Linux Driver Verification
(Position Paper)

Dirk Beyer[1] and Alexander K. Petrenko[2]

[1] University of Passau, Germany
[2] ISPRAS, Moscow, Russia

Abstract. Linux driver verification is a large application area for software verification methods, in particular, for functional, safety, and security verification. Linux driver software is industrial production code — IT infrastructures rely on its stability, and thus, there are strong requirements for correctness and reliability. This implies that if a verification engineer has identified a bug in a driver, the engineer can expect quick response from the development community in terms of bug confirmation and correction. Linux driver software is complex, low-level systems code, and its characteristics make it necessary to bring to bear techniques from program analysis, SMT solvers, model checking, and other areas of software verification. These areas have recently made a significant progress in terms of precision and performance, and the complex task of verifying Linux driver software can be successful if the conceptual state-of-the-art becomes available in tool implementations.

1 Overview

The Linux kernel is currently one of the most important software systems in our society. Linux is used as kernel for several popular desktop operating systems (e.g., Ubuntu, Fedora, Debian, Gentoo), and thus, the seamless workflow of many users depends on this software. Perhaps even more importantly, the server operating systems that currently dominate the market are based on Linux. Almost all (90 % in 2010) supercomputers run a Linux-based operating system. Increasingly many embedded devices such as smart phones run Linux as kernel (e.g., Android, Maemo, WebOS). This explains an increasing need for automatic verification of Linux components.

Microsoft had identified the device drivers as the most important source of failures in their operating systems. Consequently, the company has significantly increased the reliability of Windows by integrating the Static Driver Verifier (Sdv) into the production cycle. The foundations were developed in the Slam research project [1]. The Sdv kit is now included by default in the Windows Driver Kit (Wdk).

For Linux, an industry-funded verification project of the size of Sdv does not exist. But the development community is increasingly looking for automatic techniques for verifying crucial properties, and the verification community is using Linux drivers as application domain for new analysis techniques. During the

T. Margaria and B. Steffen (Eds.): ISoLA 2012, Part II, LNCS 7610, pp. 1–6, 2012.

last years, three verification environments were build in order to define verification tasks from Linux drivers: the Linux Driver Verification project [1] [23], the AVINUX project [27], and the DDVERIFY project [2] [32].

The Linux code base is a popular source for verification tasks [17, 22, 24, 25]. Linux drivers provide a unique combination of specific characteristics that attract researchers and practitioners to challenge their tools. The most important benefits of using the Linux code as source for verification tasks are the following:

- the software is important – many people are interested in verification results;
- every bug in a driver is potentially critical because the driver runs with kernel privileges and in the kernel's address space;
- the code volume is enormously large (10 MLOC) and continuously increases;
- the verification tasks are difficult enough to be challenging, but not too complex to be hopeless; and
- most Linux drivers are licensed as open source and therefore easy to use in verification and research projects.

Although many new advancements in the area of software verification have been made, it requires a special effort to transfer them to practice and make them applicable to complex industrial code such as Linux device drivers. The recent competition on software verification (SV-COMP'12) [3] [3] showed that even modern state-of-the-art tool implementations have problems analyzing the problems in the category on device drivers.

2 Research Directions

Pointer Analysis and Dynamic Data Structures. Many safety properties of device drivers depend on a precise analysis of pointers and data structures on the heap. The analysis of pointers is well understood, but due to the low-level code that is used in system programming, the analysis concepts are difficult to implement. The LDV project has made a significant progress on this topic with implementing a more precise pointer analysis into the software model checker BLAST [29]. This improved version of the original software model checker BLAST [5] is the SV-COMP'12 winner on the verification tasks that were derived from the Linux kernel [28].

The analysis of data structures is still an ongoing research topic, with significant progress in the last years; however, there is no large set of open benchmark verification tasks to practically compare the different implementations. The tool PREDATOR is an example of a state-of-the-art static analyzer with the ability to check data structures and memory safety [16].

