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

Anders Fischer-Nielsen
*afin@itu.dk*

KISPECI1SE

June 2020

## Abstract

I present the concept and the implementation of *monitor templates*, a way of expressing bug checkers operating on code — here on the Linux kernel. Monitor templates define properties of source code leading to resource manipulation bugs on memory cells, allowing detecting and reporting these bugs. I define monitor templates and their implementation on the example of the Linux kernel by extending the *shape-and-effect* analysis by Iago Abal et. al. [1]. Monitor templates are deterministic finite automata specified on so-called *effects* inferred from executing program points in the control flow of a program. I present an implemented reference implementation of such monitor templates as an extension to the work by Abal et. al. and an evaluation of the resulting bug checker on the Linux kernel showing greater expressibility of buc checkers and a higher accuracy for detection of bugs compared to a previous approach based on the existing theory by Abal. et. al. [9], albeit with a greater number of false positives being present as well.

# Contents

# 1 Introduction

Static analysis can aid in catching problems in programs early, allowing developers to find and fix errors before runtime. The value of static analysis has been noticed by developers and is seeing adoption in the shape of recent smart linters, type and error checkers for dynamic languages, and is even being integrated directly into compilers as in the case with the GCC compiler in version 10 [11]. GCC is the compiler of choice for a multitude of projects written in the C programming language, such as the Linux kernel development project.

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.

In this thesis I will answer the question: *"How can bug checkers utilizing shape-and-effect inference be defined using finite automata with greater expressibility, how can such bug checkers be implemented as an extension to the Effective Bug Finding (EBA) framework operating on the Linux kernel and how effective are such checker definitions?"*

This thesis will present the definition of *monitor templates* and an implementation of such monitor templates as an extension of the work by Abal et. al. [1]. The implementation is incomplete, in that it detects some bugs, but not every bug. An incomplete tool which detects some but not all bugs in the source code under analysis is still of value to developers, since this could lead to detecting unknown programming errors — detecting some bugs is beneficial compared to detecting no bugs. Such a tool should be fast, allowing quick detection and integration into deployment tools. The amount of time a tool takes to analyse a code base negatively impacts developers' time and might therefore limit the adoption of the tool. Speed should hopefully therefore increase the use of the tool.

The implementation is evaluated by comparing the detection rate of bugs on confirmed positives — files which contain bugs reported by the Linux kernel developer community which have then been fixed — by the implementation in this thesis versus the previous approach based on a subset of Computation Tree Logic (CTL). The previous approach shows limitations in how complex bug checkers can be expressed as well as detecting a small number of confirmed bugs. The expressiveness of monitor templates is also compared against the Computation Tree Logic subset definition of checkers in the previous work, in order to determine whether one approach allows expressing checkers which the other definition does not. The results of this show that monitor templates allow expressing more complex bug checkers with a higher detection rate in confirmed positives.

This thesis is structured as follows. Section 2 details the necessary background knowledge required to define monitor templates based on the *shape-and-effect* inference and related concepts

defined by Abal et. al. Section 3 defines *monitor templates*, how different types of bug checkers can be defined as monitor templates and shows the correctness of this definition. Section 4 describes the implementation of monitor templates in practice both as a concrete algorithm and as an extension to the existing implementation of the EBA framework by Abal. et. al. Section 5 will evaluate the definition of monitor templates and the implementation of a monitor template bug checker. Finally, Section 6 will describe future work and possible extensions to this thesis.

### 1.1 Related Work

EBA by Abal et. al. defines and implements *shape-and-effect inference* on the example of the Linux kernel [1] which this thesis is based upon, but employs a subset of CTL to detect bugs rather than state machines, resulting in less expressibility as shown in this thesis. My work simplifies and extends this by the use of finite state machines.

SMATCH by Engler et. al. defines *meta-level compilation* which can be used to develop system-specific compiler extensions to catch possible bugs in the Linux kernel and is used by the Linux kernel developers currently [8]. However, SMATCH does not employ finite state machines to accomplish this.

