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 Monitor Templates

Monitor automata are utilized in my work to monitor the control flow of a given input source file and detect whether possible bug are present in this control flow.

A monitor automaton will change state based on what is happening in the control flow of a program. When this monitor automaton reaches its final state, then a possible bug has been discovered. The effect analysis provided by EBA allows monitoring which effects program points have, and monitor automata can then monitor these effects in order to determine whether possible bugs are present.

Effects within EBA happen on a region in the input program, and monitor automata must therefore monitor effects happening on a given region in order to determine whether effects happening on this region lead to possible bugs. It is, for example, common to have multiple locks on different regions within programs, but a lock on one region followed by a lock on a different region does not neccesarily mean that a bug is present. Regions, along with their effects, must therefore be monitored by these monitor automata. This can be expressed formally as follows.

Given a region variable ρ, a monitor template is defined as the quintuple $X_\rho(\sum, S, s_0, \delta, F)$ 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 $\delta : S \times \sum \to S$ and F is the possibly empty set of final states and a subset of S. An illustration of such a monitor automaton can be seen in Figure 1.

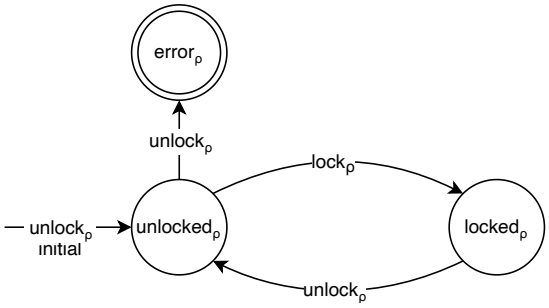

Figure 1: An illustration of a monitor automaton.

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].

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.

2.1.1 Double-unlock monitor automata

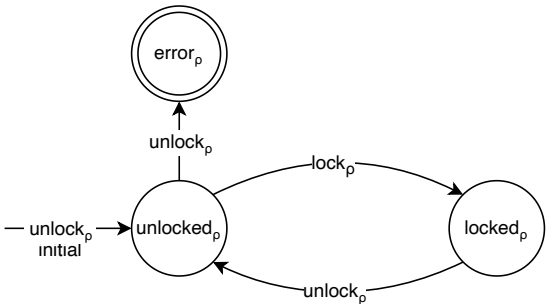

Figure 2: An illustration of a double-unlock monitor automata.

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.

2.1.2 Double-lock monitor automata

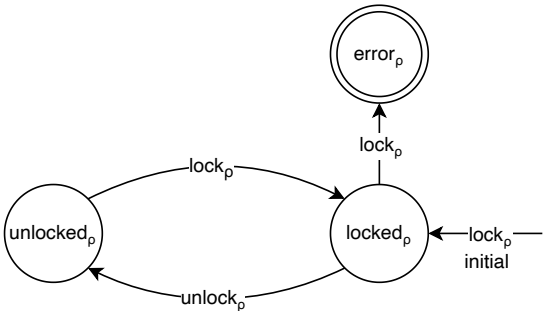

Figure 3: An illustration of a double-lock monitor automata.

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 3.

2.1.3 Double-free monitor automata

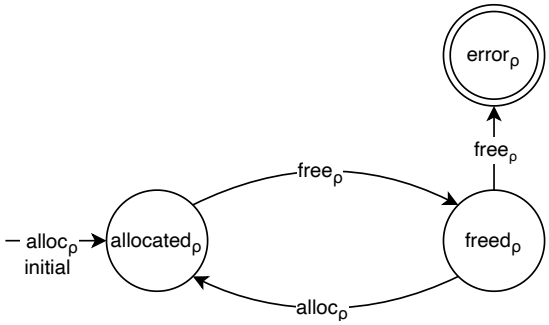

Figure 4: An illustration of a double-free monitor automata.

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 4.

