How to Cook an Automated System for Linux Driver Verification

A. Khoroshilov, V. Mutilin, V. Shcherbina, O. Strikov, S. Vinogradov, V. Zakharov

Institute for System Programming
Moscow, Russia

***Abstract*—We present the preliminary results of our work on designing an automatic toolset for verification of Linux kernel-space drivers. By now the toolset includes three components:**

- **a library of verification models; each model is a formal description of some safety property to be checked;**
- **a preprocessor unit; it is intended for simulating an environment of a driver;**
- **a verification unit BLAST; it is a general-purpose toolkit intended for automatic program verification based on Boolean abstraction and counter-example guided abstraction refinements techniques.**

We discuss in some details how our system operates and outline the directions for future work.

Operating system (OS) is the key component of every computer system including automatic control system of power station, bank information system, or just a cheap cell phone. The faults occurred in OS may result in fatal consequences: damage of the objects under the control of computer, material losses, data leakage. For the most part program errors do not exceed the bounds of erroneous software components; they can be easily localized, checked and corrected. But if a software component operates in a kernel mode then its erroneous behaviour has far-reaching implications and tends to failure many other elements of computer system. The empirical data indicate (see [10], [21]) that bugs in kernel-space device drivers cause 85% of the system crashes. Therefore verification of OS drivers is the topical problem in software designing.

The most generally employed approach to detecting bugs in application programs is testing. But in the case of OS testing encounter with a number of limitations that drastically deteriorate its effectiveness. When testing OS components one checks the correctness of their behaviour only on some specific configurations depending on the state of main and peripheral hardware, distribution of internal and external events, etc. The number of such configurations in incredibly large and even small difference in configurations may cause in significant variances in OS behaviour. That is why no reasonable test set could cover a high proportion of traces in OS code to detect all possible errors. Certainly, a highly skilled expert is able to reveal the most critical parts of OS code and thus guide the detection of errors, but in this case the reliability of a driver becomes a subject to human factor.

In this paper we report on the attempts to implement a new approach to driver verification which would be able to cope pure automatically with checking the most common errors occurred in OS drivers. When verification of drivers is concerned, it should be noticed that the most important issue which need to be checked is the safety of interaction between a driver and the other components of OS. A vast amount of drivers employ only some bounded set of conventional means for interacting with the environment. Close inspection of means and templates that are used by designers of drivers makes its possible to settle in formal terms the requirements (rules) that should be satisfied to guarantee the safety of driver's behaviour. As soon as this set of rules is formalized a partial verification of OS drivers may be performed pure automatically with the help of some software verification tool which is tailored to suit checking of these rules with high accuracy. The users of such a system need not to be experts in OS or program verification, and it can be widely dissipated in the community of drivers designers to facilitate effective preliminary checking of the most common safety requirements.

The paper is organized as follows. We begin with a brief introduction into the fundamentals of formal analysis of programs and description of a recently developed new approach to automatized program verification — counter-example guided abstraction refinement (CEGAR). Then we sketch the architecture of the work-in-progress system for OS Linux driver verification, tell about the intended meaning of its components, give a brief report about our experience in using CEGAR-based software verification toolset approach to verification of some OS Linux drivers, and outline directions of future work.

I. Formal analysis of programs

There are two main approaches to program verification — proof theoretic and model checking. In proof-theoretic approach program statements are treated as predicate transformers. The aim of verification is to check that a given program transforms a predicate φ which characterizes a set of initial states of computations (pre-condition) into a predicate ψ which covers all admissible final states of computations (post-condition). Usually this is achieved by applying logic inference technique (see [15]). The main advantage of this approach is its high accuracy — as soon as ψ is derived from φ this guarantees that *any* execution of the program is correct w.r.t. φ and ψ, — which is attained at the cost of getting over severe difficulties in building such a proof. Therefore it is used only for verification of critical applications.

Model checking approach was initially developed to reason about the logical correctness of any discrete state systems (see [11]). To apply this technique to software verification one needs to abstract a finite state transition system M from a given program and then to check if M satisfies its specification ψ. The virtues of model checking are its efficiency and the ability of building counter-examples in the cases when M does not satisfies φ. The main problem faced by model-checking tools is a combinatorial blow up of the state-space of a model; this effect puts obstacles in the way of the wide application of this technique. To make model checking effective for verification of real programs one needs to choose a proper abstraction level for a finite-state model M of a program to be analyzed. Search of this level is rather uneasy error-prone activity which requires high experience.

