Verification of Operating System Monolithic Kernels without Extensions*

Evgeny Novikov and Ilja Zakharov

Ivannikov Institute for System Programming of the Russian Academy of Sciences, Moscow, Russia
novikov@ispras.ru, ilja.zakharov@ispras.ru

Abstract. Most widely used, general-purpose operating systems are built on top of monolithic kernels to achieve maximum performance. These monolithic kernels are written in the C/C++ programming language primarily and they may exceed one million lines of code in size even without optional extensions or loadable kernel modules such as device drivers and file systems. In addition, they evolve rapidly for supporting new functionality and due to continuous optimizations and elimination of defects. Since operating systems and, in turn, applications strongly depend on monolithic kernels, requirements for their functionality, security, reliability and performance are ones of the highest. Currently used approaches to software quality assurance help to reveal quite many defects in monolithic kernels, but none of them aims at detecting all violations of checked requirements and alongside providing guarantees that target programs always operate correctly. This paper presents a new method that is based on the software verification technique and that enables thorough checking and finding hard-to-detect faults in various versions of monolithic kernels. One of its key features is the possibility to avoid considerable efforts for configuring tools and developing specifications to obtain valuable verification results while one still can steadily improve their quality. We implemented the suggested method within software verification framework Klever and evaluated it on subsystems of the Linux monolithic kernel.

Keywords: Formal Verification, Software Verification, Deductive Verification, Formal Specification, Program Decomposition, Environment Model, Operating System, Monolithic Kernel.

1 Introduction

An architecture of most widely used, general-purpose operating system kernels is either completely or mostly monolithic [1]. Like tiny microkernels, monolithic kernels usually implement such main facilities as scheduling, memory management and interprocess communication. Besides, unless specially configured, they

* The reported study was partially supported by RFBR, research project No. 16-31-60097.

also have a built-in support for low-level network protocols, security modules, servers or monitors, the cryptography API, underlying abstraction layers for different device classes and so on. One can further extend a set of monolithic kernel facilities by enabling various optional extensions or loadable kernel modules such as device drivers and file systems. There may be available thousands of such extensions but as a rule each monolithic kernel of a particular operating system instance has several dozens of them at a time. It is possible to add new extensions and remove existing ones either by recompiling monolithic kernels or loading and unloading them dynamically.

The paper focuses on verification of monolithic kernels without extensions, since many other works already address the latter [2–7]. Below for brevity we refer to monolithic kernels without extensions as *monolithic kernels*.

Sizes of typical monolithic kernels may exceed one million lines of code in the C/C++ programming language primarily. In addition, they evolve rapidly for supporting new functionality, e.g. to support new device classes or a new security model, and due to continuous improvements like optimizations and elimination of defects. For instance, from 2009 to 2016 the size of the Linux monolithic kernel grew in more than 2 times and now it exceeds 1.4 million lines of code [8].

During the boot process a monolithic kernel is loaded into memory and then it operates completely in the same address space having a full direct access to all its internal data structures as well as to all hardware. This is the main reason why the given architecture allows reaching maximum performance. As a huge drawback, even minor faults in monolithic kernels can lead to an incorrect operation, data corruption and considerable performance degradation of operating system components including monolithic kernel extensions, and, in turn, applications. Critical faults can lead to privilege escalations and confidential data breaches.

Challenge. Operating systems based on monolithic kernels operate on billions of devices[1]. That makes monolithic kernels one of the most critical software in computer systems, thus, requirements for their functionality, security, reliability and performance are ones of the highestrequirements for their functionality, security, reliability and performance are ones of the highest.

To identify defects in monolithic kernels developers and quality assurance engineers from different organizations use various methods and tools like code review, testing and static analysis [9]. However, none of these approaches aims at detecting all violations of checked requirements and providing some guarantees that target programs always operate correctly. Considering a very high importance of monolithic kernels, industry is eager for additional software quality assurance methods and tools. In some cases, e.g. for checking safety-critical computer systems based on monolithic kernels, certification authorities can specify quite rigorous requirements for such the tools [10]. However, there is a lack

[1] https://www.computerworld.com/article/3050931/microsoft-windows/windows-comes-up-third-in-os-clash-two-years-early.html

of available tools and evaluations of their applicability to fulfill such the requirements.

Below we present various formal verification methods and tools that meet the challenge.

Related Work. *Deductive verification* tends to prove the complete formal correctness of target programs. There are several quite successful projects devoted to deductive verification of microkernels [11–13]. Few works address deductive verification of small parts of special purpose and monolithic kernels [14–16]. In both cases authors show that it is necessary to do an enormous amount of manual work to develop models and specifications. Therefore, it is extremely hard to use existing specification languages, methods and tools of deductive verification for large-scale verification of target monolithic kernels since their typical sizes exceed ones of microkernels and special purpose kernels by 2-3 orders. Moreover, monolithic kernels constantly evolve that hinders deductive verification especially if yielded proofs are complex and rely on many factors.

Many researchers suggest using special programming languages and even special hardware for designing more safe and secure software, in particular operating system kernels [17–20]. This substantially simplifies formal verification but these approaches can not help for formal verification of existing general-purpose monolithic kernels.

