Model Checking Concurrent Linux Device Drivers

Thomas Witkowski [*]
Dresden University of Technology

Nicolas Blanc — ETH Zurich
Daniel Kroening — ETH Zurich
Georg Weissenbacher[†] — ETH Zurich

ABSTRACT

The SLAM toolkit demonstrates that predicate abstraction enables automated verification of real world Windows device drivers. Our predicate abstraction-based tool DDVERIFY enables the automated verification of Linux device drivers and provides an accurate model of the relevant parts of the kernel. We report on benchmarks based on Linux device drivers, confirming the results that SLAM established for the Windows world. Furthermore, we take predicate abstraction one step further and introduce a technique to verify concurrent software with shared memory.

Categories and Subject Descriptors:D.2.4[Software Engineering]: Program Verification; D.4.5[Operating Systems]: Reliability

General Terms: Verification, Reliability

1. INTRODUCTION

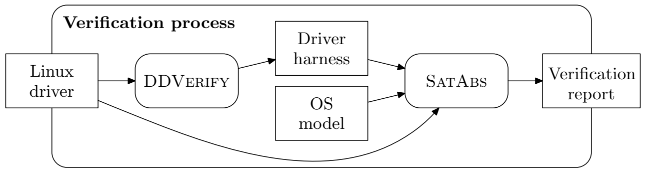
Automated software verification has taken a huge leap forward with the introduction of predicate abstraction [12] and counterexample-guided abstraction refinement (CEGAR) [8]. Pioneered by the model checking tools SLAM and BLAST, the technique has been successfully applied to analyze device drivers with more than 10,000 lines of code [1, 15]. The popularity of these tools is based on the fact that they enable *automatic* detection of bugs in *real* drivers and report a *counterexample* exposing the bad behavior. SLAM checks for violations of about 30 rules as simple as "a thread may not acquire a lock it has already acquired, or release a lock it does not hold".

Predicate abstraction succeeds in checking such properties by separating the control and the data of the program. This technique enables the detection of control-flow related bugs. Even though SLAM is able to verify properties related to locking, its scope is restricted to *sequential* programs. The most vicious bugs emerge in systems with threads that communicate via shared memory. These bugs are hard to comprehend, and it is almost impossible to reproduce them by means of testing. Fortunately, predicate abstraction can also be used to generate abstract models for multi-threaded programs: Attempts to integrate a model checker for concurrent abstract models into SLAM have been made, but not reported due to scalability issues and the lack of convincing benchmarks. Our first contribution is that we integrate the model checkers Cadence SMV [16] and BOPPO [9] into the predicate abstraction-based verification tool SATABS [7], thus enabling the verification of concurrent programs that communicate using shared memory. To the best of our knowledge, this approach has never been reported so far.

In order to *detach* the code of the device driver under test from the operating system, the service routines are replaced with a model that over-approximates their behavior. The device driver is then exposed to a hostile environment, simulated by a driver harness, trying to reveal "misuses" of the kernel API. These misuses are specified by means of assertions added to the operating system model.

The operating system as well as the driver harness is modeled using non-determinism. An inaccurate model may result in falsely reported bugs that do not exist in the actual system. In particular, in the presence of threads it is essential to truthfully model the synchronization primitives of the operating system.

Our second contribution is that we provide a concurrent model of the relevant parts of the Linux kernel API. More information on our formalization of the Linux kernel is presented in [22]. We provide the fully automated verification tool DDVERIFY, which, given the source code of a Linux device driver, generates an appropriate driver harness and uses SATABS to check whether the driver violates the pre- or post-conditions of our kernel model. The following activity diagram shows the corresponding verification process:



We present benchmarks generated by applying DDVERIFY to concurrent programs and report two previously unknown bugs that were detected in Linux device drivers.

[*]This work was carried out during a visit at ETH Zurich.

[†]Supported by Microsoft Research through its European PhD Scholarship Programme.

2. RELATED WORK

Predicate abstraction [12] is an abstraction mechanism that generates finite-state models for programs with unbounded state space. The states of the abstract model are determined by evaluating the concrete states under a finite set of first order logic predicates, which reflect properties of the original program. An insufficient set of predicates leads to falsely reported counterexamples. Such inaccurate abstractions can be refined by means of adding predicates that are extracted from these false counterexamples. This technique is known as counterexample-guided abstraction refinement [8].

