Further development of Klever in favor of verification of the Linux kernel

Klever [1-3] is a software verification framework that aims at automated thorough checking of programs developed in the GNU C programming language against specified requirements. It uses software verification tools (also known as software model checkers) implementing such methods of formal verification as Bounded Model Checking [4] and Counterexample-Guided Abstraction Refinement [5]. These tools allow finding faults that hardly can be detected by other software quality assurance techniques like code review, testing and static analysis. In addition, they are capable to prove formal correctness of programs checked against particular requirements under certain, explicitly stated assumptions.

Currently Klever supports verification of Linux kernel loadable modules, Linux kernel subsystems and BusyBox applets. This list can be extended further by developing corresponding configurations, specifications and, perhaps, appropriate Klever components that will adapt the framework for specifics of target software.

For the Linux kernel Klever includes specifications for detecting:

- Memory safety issues, e.g. buffer over-reads/writes and null pointer dereferences.
- Incorrect usages of the most popular Linux kernel API.
- Data races.

On the base of specifications Klever generates environment models for invoking:

- Interrupt and timer handlers.
- Callbacks of different device types (USB, PCI, SCSI, network, etc.).
- File system operations.
- Some other most widely used interfaces.

These environment models allow reaching more than 50% code coverage of Linux device drivers and subsystems on average. At the moment a false alarm rate ranges from 0% to 80% depending on checked requirements. One can incrementally improve verification results primarily by fixing existing specifications and developing new ones. Faults found by Klever in the Linux kernel and acknowledged by the developers are listed here.

As software verification back-ends Klever supports CPAchecker[1] [6] and Ultimate Automizer [7]. There are predefined configurations of these tools that are suitable for checking particular requirements. To obtain better verification results one can try other options. Moreover, it is possible to integrate within Klever tools participated in the Competition on Software Verification [8].

[1] CPAchecker is used in Klever by default.

For managing verification processes and for assessment of verification results Klever provides a multi-user web interface. For the former it provides means to develop and to arrange verification jobs, to start/terminate their solution and to observe a solution progress. For the latter it shows:

- Statistics like the total number of detected faults and false alarms.
- Error traces that contain statements from program entry points to places where requirements are violated as well as concrete values of variables and function arguments at every particular point.
- Code coverage demonstrating completeness of used environment models.

Experts can estimate whether issued warnings represent faults or false alarms while Klever automatically associates corresponding expert assessments with new warnings having similar error traces. In addition, experts can compare verification results obtained at different times.

This document provides possible directions for further development of Klever that can contribute verification of the Linux kernel in some way. Section 1 introduces tasks required for easier and more convenient use of Klever. These tasks are vital for newbies, but also they are helpful for advanced users and experts as well. Section 2 includes features/bugs that should be implemented/fixed to get verification results faster with better quality and fewer failures. In Section 3 there are tasks that are essentially specific for verification of the Linux kernel. All tasks are provided with descriptions, priorities and evaluation of efforts for their implementation. Evaluation assumes work of one person that has a high qualification in corresponding areas. Some tasks from Section 1 and especially from Section 2 are important for verification of different software, so we can proceed to them in advance.

1. Adaptability and usability

1.1. Automatic deployment

1.1.1. Reliable operation of services (urgent, 0.5 man-month)

At the moment deployment scripts create, start and stop a number of *init.d* services corresponding to Klever components. Some operations of these services are not reliable. For instance, starting services with some internal bugs due to, say, an invalid deployment configuration produces no error messages, stopping services can remain child processes and so on. After these issues happen, automatic deployment fails and one needs to perform some extra actions manually that is not trivial sometimes and can damage Klever installations in an unpredictable way.

We suggest switching to *systemd* and ensuring reliable operation of services in various cases. Besides, it would be great to have a command-line tool for simple checking of statuses of all Klever services as well as for some basic troubleshooting.

1.1.2. Building Docker images (high, 0.5 man-month)

Features 1.1.3 and 1.1.4 are aimed at improving native Klever deployment. A good alternative for that is to run Klever inside containers.

We suggest developing necessary configuration files and scripts that will allow building Docker images with Klever inside. This task should be completed prior to 2.5.

1.1.3. Deployment on additional distributions (normal, 1 man-month)

Klever can be deployed on Debian 9 and perhaps on some other versions of Debian and its derivatives like Ubuntu.

Customers can be interested in extending support for other distributions and various versions of these distributions. This task assumes support of automatic Klever deployment at least on Debian, Ubuntu, Fedora and openSUSE. We suggest treating 2 latest major versions of these distributions.

