Avinux: Towards Automatic Verification of Linux Device Drivers

Hendrik Post, Carsten Sinz, Wolfgang Küchlin

Symbolic Computation Group, University of Tübingen,
www-sr.informatik.uni-tuebingen.de

Abstract. Avinux is a tool that facilitates the automatic analysis of Linux and especially of Linux device drivers. The tool is implemented as a plugin for the Eclipse IDE, using the source code bounded model checker CBMC as its backend. Avinux supports a verification process for Linux that includes specification annotation in SLICx (an extension of the SLIC language), automatic data environment creation, source code transformation and simplification, and the invocation of the verification backend. We have successfully used Avinux for the automatic analysis of Linux device drivers reducing the immense overhead of manual code preprocessing that other projects incurred.

1 Introduction

Avinux[1] is an integrated software verification tool chain that comes as an Eclipse[2] plugin which significantly improves the automation in Linux Device Driver code analysis and error checking. The code transformation tool CIL [1] and the source code bounded model checker CBMC [2] provide basic functionality. For the purpose of finding bugs in Linux source code modules, Avinux adds the tools SLICx [3, 4] for annotating the source code with typical verification conditions, DEC for the construction of abstract module environments, and CLEANC [5] for massaging CIL output into a form which is acceptable for CBMC.

With Avinux, Linux kernel distributions can be checked for race conditions, deadlocks, memory safety problems, and API conformance [3]. The analysis is context-sensitive, path-sensitive and interprocedural. The tool chain of CIL, CLEANC and CBMC contained in Avinux supports many features of the GNU C dialect used in Linux sources, such as calls through function pointers, pointer arithmetic, and side-effects. Together with the sophisticated automatic data environment creation performed by DEC, we can check actual kernel distributions with minimal manual intervention and coding.

We have used Avinux to rediscover several known errors in Linux. These errors have been reported to require extensive code simplification [6] when checked with BLAST. With Avinux, the manual effort is reduced to the construction of an

[1] Project home page: `http://www-sr.informatik.uni-tuebingen.de/~post/avinux`
[2] `www.eclipse.org`

operating system model which basically implements an abstract use case that simulates the interactions between the operating system and a device driver. Rediscovered error examples and a detailed feature description of DEC and CLEANC can be obtained through the project home page.

2 Architecture and Results

The Avinux tool chain integrates five components that are orchestrated by a plugin for the Eclipse IDE. The first two, CBMC and CIL, originate from other research groups whereas the remaining three are our contribution.

CBMC [2] is a bounded model checker for C that extracts models directly from source code. Function calls are inlined, and loops and recursion are unwound up to a user provided bound. Types are reduced to a bit-level representation that models the execution on bit-level hardware. CBMC has built-in support for memory safety checking and recognizes a specification language in the form of `assert` and `assume` statements.

CIL [1] is a code transformation framework that translates several C dialects and non-standard code constructs into ANSI-C programs.

DEC[3] is our own Data Environment Construction module. It scans a C translation unit for global identifiers that are of pointer type or contain a pointer type member. Up to a user provided bound we unroll the object graph by creating appropriate objects for each pointer. These pointers are then initialized such that at the start of every control flow every pointer points to a valid object. The procedure enables memory safety analysis in a modular context, where without data environments mostly false positives would be created because CBMC would then infer that interface pointers could be `NULL`. For DEC, we extended generic tools for data environment creation such as CUTE [8], but without covering all features of CUTE. Most importantly, we complemented the generic techniques with heuristics to infer the data structures hidden behind external pointers based on Linux typical symbol names such as `struct list_head` which indicates a doubly linked list.

CLEANC [5] is an additional code cleaning facility. Although CIL transforms most code dialects into ANSI-C, we found that several additional simplifications must be made for CBMC to accept the preprocessed code as an input. Examples include empty structs, compiler attributes and some forms of nested static function pointer initializers. A complete list of eliminated hazards can be obtained from our website.

SLICx is our specification language [4], which extends the SLIC [9] language for interface specification used in the Microsoft Static Driver Verifier. We have also implemented a SLICx compiler that transforms SLICx specification into C code which is then merged with the Linux sources to be checked. The language SLICx and the detection of race conditions, deadlocks and API violations using SLICx are described in [3].

[3] A detailed problem statement is given in [7].