Several predicate-abstraction based CEGAR verification tools are available. Blast [15] improves the approach implemented in Slam [3, 1] by using a "lightweight" refinement abstraction loop that refines only relevant parts of the abstract model [15]. Henzinger et al. present a Blast-based automatic race checker for multithreaded C programs [14], which examines each thread separately using assume-guarantee reasoning. Magic is able to handle concurrent programs that communicate via message passing, but do not use shared memory [5]. Unlike the other CEGAR tools mentioned in this section, SatAbs [7] uses a bit-level accurate decision procedure, and therefore, can detect errors related to bit-level operators (e.g., overflows). Using the model checker Boppo [9], SatAbs is able to check concurrent programs that communicate via shared memory. Yasm uses a modified abstraction mechanism that enables verification of liveness properties [13], while all other tools mentioned here support only safety properties. Qadeer's KISS tool uses Slam to check concurrent programs by generating a sequential model that reflects a subset of the execution traces of the original program [19].

Besides predicate abstraction, several other model checking techniques to verify ANSI-C programs have been proposed. (We do not discuss verification tools for other languages.) The tools described below operate on the original program, i.e., no abstraction is applied. Cbmc [6] is based on *bounded model checking* (BMC), and symbolically executes all traces of a program up to a user-specified length. Due to its bounded nature, the approach is appropriate for detecting shallow bugs only. Saturn [23] performs BMC for each function of the C program separately. Loops are modeled by unrolling them a predetermined number of times. Function calls are handled by maintaining concise summaries of functions.

Post and Küchlin [18] present a heuristic for automatic generation of test harnesses for the BMC-based verification of Linux device drivers. Their method automatically infers parameters for the dispatch function calls, including pointers and data structures that typically occur in device drivers.

Other verification tools such as Daikon [10], Eraser [20], and Verisoft [11] are based on explicit execution rather than static analysis. Yang et al. verify file system implementations using Cmc [17], a tool that explicitly executes the unmodified kernel code *within* a model checker [24].

3. PREDICATE ABSTRACTION

The CEGAR approach comprises three phases, namely abstraction, model checking, and refinement. These three phases are executed repeatedly, until either a counterexample is found, or the program under test is proved correct. The CEGAR process is fully automated: Except for providing the property to be verified, no user interaction is required.

In the first phase, predicate abstraction is used to generate a finite state abstraction $\hat{M}$ of the original program M. The variables of $\hat{M}$ correspond to a finite set of predicates over the variables of M. Each predicate φ_i describes an observable "fact" about the program, and the valuation of a variable b_i in an abstract state determines whether the corresponding fact holds or not. For each instruction of M, the corresponding abstract transition is constructed independently of the surrounding instructions. SatAbs aggressively inlines function calls. Therefore, an abstract state comprises a valuation to the Boolean variables and a program counter.

The abstraction step preserves the control flow structure of the original program M. The abstract program $\hat{M}$ contains all execution traces of M, and potentially more.

In phase two, the abstract model $\hat{M}$ is examined by a model checking tool. For this purpose, SatAbs uses either Boppo or Cadence SMV, but several other efficient symbolic model checking algorithms based on summarization and saturation are known (e.g., [2, 21, 4]). If a counterexample is found in the abstraction, the third phase is entered. In that phase, symbolic simulation is used to determine whether the counterexample can be replayed in the original program. The existence of an abstract counterexample in $\hat{M}$ does not necessarily imply that the error state is reachable in the original program M. Counterexamples that are feasible in M constitute "real bugs" and are reported. We call an infeasible counterexample *spurious*. Spurious counterexamples are eliminated by adding additional predicates that increase the accuracy of $\hat{M}$. These predicates are extracted from the counterexample automatically.

The concurrent harness generated by DDVerify uses a statically bounded number of threads. The abstraction algorithm can be applied without major modifications. The state space of the resulting concurrent abstract model is still finite and can therefore be checked using SMV. Unlike SMV, Boppo is able to handle infinite dynamic thread creation by over-approximating the abstract transition relation [9]. Currently, DDVerify does not use this feature.

The counterexample trace obtained by checking a multi-threaded abstract model may contain context switches. The model checker passes this information on to the feasibility checker by attaching a thread identifier to each transition in the counterexample. We modified the implementation of the feasibility checker such that a new stack is created whenever a new thread identifier is encountered in the counterexample. Naturally, a context switch involves saving the stack and program counter of the current stack and restoring the stack and program counter of the target thread. While adding threads to Boolean programs increases the complexity of the model checking phase significantly, adding threads to the counterexample trace does not make feasibility checking a harder problem.

4. EXPERIMENTS

DDVerify introduces verification conditions for device drivers by means of assertions in the operational model of the operating system. Due to inlining, each assertion occurring in the code results in at least one claim that has to be discharged in order to show the correctness of the program.