1.1.4. Deployment within specific production environments (normal, 1 man-month)

There is the following warning in the Klever deployment documentation:

> Do not deploy Klever at your workstation or valuable servers unless you are ready to lose some sensitive data or to have misbehaved software. We hope that one day Klever will be more safe and secure, so this warning will be redundant.

This is because Klever uses inappropriate deployment means sometimes, e.g. installing Python packages globally rather than using virtual environments, running tools from a user home directory, etc. It can be deployed and operate more or less safely just on a new installation of Debian 9 without any other specific software and services.

This feature includes using more standard means for deployment that will enable using Klever within specific production environments. Together with 1.1.3 it will need more resources as various distributions can differ considerably.

1.1.5. Security (normal, 2 man-months)

We do not care about security with Klever much while this is a very important issue for production systems.

It is required to analyze potential security issues and to mitigate them. Like the previous feature together with 1.1.3 it will need more resources as various distributions can differ considerably.

1.2. User interface

1.2.1. More means for expert assessment of verification results (urgent, 1.5 man-month)

One of primary goals of Klever UI is providing means for experts assessment of verification results. Many relevant things were already developed for evaluation of warnings, analysis of error traces and code coverage, comparison of verification results. But in some use cases they are not convenient enough and too much time is necessary for expert assessment. For instance, if experts create new marks or change existing ones, it can affect previously analyzed verification results, and after that they need additional evaluation.

We have already suggested several additional means that will help experts in all of the mentioned use cases. It is necessary to implement them.

1.2.2. Generalization and simplification (high, 2.5 man-months)

At the moment different parts of Klever UI look quite differently, and they have non-uniform management tools, e.g. for performing various actions with entries, for sorting and filtering table rows and so on. This confuses even advanced users and complicates different operations.

We need to think about and to implement a more generic and convenient UI. Since this touches too many things, this feature is rather complex.

1.2.3. Improving help (high, 1 man-month)

Klever UI has several short popup help messages clarifying some things. But many non-trivial places do not have any hints. Besides, UI lacks support for large help pages that are necessary for describing, say, development of specifications and configurations.

This feature suggests adding short popup help messages for all ambiguous places of UI as well as basic support of large help pages developed using a markup language like reStructuredText. Sources of large help pages should be developed within scope of 1.3.

1.2.4. Improving troubleshooting (high, 1 man-month)

When using Klever UI different bad things can happen, e.g. users can provide invalid configuration files or data in an outdated format. At the moment many of them are not troubleshooted well enough. That leaves users in an uncertain state since they know neither issue reasons nor things to do.

We suggest improving troubleshooting at least for those bad cases that are already well-known.

1.3. Documentation

At the moment Klever lacks documentation except for deployment instructions.

1.3.1. Tutorial (urgent, 1 man-month)

It will be extremely helpful for users to have a tutorial describing all basic things from deployment to analysis of verification results. More complicated things should be treated within other documentation.

1.3.2. Documenting top-level configuration options (urgent, 1 man-month)

To initiate new verification processes users need to adjust top-level configuration options basing on predefined or previously set ones. These options describe:

- What should be checked: particular parts of the Linux kernel, configuration, architecture.
- Requirements to be checked and specifications to be used.
- Quality of service and computational resource limitations.
- Auxiliary settings like logging levels and task priorities.

There are quite many top-level configuration options that can affect verification results very much. Usually it is hard for new users to adjust them properly.

1.3.3. Documenting checked requirements (urgent, 2 man-months)

Klever can check many requirements. To understand verification results (reasons of faults, false alarms and failures) users need a comprehensive description of these requirements that will provide:

- Descriptions of requirements.
- Consequences of requirement violations.
- Examples of requirement violations.
- Most frequent reasons of faults, false alarms and failures.

In addition, this part of user documentation should describe an overall level of support of different versions, configurations and architectures of the Linux kernel.

1.3.4. Documenting development of environment model specifications (high, 3 man-months)

Above it was mentioned that environment models generated on base of specifications allows covering about 50% of source code of the Linux kernel at the moment. Besides, inaccurate environment models result in false alarms in most cases. To check drivers and subsystems of unsupported types it is necessary to develop new environment model specifications (3.5 is devoted to this work). To reduce a false alarm rate we need to fix existing environment model specifications (3.4).

A corresponding part of user documentation should describe development of environment model specifications for the Linux kernel. 2.8 can affect this task considerably as it suggests a new format for environment model specifications.

1.3.5. Documenting development of requirement specifications (high, 1 man-month)

To check specific requirements with Klever it is necessary to develop requirement specifications. This part of user documentation should describe how to do that. It will help to fix both existing requirement specifications and to develop new ones.