Promising and outstanding results in formal verification of software have been achieved using *software model checking* [21] which today is often called *software verification* [22]. This technique provides a higher level of automation relatively to deductive verification. Software verification already has many successful applications regarding operating system monolithic kernels and their extensions including:

- Verification of operating system device drivers [2–7].
- Verification of network protocols [23].
- Verification of file systems [24, 25].
- Verification of a Linux kernel memory management subsystem [26].

Contribution. Thus far researchers focus on verification of specific subsystems of monolithic kernels providing appropriate specifications and tools that do not suit for other subsystems. Moreover, nobody takes care of reusing and updating tool configurations and specifications for different versions of monolithic kernels.

This paper presents a new method that is based on the software verification technique and that enables thorough checking and finding hard-to-detect faults for various versions of monolithic kernels. The method allows avoiding considerable efforts for configuring tools and for developing specifications to obtain valuable verification results by means of:

- Verification of monolithic kernel subsystems together with extensions that use their interfaces.

– Reusing specifications developed for verification of monolithic kernel extensions.
– A high level of automatization of routine operations at various steps of the software verification workflow.

Besides, the suggested method remains room for improving verification results quality, i.e. for reducing the number of false alarms and the number of missed faults of various kinds. Primarily one can achieve that by means of developing specifications. Sometimes it may be necessary to adjust tool configurations.

Paper Outline. Before proceeding to the suggested method we give more details on monolithic kernel internals and on capabilities of software verification methods and tools (Section 2). In Section 3 we present a new method for verification of monolithic kernels. Section 4 describes the implementation of the suggested method. Its evaluation on subsystems of the Linux monolithic kernel is given in Section 5. Section 6 presents conclusions and future works.

2 Background

Both operating system monolithic kernels and software verification methods and tools are extremely wide areas of research and development. In this section we consider only those aspects that are vital for verification of operating system monolithic kernels.

2.1 Operating System Monolithic Kernels

Traditionally one considers monolithic kernels as several abstraction layers which often are referred to as subsystems. An actual implementation of these layers often does not fit well their abstract representations. Monolithic kernel subsystems can be tangled in an intricate way since this may be more efficient from the practical point of view and easier for development.

In this study we rely on the fact that target monolithic kernels are developed over decades and their current code bases are already mature and well organized. In particular, most likely developers already put closely related functionalities, that form subsystems, into corresponding groups of source files and perhaps directories. For instance, some group of source files constitutes a memory management subsystem, other source files are responsible for a particular network protocol, all source files from some directory form a subsystem for supporting some class of devices, and so on.

Each monolithic kernel subsystem has an API decorating implementation details. For instance, top-level subsystems define system calls that applications invoke for using facilities of monolithic kernels and underlying hardware. As a rule, at the bottom there is a hardware abstraction layer that introduces a uniform API for various devices for the rest monolithic kernel subsystems and extensions.

Middle-level subsystems implement either a number of interfaces, such as helper functions used in other subsystems and extensions, or event-driven APIs by registering event handling callbacks. Such events include software and hardware interrupts. Also, callbacks can be invoked in more implicit ways, e.g. on expiration of timers or during execution of queued works. It is worth noting that monolithic kernel extensions are similar to middle-level subsystems, but their APIs and interrelations are usually simpler than subsystem ones. In particular, mechanisms for defining, registering and unregistering callbacks in subsystems are the same as in extensions [27].

In order to allocate required resources and to subscribe for handling events, monolithic kernels initialize all subsystems in an appropriate order on loading into memory. This process is not so straightforward due to the necessity to ensure that event handlers are registered in advance to invocation:

– There are subsystems or some parts of subsystems to be initialized first of all. This is the case for, say, memory management and scheduling. Usually monolithic kernels perform such initialization in startup functions such as *start_kernel* in the Linux kernel and *init386* in the FreeBSD kernel.
– Most subsystems and subsystem parts are initialized in accordance with their levels. For instance, the Linux monolithic kernel of version 3.14 has 19 such the levels[2]. Initialization of its subsystem for supporting PCI devices leverages 6 of them starting from registering a PCI bus and finishing by registering file attributes for PCI devices. Monolithic kernels provide different mechanisms to set initialization levels for particular subsystem interfaces There are dedicated macros often, e.g.:
  - in the Linux kernel macros *postcore_initcall*, *arch_initcall*, *subsys_initcall*, etc. take corresponding initialization function names, e.g.:

```
postcore_initcall(pci_driver_init);
arch_initcall(acpi_pci_init);
subsys_initcall(pci_slot_init);
fs_initcall_sync(pci_apply_final_quirks);
device_initcall(pci_proc_init);
late_initcall(pci_resource_alignment_sysfs_init);
```

  - in the BSD based kernels such as FreeBSD, NetBSD and Darwin there is macro *SYSINIT* that takes corresponding initialization function names, their levels and orders within levels, e.g.:

```
enum sysinit_sub_id {
  ...
  SI_SUB_VNET_PRELINK = 0x1E00000, /* vnet init before modules */
  ...
  SI_SUB_VNET         = 0x21E0000, /* vnet 0 */
  ...
  SI_SUB_VNET_DONE    = 0xdc00000, /* vnet registration complete */
```

[2] One can see files *include/linux/init.h* and *init/main.c* for details.

```
  ...
}
enum sysinit_elem_order {
  SI_ORDER_FIRST      = 0x0000000, /* first*/
  SI_ORDER_SECOND     = 0x0000001, /* second*/
  ...
  SI_ORDER_ANY        = 0xfffffff  /* last*/
};
SYSINIT(vnet_init_prelink, SI_SUB_VNET_PRELINK, SI_ORDER_FIRST, ...);
SYSINIT(vnet0_init, SI_SUB_VNET, SI_ORDER_FIRST, ...);
SYSINIT(vnet_init_done, SI_SUB_VNET_DONE, SI_ORDER_ANY, ...);
```

– Some subsystems trigger initialization of parts of other subsystems or even their complete initialization. Usually this is the case when subsystems depend on each other.