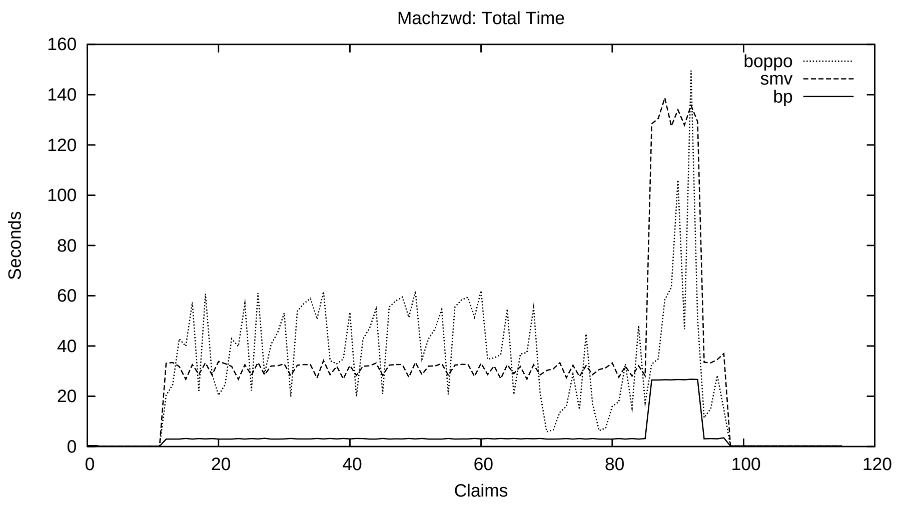


Figure 1: Sequential model: total time to prove the claims using Boppo, SMV and Bp.

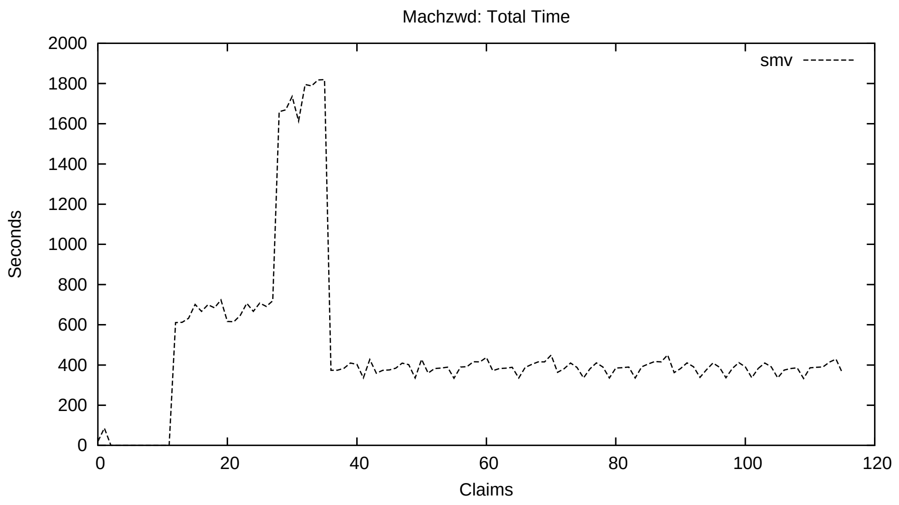


Figure 2: Concurrent model: total time to prove the claims using SMV.

Our framework contains a collection of drivers that come with the Linux kernel distribution. Using DDVerify, we found two previously unknown bugs in two device drivers. Both were discovered using a sequential driver harness.

As an illustrating example, we present the verification results for a watchdog driver. The experiments were conducted on an Intel Xeon processor running at 3GHz with 4 GB of RAM.

The *machwzd* benchmark code contains 494 lines of code and uses spinlocks, IO ports and timer functions. We checked the driver for the correct usage of IO ports, which are resources that need to be requested by the driver in order to prevent access conflicts. Drivers can call the *request_region* function to acquire a range of consecutive port numbers. Using the following assertion, we verify that any access to the port `port` is valid:

```
__CPROVER_assert(
  port >= ioport_request_start &&
  port < ioport_request_len + ioport_request_start)
```

In total, SatAbs generates 116 claims for our sequential model. SatAbs reports one previously unnoticed bug. This bug has gone undetected so far despite the fact that the faulty code is executed each time the module is loaded. First, the driver attempts to detect the presence of its related hardware logic, and only subsequently requests IO resources. This initialization step is faulty since the detection is performed by reading a port not yet requested. As a result, the IO access has unpredictable consequences.

SatAbs produces a counterexample using two predicates derived from the condition of the assertion. Fig. 1 reports the total runtime of SatAbs for each of the 116 claims generated for *machzwd*. Some of the claims are proved correct using constant propagation only. In that case, the CEGAR loop is skipped.