1.3.6. Documenting low-level configuration options (normal, 2 man-months)

When a software verification back-end consumes too much computational resources [2] or produces false alarms because of insufficient accuracy of analysis, users can adjust different low-level configuration options to overcome these problems. We suggest treating most influencing such options. They can be related with both various Klever components and software verification tools.

1.3.7. Video guides (normal, 2 man-months)

A set of video guides with subs can assist new users. We suggest developing video guides corresponding to things covered by 1.3.1 and 1.3.2. Video guides for other use cases will need additional effort.

1.3.8. Writing developer documentation (normal, 4 man-months)

Developer documentation will describe a structure, a principle of operation and inter component interfaces at various levels of details. We have some drafts of its various parts, but they are neither completed nor up to date.

[2] When a tool consumes more computational resources than allowed by limits, there are either timeouts or out of memory errors.

2. Verification workflow improvements

2.1. Support of extended violation witnesses (urgent, 3 man-months)

Software verification back-ends output violation witnesses when they find violations of checked requirements. Klever converts these violation witnesses to error traces that are visualized for further analysis by experts. By design violation witnesses lack many details necessary for visualization and manual analysis. At the moment Klever tries to recover some of these details using ad-hoc solutions. Because of this error traces visualization is broken sometimes. We suggest supporting an extended format for violation witnesses that will not miss vital details.

In addition, we plan to get rid of some auxiliary source code transformations that break visualization of error traces even more and to enable seamless usage of violation witnesses produced by different software verification back-ends. For the latter we suggest supporting translating of those violation witnesses to violation witnesses in the extended format with help of CPAchecker. Using non-default verification back-ends will allow obtaining better verification results in some cases.

2.2. More robust specification parsers and code generators (urgent, 3 man-months)

Klever takes as input specifications developed in different DSLs and perform several source code transformations before verification takes place:

- From specifications to source code with help of environment model generators and translators.
- Source code weaving with help of C Instrumentation Framework [9].
- Slicing and source files merging with help of CIL [10].

There are quite many well-known issues with parsing of specifications and generation of code. Often these issues result in internal failures that are quite hard for understanding as a rule. Sometimes they can prevent verification at all, e.g. at the moment Klever fails to verify recent versions of the Linux kernel.

We suggest fixing all most crucial issues in specification parsers and code generators. This will help to decrease the number of internal failures 2-3 times. For new versions of the Linux kernel as well as for non-standard configurations and architectures new issues can arise that will need some extra effort for their fixing.

2.3. Fixing and optimizing CPAchecker (high, 6 man-months)

We already know some crucial issues and most suboptimal operations in CPAchecker. They can either completely break verification of some Linux kernel loadable modules and subsystems, e.g. in case of parsing failures and other internal exceptions, or slow it down very considerably.

We suggest fixing all most crucial issues and make most important optimizations in CPAchecker including:

- Fixing and optimizing implementation of the Symbolic Memory Graph technique that is used for detecting memory safety issues:
    - It is not accurate enough in some cases.
    - It is extremely inefficient.
    - It can produce spurious violation witnesses.
- Fixing implementation of the Block Abstraction Memoization technique that is used for caching intermediate analysis results:
    - There are well-known internal failures.
    - Violation witnesses can be contradictory.
    - Sometimes CPAchecker with this optimization can not output violation witnesses at all.
- Fixing detection of data races:
    - More accurate memory model.
    - Filtering of outputted violation witnesses.
- Fixing function pointer analysis that can call inappropriate functions by pointers as well as miss function calls at all.

2.4. Improving troubleshooting (high, 2 man-months)

In case of various issues with configurations, specifications and an environment Klever produces internal failure reports. Unfortunately, these reports do not describe issues well enough often. For instance, neither failure reasons nor their places are provided for incorrect specifications often. This hardens corresponding fixes very much.

One needs to have good internal failure reports that will both describe issues accurately and provide some hints on how to eliminate them.

2.5. Distributed verification (normal, 2 man-months)

To verify all device drivers (3.3) against one requirements specification Klever needs from several hours up to several days when using rather powerful nodes (say, modern CPU with 4 cores and 64 GB of RAM). We have more than 30 requirement specifications in total and new ones can be developed, so, a complete verification on a single node will take very much time.

Verification of software can be distributed within a cluster or a cloud. Klever already can schedule verification jobs and tasks at different nodes, but there is not an infrastructure for managing nodes.

This feature implies development of the distributed verification infrastructure based on Kubernetes. The infrastructure should support hot-plugging of new nodes and software verification back-ends as well as it should gracefully support various issues like network disconnects, drive breaks and so on. This feature should be implemented after 1.1.2.