Automated theorem proving and model checking taken separately can not cope with such difficult problem as verification of kernel-level device drivers of OS. The huge leap forward has been taken with the introduction of predicate abstraction [16] and counter-example guided abstraction refinement (CEGAR) [12] which combines the advantages of both approaches. Predicate abstraction is a formal method for generating finite-state models of computer programs. The states of the abstract model are finite bit-strings whose elements are the values of first order logic predicates. An initial set of predicates is derived from the formal specification (property) of program to be checked. The transitions in the model are computed with the help of automated theorem provers. Then a model-checking algorithm is applied to the finite state transition system and either reports a success or build some possibly erroneous trace as a counter-example. In the last case theorem proving is used once again to check the feasibility of the counter-example trace. Usually an insufficient set of predicates leads to falsely reported counter-examples. Such an inaccurate abstractions can be refined by means of adding predicates that are extracted from these false counter-examples. This cycle is repeated until either all false counter-examples vanish, or some genuine execution trace which certifies the error is found. Mathematical fundamentals of Boolean program abstractions for software verification are thoroughly studied in [1], [2], [18], [17], [20].

A typical automated verification system implemented in the framework of CEGAR principles is comprised of three main units:

1) a tool for transforming a C program P into a Boolean program $BP(P;E)$ with respect to a given set of predicates E. This tool translates each procedure of the C program separately, enabling it to scale to large programs.
2) a model checking tool for performing reachability analysis of Boolean programs $BP(P;E)$ in attempt to find a path π leading to an erroneous state. It combines interprocedural dataflow analysis in the style of [19] with Binary Decision Diagrams to efficiently represent the reachable states of the Boolean program at each program point.
3) a tool that discovers additional predicates to refine the Boolean program, by analyzing the feasibility of erroneous paths π in the C program. There are three possible outcomes of this checking:
   - the process terminates with a feasible error path π;
   - the process terminates with discovering a path π is not a trace of any program execution; in this case the tool finds a set of predicates F_i that "explain" the infeasibility of path π in P and extends the set of predicates $E_{i+1} = E_i \cup F_i$;
   - the incompleteness of the theorem prover may bring the system to a state where no further refinements are possible or all computer resources are exhausted; in this case user input is required.

Nowadays CEGAR principles of program verification are implemented in 3 systems for program analysis — BLAST (Berkley Lazy Abstraction Software verification Tool) [7], [8], [9], [17], SDV (Static Driver Verifier) [1], [2], [3], [4], [5], [6], and DDVerify [22]. These systems differ in the techniques used for checking the feasibility of trace and computing abstraction refinement. Since BLAST is intended for academic research in software verification, we chose this toolset for our experiments with analysis of OS Linux drivers.

II. Rules and environment

The input data our verification system is applied to includes a C-code of a driver, a formal specification of a property to be checked (a rule), and a model of environment. The rules specify safety requirements a driver should satisfy when it interacts with the environment, e. g.

Rule R: "It is inacceptable to use function *memset* to zero all memory regions allocated by *kmalloc* function".

An environment of a driver includes OS, hardware, application programs, etc. whose behaviour should be specified in some formal terms. Thus, in the example above, a behaviour of OS may be specified by the statement "function *kmalloc* set the pointers only to the memory regions that were not allocated so far". As far as hardware is concerned, in most cases it is simulated by introducing non-determinism in drivers behaviour: for example, any time a call for memory allocation may be refused and *kmalloc* returns null value.

Informal description of the environment should be formalized and adapted to the program verification toolset. BLAST gives us the following means for supplying drivers to be analyzed with information about rules and verification model:

1) insertion of *assert* statements to selected program points;
2) using variable __BLAST_NONDET in conditional jumps or *switch* statements to simulate non-deterministic choice of branches and labels;
3) using function prototypes with empty bodies; BLAST treats such cases as a functions that return arbitrary values;
4) using auxiliary variables to store information about states of verification model;
5) using auxiliary functions.

A verification model corresponding the rule R can be built as follows. First of all, one have to specify explicitly the set of predicates (properties) $R_1, R_2, \dots, R_n$ such that every behaviour of a driver satisfies all these predicates iff it does not comply with the rule R. One can readily see that the rule R is violated iff the requirements below are satisfied

R_1: an address assigned to a pointer by the function *kmalloc* is used as an argument of the function *memset*;

R_2: *memset* takes zero for the second argument and releases all memory allocated be *kmalloc*.