Monolithic kernels invoke callbacks when corresponding events happen after completing initialization and even during it. In turn, callbacks can refer to interfaces provided by other subsystems, e.g. for allocating and freeing resources or for acquiring and releasing locks. Each event handling execution path can pass through many subsystems and even several extensions.

Monolithic kernel subsystems operate until either normal or abnormal operating system reboot unlike extensions that can be loaded and unloaded dynamically. In particular, subsystems do not need a final clean-up, e.g. to free resources and release locks, at the end of their work.

There are many diverse requirements for monolithic kernel subsystems. In this paper, we do not consider functional requirements since one has to spend too much efforts on developing models and specifications to check them. As for non-functional requirements, monolithic kernel subsystems should invoke used interfaces properly and obey generic rules of safe programming such as an absence of null pointer dereferences or buffer overflows.

2.2 Software Verification Methods and Tools

The method suggested in the following section is based upon methods for software verification [21, 22]. In the previous work [28] we already described an interface, features and requirements of modern software verification tools like SLAM [5] and CPAchecker [29]. The fundamental limitation of these tools is the possibility to check programs of thousands or dozens of thousands of lines of code in size at most depending on the number of conditions. Thus, one needs to decompose monolithic kernels into moderate-sized subsystems to verify them independently.

An experience of leveraging software verification tools demonstrated the necessity of modeling a target program environment in a rather accurate way [2–7, 27, 30]. Software verification tools may produce false alarms at checking spurious scenarios of interactions between the program and its environment and miss faults if some paths possible during the program execution are forbidden by the

environment model. Regarding monolithic kernel subsystems, their environment mainly consists of other subsystems, various extensions, hardware and applications. The environment model should initialize subsystems, invoke registered subsystem callbacks and provide models of used interfaces which implementations are out of verification scope but which significantly influence verification results.

Software verification tools are capable to check satisfiability of safety and liveness properties. Sometimes these properties explicitly match requirements, e.g. this is the case for memory safety. Otherwise, it is necessary to formulate specific requirements as a property supported by tools. For instance, one represents rules of correct usage of a particular API as a reachability problem usually. Software verification tool can both miss faults and obtain false alarms in case of imprecise formalization of requirements.

In contrast to methods and tools for deductive verification [11–16], the software verification technique does not require developing complete models and formal specifications covering all functional and high-level requirements. It is possible to detect faults of particular kinds as well as to prove correctness under certain assumptions even having inaccurate models and specifications. This stems from the following factors:

- One does not prove the complete formal correctness of target programs but searches for violations of quite widespread non-functional requirements using software verification methods and tools. We gave examples of such requirements for operating system monolithic kernels at the end of the previous subsection.
- Software verification tools automatically build models for all functions from target programs. These models are accurate enough for checking specified requirements.
- Software verification tools make certain assumptions either by default or being configured appropriately. For instance, tools can ignore the inline assembler. One should not expect many related problems since in monolithic kernels there are not many such statements and they are concentrated in architecture dependent subsystems [8]. Otherwise, one can develop corresponding models in the C/C++ programming language.
- Researches suggest new software verification methods and optimize the implementation of existing ones. Thanks to that today tools can automatically build accurate models and check satisfiability of specified properties for medium-sized programs using reasonable computational resources.

2.3 Klever Software Verification Framework

It is hardly possible to use software verification tools out of the box for industrial programs [28]. Fortunately, there are higher-level methods and frameworks that considerably automate the entire software verification workflow [2–7, 28]. Most existing software verification frameworks target specific software like operating system device drivers [2–7]. In contrast, Klever is an extensible framework for checking various GNU C programs by design [28].

At the moment Klever is capable to thoroughly check Linux device drivers. It includes a set of specifications allowing both to generate rather accurate environment models for invoking most popular device driver APIs and to check various requirements. These specifications are also applicable at verification of the Linux monolithic kernel after slight customizations.

3 Verification of Monolithic Kernels

Following subsections consider adaptations of the common method for verification of GNU C programs [28] that we suggest for verification of monolithic kernels. Because of the limited space, we do not provide details like formats and samples of tool configurations and specifications.

3.1 Decomposing Monolithic Kernels into Subsystems

We suggest treating all source files from specified directories built into a monolithic kernel for a specified architecture and configuration as subsystems. This simplifies updates of tool configurations for new versions of monolithic kernels since developers rarely modify directories. Provided source files of different subsystems belong to the same directory, one should divide them between these subsystems explicitly.