2.1.4 Use-before-init monitor automata

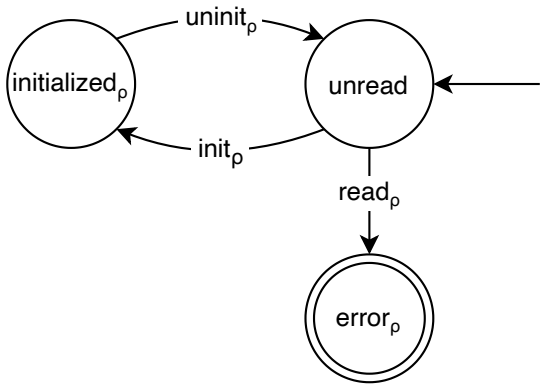

Figure 5: An illustration of a use-before-init monitor automata.

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 5.

2.2 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 `step`s. A path in this tree structure models a possible execution path, with each `step` in a path containing information about the modelled statements. A finite state machine is a quintuple $(\sum, S, s_0, \delta, F)$, where $\sum$ is an 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. I will use such state machines to represent the code under analysis and the properties I wish to detect in input source files.

The concrete tree structure modeling the resulting effects of statements can be formalized as the finite state machine $(\sum, S, s_0, \delta, F)$, where $\sum$ is the 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.

The control flow generated by EBA is acyclic, since EBA unrolls loops within a fixed depth and generates a path of this length accordingly. I keep the abstract formulation since, in principle, the monitor automata checkers will work with more general abstractions over programs.

The alphabet, $\sum$, of the of the control flow abstraction is the set of all effects EBA detects, annotated by the region variables being affected by a given effect. The states, S, are program points after the unrolling of loops. The definition of the control flow abstraction is shown in the following, with a concrete example of a control flow formulated using this abstraction in Figure 6.

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

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 6.

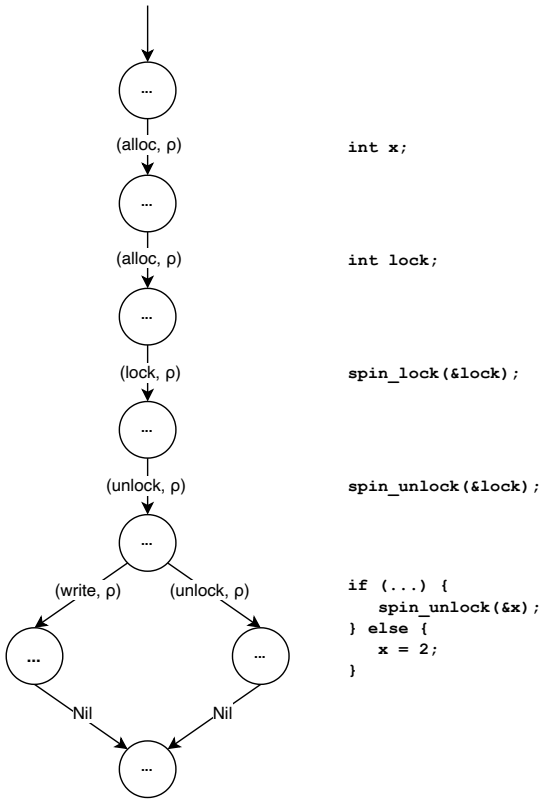

Figure 6: An illustration of a Control Flow automaton.

- $s_0 = Entry$
- $\delta =$ 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.

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 6 and the monitor generated from the template.

Given two automata $A = (\sum, S, s_0, \delta, F)$ and $A' = (\sum, S', s'_0, \delta', F')$ both working over the same alphabet $\sum$, the product automaton of A and A' is an automaton $P = (\sum, Q, s'_0, \delta_P, F_P)$ where $Q = S \times S'$ and $\delta_P : Q \times \sum \to Q$. Furthermore, $\forall q \in Q, q' \in Q'$ and $e \in \sum$, we require $\delta_P((q, q'), e) = (\delta(q, e), \delta'(q', e))$. F_N is for our purposes $F_N = F \times F'$.

