Linux kernel modules is proposed in Section 2. It allows one to configure the verification process at all its stages. Furthermore, this section describes modifications of the method required to check kernel components of other OSs. The architecture of the developed configurable toolset for static verification of Linux kernel modules is discussed in Section 3. Section 4 presents results of practical application of this toolset; here, we analyze bugs found in Linux kernel modules, causes of false alarms and missed bugs, static verification time consumption, and causes of unsuccessful terminations of the static verification toolset. Moreover, Section 4 demonstrates capabilities of the developed configurable static verification toolset for comparing static verification tools and their configurations. Directions for further development of the proposed method are discussed in conclusion.

2. METHOD FOR STATIC VERIFICATION OF LINUX KERNEL MODULES

The proposed method consists of several steps that are described in Subsections 2.1–2.5. Modifications of the method required to check kernel components of other OSs are addressed in Subsection 2.6.

2.1. Initial Processing of Source Code for Static Verification

At the first step of the method, initial processing of source code of the Linux kernel and modules is performed. In Section 1.1, we mentioned that modern static verification tools cannot be used to check the entire OS kernel. Therefore, it is required to divide source code of the Linux kernel and modules with corresponding compile and link commands (that describe build rules) so that the size of resultant *verification objects* is limited by several thousand or tens of thousands lines of code and that these verification objects comprise as much code implementing functions of the corresponding modules as possible. For example, a verification object can include source files that constitute a module being analyzed together with the corresponding build commands.[3] A kernel module can call kernel functions as well as functions from other modules; hence, these functions code can be added into a corresponding verification object.

[3] This can be done by using link commands related to modules, since output files of commands for compiling source files are input files for link commands, including module link commands.

In the proposed method, verification objects are prepared automatically using original Linux kernel build scripts. To this end, they are modified so that, along with assembling the kernel and modules, information about the corresponding compile and link commands is outputted.

2.2. Generating the Environment Model

In order to check verification objects obtained at the first stage for compliance with rules for correct usage of the Linux kernel API, the environment model for these verification objects is generated at the second stage and the entry point for static verification tools is specified.

In a typical Linux kernel module, an initialization function, callbacks (callbacks for probing and disconnecting devices, callbacks for opening, reading, writing, and closing files, hardware interrupt handlers, etc.), and an exit function are defined. The initialization function is called when the kernel loads the module. This function registers module callbacks that are called, when required, by the kernel to handle system calls from user applications, hardware interrupts, and kernel internal events. Before the module is unloaded, the exit function is called in which resources allocated to the module are released and module callbacks are deregistered.

In the proposed method for static verification of Linux kernel modules a set of scenarios of interaction between the kernel and modules, which may occur in the real environment, is described using π-processes [24, 25]. This allows one to define the environment model both on a detailed level for particular groups of module callbacks and as callback group patterns that cover most of the other groups. For example, these callback group patterns may specify that module callbacks can be called only after a successful invocation of the initialization function, that a device can be disconnected only after its successful probing, and that a file can be read only upon being opened. Thus, the π-model of environment can be specified rather compactly for all types of Linux kernel modules. Moreover, the same specification of the π-model of environment can be used for different versions of the kernel, since patterns are not restricted to any particular module callbacks or even to any particular groups of callbacks. Figure 1 shows the layout view of the environment model for a USB driver.

The user can refine the specification of the π-model of environment by describing a particular group of module callbacks, for which the environment model is generated based on a callback group pattern, if would be found that a certain known bug is missed because the model specifies not all possible scenarios of kernel–modules interaction or that a false alarm is produced because the model allows scenarios impossible in the real environment.

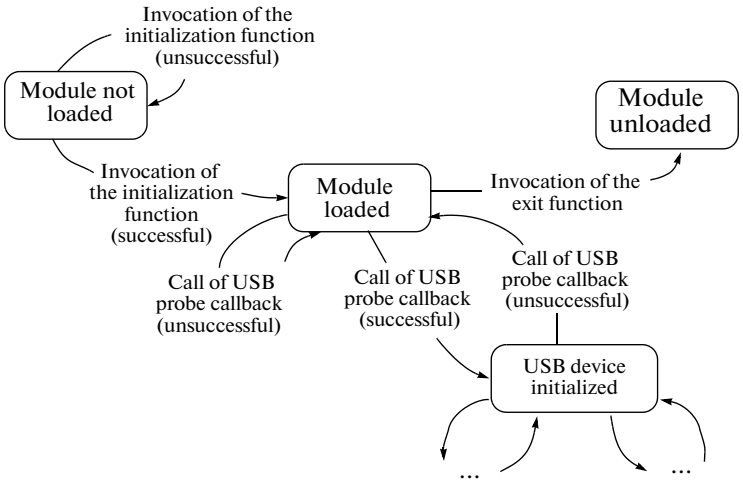

Fig. 1. Layout view of the environment model for a USB driver.

Callback group patterns from the specification of the π-model of environment are suggested to be selected and filled automatically by analyzing source files of verification objects obtained at the previous step. For each verification object several environment models can be generated, e.g., for different groups of module callbacks. In order to use various static verification tools in the future, each environment model is translated into a function written in C, from which callbacks of a verification object are called in the same manner and in the same sequence as in the real environment. This function is added to source files of a verification object and then it is used as an entry point for static verification tools.

2.3. Developing Specifications of Rules for Correct Usage of the Linux Kernel API

At the third step of the proposed method, for those elements of the Linux kernel API that match with specified rules, we develop a model[4] and, based on these rules and a state of this model, specify preconditions for given API elements. Developing specifications of rules for correct usage of the Linux kernel API is discussed in detail in [26], where an aspect-oriented extension for the C programming language used to develop specifications is described. Figure 2 presents a specification of rules for correct release of USB request blocks (URBs) to show integral parts of specifications and their relations with one another and with source code of modules being analyzed.

Note that, in the proposed method signatures of kernel API elements, for which the model is developed and preconditions are set, are described completely (see Fig. 2 for *USB_ALLOC_URB* and *USB_FREE_URB* definitions). Thus, compatibility of rule specifications with the kernel API and its implementation can be maintained automatically, since the developers as a rule modify the corresponding API elements when making essential changes in their implementation.

The user can improve these specifications if it is found that they are not adequate enough or that they miss well-known bugs or lead to false alarms. The users can also add their own rule specifications.

Based on rule specifications, we propose to instrument source files obtained at the second stage so that violations of rules for correct usage of the Linux kernel API are transformed to reachability of the *LDV_ERROR* label from given entry points. As a result, we obtain *verification tasks*: reachability problems are posed for verification objects.

