

Tools Support for Linux Kernel Deductive Verification Workflow

Denis Efremov

Institute for System Programming of
Russian Academy of Sciences
Moscow, Russia
efremov@ispras.ru

Nikita Komarov

Institute for System Programming of
Russian Academy of Sciences
Moscow, Russia
nkomarov@ispras.ru

Abstract—Errors in critically important systems may become very expensive. If such systems must provide confidentiality when working with some critically important data such as classified information or private know-how, an error cost may become difficult to evaluate. For these systems, formal verification methods should be used to prove they are error-free. In the paper, a case of formal verification of such system – a Linux kernel security module – is considered; the chosen toolset, the verification process workflow are reviewed, along with some auxiliary tools required for this process and developed by the authors.

I. INTRODUCTION

With the growth of software systems complexity, new requirements for their correctness and robustness emerge. Errors in critically important systems may become very expensive. For such systems, just a thorough comprehensive testing with maximum possible coverage is not enough; the only way to ensure the system is error-free is formal verification – mathematical proving of system correctness and compliance with requirements.

One of the formal verification bases is Hoare logic [1]. Its central part is so-called, Hoare triple, describing how an execution of code fragment changes the computation state. Hoare triple looks this way:

$$\{P\}C\{Q\}$$

where P and Q are predicates and C is a command. P is called precondition, Q is called postcondition. If the precondition is true, then running the command makes postcondition true too. So, the program is divided into such fragments, and for each of them there is some precondition and some postcondition. It can be proven that, if the precondition holds, after the execution of this code fragment, the postcondition holds too. Examining the whole program in such a way, it can be proven that, if the precondition for the whole program holds, after its execution, the postcondition for the whole program holds. Robert W. Floyd has also developed a similar to Hoare logic method at the same time, applied to flowcharts [2].


The seL4 microkernel [3] developers had a task of performing the formal, machine-checked verification of the seL4 microkernel. SeL4 is a microkernel of L4 family, and when it was developed, one of the main requirements for it were highest security and reliability possible. SeL4 is relatively small (about 8700 lines of code in C language, plus about 600 lines of code in assembly language). Therefore formal approach to its verification could have been used, that allowed to demonstrate design and prove C language implementation correctness.

SeL4 developers' approach to kernel development process is remarkable. As a first step, abstract kernel specification is developed. Based on it, high-level kernel prototype is developed, using a subset of Haskell functional language. This prototype is automatically translated into a executable specifications, and then compliance between formal and executable specifications is proven. Executable specification uses Isabelle/HOL prover language, which the developers have chosen for kernel correctness proof. As the next step the prototype was running on hardware simulator. This allows to evaluate kernel's design correctness.

Then, based on the aforementioned Haskell executable prototype, final kernel implementation is developed. C language was used. Kernel developers don't use Haskell code itself, because it would require a Haskell runtime environment for kernel to function. And this runtime environment is bigger than C kernel implementation. Also, because of using the low-level programming language, some additional optimizations and performance tweaks can be applied. After the development of the final kernel implementation, its correctness and compliance with the executable specification are proven.

There are, of course, some limitations to this approach. In particular, correctness of the parts of the kernel implemented in assembly language, which include some important parts such as MMU, is not proven. Also, correctness of compiler, boot loader and hardware is not proven either.

From some point of view, we are working on a quite similar task now. Linux kernel security module based on formal security model [4] and using standard Linux kernel security interfaces (LSM, Linux Security Modules) [5] with

This work is licensed under the Creative Commons Attribution License. 

some extensions has been developed. This module is a part of Astra Linux Special Edition distribution, which is security-enabled and certified for working with confidential data. The size of security module’s source code is about 4500 lines. The challenge is that the module was in an active development state at the time when its verification has started. The code wasn’t stabilized and was a subject to changes, including not only bugs fixing, but also adding some new features.