As most of the verification time is spent verifying the abstract program, the choice of the model checker has a significant impact on the runtime. We demonstrate this by using three different model checkers, namely Cadence SMV [16], Boppo [9], and Bp [4]. The latter is based on efficient satisfiability checking techniques and has been presented only recently. It performs significantly better than Cadence SMV and Boppo, but lacks support for concurrency.

Claim 0 corresponds to the invalid IO port access described above and turns out to be wrong. Since the broken assertion occurs at the very beginning of each execution trace of the driver, other assertions are unreachable. As a result, the verification process terminates as soon as SatAbs is able to show that the assertion that corresponds to claim 0 cannot be bypassed. To verify certain claims, SatAbs performs up to 15 refinement iterations, discovering 20 predicates. We observed no difference in the number of discovered predicates and iterations when switching between SMV, Boppo, and Bp.

We are able to verify the same properties using a concurrent model with two threads. The first thread calls the driver functions, while the second one simulates the arrival of interrupts and the execution of deferred tasks. The results we obtain by using SMV are presented in Fig. 2 (Boppo actually timed out on almost all claims). Note that the order for proving the claims depends on the model, and thus, differs from Fig. 1. However, Fig. 1 and Fig. 2 share some similarities, such as a unique peak. SatAbs needs at most 2.5 minutes to verify a claim using the sequential harness, while this number increases to 30 minutes when switching to the concurrent one. These results illustrate the state-space explosion induced by the additional thread. When the property being checked requires more predicates, the verification time increases rapidly. In particular, fixing the bug mentioned above results in a significant increase in the number of predicates required to verify a claim using the sequential model: The number of predicates increases by a factor of 10. In that case, running SatAbs using the concurrent model yields time-outs for most of the claims.

5. CONCLUSION

We present the DDVerify framework, which enables the automated verification of Linux device drivers based on the SatAbs model checker. Given a device driver, DDVerify is able to automatically generate a test harness for this driver. DDVerify provides a sequential as well as a concurrent model of the driver service routines of the Linux kernel. These models are significantly more accurate than the operating system models provided by Slam or Blast. For instance, DDVerify supports synchronization constructs, interrupts, and deferred tasks. Furthermore, we integrate a concurrent model checker for abstract models generated by predicate abstraction into a CEGAR framework.

We present benchmarks that confirm that predicate abstraction is an adequate method for verifying sequential device drivers. In principle, it is possible to detect more bugs using a concurrent harness. However, in order to apply this verification technique to concurrent device drivers, the performance of the model checker tools for concurrent abstract programs has to be improved significantly. Currently, no CEGAR tool we are aware of is able to handle device drivers of a realistic size in the presence of threads.

DDVerify and a collection of test cases are available from `http://www.verify.ethz.ch/ddverify`. With the framework and the operating system models being in place, we believe that DDVerify is a first step to make the automated verification of Linux device drivers practical.

6. REFERENCES

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *European Systems Conference (EuroSys)*, 2006.

[2] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN Workshop on Model Checking of Software*, volume 1885 of *LNCS*. Springer, 2000.

[3] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Principles of Programming Languages (POPL)*. ACM Press, 2002.

[4] G. Basler, D. Kroening, and G. Weissenbacher. SAT-based summarisation for Boolean programs. In *SPIN Workshop on Model Checking of Software*, volume 4595 of *LNCS*, 2007.

[5] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6), 2004.

[6] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*. Springer, 2004.

[7] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *LNCS*. Springer, 2005.

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, volume 1855 of *LNCS*. Springer, 2000.

[9] B. Cook, D. Kröning, and N. Sharygina. Symbolic model checking for asynchronous Boolean programs. In *SPIN Workshop on Model Checking of Software*, volume 3639 of *LNCS*. Springer, 2005.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[11] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design (FMSD)*, 26(2), 2005.

[12] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, volume 1254 of *LNCS*. Springer, 1997.

[13] A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A software model-checker for verification and refutation. In *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*. Springer, 2006.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Computer Aided Verification (CAV)*, volume 2725 of *LNCS*. Springer, 2003.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL)*. ACM Press, 2002.

[16] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[17] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. *SIGOPS Operating Systems Review*, 36(SI), 2002.

[18] H. Post and W. Küchlin. Automatic data environment construction for static device drivers analysis. In *Specification and Verification of Component-based Systems (SAVCBS)*. ACM Press, 2006.

[19] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Programming Language Design and Implementation (PLDI)*. ACM Press, 2004.

[20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *SIGOPS Operating Systems Review*, 31(5), 1997.

[21] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

[22] T. Witkowski. Formal verification of Linux device drivers. Master's thesis, Dresden University of Technology, 2007.

[23] Y. Xie and A. Aiken. Saturn: A SAT-based tool for bug detection. In *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*. Springer, 2005.

[24] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *Transanctions on Computer Systems (TOCS)*, 24(4), 2006.