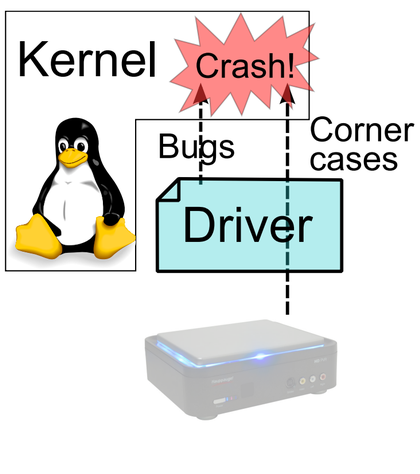
Linux Drivers Static Verifier

Pavel Shved
Institute for System Programming of Russian Academy of Sciences

Challenge

Increase the quality of Linux Kernel device drivers by building an open framework for static verification.



Bugs in drivers of external and internal devices lead to “kernel panic” (Linux version of BSoD), since drivers are executed with kernel privileges.

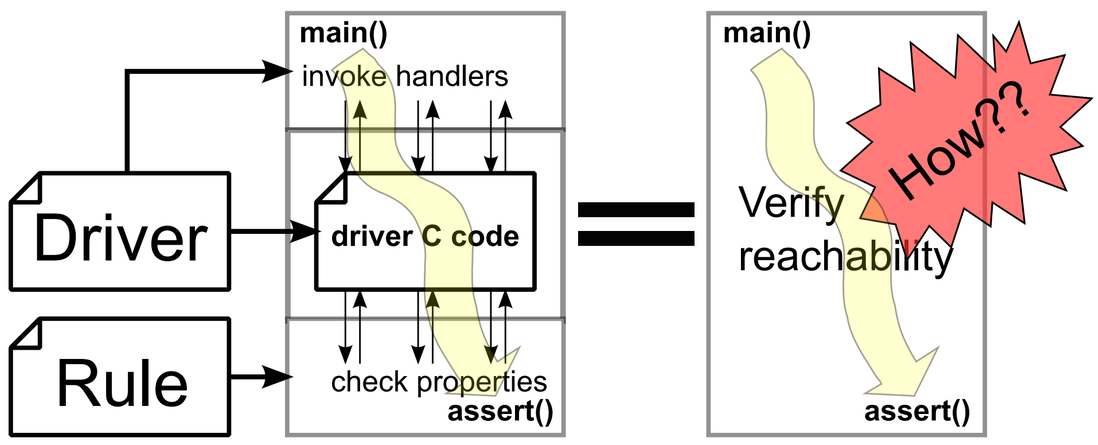
Conventional testing may miss *bugs that occur in “corner cases”* only, because it’s hard to alter the behavior of the device for comprehensive testing of a driver.

To employ static verification we do not need equipment at all, verifying that no crash happens even if the equipment is broken or malicious!

Static reachability verification of drivers

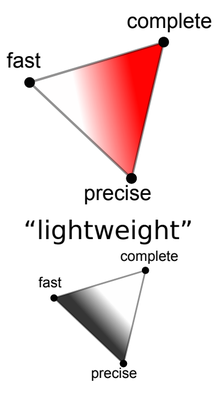
A lot of rules a kernel hacker must obey may be explicated as safety properties (*“at any time during an execution a property must hold”*). To verify a driver against them, we should:

- automatically construct an *environment model* (a C code that invokes handlers in the driver source code like the Kernel does during an execution);
- insert auxiliary code that models behavior of a certain Kernel subsystem;
- insert assertions that check if the rule is violated.



Research objective: create a static verifier capable enough to verify Linux device drivers against a hundred of different rules, consuming sensible CPU time and memory.