Because of the task’s importance, our team have decided to use formal verification methods. So, the task is as follows: to formally prove the correctness of Linux kernel module implementation in C language, and to prove the compliance of this implementation with the abstract security model. The correctness of security model has been verified separately. Also the multi-threaded nature of Linux kernel environment should be taken into account. In particular, this means to proof the absence of simultaneous memory access problems such as race conditions.

The paper is organized as follows: in section II brief description of instruments used for deductive verification process is given; in section III some features of the target module source code are described; in section IV verification process workflow is considered; in section V some additional tools which are developed by the authors and useful in verification process are described.

II. TOOLSET

There are a number of deductive software verification tools. All of them are used in a similar way: functions are annotated with pre- and postconditions; each loop is annotated with invariant that has to be preserved on each iteration; one can also specify some lemmas that allow tools to prove complicated statements. For functions that aren’t proven, but are used in the code under analysis, it is necessary to write pre- and postconditions only. Developing specifications for macros is usually impossible due to the fact that the tools work with the code already preprocessed. Let’s consider some of such tools.

A. *Verifast*

Verifast [6] is a tool for verification of single- and multi-threaded programs written in C or Java. This tool has been developed in the Belgian Katholieke Universiteit Leuven. It uses a modified Hoare logic, function annotations are written in its own original language. To prove the specification conditions correctness, Z3 SMT solver is used. There is also an IDE with graphical user interface. Unfortunately, Verifast is not free software. Its source code is not open, which might cause some problems to the point of inability to use the tool in certain situations, given the task complexity and non-triviality.

B. *Boogie*

Boogie [7] is an intermediate language (formerly called BoogiePL) and a tool for verification of programs developed by Microsoft Research. It is language-independent,

for now there is a translation support available for languages Spec#, C, Dafny, Eiffel and Java bytecode with BML. The tool supports verification of multithreaded programs [8]. Boogie also isn’t free, which might cause problems similar to those described in paragraph II-A.

C. *Frama-C + Why + Jessie*

Jessie [16] is designed for deductive program verification. It is a plugin for Frama-C, a platform for static analysis of programs written in C language. Frama-C is developed jointly by the two French organizations: CEA-LIST (Software Reliability Laboratory) and INRIA-Saclay (ProVal team, common with LRI-CNRS and Université Paris-Sud 11). Jessie is written in OCaml language. Most importantly, it is a free software, which allows one to fix quickly its shortcomings revealed during the tool usage. Specifications for Jessie are written in ACSL language [10]. ACSL gives the ability to develop specifications of different levels, from more abstract to more specific. Jessie uses the Why platform [9] for the purpose of verification conditions translation to the format required by the specific SMT solver. It supports a wide variety of output formats, including Coq, PVS, Isabelle/HOL, Simplify, Alt-Ergo, CVC3, CVC4, Z3 etc. To solve the problem, this particular toolset has been chosen.

Unfortunately, just the formal verification tool isn’t enough to solve the kernel module verification problem. First of all, the tools lack some features needed to verificate the module; for example, Jessie didn’t support function pointers and variable-arguments functions at the time of project start. Secondly, some other auxiliary tools are needed to simplify the task and to establish a more efficient collaboration between the specifications developers. For example, it was found out that time required for Jessie to start may become unacceptably long when working with large source code chunks such as the security module with all the Linux kernel headers it depends.

III. TARGET MODULE SOURCE CODE ANALYSIS

Initially, the security module source code has been developed without any clear plans of its subsequent verification, or even the conception of how this can be done. Therefore, the code hasn’t been limited to any C language subset. The module has been developed based on the optimal performance, not easiness of verifiability. Thus the code uses some GCC extensions of C standard actively. Moreover, because the security module is based on the Linux kernel code base, some of the code design pattern and language extensions are imposed and not always possible to give up.