Infer, developed for and used by Facebook, is a static analyzer based around symbolic execution and separation logic and is able to detect possible bugs in Java and C/C++/Objective-C code [6]. However, Infer does not use finite state machines to accomplish its analysis.

SDV, developed for Microsoft Inc., encodes API usage rules for kernel-level device drivers in the Windows operating system as state machines by specifying a state space as *state variables* and *events* as transitions between states, enabling the detection of bugs in the Windows kernel drivers [3]. SDV uses SLAM, also developed by Microsoft Inc., as an analysis engine which has also been shown to detect kernel-level bugs in Windows expressed as finite state machines [4]. This thesis has been inspired by the award-winning SLAM project, due to its groundbreaking use of finite state machines to detect bugs at the kernel level. However, SLAM is limited to the Windows kernel while this thesis is on the example of the Linux kernel.

## 2 Background: The EBA Framework

The EBA framework reference provides *shape-and-effect inference* and allows extracting an *effect-annotated control flow graph* (effect-CFG) from a Linux kernel program. The annotated control flow graph can be statically analyzed in order to find possible bugs in the given program. A control-flow-graph is a representation of all paths that might be traversed through a program when executed [2]. A node in the graph represents a program point where edges represent a jump in the control flow.

Figure 1 shows an example of a bug found in the Linux kernel[1] and the data types provided by the framework. These data types have been used in this thesis to develop new resource manipulation bug checkers. This section will explain the definitions by Abal et. al. [1] which I build upon in this thesis in order to develop new bug checkers.

[1] This bug was patched in commit `4dd75b33`.

```
1  static void orphan_delete(struct ubifs_info
   ↪  *c, struct ubifs_orphan *orph)
2  {
3          if (orph->del) {
4                  spin_unlock(&c->orphan_lock);
5                  return;
6          }
7          if (orph->cmt) {
8                  spin_unlock(&c->orphan_lock);
9                  return;
10         }
11 }
12
13 void ubifs_delete_orphan(struct ubifs_info
   ↪  *c, ino_t inum)
14 {
15         orphan_delete(c, orph);
16         spin_unlock(&c->orphan_lock);
17 }
```

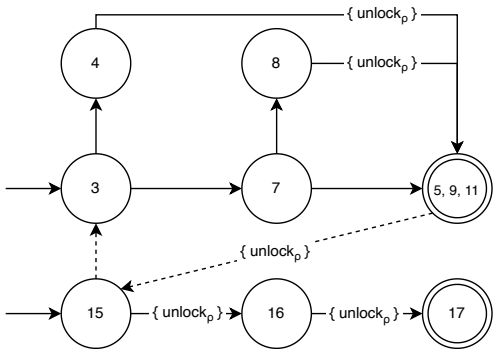


Figure 1: An illustration of a double-unlock bug (`4dd75b33`) found in the Linux kernel and the analysis data types provided by EBA. The relevant code is shown on the left-hand side and on the right-hand side is the *effect-CFG* with *lock* and *unlock effects* provided by EBA. The numbered CFG nodes show the corresponding line numbers.

I build upon EBA in this thesis and though it is generally seen as a black box a certain understanding of definitions described by Abal et. al. [1] is required to see how I build upon the existing work. Three main definitions are of note here, namely *shapes*, *regions* and *effects*. EBA employs CIL [17], a lightweight intermediate representation of the C code implemented in OCaml. Abal et. al define a base type system for a smaller language than CIL. This chapter will provide a summary of the definitions we also use in this thesis to define our work.