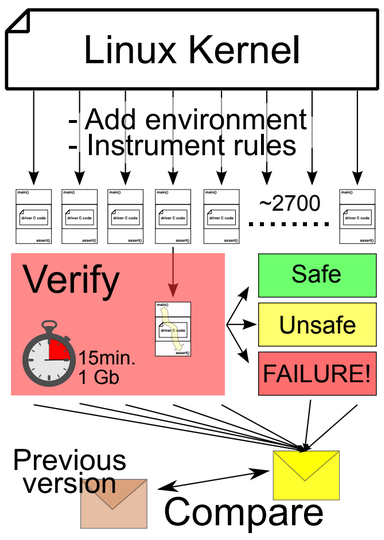
Why we use “Heavyweight” approach



There are hundreds of correctness rules the source code of a driver should follow. In addition to generic rules any C program should obey, there also is a lot of Linux driver-specific rules; they *evolve with time* as the Kernel core gets modified. To perform the complete analysis of a driver, the context it works in should be modeled, which also evolves with time and *changes across driver subsystems*.

Transparent way to specify a property to verify and the context the code is executed in is a native feature of “heavyweight” approach to static analysis. Therefore, we use “heavyweight” verifiers to check Linux drivers; such tools include: CPAchecker, CBMC, BLAST, SLAM. The major drawback of “heavyweight” verification is its speed, which is low, compared to “lightweight” approach used in Coverity, Klockwork Insight, SVACE etc.

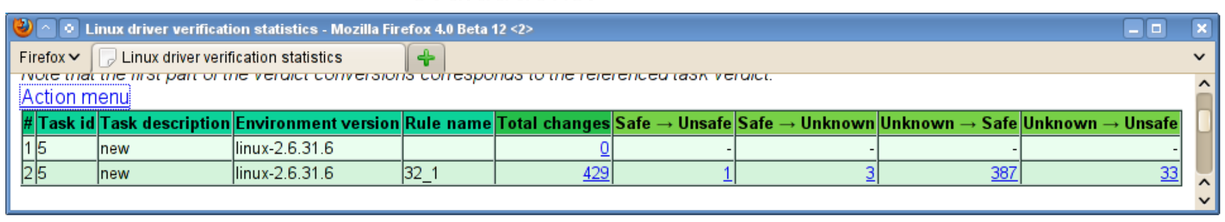
Experimental setup



We implement our new verification algorithms as patches to *BLAST* (“Berkeley Lazy Abstraction Software verification Tool”). BLAST was released in 2002 by a team in Berkeley, and we currently maintain it.

We consider the effect of our improvements “positive” if, on a large scale, the tool works faster or more precise, or finds more errors, retaining the “heavyweight” values (see the red gradient triangle above).

To carry out the comparison of results on such a massive scale, we have developed a tool that visualizes differences between several evaluations.



#	Task id	Task description	Environment version	Rule name	Total changes	Safe → Unsafe	Safe → Unknown	Unknown → Safe	Unknown → Unsafe
1	5	new	linux-2.6.31.6		0	-	-	-	-
2	5	new	linux-2.6.31.6	32_1	429	1	3	387	33

Research + Development

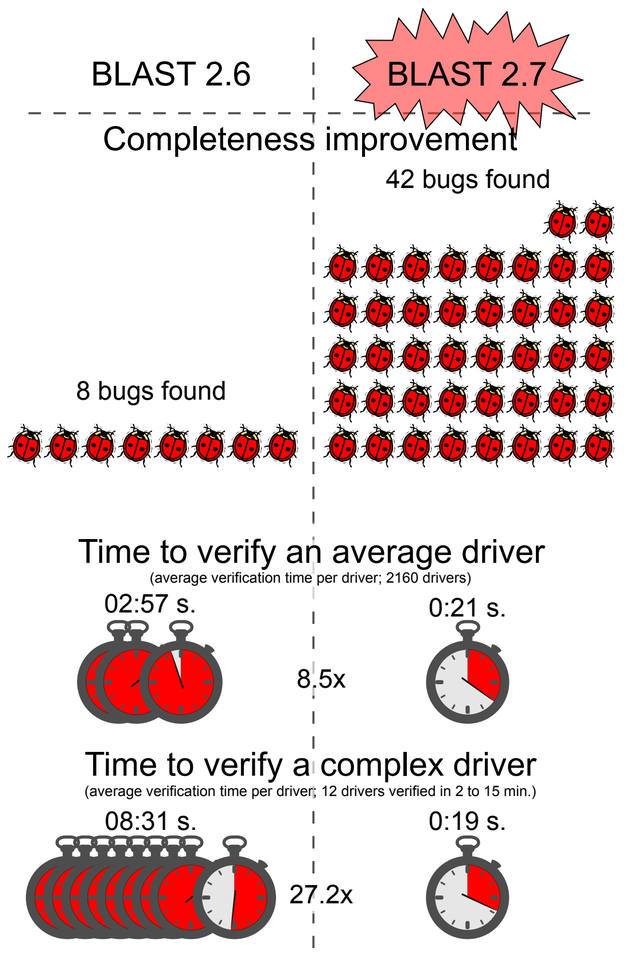
In addition to researching new verification algorithms, we also spend a fraction of our time to optimize the tool we use, because:

- an unoptimized tool biases measurements of “heavyweight” verification *applicability*;
- speed effect of novel static verification algorithms would be measured less precisely.

We deliberately addressed several issues in BLAST, and achieved the following:

- fixed errors in configurable analysis yielded 25% more bugs found, which were elicited before;
- improved speed of interaction with external SMT solvers (integration overhead is now negligible);
- tuned an open-source SMT solver (CVC3) for maximum performance
- the improved interaction with SMT solvers allowed us to eliminate non-free components from BLAST;
- improved speed of trace analysis heuristic (less unnecessary SMT solver calls).

The effect of these achievements on the number of errors found and on time it takes to verify the drivers is shown here →



Verification of certain operations with sets

A goal was to support verification of a C language extension with set operations (∪, ∩, \, ∈, ∅). Such operations could be used to model mutex locks/unlocks or memory allocation precisely, without alias analysis. We tried to create an algorithm that forward-tracks structures of user-specified set structures (i.e. locked mutexes, allocated memory chunks) used in the instrumented source code.

Our studies have demonstrated that the speed did improve over the “undetermined branching” approach. Unfortunately, we also found out that forward-tracking set structures imposed restrictions on control-flow of the programs, and didn’t scale well due to overly complicated formulæ for interpolating prover to handle. Thus, we decided not to use this extension.

Current research

Currently, we **determine the C features semantics should be verified correctly of**, in order to verify Linux drivers with significant quality. We explicate more Kernel rules and describe and collect the issues with completeness and precision to address the most substantial of them.

Currently, the most likely C features to support are:

- fast and precise interprocedural *alias analysis* with pointer type casting support;
- calling functions by pointers.

We already started works on alias analysis.

Here’s an illustration how fast and precise alias analysis would help us to model correctness rules. Suppose we are to implement a `lock()`/`unlock()` correctness rule for mutexes. A mutex in the Linux kernel is a plain structure, and it is passed by pointer to the functions that operate on it.

Assume that a straightforward model checks a value of a special `locked` field in the model of such a structure. However, to verify a sample program presented on the figure to the right, a tool should devise that formal parameters a and b are actually aliases of c, and that the corresponding fields also alias each other.

Pointer aliases in a mutex lock model

```
void mutex_lock(struct mutex *a)
{ assert( ! a->locked );
  a->locked = 1; }
void mutex_unlock(struct mutex *b)
{ assert( b->locked );
  b->locked = 0; }
int main()
{ struct mutex *c;
  mutex_lock(c);
  mutex_unlock(c); }
```

Currently, the algorithm that determines such an aliasing relation used in BLAST is prohibitively slow, but without it the verification of the code on the figure yields a false positive.

Acknowledgments

The team that works on Linux Driver Verification consists of *Vadim Mutilin*, *Eugene Novikov*, *Pavel Shved*, and *Alexey Khoroshilov*. We work at the Institute for System Programming of Russian Academy of Sciences (http://ispras.ru/en/).

Visit the project website at **http://forge.ispras.ru/projects/ldv/**