2.4. Launching Static Verification Tools

At the fourth stage, static verification tools are launched on the produced verification tasks in order to find violations of rules or to prove correct operation of the corresponding modules against these rules. We propose to run static verification tools using adapters. For a certain tool, an adapter does the following:

• Prepares C files that constitute verification objects, e.g., uses the standard C preprocessor for each file, processes files using the CIL code transformation tool [27], or merges all files into one with help of CIL (all static verification tools accept preprocessed source files as an input; some of tools can analyze only one file at a time).

• Launches a static verification tool passing the prepared C files together with a given configuration and limiting the amount of resources the (CPU time and memory) tool is allowed to consume (this is especially important, since static verification tools may run unacceptably long time and/or require excessively large memory capacity).

[4] This model is a part of the environment model for Linux kernel modules.

```
#include ⟨linux/usb.h⟩
#include ⟨verifier/rcv.h⟩

/* Set of pointers to URB structures for which memory is
allocated to a module. Header file verifier/rcv.h de
fines the "set" data type as well as operations
with objects of this type
set URBS = empty;

/* Model functions to allocate/deallocate memory
for URB structures. Header file verifier/rcv.h. */
 defines memory allocation model function ldv_alloc

struct urb *
ldv_usb_alloc_urb (void)
{
  void *urb;
  urb = ldv_alloc();
  if (urb) {
   add (URBS, urb);
  }
  return urb;
}
void
ldv_usb_free_urb (struct urb *urb)
{
  if (urb) {
   remove (URBS, urb);
  }
}
```

```
/* Description of a set of points where kernel
API elements are used */
pointcut USB_ALLOC_URB: call (
  struct urb *usb_alloc_urb (int, qfp_t))
pointcut USB_FREE_URB: call (
  void usb_free_urb (struct urb *))

/* Binding of model function calls to points
where API elements are used */
around: USB_ALLOC_URB {
  return ldv_usb_alloc_urb();
}
around: USB_FREE_URB {
  ldv_usb_free_urb($arg1);
}

/* Preconditions for kernel API elements. Macrofunction
ldv_assert is defined in the header file verifier/rcv.h as follows:
#define ldv_assert (e) (e ? 0 :
ldv_error ())
static inline void ldv_error(void) {
  LDV_ERROR: goto LDV_ERROR;
}
LDV_ERROR is the label that, upon reached,
indicates a violation of rules being checked. */
before: USB_FREE_URB {
  ldv_assert (contains(URBS, $arg1));
}
after: MODULE_EXIT {
  ldv_assert (is_empty(URBS));
}
```

Fig. 2. Specification of rules for correct release of USB request blocks.

• Handles an exit status of the static verification tool (for example, successful termination, termination by a signal due to exceeding the memory limit, or termination because of a bug in the tool) and evaluates resources consumed by the tool.

• Returns the *Unknown* verdict when the static verification tool terminates incorrectly and points to causes of the unsuccessful termination that can be determined by analyzing the output of the tool, as well as its exit status.

• Determines, based on the output of the tool, a proper verdict in the case when the static verification tool terminated correctly, indicating either that there were no violations of rules (the tool returns the *Safe* verdict) or that there possibly was a violation of rules (the tool returns the *Unsafe* verdict).

• When the *Unsafe* verdict is returned, the static verification tool provides an error trace whose format may considerably differ for different tools; therefore, to facilitate further analysis of static verification results, the adapter converts all error traces to a common format (see the following subsection).

In order to integrate third-party static verification tools, the user must develop a corresponding adapter. One can use parts of adapters for other tools, since they may be organized similarly.

2.5. Analyzing Static Verification Results

At the final step of the proposed method, automatic analysis of static verification results is performed in several directions.

First, analysis of error traces provided by static verification tools is facilitated to gain insight into causes of revealed bugs and to find out which bug reports are false alarms. For this purpose, all error traces are first converted to the common format (this is done by the corresponding static verification tool adapter), and then the error traces represented in the common format are uniformly visualized with links to the corresponding source code of the analyzed modules and the Linux kernel.

Second, bug reports are automatically marked (to indicate whether they correspond to real bugs or false alarms) if error traces are proved similar to the already analyzed ones (for example, have the same trees or stacks of function calls). Thus, we can considerably reduce efforts on analysis of results. For instance, if a static verification tool produces a false alarm when

checking a module for a particular version of the Linux kernel, then it will likely produce a quite similar error trace for this module in the next version of the kernel; therefore, owing to the automated marking of bug reports, one can avoid performing this analysis repeatedly.

Third, statistic data and ability to compare static verification results are provided, since there are a lot of Linux kernel modules, rule specifications, static verification tools, and their configurations that, in addition, evolve with time.

Fourth, collaborative analysis of static verification results is enabled without considerable expenses on interaction procedures.

2.6. Adapting the Method for Checking Kernel Components of Other OSs

The method proposed above can be used to verify kernel components of other OSs. Thus, for a target OS, it is required to

• Adopt an approach to preparing verification objects. Information about source files and build options of kernel components can be obtained using original build scripts in the same way as for Linux kernel modules.

• Develop an environment model for analyzed kernel components by studying characteristics of intercomponent interaction, as well as interaction between these components and an OS environment, and by formulating restrictions on a set of interaction scenarios. Based on these restrictions, the environment model can be developed either manually or using a certain formal description.

• Develop rule specifications for bugs of a sought-for type. For example, to find violations of rules for correct usage of a kernel API, API elements relevant to these rules should first be found and, then, a model for them should be developed and preconditions for these API elements should be specified. Further steps, integration, and launch of static verification tools, including analysis of static verification results, remain the same as for the Linux kernel modules.

2.7. Intermediate Conclusions

The proposed method for static verification of Linux kernel modules allows one to configure the verification process at all the stages:

• When preparing verification objects, one can decide how to divide source code of modules and the kernel and the corresponding build commands.

• One can refine specifications of the π-model of environment.

• One can refine available specifications of rules for correct usage of the kernel API and add new specifications.

• One can specify own configurations and resource limits for integrated static verification tools, as well as integrate third-party tools.

Such configurability helped us to raise a toolset implementing this method to a rather high level of maturity. This toolset is now evolving quite intensively and has already helped to find more than 150 critical bugs in Linux kernel modules.

3. CONFIGURABLE TOOLSET LINUX DRIVER VERIFICATION TOOLS

The proposed method for static verification of Linux kernel modules was implemented in configurable toolset Linux Driver Verification Tools (LDV Tools) [28].