The approach allows obtaining quite compact subsystems. If some subsystem is too complex for software verification tools at checking particular requirements, we suggest doing an additional decomposition using the same approach. One can expect several hundreds of subsystems for each monolithic kernel assuming a mean size of a subsystem to be about several thousands of lines of code.

However, our assumption that developers strictly follow separation of concerns is wrong sometimes. For instance, the same source file can contain functionalities of several subsystems. It is possible to extend a decomposition level further, e.g. by enabling enumeration of particular subsystem functions. But one has to provide and maintain different function name lists for various versions of monolithic kernels because developers change function names rather frequently. To avoid such difficulties we suggest considering source files shared by different subsystems indivisibly.

3.2 Verifying Monolithic Kernel Subsystems with Extensions

We suggest verifying monolithic kernel subsystems together with extensions that use their interfaces to avoid development of specifications considering corresponding interaction scenarios. There are several related assumptions:

- There should be environment model specifications for selected extensions to cover execution paths invoking target subsystem interfaces.
- Extensions should use subsystem interfaces correctly. Correctness of extensions is within the scope of other works [2–7].

One can select extensions using subsystem interfaces in different ways. If there is enough time and computational resources, we suggest taking all relevant extensions since this helps to cover all possible interaction scenarios. Otherwise, we propose to follow the algorithm:

- Obtain function coverage when verifying target monolithic kernel subsystems without extensions.
- Determine what subsystem functions are not covered and which of them are invoked by extensions.
- Obtain a minimal number of extensions invoking all uncovered subsystem functions or gather extensions in a greedy way.

3.3 Generating Environment Models for Monolithic Kernel Subsystems

We base the approach for generating environment models for monolithic kernel subsystems on the method we developed for modeling environment for Linux device drivers [27]. The method suggests specifying callbacks using a special domain specific language (DSL). Also, it has a hardcoded algorithm for initializing and exiting extensions occurring after loading and before unloading them respectively. Using environment model specifications and target extensions an environment model generator produces an extra C code to be verified together with a source code of these extensions.

To cope with monolithic kernel subsystems that have more complex APIs we suggest extending the existing method to support a number of DSLs for developing specifications and a corresponding number of environment model generators. These generators should prepare a final environment model as a parallel composition of its fragments generated independently for each DSL and the target program. Below we consider 3 such DSLs and environment model generators for monolithic kernel subsystems.

Modeling Initialization of Monolithic Kernel Subsystems and Extensions. It is vital to perform accurate initialization since during it resources are allocated and callbacks are registered. We suggest supporting a corresponding environment generator for specifying initialization of monolithic kernel subsystems and extensions as well as exit of extensions. It allows setting:

- Initialization levels and sublevels together with mechanisms to relate them with concrete initialization functions without referencing names of the latter.
- Initialization levels and sublevels for concrete initialization functions using their names (this is necessary for invoking those initialization functions for which initialization levels and sublevels are not set within target subsystems).
- Mechanisms to obtain exit functions for extensions without referencing names of the latter.

An environment model generator responsible for initializing subsystem and extensions and for exiting extensions:

– Obtains all initialization and exit functions defined by target subsystems and extensions with help of specified mechanisms.
– Properly orders obtained functions together with ones specified explicitly in accordance with respective levels and sublevels.
– Generates C code invoking initialization and exit functions in the calculated order taking into account failures of initialization functions if necessary[3].

Modeling Invocations of Monolithic Kernel Subsystem Callbacks. Regarding callbacks we propose to use the same approach to environment model generation for monolithic kernel subsystems as applied for verification of their extensions [27]. Often subsystems implement the same event-driven APIs as extensions, so, one can reuse existing environment model specifications.

Modeling Remaining Environment of Monolithic Kernel Subsystems. Sometimes nothing suggested above helps to cover subsystem interfaces, e.g. when they are invoked just by other subsystems or not invoked anywhere in a target monolithic kernel or considered extensions. To cover them we suggest to manually extend an intermediate environment model which is prepared at the previous stage since it is hard to suggest appropriate top-level specifications.

It may be necessary to develop models of interfaces invoked by target subsystems but which implementations are out of verification scope unless existing environment model specifications contain them. For these models we suggest using the C programming language with special expressions. The most important such expressions are functions which return non-determined values of their return value types. By using them one can force software verification tools to consider various paths, e.g. when functions both succeed and fail.

3.4 Checking Requirements for Monolithic Kernel Subsystems

We suggest checking those requirements for monolithic kernel subsystems that are vital but do not require much time for developing corresponding specifications. These requirements include memory safety and relevant for subsystems rules of correct usage of monolithic kernel subsystem interfaces that are traditionally checked at verification of monolithic kernel extensions [2–7]. Unlike extensions one should not check that subsystems perform a final clean up since they can not be unloaded on demand.

3.5 Improving Verification Results

Generated environment models and requirement specifications may be imprecise. As we discussed in the previous section it can lead to missed faults, which

[3] For instance, for the Linux kernel initialization functions can fail and return error codes. In this case, the environment model generator should not invoke exit functions if so, but can try to invoke failed initialization functions again.

is extremely undesirable, and false alarms substantially complicating verification results analysis. In addition, considering subsystems with or without extensions may be too hard for software verification tools at checking particular requirements. To improve verification results we suggest:

- To adjust tool configurations describing target subsystems and extensions verified together with them.
- To refine environment model and requirement specifications step by step until one obtains a reasonable coverage and an acceptable number of false alarms.