2.6. Internal optimizations (normal, 1 man-month)

Klever generates verification tasks to be solved by software verification back-ends in parallel and rather efficiently. But there are several places to be optimized.

Klever components should reuse the same models generated for various verification tasks. Several inter-component delays should be minimized by using, say, message queuing. Excessive file operations should be avoided as much as possible.

2.7. Integration of additional software verification back-ends (normal, 1 man-month per a tool)

Software verification back-ends other than CPAchecker can provide better facilities at checking the Linux kernel against particular requirements.

This feature implies developing appropriate configurations for deployment and launching as well as basic evaluation of additional software verification back-ends. Using them for large-scale verification of, say, device drivers is outside of this issue. Integration of additional software back-ends should be done after 2.1 since otherwise we will have too many issues related with processing of violation witnesses.

2.8. New DSL for environment model specifications (normal, 3 man-months)

At the moment environment model specifications are JSON files with very complex syntax and semantics that have many implicit restrictions.

We suggest developing a new DSL that will simplify development of environment model specifications quite considerably. Developing an appropriate DSL editor will take 2 times more effort. This feature affects 1.3.4 quite considerably.

3. Tasks specific for verification of the Linux kernel

3.1. Support of various Linux kernel configurations (urgent, 0.5 man-month)

Klever supports Linux kernel configuration *allmodconfig* well. For this configuration most drivers are built as loadable kernel modules and the kernel has many various features enabled. Klever can verify just that kind of modules and assumes that all these features are enabled. For other configurations some drivers can be built into the kernel while the kernel can miss some features. Built-in drivers are not verified. When there are no features expected by default environment model and requirement specifications, verification fails completely.

We suggest making Klever more flexible to avoid complicated adjustment of Linux kernel configurations as well as specifications. It should start to verify built-in drivers like drivers built as loadable kernel modules. Also, Klever should adjust its specifications automatically according to enabled kernel features.

3.2. Adapting specifications for various Linux kernel versions (urgent, 1 man-month per each next major version)

Klever verifies drivers of Linux 3.14 pretty well since there are specifications developed specially for that version. To have good verification results for, say, Linux 3.15 or Linux 3.15.10 we need to adapt these specifications. Indeed, this is not necessary to support verification of all major versions and one may miss some ones. Corresponding changes in this case will be harder since there are more things to be treated. Some related work was already done for Linux 4.6.7, 4.11.6 and 4.16.10, but it was not completed.

We suggest making necessary adaptations of specifications for all target versions of the Linux kernel that are important for customers. Without appropriate user documentation (1.3.5 and especially 1.3.4) this work can be done efficiently just by Klever developers. This feature does not include fixing of Klever components that can fail to process source code of new versions of the Linux kernel (2.2). Also, this feature does not include fixing existing specifications and development of new ones (3.4).

3.3. Verification of all device drivers[3] against all specified requirements (high, 2 man-months per each next major version)

Conducting verification, analyzing verification results, preparing patches and final accounts take quite much time. Like with the previous task, skipping major versions will result in more effort.

[3] All built-in or loadable kernel modules from directory *drivers* that are built for configuration *allmodconfig* and architecture x86_64.

We suggest doing this job for all target versions of the Linux kernel. In principle, all advanced enough Klever users with appropriate documentation (1.3.1, 1.3.2, 1.3.3) can do this without our aid.

3.4. Fixing existing specifications(high, 3 man months)

There are many known issues for existing specifications of environment models and requirements. These issues result in missed faults and annoying false alarms.

Either Klever developers or advanced users with appropriate documentation (1.3.4, 1.3.5) can fix existing specifications. Fixes of specifications will reduce a false alarm rate 2-3 times. This feature does not include fixing of Klever components that can fail to process source code of new versions of the Linux kernel (2.2). Sometimes some fixes of Klever components can be necessary.

3.5. Development of new specifications (high, 0.5 man-month per each new specification)

To support verification of new types of drivers and checking of new requirements, one needs to develop new specifications.

Either Klever developers or advanced users with appropriate documentation (1.3.4, 1.3.5) can develop new specifications step by step. Concrete specifications to be developed strictly depend on customer needs:

- New environment model specifications can increase code coverage for particular types of device drivers very considerably, but on average it will not increase much as most popular interfaces are already specified.
- Similarly, new requirement specifications will allow checking specific rules of correct usage of the kernel API. Supporting checking of vital generic rules, e.g. integer overflows, will need considerable effort.