3.1. Architecture of Linux Driver Verification Tools

The architecture of LDV Tools is shown in Fig. 3. Components and subcomponents, as well as static verification tools, are depicted in the center in the order of invocation. Input data is shown on the left. On the right, we depict the order of generating a report based on static verification results, upload of the report to the LDV database, and the LDV Analytics Center component intended for automating analysis of static verification results.

3.2. LDV Core

The process of static verification of Linux kernel modules begins with launching the LDV Core component. This component calls the Kernel Manager subcomponent that creates on a disk a copy of the Linux kernel provided by the user in the form of an archive, a directory, or a Git repository. After that, all components and subcomponents of LDV Tools use this copy only. Kernel Manager modifies original kernel build scripts to obtain later information about source files and build options of modules.

Then, Build Command Extractor is called, which initiates a build process of the Linux kernel. As the build process progresses, this subcomponent intercepts a stream of compile and link commands for kernel files. Using these commands, Build Command Extractor extracts source files related to modules being verified.

Compile and link commands are then transferred to the Command Stream Divider subcomponent that forms verification objects based on these commands. Each verification object corresponds to exactly one kernel module, so that all source files, that constitute corresponding modules, are processed and analyzed further. In the future, we intend to generate verification objects for groups of interrelated modules and to supplement them with some kernel code that the modules analyzed depend on. The user will be offered an opportunity to select a particular strategy for divid-

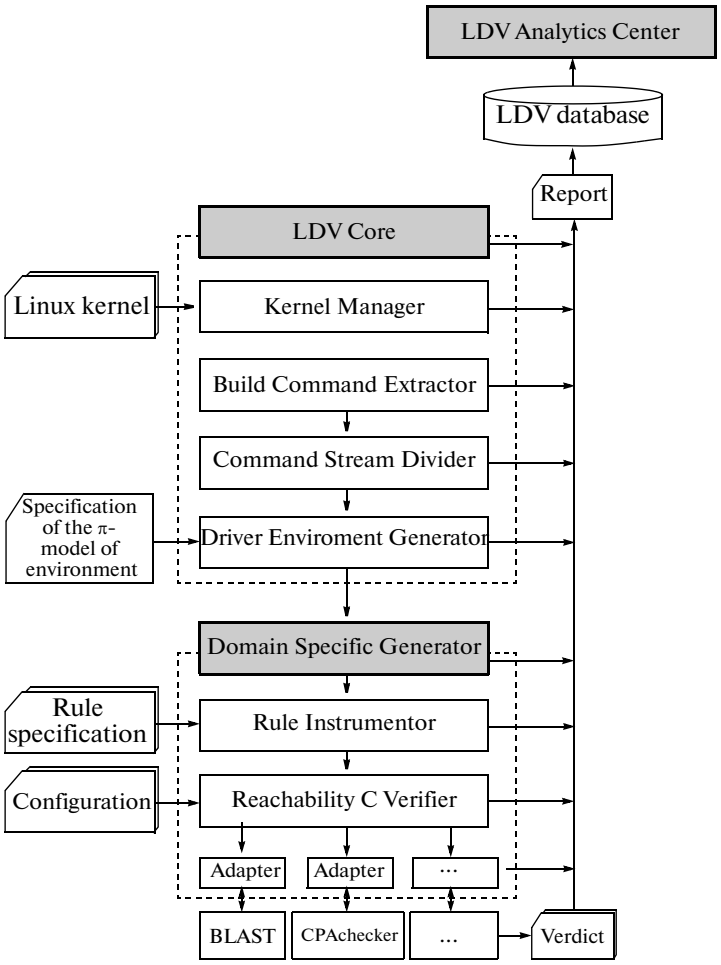

Fig. 3. Architecture of Linux Driver Verification Tools.

ing source code of the Linux kernel and modules and corresponding build commands into verification objects.

Each prepared verification object is inputted to the Driver Environment Generator subcomponent [29] along with the specification of the π-model of environment integrated into LDV Tools. The user can refine this specification if required. Driver Environment Generator finds out the initialization function, callbacks, and the exit function based on source files that constitute verification objects. Then, using this data and the specification of the π-model of environment, an intermediate representation of the environment model is generated for each verification object. This representation is translated by Driver Environment Generator into source code in C that is added to source code of verification objects.

3.3. Domain Specific C Verifier

The Domain Specific C Verifier component weaves verification objects with given rule specifications and launches a particular static verification tool with a specified configuration.

Available rule specifications are stored in a database, where each specification has a unique ID. The users can employ their own rule specification databases, add rule specifications to the available database, and modify existing specifications.

For each rule specification ID, Domain Specific C Verifier calls Rule Instrumentor. This subcomponent extracts, using the ID, information about a specification of rules being checked from the database; based on this information, it instruments verification object source files so that the reachability problem is posed for static verification tools (solution of this problem corresponds to a potential violation of rules being checked).

For static verification, LDV Tools uses either a tool and a configuration specified by the user or a tool and a configuration used by default. The Reachability C Verifier subcomponent calls a static verification tool using the corresponding adapter. The verification task, the configuration, and resource limits for the static verification tool are passed to this adapter.

#	Task	Kernel	Rule	Total	Safe	Unsafe	Unknown	Verdicts		
								True	False	?
1	Task description BLAST, 15Gb	linux-3.13-rc1	08_1a	5994	5004	92	898	2	79	11
2			101_1a	5994	5112	1	881	-	1	-
3			106_1a	5994	5008	23	963	2	20	1
4			10_1a	5994	5175	8	811	-	8	-
5			118_1a	5994	5115	13	866	-	12	1
6			129_1a	5994	5028	7	959	2	5	-
7			132_1a	5994	5055	42	897	13	14	15
8			134_1a	5994	5074	3	917	3	-	-
9			146_1a	5994	5028	11	955	-	1	10
10			147_1a	5994	5017	21	956	3	1	17
11			150_1a	5994	5119	3	872	1	2	-
12			32_7a	5994	5037	81	876	8	70	3
13			39_7a	5994	5050	76	868	5	58	13
14			68_1	5994	4867	119	1008	4	115	-
15			77_1a	5994	5184	1	809	1	-	-

Fig. 4. Static verification results grouped by the Linux kernel version (3.13-rc1) and rule specification ID.