Tie requirements below together with definition above.

Since monitors generated from the monitor template only monitor a given input region, it is necessary to define rules for the state changes within this product, given that a monitor only accepts effects on a given region. In other words, the state of the automata should not change if the effect does not happen on the monitored region, but the automaton representing the control flow *should*. These requirements can be formalized as follows.

$$\frac{s \xrightarrow{e} s' \qquad p \xrightarrow{E} p' \qquad e \in E}{(s, p) \to (s', p')} \qquad\qquad \frac{p \xrightarrow{E} p' \qquad \forall e \in E \,.\, s \overset{e}{\not\to}}{(s, p) \to (s, p')}$$

E is the set of effects happening in a given program step, $p \to p'$. The state s will change given that the transition happens on the effect e, present in E. If this is not the case, the control flow will have changed state to p', while the monitor has not and stays the same, s.

The product of the control flow example shown in Figure 6 and a generated double-unlock monitor automaton can now be found in order to demonstrate that a possible bug is detected by the monitor, resulting in the following definition. This definition is illustrated in Figure 7.

- $\sum = \{Entry, Nil, (alloc, \rho), (free, \rho), (read, \rho),$
  $(write, \rho), (uninit, \rho), (call, \rho), (lock, \rho), (unlock, \rho)\}$
- $S = \{(allocated, \rho), (freed, \rho), (read, \rho), (written, \rho),$
  $(uninitialized, \rho), (called, \rho),$
  $((locked, \rho), locked_\rho), ((unlocked, \rho), unlocked_\rho),$
  $((unlocked, \rho), error_\rho),$
  $End\}$
- $s_0 = Entry$
- $\delta =$ 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 is illustrated in Figure 7.

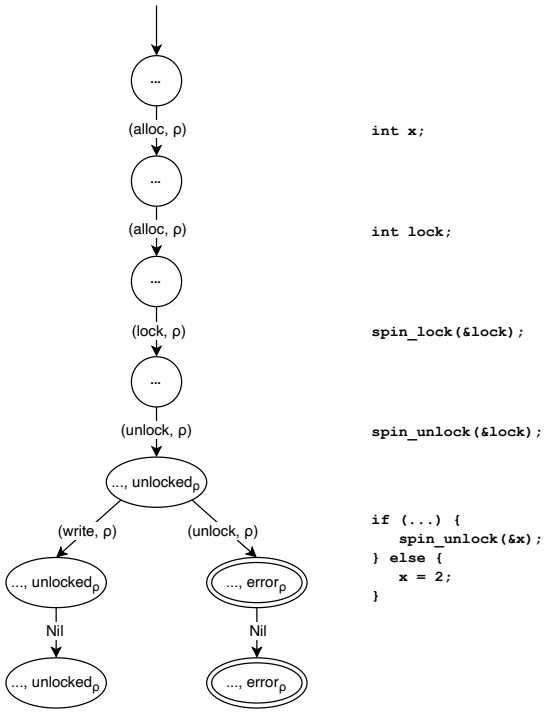

Figure 7: An illustration of the product construction of a double-unlock monitor automata and a control flow.

I have shown that it is possible to construct monitor templates, which allows generating monitor automata monitoring effects happening on a region, and shown that such a monitor can detect a possible bug in an example control flow. In order to implement this approach in practice, two things are needed, namely

- A control flow abstraction
- Concrete definitions of monitor templates

It is therefore necessary to implement these monitor templates, since the EBA framework can provide the control flow abstractions over a given input file. I will present the implemetation of the monitor templates as part of my work in the following section.

3 Finding Double-Unlock Bugs

This section will detail how the abstractions defined previously are integrated into the EBA framework in order to explore the control flow of input programs while generating the product of the control flow and monitor automata in order to detect possible bugs.