The base type language of the *Shape-and-Effect System* is defined to be:

$$\begin{array}{lcl} \text{l-value types } T^L & : & \text{ref } T^R \mid \text{ref } \left(T_1^R \times \cdots \times T_n^R \to T_0^R\right) \\ \text{r-value types } T^R & : & \texttt{int} \mid \text{ptr } T^L \end{array}$$

The l-value ($T^L$) types and r-value ($T^R$) types correspond to the left and right side of assignments in C. A reference type, `ref` $T$ represents a memory cell, holding objects of the type $T$. For example, ptr ref $T$ is the current address of the reference for the objects T in memory. The corresponding tiny programming language is described by the following grammar:

$$\begin{array}{lcl} \text{l-value expressions } L & : & x \mid f \mid *E \\ \text{r-value expressions } E & : & n \mid E_1 + E_2 \mid \texttt{if (}E_0\texttt{) } E_1 \texttt{ else } E_2 \mid (T)E \\ & \mid & \texttt{new } x : T = E_1;\ E_2 \mid !L \mid \&L \mid L_1 := E_2;\ E_3 \\ & \mid & \texttt{fun } T\ f\ (T_1\ x_1, \cdots, T_n\ x_n) = E_1;\ E_2 \mid L_0(E_1, \cdots, E_n) \end{array}$$

L-value expressions (L) represent memory locations and will always be assigned reference types ($T^L$). Function values are immutable, while other variables ($x$) are not. $*E$ represents the dereferencing of a pointer, which is looking up the reference cell in memory, as seen in C. R-value expressions are *values*, such as integers ($n$) and pointers. $(T)E$ is a cast, as found in C, and will convert the value $E$ to the type $T$.*new* $x : T = E_1; E_2$ represents the introduction of a new

variable, $x$, which is initialized in $E_1$ and available in $E_2$. $x$ is the name of the memory cell where the value of $E_1$ is stored, and has the type `ref` $T$. The expression $!L$ will read an l-value, and pointer values can be obtained with $\&L$. The assignment expression $L_1 := E_2$; $E_3$ allows assigning a new value $E_2$ to the value $L_1$ before evaluating $E_3$. The declaration of a function, $f$, `fun` $T\ f\ (T_1\ x_1, \cdots, T_n\ x_n) = E_1$; $E_2$, $f$ will then be visible in $E_2$, similar to `new`. The function $f$ will bind the parameters $x_1, \cdots, x_n$ and evaluate the body expression $E_1$. Loops and `goto`s are not modelled in this system.

## 2.1 Regions

Regions are an abstract representation of memory. Variables are names for memory cells on the stack. Aliasing — when multiple variable names actually point to the same memory — can therefore happen. These possibly aliased memory cells are tracked by the shape-and-effect system using regions. The system will assign a region, $\rho$, to each reference value in the source code, and attempt to detect aliased variables, by unioning these regions when it can no longer distinguish the regions.

## 2.2 Effects

Effects represent how expressions affect regions. For example, an expression which reads a memory location will have the effect of reading that region. Likewise, expressions writing to memory locations will have the effect of writing to that region. An example of a set of effects is $\varphi = \{read_\rho, read_{\rho'}, write_{\rho'}\}$, where the region $\rho$ is being read and the region $\rho'$ is being both read and written.