Task	Kernel	Rule	Module	Entry point	Verifier	Error trace	KB Verdict	KB Tags
Task description BLAST, 15Gb	linux-3.13-rc1	106_1a	drivers/base /firmware_class.ko	ldv_main0	blast	...	False positive	env_gen
			drivers/char /ppdev.ko	ldv_main0	blast	...	False positive	multi_module
			drivers/hid/hid.ko	ldv_main1	blast	...	False positive	kernel_model
			drivers/hsi/clients /hsi_char.ko	ldv_main0	blast	...	False positive	env_gen
			drivers/iio /industrialio.ko	ldv_main0	blast	...	False positive	rule_model
			drivers/infiniband /core/ib_ucm.ko	ldv_main0	blast	...	False positive	init_global_var
			drivers/media/pci /ddbridge /ddbridge.ko	ldv_main0	blast	...	False positive	multi_module
			drivers/media/usb /pvrusb2/pvrusb2.ko	ldv_main5	blast	...	True positive	obsolete
			drivers/mtd/ubi /ubi.ko	ldv_main3	blast	...	False positive	pointer_analysis
				ldv_main4	blast	...	False positive	env_gen
			drivers/net/wireless /mac80211_hwsim.ko	ldv_main0	blast	...	True positive	new
			drivers/scsi /dpt_i2o.ko	ldv_main0	blast	...	False positive	init_global_var
			drivers/scsi /scsi_mod.ko	ldv_main0	blast	...	False positive	rule_model

Fig. 5. Analysis of bug reports produced by BLAST when checking the specification of rules describing correct registration of USB gadget devices (results of marking the bug reports are shown on the right: "True positive" means that a real bug is found and "False positive" means a false alarm).

Kernel	Rule	Total changes	Safe → Unknown	Unsafe → Safe	Unsafe → Unknown	Unknown → Safe	Unknown → Unsafe
linux-3.12-rc1 → linux-3.13-rc1	39_7a	310	23	1	2	75	10

Fig. 6. Comparison of static verification results for modules of Linux kernel 3.12-rc1 and 3.13-rc1: some bugs were fixed (transitions from "Unsafe") and, for more modules, correctness was proved (transitions to "Safe") or possible bugs were found (transitions to "Unsafe").

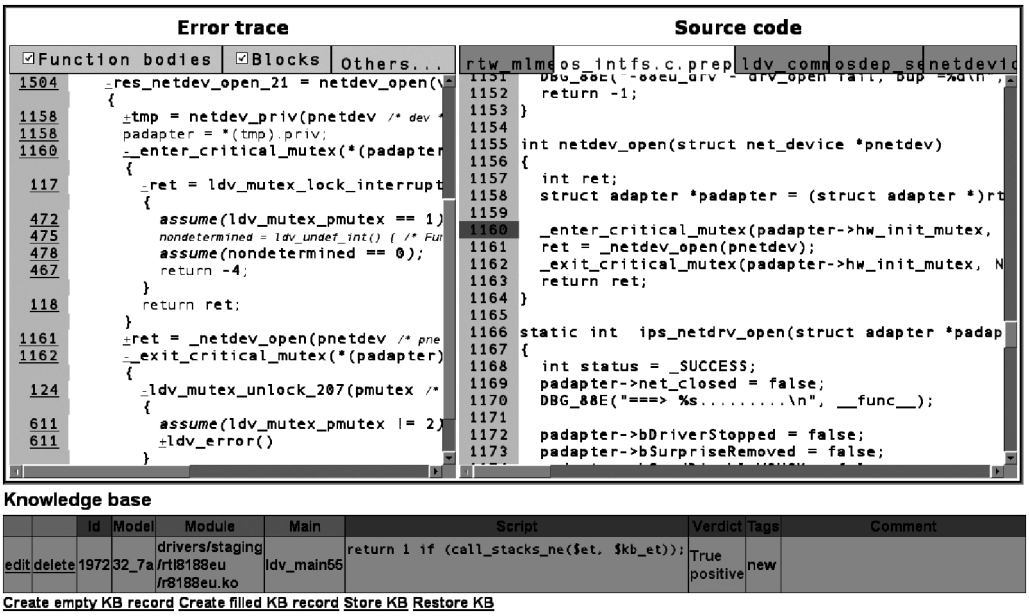

Fig. 7. Visualized error trace: the specification of rules for correct usage of mutexes in one thread is violated.

At the present time, LDV Tools includes adapters for BLAST [30] (used by default), CPAchecker [31], UFO [32], and CBMC [22]. In addition, the user can integrate third-party static verification tools into LDV Tools by implementing the corresponding adapters.

3.4. Generating the Final Report and Uploading It to the LDV Database

Verdicts returned by a static verification tool are processed by all components and subcomponents of LDV Tools in the reverse order (see Fig. 3). At each stage, an intermediate report is supplemented with information concerning operation of the corresponding components and subcomponents. As a result, LDV Tools generates a final report, which includes static verification results for Linux kernel modules. This final report can be uploaded to the LDV database.

3.5. LDV Analytics Center

LDV Analytics Center is a component that provides several types of automated analysis of static verification results uploaded to the LDV database:

- Statistical analysis that makes it possible to group static verification results according to the Linux kernel version, rule specification ID, etc. (Fig. 4).
- Analysis of static verification results obtained for particular rule specifications, modules, etc. (Fig. 5).
- Comparative analysis that (allows one to compare static verification results obtained for different versions of the Linux kernel, (Fig. 6).

To facilitate analysis of error traces, LDV Analytics Center employs the Error Trace Visualizer subcomponent [33] to visualize error traces, represented in the common format, as Web pages. The user is provided with easy navigation over error traces and over corresponding source files of modules, kernel, and contract specifications. An example of a visualized error trace is given in Fig. 7.

Figure 7 also shows the Web interface of Knowledge Base, which is another subcomponent of LDV Analytics Center. This Web interface makes it possible to save results of error trace analysis. Scripts used in Knowledge Base automatically mark all newly uploaded error traces if they are similar according to one or another criteria to previously obtained ones.

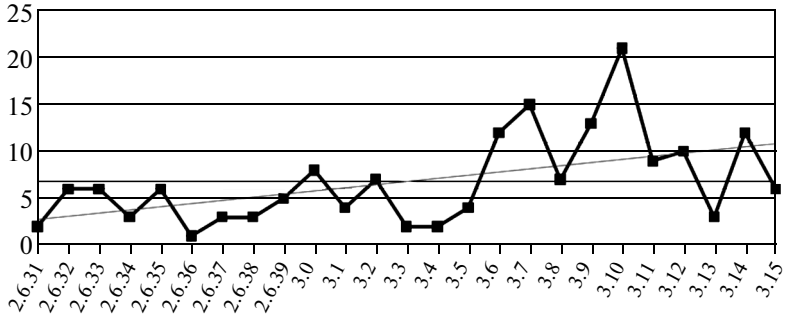

Fig. 8. Dependence of the number of bugs found in Linux kernel modules using LDV Tools on the kernel version: the horizontal line represents the average number and the inclined line is the linear regression.