So, first of all, the security module source code has been reviewed in order to identify all the features that are impossible or very difficult to work with using the verification tools. This often includes all the extensions of the C language standard, and also a well-defined set of language features [11]. This language features set may be implied by the specific verification tools, but often they

are just restrictions on the possibility to prove the code compliance with the specifications. Some coding standards include all of these limitations in advance [13] [12].

All the identified features were analyzed in respect to the possibility to avoid any usage of them in each case. Unfortunately, it was not always possible because the security module code is based on the Linux kernel code. And if some of these features are found in the kernel header files, the denial can be impossible at all. (It should be noted that verification tools often just cannot parse some unsupported language features and stop working when encounter them.) In such cases, decisions were made to add limited support of these features to verification tools. In particular, such decisions have been made on function pointers, variable-argument functions, `asm goto`, `_Bool` type support etc.

The results of this analysis of the security module source code were reported to its developers, with the emphasis on the parts that cannot be verified and need rewriting. And the parts that need to be simplified, since their proving is an almost impossible task, as well as some recommendations on coding style.

Since the verification is performed not for the entire module, but only for the part of it that is described by the mathematical model [4], some patches have been added to the project source code, introducing the preprocessor directives to make the module conditionally compile. Initially the module source code has been split into several subsystems. But no clear distinction between the part that is based on formal model and the other parts that provide additional functionality (such as logging or system calls audit subsystems) was made. These patches make the module source code easier to work with both for the verification tools (they work with the preprocessed source code) and the specifications developer. It should be also noted that the deductive verification tools are designed to work with code size of about tens of thousands of lines, and any reduction of this code base is significant for them. The security module consists of about 10 thousand lines of code, but only half of them correspond to the part that is based on formal model and requires verification.

As a next step, the authors analyzed the security module source code from the verification works plan development point of view. As the specifications are just pre- and post-conditions for functions, the module functions were analyzed with respect to the frequency of their use, their call dependency, kernel functions used by them. Although the kernel functions are not verified, preconditions and postconditions still have to be developed for them.

Macros were also investigated. It is necessary to provide some additional clarification. Verification tools work with the preprocessed code only. Therefore, writing specifications for macros is impossible. However, macros can be used in the bodies of specifications themselves, if they do not violate their syntax. For example, such macros are those replaced by the constants or references to the

structures fields. All the macros in project were classified into those that can be used in the specifications bodies and those that can't. For the second type, a recommendation was given to the developers to rewrite them as inline functions.

To collect the aforementioned information on the functions and macros of security module and kernel special software was developed. Its detailed description is given in the section V.

Function code can not be proven to comply to its specifications until all the functions it calls are verified too. Accordingly, the process of code verification starts from the bottom up to the top of the call graph. We made a map (Figure 1) of the module source code based on the module functions call graph. A special program was developed for this, a detailed description of which is given in the section V.

The map shows the amount of work to be done, provides an opportunity to develop the well-founded verification plan and helps to coordinate the people involved in verification process. In addition, it has immediately allowed to identify several errors in the conditional compilation directives and to find some sections of code that are most difficult to verify.

Based on the data that we got after the analysis of the security module source code, all the module functions were assigned one of 5 priority levels for specification. These levels were elaborated based on the current and planned language features support by instruments, so that the functions that use some unsupported features would be verified later, with respect to functionality and safety significance of individual sections of code, effort and functions dependencies.

IV. VERIFICATION PROCESS WORKFLOW

The process of kernel module source code deductive verification is as follows. As a first step, the specification for the function is developed. Secondly, the developer attempts to prove correctness of this function. Then some errors may be found in the specification and/or the source code. After that, the specification and the source code are reworked to the point when the specification can be proven. It should be also noted that the specification development, its editing and the proof are carried out in a different programs.

Currently, the verification tools can not cope with the full source code of the module (about 4500 lines of code) along with all the required kernel header files (about 70000 lines of code), because of both their size and the presence of number of language features that are unsupported for now. We can state with certainty that in the future the unsupported language features problem will be solved. However, launching the tools on the full source code will still take considerable time, and it greatly complicates the specifications proof process, because their constant

refinement with external tools and, accordingly, frequent restarts of proof tools.