This feature does not include fixing of Klever components that can fail to process source code of new versions of the Linux kernel (2.2). Sometimes some fixes of Klever components can be necessary.

3.6. Verification of Linux kernel subsystems (high, 1 man-month per each subsystem)

Recently Klever started to support verification of Linux kernel subsystems. We already have some preliminary results for 3 subsystems [11]. But there is still much work to do.

Customers can suggest subsystems to be verified, and we can investigate the ability of this.

3.7. ARM support (normal, 0.5 man-month)

Klever supports just architectures x86 and x86_64. There are several components that depend on a target architecture much, namely, CIF and CPAchecker.

One may need support of other architectures rather than x86 and x86_64. We made some preliminary experiments for ARM, so its support will be not so hard. In case of other architectures the situation can be worse. It is better to implement this feature after 3.1 since architectures can affect configurations.

4. Tasks specific for verification of other software

Different software has different specifics that can affect the verification workflow quite considerably, thus, it has less sense to provide any particular values here. We can evaluate an amount of these specifics for one or another target program if customers will be interested in verification of particular software in addition to the Linux kernel.

Supporting verification of software written in programming languages other than C (e.g. C++, Java, etc.) will take a lot of time as we will need to change many core things in Klever. Nevertheless, this can be achieved since many software verification tools support different programming languages as well as intermediate representations like Boogie or LLVM bitcode.

References

1. Novikov E., Zakharov I. Towards automated static verification of GNU C programs. In: Petrenko A., Voronkov A. (eds) Proceedings of the 11th International Andrei P. Ershov Informatics Conference on Perspectives of System Informatics (PSI'17). LNCS, volume 10742, pp. 402–416. Cham, Springer. 2017. https://doi.org/10.1007/978-3-319-74313-4_30

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. Programming and Computer Software, volume 41, issue 1, pp. 49–64. Pleiades Publishing. 2015. https://doi.org/10.1134/S0361768815010065

3. Beyer D., Petrenko A.K. Linux Driver Verification. In: Margaria T., Steffen B. (eds) Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation. Applications and Case Studies (ISoLA'12). LNCS, volume 7610, pp. 1-6. Springer, Berlin, Heidelberg. 2012. https://doi.org/10.1007/978-3-642-34032-1_1

4. Clarke E., Biere A., Raimi R., Zhu Y. Bounded model checking using satisfiability solving. Formal Methods in System Design, volume 19, issue 1, pp. 7–34. Kluwer Academic Publishers. 2001. https://doi.org/10.1023/A:1011276507260

5. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-guided abstraction refinement. In: Emerson E.A., Sistla A.P. (eds) Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00). LNCS, volume 1855, pp. 154-169. Springer, Berlin, Heidelberg. 2000. https://doi.org/10.1007/10722167_15

6. Beyer D., Keremoglu M.E. CPAchecker: A tool for configurable software verification. In: Gopalakrishnan G., Qadeer S. (eds) Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, volume 6806, pp. 184–190. Berlin, Heidelberg, Springer. 2011. https://doi.org/10.1007/978-3-642-22110-1_16t

7. Heizmann M., Christ J., Dietsch D., Ermis E., Hoenicke J., Lindenmann M., Nutz A., Schilling C., Podelski A. Ultimate Automizer with SMTInterpol. In: Piterman N., Smolka S.A. (eds) Proceedings of 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13). LNCS, volume 7795. Springer, Berlin, Heidelberg. 2013. https://doi.org/10.1007/978-3-642-36742-7_53

8. Beyer D. Software verification with validation of results. In: Legay A., Margaria T. (eds) Proceedings of 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17). LNCS, volume 10206. Springer, Berlin, Heidelberg. 2017. https://doi.org/10.1007/978-3-662-54580-5_20

9. Novikov E.M. An approach to implementation of aspect-oriented programming for C. Programming and Computer Software, volume 39, issue 4, pp. 194–206. Pleiades Publishing. 2013. https://doi.org/10.1134/S0361768813040051

10. Necula G.C., McPeak S., Rahul S.P., Weimer W. CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool R.N. (eds) Proceedings of the 11th International Conference on Compiler Construction (CC'02). LNCS, volume 2304, pp. 213–228. Berlin, Heidelberg, Springer. 2002. https://doi.org/10.1007/3-540-45937-5_16

11. Novikov E., Zakharov I. Verification of operating system monolithic kernels without extensions. In: Margaria T., Steffen B. (eds) Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation. Industrial Practice (ISoLA'18). LNCS, volume 11247, pp. 230–248. Springer, Cham. 2018. https://doi.org/10.1007/978-3-030-03427-6_19