This process can be hardly formalized. There are strict deadlines usually, so one has to balance time spent on setting tool configurations, developing specifications, verification results analysis and preparing final accounts.

4 Implementation

We implemented the suggested method within the Klever software verification framework [28]. The implementation employs all existing Klever components and specifications intended for verification of Linux device drivers. At the moment it aims at thorough checking subsystems of the Linux monolithic kernel but it can be extended for other monolithic kernels as well.

For decomposing the Linux monolithic kernel into subsystems we allowed specifying directories and particular source files belonging to subsystems. Also, we extended a Klever component responsible for program decomposition so that it automatically triggers required build actions and filters out all required subsystem source files. Moreover, this component started to generate program fragments incorporating both subsystems and device drivers together.

We considerably extended the corresponding Klever component to generate environment models for monolithic kernels. The new version supports additional kinds of specification DSLs and generates corresponding environment model parts on their basis. In addition, we implemented a common environment model specification taking care of multilevel initialization for Linux monolithic kernel subsystems. We allowed disabling checking a final state since monolithic kernel subsystems do not need that.

5 Evaluation

For evaluating the suggested method we considered 3 subsystems of the Linux monolithic kernel that was built for architecture *x86_64* and configuration *allmodconfig* (Table 1). We limited the number of target subsystems as we did a thorough analysis of results to investigate various aspects of verification quality while analyzing only found violations does not take much time.

All experiments were conducted on OpenStack virtual machines each with 8 virtual cores of the Intel Xeon E312xx (Sandy Bridge) CPU, 64 GB of memory and Debian 9 (Stretch) on board[4]. We used Klever Git branch *kernel-verification* [28], CPAchecker Subversion revision *trunk:27583* [29] and, unless particularly pointed, those specifications and tool configurations that are used in Klever by default. In particular, CPAchecker could spend 15 minutes of CPU time and 10 GB of memory for checking each subsystem against any requirements specification. Earlier in subsection 2.2 we discussed major limitations of CPAchecker and other software verification tools as well as how these tools operate.

Table 1. Target Linux monolithic kernel subsystems (numbers of source files and lines of code are given for Linux 3.14)

Subsystem name	Directory	Source files	Lines of code
Character Devices Support (*CHAR*)	drivers/char	5	4194
General-Purpose I/O (*GPIO*)	drivers/gpio	6	4472
Terminal Devices Support (*TTY*)	drivers/tty	11	12129

5.1 Verification of Linux Monolithic Kernel Subsystems

To confirm that the suggested method meets one of its major expectations we verified target subsystems for all major versions of the Linux kernel issued from 2013, April 28 (version 3.9) to 2015, February 8 (version 3.19). This period covers almost 2 years of development and includes 11 major versions. Table 2 provides generic information on changes made in target Linux monolithic kernel subsystems.

Table 2. Changes of target Linux monolithic kernel subsystems (percentages were calculated relatively Linux 3.14)

Subsystem name	Source files added/removed	Lines of code added/removed
CHAR	+0/-1 (+0%/-20%)	+950/-712 (+23%/-17%)
GPIO	+2/-3 (+33%/-50%)	+5074/-3079 (+113%/-69%)
TTY	+1/-0 (+9%/-0%)	+4012/-3221 (+33%/-27%)

To launch Klever we used the same tool configuration and specifications for all treated versions of the Linux kernel. To get better function coverage and to get rid of annoying false alarms we made following improvements in environment model specifications that are specific for Linux monolithic kernel subsystems:

[4] http://www.bigdataopenlab.ru/about.html

– Explicitly specify initialization levels for two initialization functions from the *CHAR* and *TTY* subsystems.
– Develop a model for function *panic* that abnormally terminates kernel operation.
– Place memory allocated in environment models into global lists to avoid detection of memory leaks after termination of subsystems.

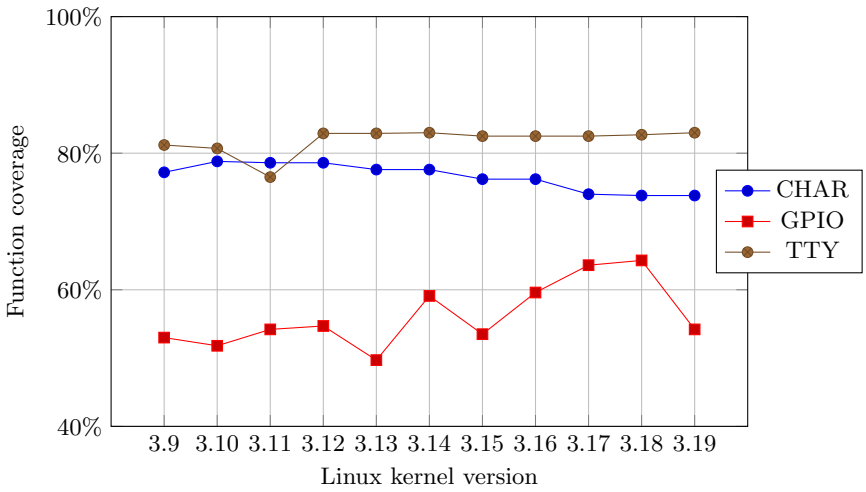

Fig. 1. Function coverage for target subsystems of the Linux monolithic kernel

Function Coverage. Fig. 1 demonstrates function coverage for target subsystems of the Linux monolithic kernel. One can see that for *CHAR* and *TTY* subsystems function coverage changes rather slightly except for Linux 3.11. In this version developers added to the *TTY* subsystem a new source file defining specific semaphores but there were no users of an introduced API at that time yet.