## 2.3 Shapes

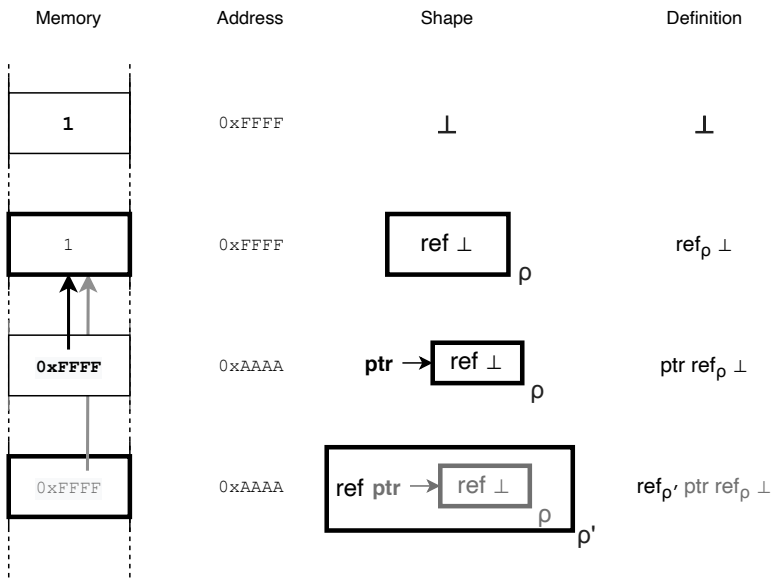


Figure 2: An illustration of *shapes* and how they represent values in memory. Arrows represent pointers, and bold cells refer to the cell itself, not its contents.

A shape approximates the memory representation of an object, and this shape is fixed and kept across type casts. Shapes are annotated with regions showing a `"points-to"` relationship between references. Abal et. al. define shapes in the following terms.

$$\begin{aligned} \text{l-value shapes } Z^L &: \quad \text{ref}_\rho\, Z^R \quad | \quad \text{ref}_\rho\, (Z_1^L \times \cdots \times Z_n^L \xrightarrow{\varphi} Z_0^R) \\ \text{r-value shapes } Z^R &: \quad \bot \quad | \quad \text{ptr } Z^L \quad | \quad \zeta \end{aligned}$$

Shapes are also divided into l-value and r-value shapes in a similar fashion as the aforementioned type system, albeit without an integer type and with shape variables, $\zeta$. An r-value is the shape objects where the *atomic* shape, $\bot$, indicates that an object has no relevant structure — e.g. an integer. A pointer expression has the pointer shape, ptr $Z^L$, where $Z^L$ is the shape of the target reference cell of the pointer. This means that a pointer represents the address of a reference cell, and therefore a pointer shape necessarily encloses a reference shape. If a pointer is being cast, its integer value will then have a pointer shape, ptr $Z^L$. $\zeta$ represents arbitrary r-value shapes. These definitions and how they represent values in memory are illustrated in Figure 2.

Functions receive *function shapes*,

$$\text{ref}_{\rho_0} \left( Z_1^L \times \cdots \times Z_n^L \xrightarrow{\varphi} Z_0^R \right),$$

where the memory region $\rho_0$ identifies the function and is used by EBA to keep track of calls to the function. Functions need to be allocated a region, due to the use of function pointers in C.

Function shapes represent an abstraction of the shapes given to a function as well as the shape of the result of the function. The parameters $Z_1^L \times \cdots \times Z_n^L$ correspond to the shapes of parameters the function is given and $Z_0^R$ is the shape of the result of the function. Since function parameters are stored in stack variables, these parameters are l-value shapes. The *latent effect*, demonstrated as $\xrightarrow{\varphi}$ above represents the effects that may happen when executing the function. Function shape schemes along with the correlation between types and shapes are described in more detail by Abal et. al. [1].

An environment $\Gamma$ maps variables $v$ to their corresponding reference shapes: $\Gamma(v) = ref_\rho Z$. $v$ is effect variables, described in the following. As already mentioned, function shapes are represent effects that may happen when executing a function, but this *may* can be made more concrete using function inlining, in which case the actual effects of calling a function within another function can be inferred concretely. The use of inlining is detailed in Section 4.

## 2.4 Shape-and-effect Inference

Abal et. al. present inference rules $\vdash \subseteq$ ENV $\times$ VALUE $\times$ SHAPE $\times$ EFFECT for *shape-and-effect inference* [1], which allows determining the shape of a given expression as well as determining what the effects of evaluation the expression are. This is expressed as the judgment

$$\Gamma \vdash E : Z \& \varphi$$

specifying that under the enviroment $\Gamma$, the value $E$ has the shape $Z$ resulting in the effects $\varphi$.