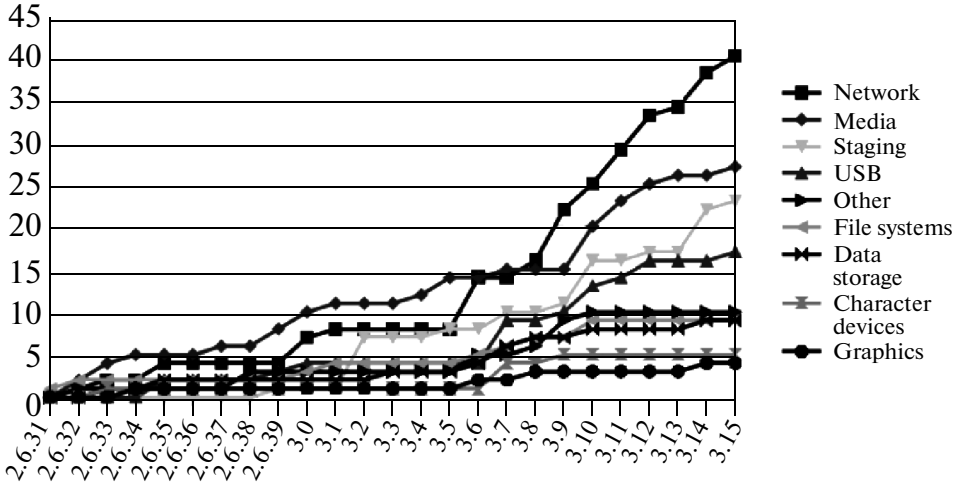

Fig. 9. Dependence of the number of bugs found in Linux kernel modules using LDV Tools on the kernel version for different kernel subsystems.

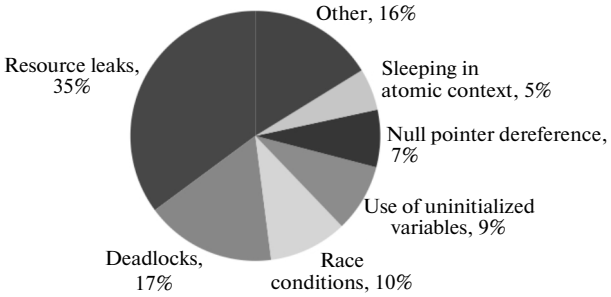

Fig. 10. Distribution of potential impacts of bugs found in Linux kernel modules.

For example, for the error trace shown in Fig. 7, the verdict was generated automatically by comparing the function call stack with that of the error trace previously analyzed for an earlier version of the Linux kernel for rule specification ID *32_7a*, module *drivers/staging/rtl8188eu/r8188eu.ko* and entry point *ldv_main55*. Knowledge Base compares error traces in terms of function call stacks by default. The user can compare error traces in terms of function call trees or can design own comparison scripts.

The Web interface of LDV Analytics Center can be easily shared between several users. This makes it possible to considerably reduce efforts on analysis of static verification results obtained.

3.6. Intermediate Conclusions

LDV Tools can be configured at each stage of its operation. The user can decide how to divide source files of the kernel and modules and the corresponding

Table 1. Distribution of false alarms over their causes for three rule specifications and all modules of Linux kernel 3.12-rc1

Cause of false alarms \ Rule specification	Correct usage of mutexes in one thread[13]	Correct release of USB request blocks[14]	Correct registration of USB gadget devices[15]	Total
Lack of source code of interrelated modules	4	29	5	38
Inaccurate environment model	24	43	7	74
Inaccurate rule specification	18	6	3	27
Inaccurate analysis by a static verification tool	17	34	5	56
Total	63	112	20	195

[13] See http://forge.ispras.ru/issues/1940.
[14] See http://forge.ispras.ru/issues/3233.
[15] See http://forge.ispras.ru/issues/2742.

build commands into verification objects, refine the specification of the π-model of environment, modify available rule specifications, as well as develop new ones, and specify a particular static verification tool, its configuration, and resource limits to be used for checking Linux kernel modules. Thus, LDV Tools fully implements the method proposed in Section 2.

4. PRACTICAL APPLICATION OF LDV TOOLS

LDV Tools has been used in practice for more than four years. During this time, LDV Tools helped to find more than 150 bugs in Linux kernel modules and revealed the most critical problems in the static verification toolset and static verification tools used.

4.1. Analyis of Bugs Found in Linux Kernel Modules

Thus far LDV Tools helped to find 150 bugs that were acknowledged by the developers of the Linux kernel[5] [34]. Figure 8 shows the number of bugs fixed in Linux kernel modules versus the version of the kernel. On the average, for the kernel versions from 2.6.31 (September 9, 2009) to 3.15-rc2 (April 20, 2014), about six bugs were fixed in each version. Moreover, the number of fixed bugs is increasing with time. This is due to the fact that, after the four years, LDV Tools has reached such a high level of maturity that increasingly intensifies its application by the developers and to the fact that the list of rules being checked is gradually expanding. Today, LDV Tools allows to check more than 40 rules for correct usage of the Linux kernel API.

Note that, in the last one–two years, LDV Tools helped to find more bugs in Linux kernel modules than the developers of the toolset manage to analyze and report to authors of the corresponding modules. This obviously demonstrates that LDV Tools have a rather high potential in finding new bugs in kernel modules.

Figure 9 shows, using cumulative sum, the dependence of the number of bugs fixed in Linux kernel modules on the version of the kernel for various subsystems of the kernel. It can be seen that, on the whole, the number of bugs fixed in the subsystems remains to be proportional. The jumps on the plots are explained by changes in the code base being analyzed. For example, for the network subsystem of Linux kernel 3.6, the jump occurred because LDV Tools were used to verify, in addition to modules that represent drivers of network devices, modules incorporated into the main network subsystem of the kernel. New kernel modules that already have rather high levels of quality but, for some technical reasons, cannot yet be placed into one of main subsystems find themselves in the staging subsystem. Since there may be a great number of bugs in new modules, the behavior of the corresponding plot is quite unpredictable.