To check the requirement R_1 we introduce an auxiliary variable to keep track of all addresses returned by *kmalloc*. The requirement R_2 can be checked with the help of one more auxiliary variable which stores the size of memory allocated by *kmalloc*; we also need to check if the second argument of *memset* is equal to zero. It is also necessary to distinguish pointers processed by different calls of *kmalloc*. To this end introduce a global counter which will keep track of all calls of *kmalloc*: we will assume that in the framework of our verification model *kmalloc* always returns the registration of this counter. As the result we obtain the following code which specifies the verification model corresponding to the rule R.

```
void * guard_kmalloc_Ptr = 0;
void * guard_kmalloc_counter = 0;
int guard_kmalloc_size = 0;

void * kmalloc(size_t size, int flags)
  {
    guard_kmalloc_size = size;
    guard_kmalloc_Ptr = ++guard_kmalloc_counter;
    if (__BLAST_NONDET)
      guard_kmalloc_Ptr = 0;
    return guard_kmalloc_Ptr;
  }

void memset(void * pdst, int c, size_t plen)
  {
    assert(pdst != 0);
    assert(pdst != guard_kmalloc_Ptr || c != 0
          || plen != guard_kmalloc_size);
  }
```

These pieces of code correctly specify the verification model corresponding to the rule R. Thus we brought a collection of 24 rules that may be used uniformly for verificatiopn of all OS Linux drivers.

III. Drivers preprocessing

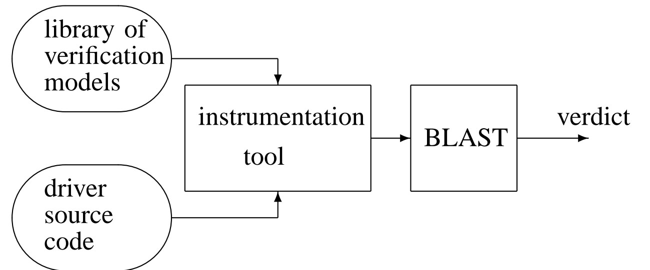
All programs fall into two classes: reactive (interactive) programs and non-interactive programs. Non-interactive programs, such as compilers, preprocessors, etc. get inputs and stimulus only at program start and this is the only point where the environment may affect at the program behaviour. Drivers, network servers as well as programs augmented with user's interface are called reactive applications; these programs are designed to respond to external stimulus and usually they interact with the environment over the course of their execution. The external stimulus for drivers are interruption signals received from perepherial devices and service requests received from other applications.

Since most of drivers are reactive programs, one needs to supply BLAST with some information about interative capabilities of drivers. This is achieved by inserting some additional statements into the code of driver to simulate explicitly the interaction of driver with its environment. The information of interactive capabilities of drivers is extracted from the rules or/and from the specifications of system functions. Based on this information some appropriate prototypes of system functions are added to the code of a driver to simulate all possible stimulus and responses of the environment interacting with the driver.

To explain the preprocessing machinery in more details consider a driver LSI Fusion (linux/drivers/message/fusion/mptctl.c) as an example. The source code of the driver is unsuitable for verification since it does not include an appropriate information about event handlers registration. To build an adequate model of interactive driver one needs to provide BLAST with some description of system functions that are implicitly invoked by the driver. To this end we designed a special-purpose bash-script which generates a new function *main* and puts into it all function calls to be checked. The *main* function begins with invocation of driver initialization procedure, next follow event-handler function calls, and, finally, an invocation of driver deactivation procedure. Initialization and deactivation procedures (module_init and module_exit) are separated out of their logging macrocalls, whereas handler functions are taken from the fields of logged structures in the driver initialization procedure. Parameters of the handler functions are declared but not initialized; this indicates that they may be assigned an arbitrary value. On generating the function *main* the transformation of the driver into non-interactive form is completed.

IV. Drivers verification

Our driver verification toolset operates as follows (see fig.).



The verification system takes a source code of a driver, applies preprocessing to transform it into a non-interactive form, inserts a verification model corresponding to some rule into a preprocessed code, and inputs the file obtained thus to BLAST. Using Boolean abstraction technique BLAST builds a finite state model of a driver and checks the reachability of any

state which falsifies the assertions of the verification model. If no such states are reachable then the driver complies with the rule. Otherwise, BLAST extracts counter-example traces from the abstract model of the driver and tries to check their feasibility with the help of theorem-prover. If one of such traces is proved to be feasible then BLAST outputs it to certify that the driver does not meet the requirements of the rule. If all counter-example traces are unfeasible then BLAST refines Boolean abstraction and checks it again.

Thus, for example, when checking driver LSI Fusion against the rule R BLAST outputs a feasible trace

```
mem = kmalloc(sz, GFP_KERNEL);
if mem == NULL)
        {
          err = -ENOMEM;
          goto out_fail;
        }
memset(mem, 0, sz);
```

which demonstrates how the rule may be violated along some execution of the driver.