In other words, the environment $\Gamma$ keeps track of the values and which effects accompany them. The inference rules defined by Abal et. al., lead to these values having effects accompanying them based on the determining of their shapes. I present two of these inference rules, [FETCH]

and [ASSIGN], in the following to give an intuition of how the inference system works and how effects are produced. The previously shown examples of effects, *read* and *write*, are found by the use of these two inference rules, since [FETCH] and [ASSIGN] produce the effects of reading from or writing to a given memory region, $\rho$.

$$[\text{FETCH}] \quad \frac{\Gamma \vdash L : \text{ref}_\rho Z \& \varphi}{\Gamma \vdash !L : Z \& \varphi \cup \{read_\rho\}}$$

The [FETCH] rule allows, given a reference to a shape $\text{ref}_\rho Z$ in the environment $\Gamma$, that the shape can be dereferenced by the use of the bang-operator !, resulting in the shape $Z$. This has the effect of reading the memory region $\rho$, in turn adding a $read_\rho$ effect to the preexisting effects, $\varphi$. Intuitively preserving the preexisting effects makes sense, since the act of reading a value should only produce effects, not remove preexisting effects of determining that value. Computing the value of the form $\text{ref}_\rho Z$ could have produced other effects by the use of the other inference rules of the system and these effects therefore need to be preserved.

$$[\text{ASSIGN}] \quad \frac{\Gamma \vdash L : \text{ref}_\rho Z \& \varphi_1 \quad \Gamma \vdash E_1 : Z \& \varphi_2 \quad \Gamma \vdash E_2 : Z' \& \varphi_3}{\Gamma \vdash L := E_1; E_2 : Z' \& \varphi_1 \cup \varphi_2 \cup \{write_\rho\} \cup \varphi_3}$$

[ASSIGN] allows assigning a value $E_1$ to the value $L$. This will evaluate both expressions $E_1$ and $E_2$, resulting in effects for both $E_1$ and $E_2$ being determined. The result of the assignment is a $\text{write}_\rho$ effect, which is added to the sets of effects of $E_1$ and $E_2$ as well as existing effects of $L$. The left and right-hand sides of the expression, $L$ and $E_1$ must be of the same shape, $Z$, while the result of evaluating $E_2$ can be of a different shape, $Z'$. While the asasignment of a new value to $L$ will bind the value in the environment, the effects of getting to this stage of the inference are still interesting. Assignment leading to producing an effect and not removing preexisting effects makes sense, since effects are produced by computing the value being written, as well as determining what is being written to. [ASSIGN] will, by the evaluation of $\Gamma \vdash L : \text{ref}_\rho Z$, result in both *read* and *write* effects being added to the effects $\varphi$, since the value of $L$ is dereferenced by assignment.

Abal et. al. axiomize the behaviour of certain operations, $f$, with a signature, $Z_i^L \xrightarrow{\varphi} Z_0$. The axiom specifies shapes of the input arguments expected by the function, $Z_i^L$, the shape of the output, $Z_0$, and the resulting effects, $\varphi$. The authors present an example of two axioms for locking and unlocking functions found within the Linux kernel. These axioms are used extensively by the inference system, in order to infer how resource manipulation is used in the kernel.

$$\begin{aligned} \texttt{spin\_lock}: &\quad \text{ref}_{\rho_1}\text{ptr ref}_{\rho_2}\zeta \xrightarrow{\texttt{lock}_{\rho_2}} \bot \\ \texttt{spin\_unlock}: &\quad \text{ref}_{\rho_1}\text{ptr ref}_{\rho_2}\zeta \xrightarrow{\texttt{unlock}_{\rho_2}} \bot \end{aligned}$$

Axioms have been defined by Abal et. al. for multiple functions in the kernel, allowing inferring what the effects are of using these built-ins. These specify that `spin_lock` and `spin_unlock` receive pointers as arguments, $\rho_1$, pointing to an object, $\rho_2$. The effects above the arrows indicate that the functions `lock` and `unlock` the object in $\rho_2$, respectively.