The diagram in Fig. 10 shows the distribution of potential impacts of the found bugs. It can be seen that, in most cases, we manage to avoid resource leaks[6]. The second and third places are occupied by deadlocks[7] (as a result of unreleasing of synchronization primitives in a thread in which they were acquired) and race conditions[8] (as a result of releasing of synchronization primitives in a thread in which they were not acquired), respectively. Besides some bugs that have specific impact on Linux kernel modules were fixed. For example, sleeping in atomic context may considerably slow down operation of the kernel and, sometimes, the entire OS may hang.

[5] In addition to bugs acknowledged by the Linux developers, bugs that were fixed by that time were found along with bugs in unsupported kernel modules. These bugs are not included into statistic data presented in this paper.

[6] See definition of a memory leak on http://cwe.mitre.org /data/definitions/401.html.
[7] See definition of a deadlock on http://cwe.mitre.org/data/definitions/833. html.
[8] See definition of a race condition on http://cwe.mitre.org/data/definitions/362. html.

Table 2. Winners of the annual International Competition on Software Verification on the DeviceDrivers64 benchmark set

DeviceDrivers64 benchmark set	Place 1	Place 2	Place 3
SV-COMP 2012 41 tasks, maximum 66 points	BLAST 55 points, 1400 s	CPAchecker-Memo 49 points, 500 s	SATabs 32 points, 3200 s
SV-COMP 2013 1237 tasks, maximum 2419 points	UFO 2408 points, 2500 s	CPAchecker-Explicit 2340 points, 9700 s	BLAST 2338 points, 9700 s
SV-COMP 2014 1428 tasks, maximum 2766 points	BLAST 2682 points, 13000 s	UFO 2642 points, 5700 s	FrankenBit 2639 points, 3000 s

Most of the bugs (about 70%) were found in error handling code, e.g., after unsuccessful memory allocation or problems occurred when initializing a device. This is explained by the fact that, when using and testing the OS, such situations are quite rare, while static verification considers all possible execution paths.

Almost all bugs were found by using the BLAST static verification tool [30]. This is due to the fact that this tool is used in LDV Tools by default and that it was purposely optimized to analyze Linux kernel modules. Four bugs were found with the help of the CPAchecker static verification tool [31], which offers extra opportunities to analyze function pointers and bit arithmetic as compared to BLAST.

In several cases, based on discussions on found bugs with the developers of Linux kernel modules, they revised the design of corresponding subsystems of the kernel or modules.[9] The bugs found often occur because of incorrect error handling[10] or uninitialized data[11] (memory leaks were found initially).

In theory, other approaches to ensure software quality could find bugs detected by LDV Tools. Some of these bugs could manifest themselves when using the Linux kernel or in the course of testing. However, on the one hand, this can take much time, including time on designing specific test scenarios, since most of bugs were found in error handling code. On the other hand, for bugs like resource leaks and race conditions, it is quite difficult to exactly find what causes these bugs.

Probably, most of the bugs could be found by tools implementing lightweight methods of static code analysis. To do this, appropriate formal descriptions of rules for correct usage of the Linux kernel API should be developed; then, tools should be launched, and results should be analyzed. Still, some bugs may be missed (for example, several dozens of bugs were found by interprocedural analysis and by analysis of a control flow with complex dependencies, which is supported to a limited extent by lightweight approaches to static code analysis) and analysis of results may be complicated by a great number of false alarms.

4.2. Analysis of Causes of False Alarms

In experiments described in this and the following three subsections, LDV Tools 0.5 was launched on a computer with Intel Core i7-2600 (4 cores, 3.4 GHz), 16 GB RAM, and OS Ubuntu 12.04 (Linux kernel 3.5, 64-bit architecture). Static verification tool BLAST 2.7.2 was used in the default LDV configuration. The maximum amount of CPU time and RAM to be used by BLAST for solving one verification task were set to be 15 min and 15 GB, respectively.

Table 1 gives the distribution of false alarms over their causes for three rule specifications. To construct this distribution, we analyzed static verification results for all modules of Linux kernel 3.12-rc1 (about 4200 modules[12]). It can be seen from Table 1 that there are few modules for which false alarms were produced with respect to the total number of analyzed modules (on the average, about 1.5%). Primary causes of false alarms are as follows:

- Lack of source code of interrelated modules, e.g., when a module being analyzed calls functions defined in other modules.
- Inaccurate environment model; e.g., module callbacks are called in a wrong sequence.
- Inaccurate analysis by BLAST, e.g., due to incomplete analysis of aliases.

4.3. Analysis of Causes of Missed Bugs

Static verification methods were initially focused on finding all possible bugs of a certain type or on proving correctness of analyzed programs against particular rules. Nevertheless, for various reasons, bugs can be missed.

To evaluate the number of missed bugs, LDV Tools was run on 34 modules of the Linux kernel of various versions where there were well-known violations of rules

[9] Discussion of bugs found in network device drivers is available on https://lkml.org/lkml/2012/8/14/128.

[10] See an example of considerable fixes in a driver at http://linuxtesting.ru/results/report?num=L0130.

[11] See an example of a fix in initialization of driver structures at http://linuxtesting.ru/results/report?num=L0116.

[12] In LDV Tools, several environment models can be generated for a module. The results are presented taking into account all environment models (see Table 1 and further))

for correct usage of the kernel API. We managed to find 16 bugs using BLAST and 14 bugs using CPAchecker.[13] BLAST found the same bugs as CPAchecker. CPAchecker could not find two bugs found by BLAST due to lack of time.

For the static verification tools used, primary causes of missing other bugs are the lack of source code of interrelated modules in verification tasks (5), inaccurate environment model (5), inaccurate rule specifications (3), bugs in static verification tools (4 for BLAST and 3 for CPAchecker), out-of-memory exceptions (1 for BLAST), and timeouts (2 for CPAchecker).

4.4. Analysis of Verification Time for Linux Kernel Modules

The experiments showed that verification of all drivers that are delivered with Linux kernel 3.12-rc1 and that can be represented as modules (about 3300 modules) requires, on the average, about 25 hours of CPU time for one rule specification. Verification of all kernel modules required about 36 hours of CPU time. Approximately 70% of the overall analysis time was consumed by the static verification tool.

Note that LDV Tools do not operate in the parallel mode, since static verification tools require great amount of RAM; therefore, the number of processor cores slightly affect the overall time of verification.