Function coverage for the *GPIO* subsystem changes more often and more significantly because this subsystem is relatively new. It was introduced in 2008[5] while other target subsystems have been developing from the nineties of the previous century. One can see that this correlates with numbers of added and removed source files and lines of code from Table 2.

Fig. 2 demonstrates reasons why remaining functions of target subsystems of the Linux monolithic kernel are not covered. In the most cases it is necessary to develop additional environment model specifications to invoke specific callbacks.

[5] https://lkml.org/lkml/2008/1/5/137

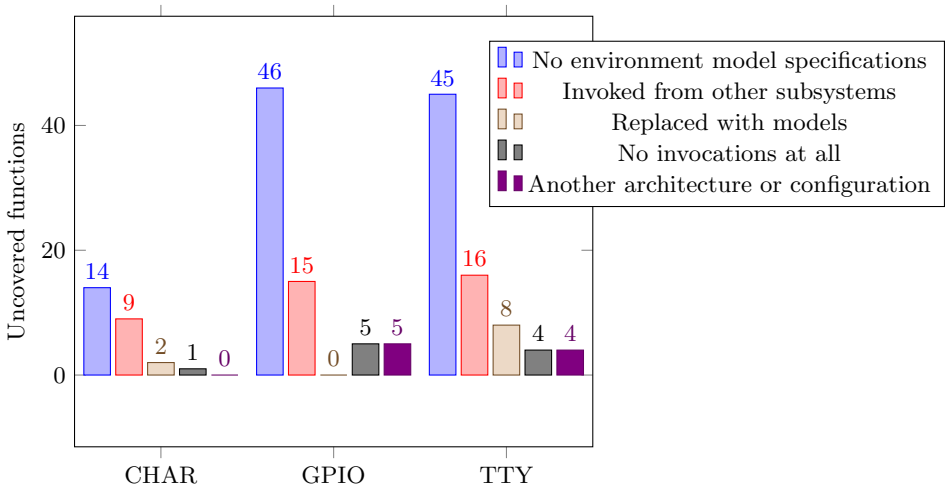

Fig. 2. Reasons of an absence of function coverage for target subsystems of the Linux monolithic kernel (for Linux 3.14)

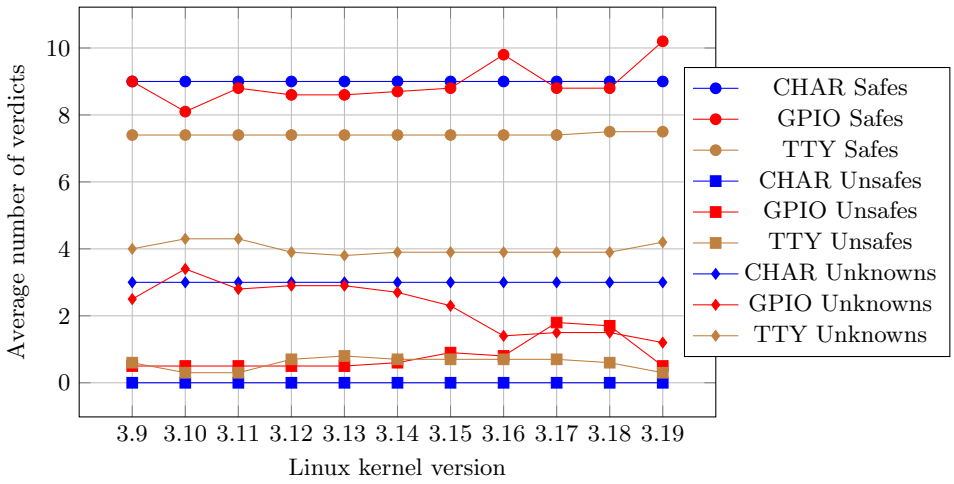

Fig. 3. Average number of verdicts for target subsystems of the Linux monolithic kernel

Verification Results. We verified all target subsystems of the mentioned versions of the Linux kernel against the most relevant requirement specifications, namely, *generic:memory*, *linux:*{*alloc:*{*irq, spinlock*}*, arch:io, drivers:base:*{*class, dma-mapping*}*, fs:sysfs, kernel:locking:*{*mutex, rwlock, spinlock*}*, kernel:*{*module, rcu:update:lock*}} (12 specifications in total).

Fig. 3 shows a dependency of obtained verdicts for each Linux kernel version. The *Safe* verdict means that the software verification tool was able to prove the absence of violations of checked requirements. The *Unsafe* verdict corresponds to a violation. The *Unknown* verdict means that the software verification tool was not able to issue either *Safe* or *Unsafe*, e.g. because of it needed more CPU time than it had. All subsystems were verified with different numbers of extensions depending on a Linux kernel version. To place plots for all subsystems in one figure, we provide average numbers of verdicts by dividing absolute ones on corresponding numbers of extensions.