To reduce these difficulties, the authors have developed the following workflow of the verification process:

- 1) Fetching all the dependencies of the module/kernel code for the function to be proven. This means collecting all the data necessary to create a separate object file, with the inclusion of definitions of all functions necessary, not just their declarations.
- 2) Developing and proving the specifications. This work is carried out with the code obtained on the step 1.
- 3) Transferring the specifications back to the security module source code after they have been proven.
- 4) A full re-proof, carried out on the whole security module source code. This, for example, is needed if the specification for one of the functions fetched on the first step has changed, so that the other functions in the security module can still be proven. Also, this is needed to make sure that the preconditions for the proven functions are held at the points of their calls.

Previously, all the work was conducted manually. Later, some instruments have been developed to perform the first and the third steps of the workflow. The last step of the workflow hasn't been executed yet, because the verification tools are now being adapted to run on the full security module source code.

V. TOOLS SUPPORT

A. A tool for source code map building

There is a number of tools that allow a developer to simplify source code navigation. Some of these tools, such as doxygen [15] and cscope [14], are also able to build call graphs. However, in the case of our project, we are faced with the fact that these tools don't work correctly with the source code of the kernel module, because it's a part of a bigger project. It is impossible to build a call graph containing only module functions, but not containing kernel functions, with them. The whole graph including kernel functions would significantly complicate the picture, because the number of nodes and edges in this graph would prevent it to be displayed with clarity. Furthermore, these tools are not able to display the entire call graph at once, only in parts.

Because of the aforementioned tools weaknesses, it was decided to develop a software for the construction of function call graphs for Linux kernel modules. You can see the graphical result of its work on Figure 1.

From the perspective of program algorithm, it may be noted that the program works with preprocessed source code. The first step is building an index of all the functions in the module source code. The second is the analysis of indexed functions names occurrence in functions bodies and the constructing of the call graph. The third is setting some additional attributes to graph vertices (such as colors marking their belonging to the priority queues). The

fourth stage is the output of the built graph in the DOT format. To build a graphical representation of the graph, *dot* program from the *graphviz* package [17] is used. The developed program imposes a restriction of the function name uniqueness in the source code.

B. A tool to gather the statistics of Linux kernel functions and macros usage

The second tool, that we use to simplify the coordination of verification efforts, project planning and management, is a mean for tracking the kernel functions and macros used by security module. These dependencies cannot be displayed on the map, as there are a number of them and their display only decrease the map's clearness.

Because the task is quite unusual, our team failed to find the software which provides a turnkey solution. Thus another software has been developed.

The software works with an unpreprocessed source code of the kernel module. It scans the entire source code, that falls under a template of function call. Then C language keywords, the security module functions and macros are removed from the collected data. The list of functions that kernel exports is created by the kernel's build system. The functions that are not in the kernel's export list, but remain in the data after filtering, are considered as static inline functions and are taken into account along with the others. Kernel's macros list is created using the C preprocessor's ability to dump all the macros definitions that it encounters during its operation. The software's final output is a table of kernel functions and macros ordered by the frequency of their calls. Also, the functions that already have a specification developed are labeled in this table.

The first two tools considered, one for working with the module source code structure and another for its external dependencies analysis, have been created as aids for the verification process. The need to create them arises from the fact that security module's source code isn't stabilized yet and is under development. This development also includes regular adaptation to the new Linux kernel releases. The tools mentioned above allow the developers to track and observe these changes, and to adapt the already developed specifications in accordance with them.

C. Slicer

During the specifications development, it's often needed to review a large amount of code at once. Constant switching between different files creates a distraction for a specifications developer. However, a function, its code and data dependencies are not always localized and located close to each other. Additionally, verification tools performance depend dramatically on the amount of code they are run on, independently of whether there are specifications for this code and even whether it's really in use.

