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 concrete tree structure modeling the resulting effects of statements can be formalized 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, δ is the state-transition function $\delta : S \times \sum \rightarrow S$ and F is the possibly empty set of final states and a subset of S.

- $\sum = \{Entry, Nil, (alloc, \rho), (free, \rho), (read, \rho),$
  $(write, \rho), (uninit, \rho), (call, \rho), (lock, \rho), (unlock, \rho)\}$
- $S = \{(allocated, \rho), (freed, \rho), (read, \rho), (written, \rho),$
  $(uninitialized, \rho), (called, \rho), (locked, \rho), (unlocked, \rho), End\}$

The remainder of the automaton definition is defined according to the control flow being modelled, where the initial state, s_0, is dependent on the control flow being modelled and therefore is an element of $\sum$. This also applies to the transition function δ and the set of final states F, which is also a subset of $\sum$. A concrete definition of an example control flow is shown below with an accompanying illustration of this in figure 1.

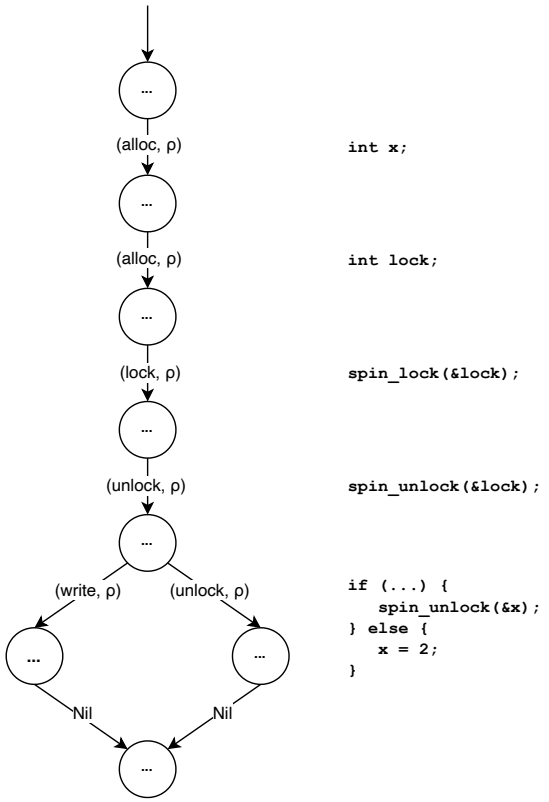

Figure 1: An illustration of a Control Flow automaton.

- $s_0 = Entry$
- $\delta =$ the relation $\{$ $(Entry, (\texttt{alloc}, \rho), (allocated, \rho))$,
  $(allocated, (\texttt{alloc}, \rho), (allocated, \rho))$,
  $((allocated, \rho), (\texttt{lock}, \rho), (locked, \rho))$,
  $((locked, \rho), (\texttt{unlock}, \rho), (unlocked, \rho))$,
  $((unlocked, \rho), \texttt{unlock}, (unlocked, \rho)$,
  $((write, \rho), \texttt{write}, (written, \rho))$,
  $((unlocked, \rho), \texttt{Nil}, End)$,
  $((written, \rho), \texttt{Nil}, End)$ $\}$
- $F = End$

A few things are of note here; Nil indicates the end of a path in the tree structure. Branches occur when an if-branch is encountered in the input source file and models the effects of the statements within if-statements.

2.2 Monitor Templates

How should the template generation function be defined?

A monitor template is defined as the quintuple $X(\rho) \rightarrow (\sum, S, s_0, \delta, F)$ or $X_\rho(\sum, S, s_0, \delta, F)$ where $\sum$ is the input alphabet, S is a finite non-empty set of states happening on the region ρ, s_0 is an element of S and initial state, δ is the state-transition function $\delta : S \times \sum \rightarrow S$ and F is the possibly empty set of final states and a subset of S.

Which one? Different one?

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

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

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

- $\sum = \{\texttt{unlock}_\rho, \texttt{lock}_\rho\}$, a subset of E
- $S = \{locked_\rho, unlocked_\rho, error_\rho\}$
- $s_0 = unlocked_\rho$
- $\delta =$ 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 2.

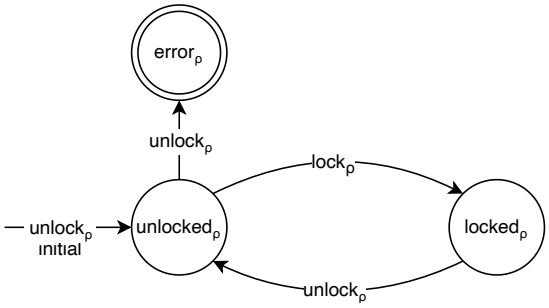

Figure 2: An illustration of a double-unlock monitor automata.

To show the detection of a possible double-unlock bug where a double-unlock monitor automaton has reached the final state, we find the product of the control flow example shown in figure 1 and the monitor generated from the template.

