# 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

Section introduction

## 2.1 Control Flow

EBA provides a representation of the control flow of the input source files which is utilized in order to detect bugs. EBA generates a tree structure of the input, modeling statements as so-called `steps`. A path in this tree structure models a possible execution path, with each `step` in a path containing information about the modelled statements. The tree structure consists of $t$ nodes, which can be formalized as follows.

$$\begin{aligned} t = \; & Nil \\ & Assume(cond, bool, t) \\ & Seq(step, t) \\ & If(t, t) \end{aligned}$$

A few things are of note here; $Nil$ indicates the end of a path in the tree structure. $Assume$ is effectively ignored in my work, and is merely used in order to continue the exploration of the tree structure. $If$ indicates a branch in the input source file and models possible branches in if-statements. $Seq$ is the most interesting of $t$, since this type consists of the aforementioned *step*, providing access to information about the effects of a statement. It is also worth noting that all types provide the next $t$ in the current branch, allowing recursively exploring the tree until a $Nil$ is reached.

*step*s contain information about a statement in the input source file, such as the type of statement, location of the statement and the effects of the statement. The latter of the three is relevant for my analysis, since we are interested in monitoring effects of statements using monitor automata. A step contains two internal sets, each modelling which effects EBA analyzes a step *must* and *may* have. Each set consists of the effect type $e$, formalized below.

Effects which operate on memory regions are modelled as the $Mem$ type. $Mem$ provides information about the type of effect and on which region the effects happen. Monitor automata can of course monitor all $e$s, but given that we are interested in detecting ressource manipulation errors and these ressources are stored in memory, $Mem$ is what I will describe in greater detail. The type of effect is modelled as $mem\_kind$, as seen in the definition of $e$. These types can be formalized as follows.

$$
\begin{aligned}
e = {} & Mem(mem\_kind, region) \\
& Noret \\
& IrqsOn \\
& IrqsOff \\
& BhsOn \\
& BhsOff \\
& Var(effectvar) \\
& Sleep
\end{aligned}
\qquad\qquad
\begin{aligned}
mem\_kind = {} & Alloc \\
& Free \\
& Read \\
& Write \\
& Uninit \\
& Call \\
& Lock \\
& Unlock
\end{aligned}
$$

A control flow defining effects of statements can be defined as a finite state machine $(\sum, S, s_0, \delta, F)$ as follows 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, $\delta$ 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 $S$.

## 2.2 Monitor Automata

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, $\delta$ 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 are defined as $E = \{$`alloc`, `free`, `read`, `write`, `uninit`, `call`, `lock`, `unlock`$\}$ by Abal [1]. These have been defined previously as the internal type, $mem\_kind$, generated by EBA.

Monitor automata in this thesis all operate on a subset of $E$ and have a non-empty set of final states, $F$, indicating that a possible bug is discovered.

### 2.2.1 Double-unlock monitor automata

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

- $\sum = \{$`unlock`, `lock`$\}$, a subset of $E$
- $S = \{locked_\rho, unlocked_\rho, error_\rho\}$, bound to a region $\rho$
- $s_0 = unlocked_\rho$
- $\delta =$ 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 figure 1.

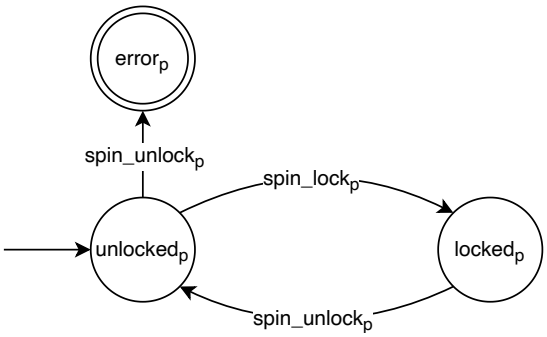


Figure 1: An illustration of a double-unlock monitor automata.

### 2.2.2 Double-lock monitor automata

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

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

An illustration of this monitor automata can be seen in figure 2.

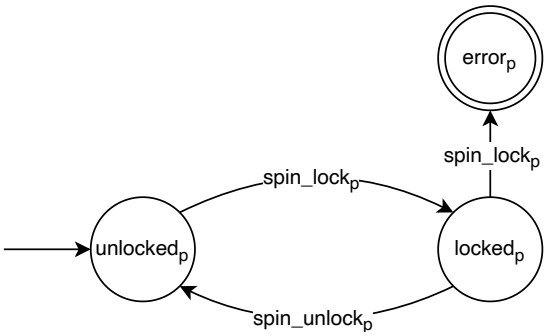


Figure 2: An illustration of a double-lock monitor automata.