These considerations has encouraged the authors to use a tool that would extract all the code and data

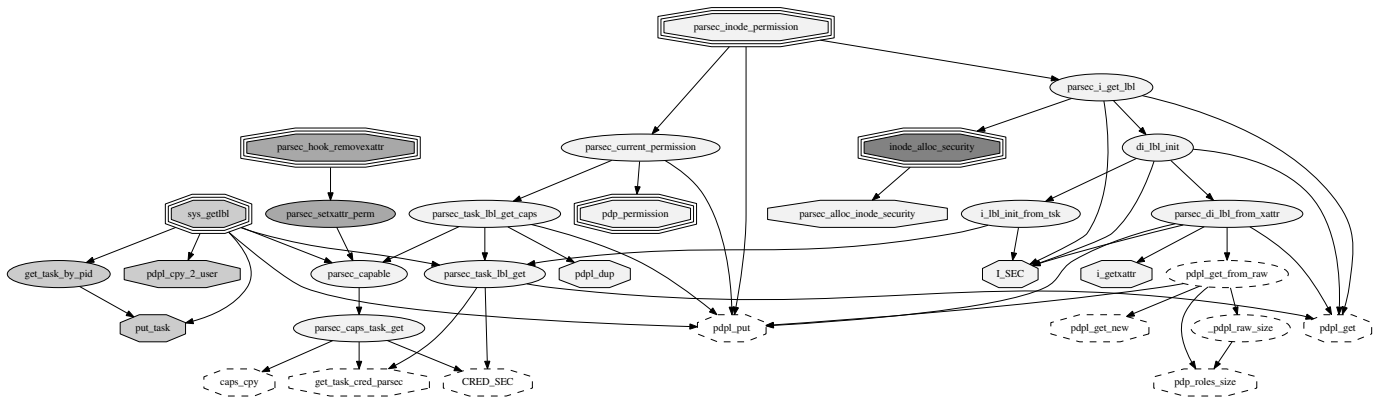


Figure 1. Map sample

dependencies for the given function from both the security module source code and the kernel header files. A study of existing open source C code slicing tools such as [18] and [19] has revealed that these instruments are not useful when working with the Linux kernel source code because of some non-standard extensions present in it. However, it isn't required for the task to analyze the possible paths of the program execution. The main requirement is to keep the original functions code structure, because its change would significantly complicate the reverse specifications transfer to the module source code.

The authors have developed a software that works with an unprocessed code without performing its full parsing. It extracts macros, structures, functions, typedefs etc. definitions and declarations based on some heuristics. It builds a global graph of these objects then by searching each object's identifier in the code of the other objects. After the graph cycle resolution procedure, the code corresponding to the graph vertices that are predecessors of the particular function is output in sorted order. The result is just one C source file with functions and several C header files: one for kernel data, one for module data, and one for the kernel specifications library. Information contained in these files is enough for the compiler to build an object file.

Despite the fact that the software is based on a number of heuristics and the output files may include redundant data in some cases (for example, if a function and a structure have the same name, they would be output, regardless of whether they are both used in the code), the software lets one to get an adequate result within a reasonable time.

D. A tool for the specifications transport

Code specifications are written as simple C comments before function declarations or definitions. After the function specification has been developed, it is required to transport it back to the full kernel module source code. This allows other developers to use this specification for their specifications development process.

We have created a software that transports the specifications from one code version to another. The software processes the two source code collections with function granularity, so its work doesn't depend on which file is which part of code located in. In the case of some specifications are already present in the old code, they are replaced with the new ones. The software recognizes the differences between code versions caused by the conditional compilation directives in the first version of the code, and automatically takes into account the absence of this code part in the second version. In the case when there are some other changes in the second part of the code, they are automatically transferred to the first. Additionally, the patch is created in this case, which can then be sent to the security module developers. When there is a conflict caused by the too many differences between code versions, which the program cannot resolve by itself, it starts an external code merge software (meld, kdiff3) to resolve them by hand.

