Replication Package for On-The-Fly Decomposition of Specifications in Software Model Checking

Sven Apel [1], Dirk Beyer [1], Vitaly Mordan [2], Vadim Mutilin [2], and Andreas Stahlbauer [1]

{mordan, mutilin}@ispras.ru {apel, andreas.stahlbauer}@uni-passau.de {dirk.beyer}@sosy-lab.org

[1] University of Passau, Germany
[2] Institute for System Programming (ISP RAS), Russia

ABSTRACT

We provide a *replication package*, which is *useful* for reproducing our experimental results, or for conducting other research endeavors: All verification tasks, tools, and scripts for automatically re-running our experiments are included. To ensure that it is easily *usable*, we made it publicly available on a supplementary Web site, and provide detailed instructions for re-running the experiments; our tools are available for download, and are easy to install and to execute. An implementation of our approach is part of the open-source software verification framework CPAchecker; its source code is freely usable for research and industry under the Apache 2 license. The verification tasks have been derived from 4336 Linux kernel modules, and a set of safety properties that define the correct usage of the Linux API. The replication package provides additional *insights* on the relevance of different properties for different Linux kernel modules.

Table 1: Not all properties are relevant for each kernel module. The number of tasks for which a property (table header) is relevant is given for two sets of modules: the full set *All* with 4336 kernel modules, and a subset *Sub* consisting of 250 modules with, at least, two relevant properties each.

	08_1a	10_1a	32_1a	43_1a	68_1a	68_1b	77_1a	101_1a	106_1a	118_1a	129_1a	132_1a	134_1a	147_1a
Sub	29	119	129	191	31	14	2	4	19	19	20	13	43	23
All	129	783	846	1054	150	65	6	10	111	82	125	52	200	123

Artifact: Replication Package

To make our results easier to reproduce, we provide a replication package that includes all verification tasks, tools, and scripts for automatically re-running our experiments. The verification tasks have been derived from 4336 Linux kernel modules, and a set of safety properties that define the correct usage of the Linux API. An implementation of our approach is as part of the open-source software verification framework CPAchecker [1]; its source code is freely usable for research and industry under the Apache 2 license. The replication package—a large fraction of it can be reused for other research endeavours—is provided on a supplementary Web site [1] which contains detailed instructions for reproducing our experimental results.

Benchmark Suite. We evaluated our approach on a set of modules from Linux kernel version 4.0-rc1. The modules were extracted and prepared using the Linux Driver Verification (LDV) toolkit [2] [4], which also takes care of enriching the modules with an environment model of the Linux kernel. Each module has several entry functions which represent, for example, different interrupt handlers; we only consider one entry function (`main0`) per module. We use two sets of modules for our experiments: the full set with 4336 modules and a subset with 250 modules. The subset consists of randomly chosen tasks from the full set of modules, for which, at least, two properties are relevant. The preprocessed Linux kernel modules are licensed under GNU GPL 2.0.

The specification consists of 14 safety properties that are relevant for the Linux kernel; a detailed description of these properties can be found in Table 2. Not all properties are relevant for all kernel modules, for example, property `77_1a` is relevant for only two modules from the subset with 250 elements. Table 1 provides an overview on the relevance of the properties for the two sets of kernel modules. For 1989 of the kernel modules, at least one property is relevant; at least two properties are relevant for 1059 modules.

Experimental Setup. We have implemented our approach on top of the CPAchecker framework. We used revision 21027 from the branch *muauto*[3] for our experiments, with SMTInterpol as SMT solver. All experiments have been executed on machines with Linux 4.2 and Java 1.7.0_101, equipped with Intel Xeon E5-2650 CPUs and 135 GB of RAM. As we assume that a software verifier can always make use of the full memory installed on a (typical) machine, we limit runs where only single properties are verified (baseline) and runs where several properties get checked at once to the same amount of memory: 26 GB of Java heap and 30 GB for the process itself; each process was limited to use, at most, 4 cores.

Table 2: The set of safety properties that we checked for the Linux kernel modules.

Property	Description
08_1a	Each module that was referenced with `module_get` must be released with `module_put` afterwards.
10_1a	Each memory allocation that gets performed in the context of an interrupt must use the flag `GFP_ATOMIC`.
32_1a	The same mutex must not be acquired or released twice in the same process.
43_1a	Each memory allocation must use the flag `GFP_ATOMIC` if a spinlock is held.
68_1a	All resources that were allocated with `usb_alloc_urb` must be released by `usb_free_urb`.
68_1b	Each DMA-consistent buffer that was allocated with `usb_alloc_coherent` must be released by calling `usb_free_coherent`.
77_1a	Each memory allocation in a code region with an active mutex must be peformed with the flag `GFP_NOIO`.
101_1a	All structs that were obtained with `blk_make_request` must get released by calling `blk_put_request` afterwards.
106_1a	The modules `gadget`, `char`, and `class` that were registered with `usb_gadget_probe_driver`, `register_chrdev`, and `class_register` must be unregistered by calling `usb_gadget_unregister_driver`, `unregister_chrdev` and `class_unregister` correspondingly in reverse order of the registration.
118_1a	Reader-writer spinlocks must be used in the correct order.
129_1a	An offset argument of a `find_bit` function must not be greater than the size of the corresponding array.
132_1a	Each device that was allocated by `by usb_get_dev` must get released with `usb_put_dev`.
134_1a	The `probe` functions must return a non-zero value in case of a failed call to `register_netdev` or `usb_register`.
147_1a	RCU pointer/list update operations must not be used inside RCU read-side critical sections.

Evaluation and Reproducibility. Since CPAchecker is written in Java, special characteristics of a JVM have to be considered [3]. In particular, a scenario where we compare multiple launches of the JVM to a single launch, but also comparing multiple iterations to a single iteration of an algorithm, requires special care. To mitigate the effects of the just-in-time compiler of the JVM, we force the JVM to already compile most of the byte code during its startup. The initial size of the Java heap is set to the maximum.

We measure and control computing resources using BenchExec [2], a framework for reliable and accurate benchmarking, which is provided under Apache 2 licence. All requirements on the hardware, the command-line parameters for CPAchecker, and the verification tasks to use, are specified in benchmark definition files (XML).

References

[1] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.

[2] D. Beyer, S. Löwe, and P. Wendler. Benchmarking and resource measurement. In *Proc. SPIN*, LNCS 9232, pages 160–178. Springer, 2015.

[3] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proc. OOPSLA*, pages 57–76. ACM, 2007.

[4] A. V. Khoroshilov, V. Mutilin, A. K. Petrenko, and V. Zakharov. Establishing Linux driver verification process. In *Proc. Ershov Memorial Conference*, LNCS 5947, pages 165–176. Springer, 2009.

[1]sosy-lab.org/~dbeyer/spec-decomposition/

[2]linuxtesting.org/project/ldv

[3]svn.sosy-lab.org/software/cpachecker/branches/muauto/