2.1 The Verification Process

The following steps are necessary for analyzing a Linux device driver with Avinux:

1. Configuration of the Linux kernel so that the relevant modules are built.
2. Formulation of an interface rule to be checked as a SLICx statement.
3. Automatic annotation of all drivers with the above rule (SLICx Compiler).
4. (Automatic annotation of subsystems or other source files that are involved.)
5. Compilation of the Linux kernel CIL and additional header files.
6. Merging of all relevant source code files with CIL.
7. Automatic code simplification with CLEANC.
8. Manual creation of a `main` simulating the operating system's use of the driver.
9. Automatic creation of a data environment for `main` with DEC.
10. Running CBMC on `main`.

The verification process bottleneck of this process is step 8 because the creation of `main` has to be done for every driver. This problem has been solved for some Windows driver architectures (WDM and KMDF [10]), but Linux drivers have less standardized architectures and interfaces. For Linux, such an operating system's model cannot be implemented in a generic way. We created individual models manually for the processed examples.

2.2 Results

BLASTing Linux code [6] is a technical case study that summarizes the transformations necessary to use the model checker BLAST on real Linux Device Drivers. We have inspected their error examples and rediscovered them with Avinux avoiding the described manual code transformations. The construction

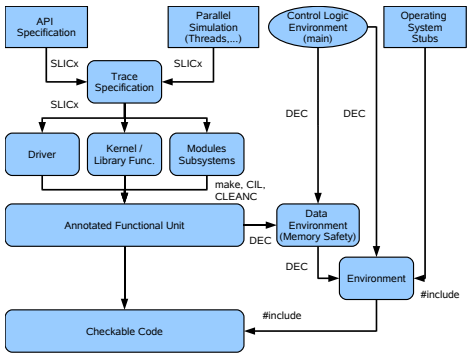

Fig. 1. The creation of a checkable unit requires multiple transformations. The ellipsoid indicates the `main` that must be created for each device driver. Round-rectangular shapes mark components that are created automatically. Rectangular shapes must be manually provided once per kernel version.

of the environment model was the only manual task required to be done for each driver. Other work as the creation of specifications and header files had to be done once per kernel version. The time needed to analyze the drivers was clearly dominated by the compilation of the Linux kernel using CIL (several hours). The analysis of a driver could be performed within minutes[4]. A preprocessed driver has about 13 k lines of code, including empty lines. The analysis by CBMC took up to several minutes, but was dominated by the model construction. We found that several SAT instances (in CNF) with up to 1 million variables and 3.6 million clauses were generated, but they could be solved within seconds.

We experienced that Avinux accelerates many verification tasks on Linux Device Drivers. Before the development of Avinux, we needed days to reproduce a known error in one device driver. Now the same task can be performed within minutes. We therefore believe that Avinux is highly effective. Verification examples can be found on our project website. Avinux is not yet available for download, but we are planning to make it available soon. Inspired by the success of SDV we believe that Avinux has great potential in improving the quality of Linux device drivers.

References

1. Necula, G.C., McPeak, S., Rahul, S., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Proc. Conf. on Compiler Construction. (2002)
2. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In Jensen, K., Podelski, A., eds.: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 2988 of LNCS., Springer (2004) 168–176
3. Post, H., Küchlin, W.: Integration of static analysis for linux device driver verification. Submitted to: The 6th Intl. Conf. on Integrated Formal Methods (IFM) (2007)
4. Meissner, F.: Regelbasierte Spezifikation von Linux Kernel-Schnittstellen mit SLIC. Master Thesis (To appear) (2007)
5. Sauter, M.: Automatisierung und Integration regelbasierter Verifikation für Linux Gerätetreiber. Master Thesis (To appear) (2007)
6. Mühlberg, J.T., Lüttgen, G.: BLASTing Linux Code. In: Proc. 11th Intl. Workshop on Formal Methods for Industrial Critical Systems. (2007) (To appear).
7. Post, H., Küchlin, W.: Automatic data environment construction for static device drivers analysis. In: SAVCBS'06: Proc. 2006 Conf. on Specification and verification of component-based systems, New York, NY, USA, ACM Press (2006) 89–92
8. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE'05), Lisbon, Portugal (2005) 263–272
9. Ball, T., Rajamani, S.K.: SLIC: A specification language for interface checking. Technical report, Microsoft Research (2001)
10. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys'06. (2006) 73–85

[4] However, this does not include the creation of the main function. This task took days as we had to figure out the internal workings of poorly documented subsystems