Shapes and effects are computed efficiently by EBA using type inference. The shape-and-effect system — and by extension this thesis — is implemented for C and not for the tiny language used for the introduction of the definitions above.

### 2.5 Effect-CFG Abstraction

Abal et. al present the *Effect-based Control-Flow Graph ($\varphi$-CFG)*. This is described as a CFG where nodes represent program points and edges specify the control flow, annotated with variables and their memory shapes, and nodes annotated with the effects inferred for their corresponding points.

## 3 An Algorithm for Finding Bugs

In order to detect resource manipulation bugs, I will present an algorithm for detecting these bugs utilizing the region, shape and effect abstractions computed by EBA. I begin with defining *monitor templates* as finite state machines operating on *effects* and *regions*, allowing the definition of buggy behaviour and the detection of such behaviour.

A finite state machine is a tuple $(\sum, S, s_0, \delta, F)$, where $\sum$ is an alphabet, $S$ is a finite non-empty set of states, $s_0$ is an element of $S$ and initial state, $\delta$ 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 $S$. I will use such state machines to represent the code under analysis and the properties I wish to detect in input source files.

### 3.1 Monitor Templates

I use monitor automata to analyze the control flow of a given input source file and detect whether possible bugs are present. A monitor automaton changes state based on what is happening in the control flow of the program. When the automaton reaches an error 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.

EBA infers what effects happen on a memory region, which is an abstract variable or value in the heap. Monitor automata track effects happening on a given region in order to determine whether they could be manifesting buggy behaviour. Both effects and regions must be tracked. For example, it is common to have multiple locks represented by different regions active simultaneously, but a lock on one region followed by a lock on a different region does not neccesarily mean that a locking bug is present unless these locks happen consequently on the same region. Monitor templates are defined formally as follows.

Given a region variable $\rho$, a monitor template is defined as the tuple $X_\rho(\sum, S, s_0, \delta, F, E)$ where $\sum$ is the alphabet of effects registered on memory objects represented by a region $\rho$, $S$ is a finite non-empty set of states, $s_0$ an initial state and an element of $S$, $\delta$ is the state-transition function $\delta : S \times \sum \to S$, $F$ is the non-empty set of final states and a subset of $S$, $E$ is the non-empty set of error states, and a subset of $F$. In other words, $X_\rho(\sum, S, s_0, \delta, F)$ is a finite state machine over regions and effects, augmented with the set of error states $E$. An illustration of such a monitor template can be seen in Figure 3. The monitor templates currently track a single region, $\rho$, but this restriction is not essential and can be lifted if the need arises.

From this definition, I distinguish two kinds of monitor templates: *long-term* and *short-term*. *Short-term* templates will monitor effects happening on a region only until a final state or error state is reached. *Long-term* templates operate as *short-term* templates, though they only include

a single final state, which is the error state. They will therefore monitor indefinititely until only an error-state is found; no early termination is possible. This distinction is made since early termination of monitors might result in performance improvements in the implementation of these monitors, since it may especially reduce the number of monitors active in the memory of the analyzers.

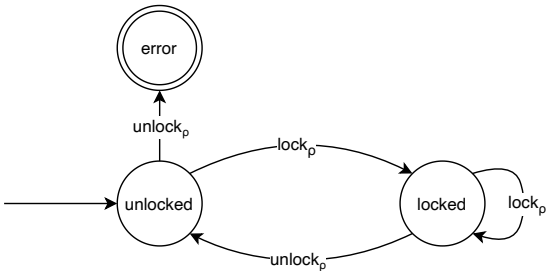


Figure 3: An illustration of a monitor template.

I will present several examples of these monitors in the following sections. Though these are examples, they are important since my implementation of the monitors follows these examples.