One can see that *CHAR* and *TTY* subsystems are quite stable while there are several jumps on plots for the *GPIO* subsystem. Here there are additional reasons of these deviations than for function coverage at Fig. 1. For instance, in Linux 3.17 and Linux 3.18 developers violated requirements specification *linux:kernel:locking:spinlock* that was detected by Klever.

We could not find any faults for *CHAR* and *TTY* subsystems that confirms their maturity. For the *TTY* subsystem about 62% of false alarms are relevant to the subsystem itself and 38% ones are relevant to device drivers verified together with it. All false alarms issued for the subsystem are due to inaccurate specifications. For the *GPIO* subsystem about 51% of *Unsafes* correspond to faults. One fault is in the subsystem. We already mentioned it, it was introduced in Linux 3.17 and fixed in Linux 3.19. Other 2 faults were detected at error paths in device drivers and they still exist in the newest versions of the Linux kernel. Regarding *GPIO* 59% of false alarms are for the subsystem and 41% are for device drivers. To get rid of about 86% of false alarms in the subsystem it is necessary to fix existing specifications and to develop new ones. Remaining 14% ones were reported due to inaccuracies of CPAchecker.

There are several directions of development to improve verification results. The first direction is an improvement of specifications to mitigate false alarms. The second one is to simplify target subsystems by replacing complex functions with models or by splitting them into several subsystems for independent verification. This can help obtaining more *Safes* and *Unsafes* instead of *Unknowns* (timeouts).

5.2 Finding Known Faults in Linux Monolithic Kernel Subsystems

We tried to find already fixed faults in target Linux monolithic kernel subsystems for estimating the suggested method ability to perform thorough checking. We analyzed manually all commits except merging ones made to the mainline Git repository[6] between tags *v3.9* and *v3.19* to source files of target subsystems. In

[6] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

total 488 commits matched these conditions. Among them we chose commits that fix violations of requirements for which there are corresponding specifications in Klever. There were fixes of 8 such faults (Table 3).

Table 3. Faults fixed in target Linux monolithic kernel subsystems

Subsystem name	Commit hash	Requirements specification	Detection status
CHAR	08d2d00b291e	*generic:memory*	✗ (another architecture)
	b5325a02aa84	*generic:memory*	✓ (extra source files)
	61c6375d5523	*generic:memory*	✗ (another configuration)
GPIO	e9595f84a627	*generic:memory*	✓ (extra source files)
	00acc3dc2480	*linux:kernel:locking:spinlock*	✓
TTY	b216df538481	*generic:memory*	✗ (needs specification)
	07584d4a356e	*linux:kernel:module*	✓ (dead code)
	1d9e689c934b	*generic:memory*	✗ (too complex)

A fault fixed in commit *00acc3dc2480* was already considered in the previous subsection. Klever could find it without additional efforts. We analyzed target subsystems together with those parts of other subsystems that define several helper functions to detect faults fixed in commits *b5325a02aa84* and *e9595f84a627*. Klever could not find a fault fixed in commit *07584d4a356e* since it proved that corresponding code is dead.

To reveal a fault fixed in commit *08d2d00b291e* it is necessary to verify the *CHAR* subsystem for architecture *x86_32*. For a fault fixed in commit *61c6375d5523* a corresponding source file is built for another configuration rather than for *allmodconfig*. For detecting a fault fixed in commit *b216df538481* one should develop an environment model specification for work queues. We could not find out a fault fixed in commit *1d9e689c934b* because of it turned out to be too complex for the software verification tool.

6 Conclusion

Researchers and proof engineers verify formally either special purpose operating system kernels or relatively small parts of large monolithic kernels that form a basis of most widely used, general-purpose operating systems. Therefore, there is still a huge gap between one of the most critical software used by billions of people and formal verification methods and tools. As a step towards closing this gap, this paper introduces a new method that enables rather thorough checking and finding hard-to-detect faults for various versions of monolithic kernels without requiring considerable efforts for configuring tools and developing specifications. Also, the method allows improving verification results step by step.

Evaluation of the suggested method on subsystems of the Linux monolithic kernel showed that the same tool configurations and specifications are suitable for verifying subsystems of a large range of Linux kernel versions. We could

detect one fault in one of target subsystems, but there is room for improvement primarily by means of developing specifications. Also, we found 2 unknown faults in device drivers analyzed together with target subsystems. Regarding known faults, we were able to reveal 4 of 8 of them after slight adjustments. For finding the remaining faults it is necessary to perform verification for other architecture and configuration and to develop an additional environment model specification.

We encourage researchers to adapt the suggested method and its implementation for verification of other operating system monolithic kernels. But one should clearly realize that it will be necessary to spend quite a long time for developing models and specifications unless some of them were developed in advance like for operating system device drivers [2–7].