4.5. Analysis of Causes of Unsuccessful LDV Tools Terminations

Out of all modules of Linux kernel 3.12-rc1 with all generated environment modes, we fail to verify about 800 modules (approximately 15% of the total number) due to the following reasons: bugs in Command Stream Divider (about 12% of 800), bugs in Driver Environment Generator (about 11%), bugs in Rule Instrumentor (about 19%), bugs in BLAST (about 29%), out-of-memory exceptions (about 22%), and timeouts (about 7%) for BLAST.

4.6. Capabilities of LDV Tools in Comparing Static Verification Tools and Their Configurations

LDV Tools can be used to compare static verification tools and their configurations on large, complex, and dynamically evolving programs that are Linux kernel modules [35, 36]. Thus, for static verification of Linux kernel modules, one can select those tools and their configurations that are optimal in terms of decreasing the amount of missed bugs, false alarms, and consumed resources. In addition, difficulties in practical application of static verification tools can be revealed, e.g., certain bugs in tools or lack of information to visualize error traces.

[13] SVN 8244 revision, the default configuration used in LDV Tools.

In the last few years, using LDV Tools, the DeviceDrivers64 benchmark set was prepared for the annual International Competition on Software Verification [12, 13]. For DeviceDrivers64 the developers of LDV Tools selected verification tasks for which violations of rules for correct usage of the kernel API were found or for which static verification tools needed a large amount of resources. Since 2013, DeviceDrivers64 became the largest set comprising more than a half of all the present verification tasks.

Table 2 gives the number of verification tasks in the DeviceDrivers64 benchmark set, winners of the annual International Competition on Software Verification on DeviceDrivers64, and their final results (scores and time). These results show which static verification tools should be used for checking Linux kernel modules.

4.7. Intermediate Conclusions

Analysis of practical application of LDV Tools clearly demonstrates that static verification tools can advantageously be employed to find violations of rules for correct usage of the Linux kernel API. Moreover, the following most essential problems were identified in LDV Tools itself:

- Lack of source code of interrelated modules in verification tasks.
- Inaccurate specification of the π-model of environment.
- Inaccurate rule specifications.
- Bugs in components and subcomponents of LDV Tools,

as well as in the used static verification tools:

- Inaccurate analysis of aliases, functional pointers, bit arithmetic, etc.
- Bugs in static verification tools, e.g., when parsing input source files.
- Resource overconsumption.

In the future, by solving these problems, the number of missed bugs and false alarms can be considerably reduced, unsuccessful terminations of the static verification toolset can be overcome, and its performance can be improved.

5. CONCLUSIONS

Modern methods and tools for static verification already can prove feasibility of specified properties for medium-scale programs in a reasonable time. Thus, these tools can be used to find all violations of rules for correct usage of a kernel API in modules. Static verification of a great number of complex Linux kernel modules showed importance of solution of many engineering problems without which it is impossible to advantageously use even the most advanced static verification methods.

In the proposed static verification method, one can configure a verification process at all its stages. Due to

this fact, LDV Tools implementing this method continues to advance and has already helped to find 150 critical bugs in Linux kernel modules.

In this paper, we showed that Linux kernel modules are a unique domain for benchmarking static verification tools to find optimal, in terms of certain criteria, tools and their configurations. On the other hand, the successful results of such a comprehensive approach are a new driving force for developing methods and static verification tools themselves.

Practical application of LDV Tools revealed problems that define the following main directions for further development:

- Analyzing groups of interrelated modules.
- Refining the specification of the π-model of environment.
- Refining present rule specifications.
- Fixing bugs in components and subcomponents of LDV Tools.

Moreover, we intend to pay attention to the following:

- Analysis of commits to Linux kernel modules to find new critical rules for correct usage of the kernel API and to develop new rule specifications.
- Integration of new static verification tools and search for their optimal configurations to check particular rules.
- Multi-aspect static verification of Linux kernel modules to find all possible violations of several rules in a single launch of a static verification tool.
- Static verification of kernel modules for various architectures, such as, for example, ARM, MIPS, S/390, and PowerPC, as well as for most frequently used configurations of the kernel.
- Application and development of methods for regression static verification to verify changes in source code of the Linux kernel and modules.
- Parallelization of static verification of Linux kernel modules.
- Application of the proposed static verification method to check kernel components of other OSs.

Developers of static verification tools are invited to develop more accurate methods of analysis, to fix bugs in their tools, and to optimize their methods to reduce resource consumption.

Current achievements, problems, prospects, and plans in the field are discussed on a regular basis at the Linux Driver Verification workshops and ETAPS competitions on software verification where the most active role is played by the Institute for System Programming of the Russian Academy of Sciences (the developer of LDV Tools) and the University of Passau (the main developer of the CPAchecker static verification tool).

ACKNOWLEDGMENTS

We thank P. Andrianov, V. Gratinskii, V.A. Zakharov, M. Makienko, V. Mordan', O. Strikov, A. Strakh, P. Shved, and I. Shchepetkov for their active participation in designing and developing LDV Tools.

This work was supported by the Ministry of Education and Science of Russia, project no. RFMEFI61614X0015.

REFERENCES

1. Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D., An empirical study of operating system errors, *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP),* 2001, pp. 73–88. doi: 10.1145/502034.502042
2. Swift, M., Bershad, B., and Levy, H., Improving the reliability of commodity operating systems, *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 73–88. doi: 10.1145/502034.502042
3. Palix, N., Thomas, G., Saha, S., Calves, C., Lawall, J., and Muller, G., Faults in Linux: Ten years later, *Proc. 16th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 305–318. doi: 10.1145/1950365.1950401
4. Mutilin, V.S., Novikov, E.M., and Khoroshilov, A.V., Analysis of typical faults in Linux operating system drivers, *Trudy ISP RAN* (Proc. ISP RAS), 2012, vol. 22, pp. 349–374.
5. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., and Ustuner, A., Thorough static analysis of device drivers, *Proc. 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2006, pp. 73–85. doi: 10.1145/1218063.1217943
6. Glass, R.L., *Facts and Fallacies of Software Engineering*, Addison-Wesley Professional, 2002.
7. Engler, D., Chelf, B., Chou, A., and Hallem, S., Checking system rules using system-specific, programmer-written compiler extensions, *Proc. 4th Symposium on Operating System Design and Implementation (OSDI)*, 2000, vol. 4, pp. 1–16.
8. Avetisyan, A., Belevantsev, A., Borodin, A., and Nesov, V., Using static analysis for finding security vulnerabilities and critical errors in source code, *Trudy ISP RÀN* (Proc. ISP RAS), 2011, vol. 21, pp. 23–38.
9. Lawall, J.L., Brunel, J., Palix, N., Rydhof, H.R., Stuart, H., and Muller, G., WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code, *Proc. 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2009, pp. 43–52. doi: 10.1109/DSN.2009.5270354
10. Mandrykin, M.U., Mutilin, V.S., and Khoroshilov, A.V., Introduction to CEGAR: Counter-Example Guided Abstraction Refinement, *Trudy ISP RAN* (Proc. ISP RAS), 2013, vol. 24, pp. 219–292.
11. Engler, D. and Musuvathi, M., Static analysis versus model checking for bug finding, *Proc. 5th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI),* 2004, vol. 2937, pp. 191–210. doi: 10.1007/978-3-540-24622-0_17