3.1 Integration into EBA

In order to implement a new bug checker into EBA, the implementation has to conform to the structure set up by the framework of how these checkers should behave. Furthermore, signatures must be defined for these checkers in order to allow the framework to instantiate these checkers according to user input. This section will describe how this has been accomplished.

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.

In order to detect possible bugs in input programs, a control flow graph and an instantiation of the given monitor automata monitoring the control flow are required. The control flow is provided by the EBA framework, while the monitor automata have been defined and implemented as part of my work.

A signature for a checker which allows instantiation of monitor automata bug checkers has been defined as part of my 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.

EBA can generate the required control flow abstraction, which is used for analysis. My work consists of integrating the implementation of monitor templates into the existing framework, using the control flow abstractions provided by EBA. In order to present how monitor templates are instantiated based on the given control flow, it is necessary to describe the control flow abstractions.

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. An illlustration of these types can be seen in Figure 8. These *Seq*-nodes are of interest, since they allow analysis on effects.

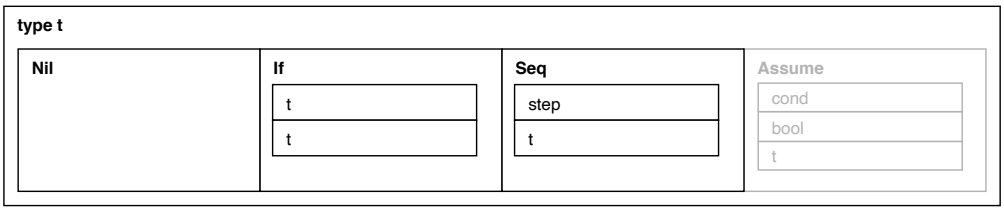

Figure 8: An illustration of the CFG node types found in EBA.

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. An illustration of this node can be seen in Figure 9. These possibly multiple effects 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 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.

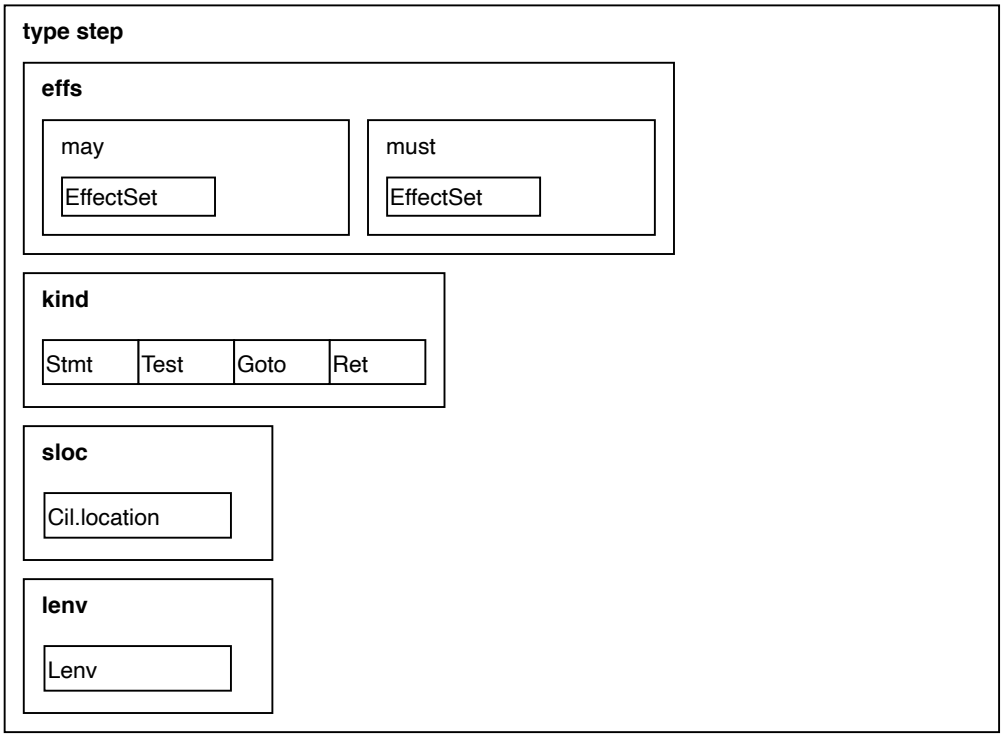

Figure 9: An illustration of the step type and its containing structures found in EBA.

