Finding Resource Manipulation Bugs with Monitor Automata on the Example of the Linux Kernel

Anders Fischer-Nielsen
afin@itu.dk

2020

Contents

1 Introduction

The Linux kernel supports a vast array of computer architectures and runs on a multitude of devices from embedded devices, through personal computers to large servers; on wireless access points, smart TVs, smartphones, refrigerators. Errors in the Linux kernel therefore affect a multitude of devices and can therefore have a potential significant negative impact.

An important aspect of kernel programming is management and manipulation of resources, be it devices, file handles, memory blocks, and locks. Shared-memory concurrency and locks are used extensively in the C source code of the Linux kernel in order to allow parallelization of subsystems within the kernel while at the same time avoiding race conditions. Static analysers allow detection of errors in the C source code of the Linux kernel by reasoning about this resource manipulation. A control flow graph can be found for the components of the kernel, which can then in turn be statically analysed to detect possible ressource manipulation errors.
[2]

2 Background

Monitor automata are defined as the quintuple $(\sum, S, s_0, \delta, F)$ where $\sum$ is the input alphabet, S is a finite non-empty set of states, s_0 is an element of S and initial state, δ is the state-transition function $\delta : S \times \sum \to S$ and F is the possibly empty set of final states and a subset of S.

Monitor automata operate on the set of possible effects of a statement in the Control-flow Graph which is defined as $E = \{\texttt{alloc}, \texttt{free}, \texttt{read}, \texttt{write}, \texttt{uninit}, \texttt{call}, \texttt{lock}, \texttt{unlock}\}$ by Abal [1].

2.1 Double-unlock monitor automata

A double-unlock monitor automata is defined as the quintuple $(\sum, S, s_0, \delta, F)$ where:

- $\sum = \{\texttt{unlock}, \texttt{lock}\}$, a subset of E
- $S = \{locked_\rho, unlocked_\rho, error_\rho\}$, bound to a region ρ
- $s_0 = unlocked_\rho$
- $\delta =$ the relation $\{(locked_\rho, \texttt{unlock}_\rho, unlocked_\rho), (locked_\rho, \texttt{lock}_\rho, locked_\rho), (unlocked_\rho, \texttt{lock}_\rho, locked_\rho), (unlocked_\rho, \texttt{unlock}_\rho, error_\rho)\}$
- $F = error_\rho$

An illustration of this monitor automata can be seen in 1.

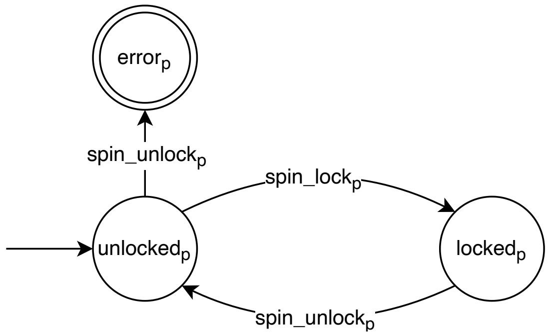

Figure 1: An illustration of a double-unlock monitor automata.

3 Finding Double-Unlock Bugs

3.1 EBA Integration

The EBA framework allows specifying checker signatures whose implementations are executed on a given input source file. Checker signature implementations instantiate a given bug checker for a given bug type and the internal logic of the bug checker is run by the framework.

A signature for a checker which allows instantiation of monitor automata bug checkers has been defined as part of my work. The function, `check`, is the only requirement for implementing this signature. This function takes two parameters and returns a list of strings for each detected possible bug in the input source file. These two parameters are an abstraction of the input file and each global function defined in this file. This signature mimics the existing CTL checkers in EBA and allows for easy integration into the framework.

This signature is implemented as a module, `Make`, which is used by EBA in order to run automata bug checkers. The `Make` module expects an implementation of the `AutomataSpec` signature which defines a monitor automata, detailed in the following section.

The Make module explores the CFG tree structure and applies a transition function defined in the monitor automata signature. Depending on the type of the given tree node different actions are executed and the tree is then explored further until the end of each path in the tree is explored.

Describe PathTree tree structure in detail

If-statements in the source input result in an If-node in the tree. If such a node is discovered the two branches from that node are explored and the union of the resulting states is found.

Describe Assume nodes

All other nodes than the ones described are modelled as Seq nodes. These Seq nodes then contain a step which models an execution step in the input source code. When a Seq is discovered in the tree, the given effects of its containing step are explored. These effect raise a problem; since a given step contains a set of effects, the order of these effects are therefore not known and all orders of executing these effects need to be explored. All permutations of the set of effects need to be found and mapped to the given region, while also preserving the information of the other permutations for that given region. Furthermore, the transition function of the monitor automata needs to be evaluated on the current input, resulting in a new state of that automata which must be stored for that region.

Implementing this evaluation using a mapping from a region to the monitor automata which is monitoring that given region is utilized to great effect solve the aforementioned problems and keep track of automata states for regions. This map is continuously updated when encountering previous and new regions with the new state of evaluating the transition function of a given automata with the current effects for a given execution step.

When all paths in the CFG have been explored, the regions which map to error states along with their location and traces are extracted from the mapping and presented to the user as error messages. traces?

3.2 Automata Signatures

4 Results

5 Future Work

6 Conclusion

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. Effective bug finding in c programs with shape and effect abstractions. pages 34–54, 01 2017.

[2] IEEE and The Open Group. pthread_spin_unlock - unlock a spin lock object. `https://pubs.opengroup.org/onlinepubs/9699919799/`, 2017. Accessed: 2019-11-25.

7 Appendix