12. Beyer, D., Competition on software verification, *Proc. 18th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2012, vol. 7214, pp. 504–524. doi: 10.1007/978-3-642-28756-5_38

13. Beyer, D., Second competition on software verification, *Proc. 19th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013, vol. 7795, pp. 594–609. doi: 10.1007/978-3-642-36742-7_43

14. Mandrykin, M.U., Mutilin, V.S., Novikov, E.M., and Khoroshilov, À.V., Static verification tools for C programs and Linux device drivers: a survey, *Trudy ISP RAN* (Proc. ISP RAS), 2012, vol. 22, pp. 293–326.

15. Corbet, J., Kroah-Hartman, G., and 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.

16. Ball, T., Levin, V., and Rajamani, S.K., A decade of software model checking with SLAM, *Commun. ACM*, 2011, vol. 54, no. 7, pp. 68–76. doi: 10.1145/1965724.1965743

17. Ball, T., Bounimova, E., Kumar, R., and Levin, V., SLAM2: Static driver verification with under 4% false alarms, *Proc. 10th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD),* 2010, pp. 35–42.

18. Ball, T. and Rajamani, S.K., SLIC: A specification language for interface checking of C, *Technical Report MSR-TR-2001-21*, Microsoft Research, 2001.

19. Ball, T., Bounimova, E., Levin, V., Kumar, R., and Lichtenberg, J., The static driver verifier research platform, *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, 2010, vol. 6174, pp. 119–122. doi: 10.1007/978-3-642-14295-6_11

20. Witkowski, T., Blanc, N., Kroening, D., and Weissenbacher, G., Model checking concurrent Linux device drivers, *Proc. 22nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE),* 2007, pp. 501–504. doi: 10.1145/1321631.1321719

21. Post, H. and Kuchlin, W., Integrated static analysis for Linux device driver verification, *Proc. 6th Int. Conf. on Integrated Formal Methods (IFM)*, 2007, vol. 4591, pp. 518–537. doi: 10.1007/978-3-540-73210-5_27

22. Clarke, E., Kroening, D., and Lerda, F., A tool for checking ANSI-C programs, *Proc. 10th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS),* 2004, vol. 2988, pp. 168–176. doi: 10.1007/978-3-540-24730-2_15

23. Clarke, E., Kroening, D., Sharygina, N., and Yorav, K., SATABS: SAT-based predicate abstraction for ANSI-C, *Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005, vol. 3440, pp. 570–574. doi: 10.1007/978-3-540-31980-1_40

24. Mutilin, V.S., Linux drivers verification with help of predicate abstractions, *Cand. Sci. (Phys.-Math.) Dissertation*, Moscow: ISP RÀS, 2012.

25. Zakharov, I.S., Mutilin, V.S., Novikov, E.M., and Khoroshilov, A.V., Environment modeling of Linux operating system device drivers, *Trudy ISP RAN* (Proc. ISP RAS), 2013, vol. 25, pp. 85–112.

26. Novikov, E.M., Development of contract specifications method for verification of Linux kernel modules, *Cand. Sci. (Phys.-Math.) Dissertation*, Moscow: ISP RAS, 2013.

27. Necula, G.C., McPeak, S., Rahul, S.P., and Weimer, W., CIL: Intermediate language and tools for analysis and transformation of C programs, *Proc. 11th Int. Conf. on Compiler Construction*, 2002, vol. 2304, pp. 213–228. doi: 10.1007/3-540-45937-5_16

28. Mutilin, V.S., Novikov, E.M., Strakh, À.V., Khoroshilov, A.V., and Shved, P.E., Linux driver verification architecture, *Trudy ISP RAN* (Proc. ISP RAS), 2011, vol. 20, pp. 163–187.

29. Khoroshilov, A., Mutilin, V., Novikov, E., and Zakharov, I., Modeling environment for static verification of Linux kernel modules, *Proc. 11th International Andrei Ershov Memorial Conference (PSI)*, 2014.

30. Beyer, D., Henzinger, T., Jhala, R., and Majumdar, R., The software model checker BLAST: Applications to software engineering, *Int. J. Software Tool Tech. Tran.*, 2007, vol. 5, pp. 505–525. doi: 10.1007/s10009-007-0044-z

31. Beyer, D. and Keremoglu, M.E., CPAchecker: A tool for configurable software verification, *Proc. 23rd Int. Conf. on Computer Aided Verification (CAV)*, 2011, vol. 6806, pp. 184–190. doi: 10.1007/978-3-642-22110-1_16

32. Albarghouthi, A., Li, Y., Gurfinkel, A., and Chechik, M., UFO: a framework for abstraction and interpolation-based software verification, *Proc. 24th Int. Conf. on Computer Aided Verification (CAV)*, 2012, vol. 7358, pp. 672–678. doi: 10.1007/978-3-642-31424-7_48

33. Novikov, E.M., Simplification of static verifier traces analysis, *Trudy nauchno-prakticheskoi konferencii Aktual'nye Problemy Programmnoi Inzhenerii* (Proc. Res. and Pract. Sci. Conf. Actual Problems of Software Engineering), 2011, pp. 215–221.

34. Institute for System Programming of RAS, Linux Verification Center, Problems in Linux Kernel. http://linuxtesting.org/results/ldv.

35. Mandrykin, M.U., Mutilin, V.S., Novikov, E.M., Khoroshilov, A.V., and Shved, P.E., Using Linux device drivers for static verification tools benchmarking, *Program. Comput. Software*, 2012, vol. 38, no. 5, pp. 245–256.

36. Beyer, D. and Petrenko, A., Linux driver verification, *Proc. 5th Int. Symposium on Leveraging Applications of Formal Methods, Verification, and Validation: Applications and Case Studies,* 2012, vol. 7610, pp. 1–6. doi: 10.1007/s10009-007-0044-z

Translated by Yu. Kornienko