References

1. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts. 9th edn. Wiley Publishing (12 2012)
2. Zakharov, I.S., Mandrykin, M.U., Mutilin, V.S., Novikov, E.M., Petrenko, A.K., Khoroshilov, A.V.: Configurable toolset for static verification of operating systems kernel modules. Program. Comput. Soft. **41**(1) (2015) 49–64
3. Lal, A., Qadeer, S.: Powering the Static Driver Verifier using Corral. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE'14, New York, NY, USA, ACM (2014) 202–212
4. Beyer, D., Petrenko, A.K.: Linux driver verification. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. ISoLA'12, Berlin, Heidelberg, Springer-Verlag (2012) 1–6
5. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7) (2011) 68–76
6. Post, H., Sinz, C., Küchlin, W.: Towards automatic software model checking of thousands of linux modules - a case study with avinux. Softw. Test., Verif. Reliab. **19**(2) (2009) 155–172
7. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent Linux device drivers. In: Proceedings of the 22nd International Conference on Automated Software Engineering. ASE'07, New York, NY, USA, ACM (2007) 501–504
8. Novikov, E.: Evolution of the Linux kernel. Trudy ISP RAN/Proc. ISP RAS **29**(2) (2017) 77–96
9. Novikov, E.: Static verification of operating system monolithic kernels. Trudy ISP RAN/Proc. ISP RAS **29**(2) (2017) 97–116
10. Black, P., Ribeiro, A.: SATE V Ockham sound analysis criteria. NIST Interagency/Internal Report 8113 (2016) 1–31
11. Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X.N., Weng, S.C., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'15, New York, NY, USA, ACM (2015) 595–608
12. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1) (February 2014) 1–70

13. Alkassar, E., Paul, W.J., Starostin, A., Tsyban, A.: Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In: Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, Experiments. VSTTE'10, Berlin, Heidelberg, Springer-Verlag (2010) 71–85
14. Efremov, D., Mandrykin, M.: Formal verification of Linux kernel library functions. Trudy ISP RAN/Proc. ISP RAS **29**(6) (2017) 49–76
15. Ferreira, J.F., Gherghina, Cristian He, G., Qin, S., Chin, W.N.: Automated verification of the FreeRTOS scheduler in HIP/SLEEK. Int. J. Softw. Tools Technol. Transf. **16**(4) (2014) 381–397
16. Gotsman, A., Yang, H.: Modular verification of preemptive OS kernels. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. ICFP'11, New York, NY, USA, ACM (2011) 404–417
17. Azevedo de Amorim, A., Collins, N., DeHon, A., Demange, D., Hriţcu, C., Pichardie, D., Pierce, B.C., Pollack, R., Tolmach, A.: A verified information-flow architecture. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'14, New York, NY, USA, ACM (2014) 165–178
18. Leino, K.R.M.: Developing verified programs with Dafny. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE'13, Piscataway, NJ, USA, IEEE Press (2013) 1488–1490
19. DeHon, A., Karel, B., Knight, Jr., T.F., Malecha, G., Montagu, B., Morisset, R., Morrisett, G., Pierce, B.C., Pollack, R., Ray, S., Shivers, O., Smith, J.M., Sullivan, G.: Preliminary design of the SAFE platform. In: Proceedings of the 6th Workshop on Programming Languages and Operating Systems. PLOS'11, New York, NY, USA, ACM (2011) 1–5
20. Yang, J., Hawblitzel, C.: Safe to the last instruction: Automated verification of a type-safe operating system. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI'10, New York, NY, USA, ACM (2010) 99–110
21. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4) (October 2009) 1–54
22. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In Legay, A., Margaria, T., eds.: Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'17, Berlin, Heidelberg, Springer (2017) 331–349
23. Musuvathi, M., Engler, D.R.: Model checking large network protocol implementations. In: Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation. NSDI'04, Berkeley, CA, USA, USENIX Association (2004) 12–12
24. Galloway, A., Lüttgen, G., Mühlberg, J.T., Siminiceanu, R.I.: Model-checking the Linux virtual file system. In: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation. VMCAI'09, Berlin, Heidelberg, Springer-Verlag (2009) 74–88
25. Yang, J., Twohey, P., Engler, D., Musuvathi, M.: Using model checking to find serious file system errors. ACM Trans. Comput. Syst. **24**(4) (November 2006) 393–423
26. Liakh, S., Grace, M., Jiang, X.: Analyzing and improving linux kernel memory protection: A model checking approach. In: Proceedings of the 26th Annual Computer Security Applications Conference. ACSAC'10, New York, NY, USA, ACM (2010) 271–280

27. Khoroshilov, A., Mutilin, V., Novikov, E., Zakharov, I.: Modeling environment for static verification of Linux kernel modules. In Voronkov, A., Virbitskaite, I., eds.: Proceedings of the 9th International Andrei Ershov Memorial Conference on Perspectives of System Informatics. PSI'15, Berlin, Heidelberg, Springer (2015) 400–414
28. Novikov, E., Zakharov, I.: Towards automated static verification of GNU C programs. In Petrenko, A., Voronkov, A., eds.: Proceedings of the 11th International Andrei Ershov Memorial Conference on Perspectives of System Informatics. PSI'17, Cham, Springer (2017) 402–416
29. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proceedings of the 23rd International Conference on Computer Aided Verification. CAV'11, Berlin, Heidelberg, Springer (2011) 184–190
30. Engler, D., Musuvathi, M.: Static analysis versus model checking for bug finding. In Steffen, B., Levi, G., eds.: Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation. VMCAI'04, Berlin, Heidelberg, Springer (2004) 191–210