This program is used by the authors not only as part of the verification process workflow, but also when getting a new release of the security module source code.

VI. CONCLUSION

The paper considers the organization of Linux module deductive verification process. Verification is performed in the conditions of continuing developing process of the module's code and in the absence of requirements to code written in a formal way.

During the verification activities the authors had to face restrictions of deductive verification tools and an inability to completely follow certain standards of safe coding.

The success of the code verification depends on clear organization and coordination of work. The authors have developed an approach that allows to mitigate a shortcomings of deductive verification tools and facilitate the development of specifications from the standpoint of ease of reading and analyzing the structure of the code by the developer.

Tool support necessity of worked out verification process workflow and the absence of turnkey solutions led to the development of additional tools. These tools have been used successfully by the authors and bring the results in the form of verification problem solving approach systematization and workflow stabilization.

REFERENCES

- [1] C. A. R. Hoare, *An Axiomatic Basis for Computer Programming*, Communications of the ACM, Vol. 12, Num. 10, October 1969.
- [2] Robert W. Floyd, *Assigning Meanings to Programs* Proceedings of Symposia in Applied Mathematics, Vol. 19, 1967.
- [3] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, *seL4: Formal Verification of an OS Kernel*, Proceedings of the 22nd ACM Symposium on Operating Systems Principles, 2009.
- [4] Pyotr Devyanin, *The Administration of System in Course of Mandate Entity-Role DP Access and Data Flow Control Model within Linux Family OS*, Mathematical Basics of Computer Security, Issue 4 (22) 2013, in Russian.
- [5] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, Greg Kroah-Hartman, *Linux Security Modules: General Security Support for the Linux Kernel*, USENIX Security 2002.
- [6] Bart Jacobs, Frank Piessens, *The VeriFast Program Verifier*, Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.
- [7] K. Rustan, M. Leino, *This is Boogie 2*, Microsoft Research, Redmond, WA, USA, June 2008.
- [8] Shuvendu K. Lahiri, Shaz Qadeer, Zvonimir Rakamari, *Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers*, Computer Aided Verification '09, February 2009.
- [9] Francois Bobot, Jean-Christophe Filliatre, Claude Marche, Andrei Paskevich, *Why3: Shepherd Your Herd of Provers*, Boogie 2011: First International Workshop on Intermediate Verification Languages, August 2011.
- [10] *ACSL: ANSI/ISO C Specification Language* CEA LIST and INRIA, 2009-2013.
- [11] Gerard J. Holzmann, *The Power of Ten – Rules for Developing Safety Critical Code*, IEEE Computer, June 2006, pp. 93-95.
- [12] *PL Institutional Coding Standard for the C Programming Language*, Laboratory for Reliable Software (LaRS) at the Jet Propulsion Laboratory, California Institute of Technology, March 2009.
- [13] *Guidelines for the use of the C language in critical systems*, Motor Industry Software Reliability Association (MISRA), MISRA-C: 2012, March 2013.
- [14] *Cscope – a developer’s tool for browsing source code*, <http://cscope.sourceforge.net/>
- [15] *Doxygen – generate documentation from source code*, <http://www.stack.nl/~dimitri/doxygen/>
- [16] *Krakatoa and Jessie: verification tools for Java and C programs* <http://krakatoa.lri.fr/>
- [17] Emden R. Gansner and Stephen C. North, *An open graph visualization system and its applications to software engineering*, Software – Practice and Experience, 2000, vol. 30, num. 11, pp. 1203–1233
- [18] *Frama-C - Slicing plug-in*, <http://frama-c.com/slicing.html>
- [19] Wisconsin Program, *Slicing Tool 1.1 Reference Manual*, http://www.cs.wisc.edu/~wpis/slicing_tool/slicing-manual.ps