In order to accomplish this, the current state of the monitor monitoring a region needs to be copied and stored in the map for each permutation found for a given set of effects in order to explore all possible effect orderings. These copies are then stored in the map for the given region and future effects on that region are then applied on these copies. An illustration of this copying on two permutations followed by another effect happening on the same region can be seen in Figure 10, leading to a possible double-unlock bug.

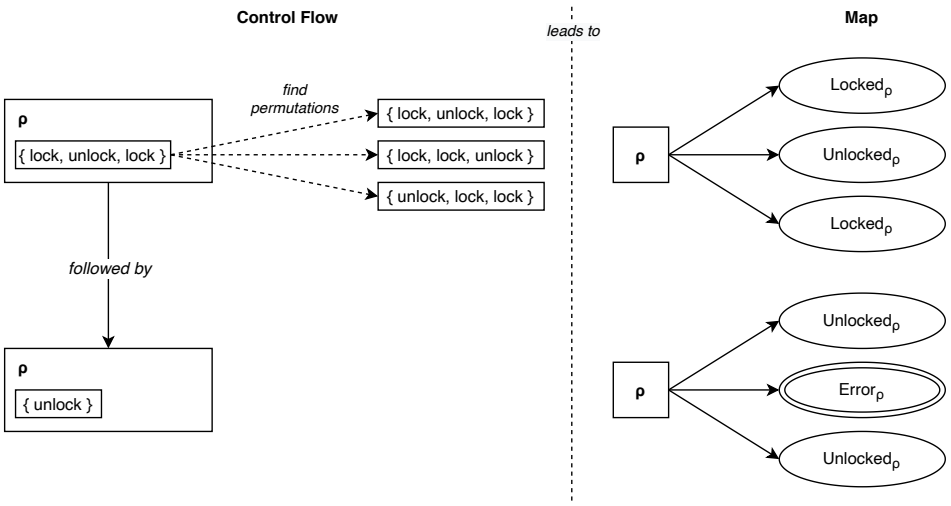

Figure 10: An illustration of how copies of a double-unlock monitor are instantiated given multiple permutations of effect orderings.

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.

Monitors need to be instantiated whenever a new region is encountered and stored in the map. 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.

```
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 ← 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

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

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

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 $transition : checker\_state \rightarrow effect \rightarrow step \rightarrow checker\_state$. A concrete example of a *checker_state* structure can be seen in Figure 11.

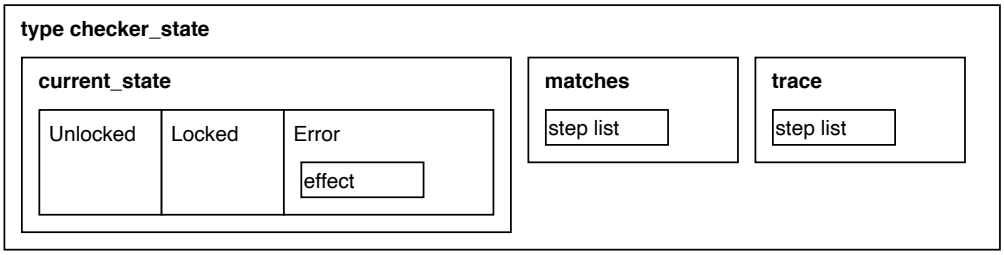

Figure 11: An illustration of the *checker_state* structure for a double-unlock monitor.

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