- $\sum = \{Entry, Nil, (alloc, \rho), (free, \rho), (read, \rho), (write, \rho), (uninit, \rho), (call, \rho), (lock, \rho), (unlock, \rho)\}$

- $S = \{(allocated, \rho), (freed, \rho), (read, \rho), (written, \rho),$
  $(uninitialized, \rho), (called, \rho),$
  $((locked, \rho), locked_\rho), ((unlocked, \rho), unlocked_\rho),$
  $((unlocked, \rho), error_\rho),$
  $End\}$
- $s_0 = Entry$
- $\delta =$ the relation $\{$ $(Entry, (\texttt{alloc}, \rho), (allocated, \rho)),$
  $(allocated, (\texttt{alloc}, \rho), (allocated, \rho)),$
  $((allocated, \rho), (\texttt{lock}, \rho), (locked, \rho)),$
  $((locked, \rho), (\texttt{unlock}, \rho), ((unlocked, \rho), unlocked_\rho),$
  $(((unlocked, \rho), unlocked_\rho), \texttt{unlock}, ((unlocked, \rho), error_\rho)),$
  $(((write, \rho), unlocked_\rho), \texttt{write}, ((written, \rho), unlocked_\rho)),$
  $(((unlocked, \rho), error_\rho), \texttt{Nil}, (End, error_\rho)),$
  $(((written, \rho), unlocked_\rho), \texttt{Nil}, (End, error_\rho))$ $\}$
- $F = (End, error_\rho)$

This product can be illustrated as follows.

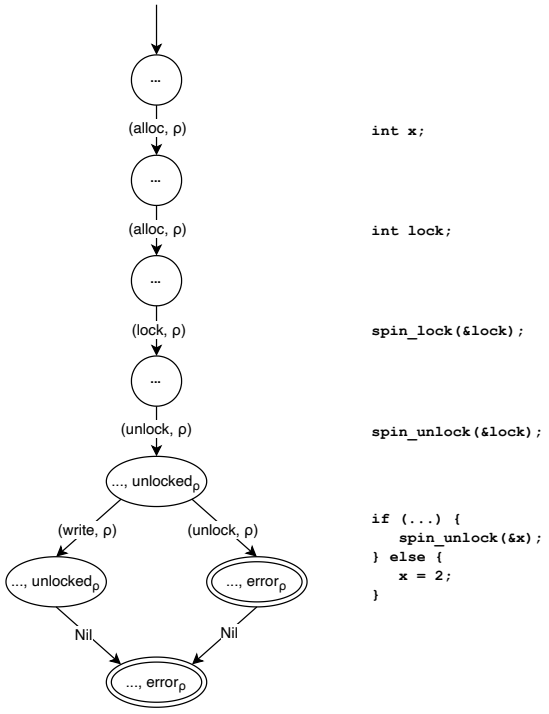

Figure 3: An illustration of the product construction of a double-unlock monitor automata and a control flow.

2.2.2 Double-lock monitor automata

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

- $\sum = \{\texttt{lock}_\rho, \texttt{unlock}_\rho\}$, a subset of E
- $S = \{locked_\rho, unlocked_\rho, error_\rho\}$
- $s_0 = unlocked_\rho$
- $\delta =$ 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 4.

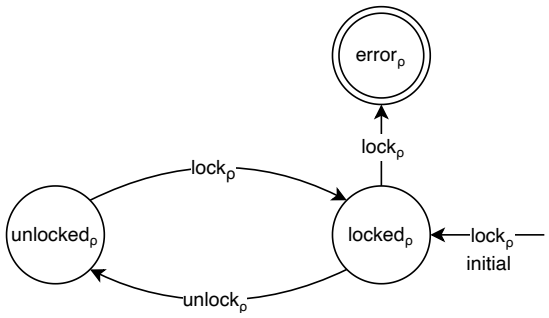

Figure 4: An illustration of a double-lock monitor automata.

2.2.3 Double-free monitor automata

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

- $\sum = \{\texttt{free}_\rho, \texttt{alloc}_\rho\}$, a subset of E
- $S = \{allocated_\rho, freed_\rho, error_\rho\}$
- $s_0 = freed_\rho$
- $\delta =$ 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 5.

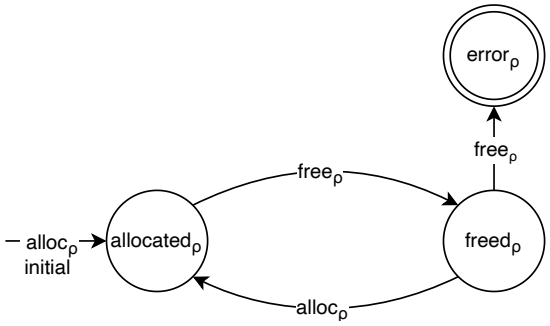

Figure 5: An illustration of a double-free monitor automata.

2.2.4 Use-before-init monitor automata

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

- $\sum = \{\texttt{read}_\rho, \texttt{init}_\rho\}$, a subset of E
- $S = \{unread_\rho, initialized_\rho, error_\rho\}$
- $s_0 = unused_\rho$
- $\delta =$ 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 6.