### 2.2.3 Double-free monitor automata

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

- $\sum = \{\texttt{free}, \texttt{alloc}\}$, a subset of $E$

- $S = \{allocated_\rho, freed_\rho, error_\rho\}$, bound to a region $\rho$
- $s_0 = freed_\rho$
- $\delta =$ the relation $\{(freed_\rho, \texttt{alloc}_\rho, allocated_\rho), (allocated_\rho, \texttt{free}_\rho, freed_\rho), (freed_\rho, \texttt{free}_\rho, error_\rho), (allocated_\rho, \texttt{alloc}_\rho, allocated_\rho)\}$
- $F = error_\rho$

An illustration of this monitor automata can be seen in figure 3.

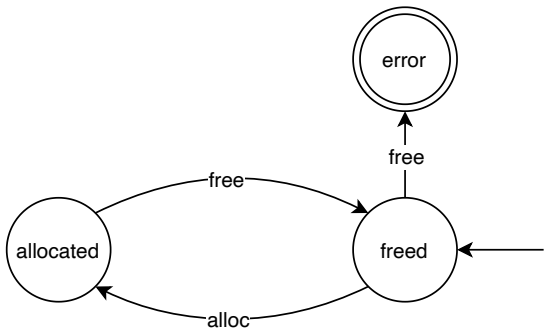


Figure 3: An illustration of a double-free monitor automata.

#### 2.2.4 Use-before-init monitor automata

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

- $\sum = \{\texttt{read}, \texttt{init}\}$, a subset of $E$
- $S = \{unread_\rho, initialized_\rho, error_\rho\}$, bound to a region $\rho$
- $s_0 = unused_\rho$
- $\delta =$ the relation $\{(unread_\rho, \texttt{init}_\rho, initialized_\rho), (initialized_\rho, \texttt{uninit}_\rho, unread_\rho), (unread_\rho, \texttt{read}_\rho, error_\rho), (initialized_\rho, \texttt{init}_\rho, initialized_\rho)\}$
- $F = error_\rho$

An illustration of this monitor automata can be seen in figure 4.

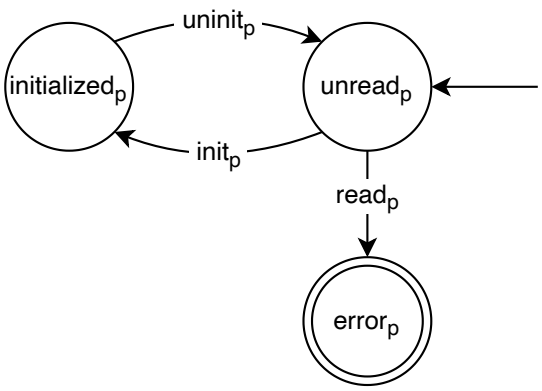


Figure 4: An illustration of a use-before-init monitor automata.

# 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.

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.

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. This map can be formalized as the function $m : region \rightarrow checker\_state$ where $checker\_state$ is the internal state of the monitor automata. Using this map it is possible to apply each permutation of effects and fold this list of effects into a modified map with possibly altered automata states for their corresponding regions.

Given that the mapping maps a discovered region to the state of monitor automatas, the length of the map will never be larger than the number of regions in the input source file. The size of the set of possible monitor automata states for a given region depends on the effects of a statement operating on a given region. Given a large number of possible effects of a statement the resulting set of permutations of these effects will naturally grow. A set of $N$ effects will result in $N!$ permutations; in other words, the number of monitor automata states for a given region will therefore in the worst case be $|effects|!$. Statements have a small number of effects in practice.

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

The signature of monitor automata must be implemented in order to use the bug checker with EBA. The implementation of a given monitor automata is passed to the aforementioned `Make` module and is then used to evaluate states based on the effects of regions. The signature of the monitor automata specifies a `state` discriminated union type, describing the possible states of the automata as well as a transition function which requires a previous state of the state machine along with an input effect. In order to provide the user with detailed error reports this state is encapsulated in a checker state structure which keeps track of the current trace through the CFG along with granular location details for discovered bugs. Providing this information requires that the current CFG step must also be passed to the automata, due to the architecture of the EBA framework. The full signature for the transition function is therefore $transition : checker\_state \rightarrow effect \rightarrow step \rightarrow checker\_state$.

## 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