[1] `http://linuxtesting.org/project/ldv`
[2] `http://www.cprover.org/ddverify`
[3] `http://sv-comp.sosy-lab.org`

Symbolic Verification. Due to the progress in SMT solving, formula-based symbolic representations of abstract states are nowadays effective and efficient. Microsoft's SDV and SLAM [1], and several current research tools are based on predicate abstraction [5, 8, 12, 18]. Several tool implementations integrate the concepts of counterexample-guided abstract refinement (CEGAR) [11], various kinds of shape analysis, abstract reachability trees [5], lazy abstraction [21], interpolation [20], and large-block encoding [4, 9]. Also bounded model checking [10] is a technique of practical relevance and with impressive results in the verification competition [14, 30].

Not yet sufficiently addressed in research projects are the problems of determining the interpolants (there is a wide range between weak and strong interpolants), block-sizes (which criterion should be used to determine the end of a block that is completely encoded in one post operation), and traversal orders (coverage-directed verification, BFS, DFS, etc.). Another important and promising technique that has been largely ignored in software verification is the possibility to encode abstract states and transition relations completely as binary decision diagrams (BDD). There was some progress on this topic, e.g., the extension of CPACHECKER and JAVA PATHFINDER to using BDDs to represent the state space that boolean variables span in code of product-line simulators [31].

Explicit-State Verification. Some explicit-state model checkers are successful in their application domain (e.g., SPIN and JAVA PATHFINDER). In order to apply this technology to the verification of driver software in a scalable manner, it would be interesting to incorporate state-of-the-art techniques that are successful in symbolic verification. For example, CEGAR should be used to automatically create an abstraction, and Craig interpolation for explicit-value domains could identify which parts of the state space are necessary to be analyzed.

Combination of Verification Techniques. In the past, several combination techniques have been proposed for assembling new analyses that are created by parallel combination of different existing analyses [7, 15]. This is extremely effective and should receive more attention and be used in practical applications. The practical application of parallel combinations is hindered by technical barriers: the two analyses have to use the same traversal algorithm, have to be implemented in the same programming language and in compatible tool environments, and need to run on the same machine at the same location (e.g., not distributed in a computing cloud).

Sequential combination using conditional model checking is an effective solution to this problem [6]. Different tools and techniques can be run one after the other, and try to solve the verification task using the various strengths. A conditional model checker is instructed when to give up (by an input condition). The input conditions represent a flexible way of bounding or restricting the verification process. Output conditions represent the state space that was successfully verified already. A successive verifier can use such conditions of previous runs to not perform the same verification work again, but concentrate on applying its strengths to the remaining task.

Termination Analysis. An area that needs more attention is termination analysis. There are a few tools for termination checking (most prominently, ARMC [26]) but the technology is not yet as wide-spread as it should be. The technology has been adopted and further improved by Microsoft's TERMINATOR project [13].

Concurrency. Due to the increasing availability of multi-core machines, the verification of multi-threaded software becomes an important research direction. Checking for race conditions and deadlocks is an essential quality assurance means that needs to be applied to Linux driver software as well. The verification community actively invents new concepts and implements new tools to approach this problem (for example, ESBMC [14], SATABS [2], THREADER [19]). It is interesting to observe that the best tool for checking concurrency problems in the last competition was a bounded model checker [14].

3 Conclusion

We outlined the motivation for considering Linux device drivers as application domain for verification research. It is important to develop verification tools that are efficient and effective enough to successfully check software components that are as complex as device drivers. The benefits are twofold: for the society it is important to get such crucial software verified; for the verification community it is important to get realistic verification tasks in order to tune and further develop the technology. We provided an overview of the state-of-the-art and pointed out research directions in which further progress is essential.