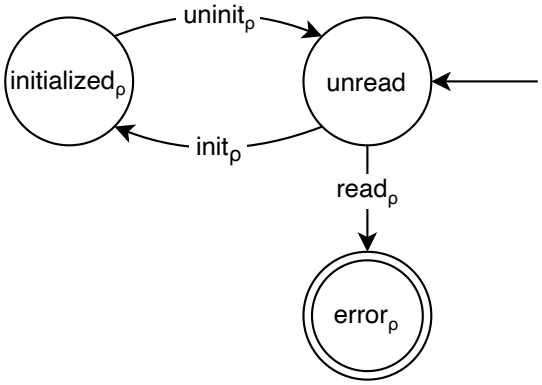

Figure 6: An illustration of a use-before-init monitor automata.

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.

These bug checkers must conform to the existing signatures of EBA in order to allow the framework to instantiate a given bug checker after which an abstract representation of the input source file is passed to the instantiated checker.

Such a signature for a checker which allows instantiation of monitor automata bug checkers has been defined as part of this work. This signature is then implemented in order to let EBA instantiate the implemented checker. A function, `check`, is the only requirement for implementing this signature and takes two parameters after which it returns a list of strings for each detected possible bug in the input source file as expected by the EBA framework. These parameters are the abstractions of the input file and each global function defined in this file, both of which are passed to the function by the framework. This mimics the implementation of the existing CTL checkers in EBA and allows for easy integration into the framework.

The aforementioned signature is implemented as a module, `Make`, which is used by EBA in order to run automata bug checkers conforming to an automata signature. Specifying a signature which all automata-based checkers must conform to ensures that the automata expose the required state and transition functions for them to run. This `Make` module expects an implementation of this `AutomataSpec` signature.

The `check` of the `Make` module explores the CFG provided by the EBA framework of the given file and applies the transition of the monitor automata using the effects of statements, which are represented as nodes in the CFG. The tree is then explored further until the end of each path in the tree is explored, resulting in a set of monitor automata states, which can then be explored in order to determine if any automata have reached accepting states. If such a state is present, a possible bug has been discovered.

Nodes in the CFG structure provided by EBA be one of four different types, each representing the input statement. Nodes representing if-statements in the source input result are *If*-nodes in the tree, containing two branches. If an If-node is discovered, the two branches from that node are explored and the union of the resulting states is found. Nodes representing the end of a branch are *Nil*-nodes in the tree. Nodes representing assumptions made after if-statements are either true or false are *Assume*-nodes, but are not used in this work since all branches are explored. Finally all other statements are represented as *Seq*-nodes, which contain information about the shapes and effects of statements. These *Seq*-nodes are of interest, since they allow analysis on effects.

Seq-nodes contain a *step* which models a statement in the input source code. When a *Seq*-node is discovered in the tree, the — possibly multiple — effects of its containing step are explored. This raises 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 must be explored. This must be done, since a given ordering of effects can lead to a bug, while a different order might not. All permutations of the set of effects must therefore be found and mapped to a given region, while also preserving the information of the other permutations for that given region. Furthermore, the transition function of the monitor automata must be evaluated on the current input, resulting in a new state of that automata which again must also be stored for that region.

This evaluation has been implemented as a map from a region to the monitor automata which are monitoring that given region in order to solve the aforementioned problems and keep track

of automata states for regions. The map is continuously updated when encountering regions and effects by applying the transition function of the automata for a given execution step with the current effects and previous state of the automata. 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. The resulting map can be explored in order to find any accepting states of monitor automata for a given region.

Given that the map maps a 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 $|effects|!$. In practice statements lead to a small number of effects and this has not posed a problem during testing.

When all paths in the CFG tree structure have been explored, the regions which map to accepting states along with their location and traces are extracted from the mapping and presented to the user as possible bugs.

This approach can be described in pseudocode as follows.

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` as a discriminated union type, describing the possible states of the automata as well as a transition function, `transition`, which takes a previous state of the monitor 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 possible bugs. Providing this information requires that the current CFG node 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

```
function APPLY_TRANSITION(effect, map)
    region ← effect.region
    previous_states_for_region ← FIND_DEFAULT([initial_state], r, map)
    states ← FOLD(TRANSITION(state, effect, previous_states_for_region))
    return ADD(map, r, states)
end function

function EXPLORE_PATHS(tree_node) map
    if tree_node is Nil then
        return map
    else if tree_node is If(t, f) then
        true_branch ← EXPLORE_PATHS(t)
        false_branch ← EXPLORE_PATHS(f)
        return UNION(true_branch, false_branch)
    else if tree_node is Seq(step, next) then
        effects ← FILTER(is_in_transition_labels, step.effects)
        if IS_EMPTY(effects) then return EXPLORE_PATHS(remaining, map)
        end if
        permutations ← FIND_PERMUTATIONS(effects)
        return
        FOLD(FOLD(APPLY_TRANSITION(effect, map)) map, effects))
                                          ▷ Find all possible states for permutations.
                                              ▷ Add these states to the region map.
    end if
end function

tree ←GENERATE_TREE(input_file)
map ← EXPLORE_PATHS(tree)
```

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