V. Advantages and shortcomings

The advantages of this approach to drivers verification is obvious: when the library of verification models corresponding to the rules is assembled then verification of drivers against the collected set of rules may be performed pure automatically. Unfortunately, there are many important rules that are beyond the scope of this technique. By the time of writing this paper we have collected 55 rules and only 24 of them have been accepted for formalization and checking with the help of BLAST. We have built verification models for 15 rules and run a series of successful experiments on checking these rules for a number of Linux character device drivers. All rules fall into the following classes:

1) safety rules,
2) liveness rules,
3) syntactic rules,
4) data rules.

Now we can deal only with reachability (safety) rules. It would be nice also to show that a driver always terminates or that some progress along its execution is achieved. However these properties are not safety properties but liveness properties. Both tools BLAST and SDV currently do not support analysis of such properties. Some progress in this direction is achieved in [13]. Syntactic rules require checking of deprecated types, fields and functions. This can be put into practice with the help of advanced syntactic analyzer; development of this analyzer is a topic for our future work. Data rules refer to bounds, types and attributes of data structures involved in drivers. Unfortunately, BLAST is not well suited for the analysis of such rules. Many of such properties can be checked via an extended data-flow analysis based on abstract interpretations supplied with counter-example guided refinement [14]. The implementation of this new perspective approach to program analysis is another direction for the future work on the improvement of our verification tool.

References

[1] T. Ball, S. K. Rajamani, Boolean programs: a model and process for software analysis, MSR Technical Report 2000-14.

[2] T. Ball, R. Majumdar, T. D. Millstein, S. K. Rajamani. Automatic Predicate Abstraction of C Programs, *SIGPLAN Conference on Programming Language Design and Implementation*, 2001, p. 203-213.

[3] T. Ball, A. Podelski, S. K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs, 2001, LNCS, v. 2031, p. 268–280.

[4] T. Ball, S. K. Rajamani. The SLAM project: debugging system software via static analysis, In *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 2002, p. 1–3.

[5] T. Ball, B. Cook, S. Das, S. K. Rajamani. Refining approximations in software predicate abstraction, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004, p. 388–403.

[6] T. Ball, E. Bounimova , B. Cook, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, A. Ustuner. Thorough static analysis of device drivers. In *Proc. of EuroSys'06: European Systems Conference*, 2006.

[7] D. Beyer, T. A. Henzinger, G. Theoduloz. Lazy Shape Analysis, In *Proc. of the 18-th Int. Conf. on Computer Aided Verification*, 2006, LNCS, v. 4144, p. 532–546.

[8] D. Beyer, T. A. Henzinger, G. Theoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis, In *Proc. the 19-th Int. Conf. on Computer Aided Verification* , 2006, LNCS, v. 4590, p. 504–518.

[9] D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar. The Software Model Checker Blast: Applications to Software Engineering, *Int. Journal on Software Tools for Technology Transfer*, v. 9, N 5–6, 2007, p. 505–525.

[10] A. Chou et al., An Empirical Study of Operating System Errors, In *Proc. of the 18-th ACM Symp. Operating System Principles*, ACM Press, 2001.

[11] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking, MIT Press, 1999.

[12] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-guided abstraction refinement. In *Proc. of the 18-th Int. Conf. on Computer Aided Verification*, 2000, LNCS, v. 1855.

[13] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *Proc. of Static Analysis Symp.*, 2005, p. 87–101.

[14] P. Cousot, P. Ganty, J.-F. Raskin. Fixpoint-Guided Abstraction Refinements, In *Proc. of the 14th Int. Symp. on Static Analysis*, 2007, LNCS, v. 4634, p. 333-348

[15] E. Dijkstra. A Discipline of Programming. Prentice-Hall, 1976.

[16] S. Graf, H. Saidi. Construction of abstract state graphs with PVS. In *Proc. the 10-th Int. Conf. on Computer Aided Verification* , 1997, LNCS, v. 1254.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, K. L. McMillan. Abstractions from proofs, *SIGPLAN Not.*, v. 39, N 1, 2004, p. 232–244.

[18] D. Kroening, A. Groce, E. Clarke. Counterexample guided abstraction refinement via program execution, In *Proc. the 6-th International Conf. on Formal Engineering Methods*, 2004, p. 224–238.

[19] T. Reps, S. Horwitz, M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conf. Rec. of the 22-nd ACM Symp. on Principles of Programming Languages*, 1995, p. 49–61.

[20] H. Saidi. Model checking guided Abstraction and analysis. *Static Analysis Symposium*, LNCS, 2000, v. 1824.

[21] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP'03: Symposium on Operating System Principles*, 2003, p. 207–222.

[22] T. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher. Model checking concurrent linux device drivers, *ASE'07: Proceedings of the 22-nd IEEE/ACM Int. Conf. on Automated Software Engineering*, 2007, p. 501–504.