References

1. Ball, T., Rajamani, S.K.: The SLAM Project: Debugging System Software via Static Analysis. In: Proc. POPL, pp. 1–3. ACM (2002)
2. Basler, G., Donaldson, A., Kaiser, A., Kröning, D., Tautschnig, M., Wahl, T.: SatAbs: A Bit-Precise Verifier for C Programs. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 552–555. Springer, Heidelberg (2012)
3. Beyer, D.: Competition on Software Verification. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012)
4. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software Model Checking via Large-block Encoding. In: Proc. FMCAD, pp. 25–32. IEEE (2009)
5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The Software Model Checker BLAST. Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)
6. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional Model Checking: A Technique to Pass Information Between Verifiers. In: Proc. FSE. ACM (2012)
7. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program Analysis with Dynamic Precision Adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008)
8. Beyer, D., Keremoglu, M.E.: CPACHECKER: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)

9. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate Abstraction with Adjustable-Block Encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010)
10. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
11. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. J. ACM 50(5), 752–794 (2003)
12. Clarke, E., Kröning, D., Sharygina, N., Yorav, K.: SatAbs: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
13. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond Safety. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 415–418. Springer, Heidelberg (2006)
14. Cordeiro, L., Morse, J., Nicole, D., Fischer, B.: Context-Bounded Model Checking with ESBMC 1.17. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 534–537. Springer, Heidelberg (2012)
15. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of Abstractions in the ASTRÉE Static Analyzer. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 272–300. Springer, Heidelberg (2008)
16. Dudka, K., Müller, P., Peringer, P., Vojnar, T.: Predator: A Verification Tool for Programs with Dynamic Linked Data Structures. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 545–548. Springer, Heidelberg (2012)
17. Galloway, A., Lüttgen, G., Mühlberg, J.T., Siminiceanu, R.I.: Model-Checking the Linux Virtual File System. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 74–88. Springer, Heidelberg (2009)
18. Grebenshchikov, S., Gupta, A., Lopes, N.P., Popeea, C., Rybalchenko, A.: HSF(C): A Software Verifier Based on Horn Clauses. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012)
19. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: A Constraint-Based Verifier for Multi-threaded Programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 412–417. Springer, Heidelberg (2011)
20. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from Proofs. In: Proc. POPL, pp. 232–244. ACM (2004)
21. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Proc. POPL, pp. 58–70. ACM (2002)
22. Khoroshilov, A., Mutilin, V., Novikov, E., Shved, P., Strakh, A.: Towards an Open Framework for C Verification Tools Benchmarking. In: Clarke, E., Virbitskaite, I., Voronkov, A. (eds.) PSI 2011. LNCS, vol. 7162, pp. 179–192. Springer, Heidelberg (2012)
23. Khoroshilov, A., Mutilin, V., Petrenko, A., Zakharov, V.: Establishing Linux Driver Verification Process. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 165–176. Springer, Heidelberg (2010)
24. Mühlberg, J.T., Lüttgen, G.: Blasting Linux Code. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 211–226. Springer, Heidelberg (2007)
25. Penninckx, W., Mühlberg, J.T., Smans, J., Jacobs, B., Piessens, F.: Sound Formal Verification of Linux's USB BP Keyboard Driver. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 210–215. Springer, Heidelberg (2012)

26. Podelski, A., Rybalchenko, A.: Transition Predicate Abstraction and Fair Termination. In: Proc. POPL, pp. 132–144. ACM (2005)
27. 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), 155–172 (2009)
28. Shved, P., Mandrykin, M., Mutilin, V.: Predicate Analysis with BLAST 2.7. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg (2012)
29. Shved, P., Mutilin, V., Mandrykin, M.: Experience of Improving the Blast Static Verification Tool. Programming and Computer Software 38(3), 134–142 (2012)
30. Sinz, C., Merz, F., Falke, S.: LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 542–544. Springer, Heidelberg (2012)
31. von Rhein, A., Apel, S., Raimondi, F.: Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In: Proc. Java Pathfinder Workshop (2011)
32. Witkowski, T., Blanc, N., Kröning, D., Weissenbacher, G.: Model Checking Concurrent Linux Device Drivers. In: Proc. ASE, pp. 501–504. ACM (2007)