Our monitors operate on the set of possible effects of a statement in the Control-flow Graph. EBA allows defining new effects tracking a host of different operation. In this thesis I will only be using the following effects, $E = \{$`alloc`, `free`, `read`, `write`, `uninit`, `call`, `lock`, `unlock`$\}$. The effects and what they represent can be seen in Table 1.

| | |
|---|---|
| `alloc` | The allocation of a memory location |
| `free` | The freeing of a memory location |
| `read` | The reading of a memory location |
| `write` | The writing to a memory location |
| `call` | The call of a function |
| `lock` | The locking of a memory location |
| `unlock` | The unlocking of a memory location |

Table 1: Effects and what they represent.

**Short-term Double-lock Monitor Template**

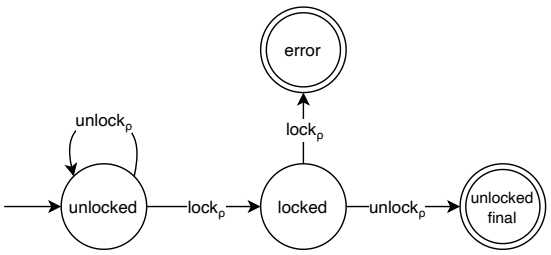


Figure 4: An illustration of a short-term double-lock monitor template.

A double-lock monitor detects two consecutive locks on a memory location with no unlock in between them leading to an infinite spinlock. Given a region $\rho$, a double-lock monitor template is defined as the tuple $(\sum, S, s_0, \delta, E, F)$ where:

- $\sum = \{lock_\rho, unlock_\rho\}$, a subset of $E$
- $S = \{locked, unlocked, error\}$
- $s_0 = unlocked$
- $\delta =$ the relation $\{(unlocked, lock_\rho, locked), (locked, unlock_\rho, unlocked_{final}),$ $(locked, lock_\rho, error), (unlocked, unlock_\rho, unlocked)\}$
- $E = \{error\}$
- $F = \{unlocked, error\}$

It is worth noting that this is a *short-term* monitor, as $F \neq E$. This monitor will therefore terminate when encountering a legal use of locks followed by an unlock. An illustration of this monitor template can be seen in Figure 4.

**Long-term Double-unlock Monitor Template**

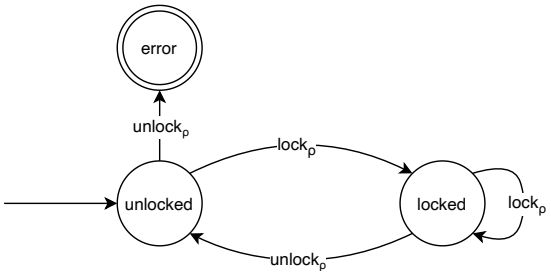


Figure 5: An illustration of a long-term double-unlock monitor template.

A double-unlock monitor detects two consecutive unlocks on a memory location with no lock in between them, leading to undefined behaviour. Given a region $\rho$, a double-unlock monitor template is defined as the tuple $(\sum, S, s_0, \delta, E, F)$ where:

- $\sum = \{unlock, lock\}$, a subset of $E$
- $S = \{locked, unlocked, error\}$
- $s_0 = unlocked$
- $\delta =$ the relation $\{(locked, unlock_\rho, unlocked), (locked, lock_\rho, locked),$ $(unlocked, lock_\rho, locked), (unlocked, unlock_\rho, error)\}$
- $E = \{error_\rho\}$
- $F = E$

It is worth noting that this is a *long-term* monitor, as $F = E$. An illustration of this monitor template can be seen in Figure 5. Note that this monitor template allows multiple consecutive locks, which results in a double-lock bug being present in the code under analysis. This is due to separation of concerns such that a single monitor checks for a single bug type. The monitor templates can be run in combination in order to detect double-lock bugs as well as double-unlock bugs.

**Long-term Double-free Monitor Template**

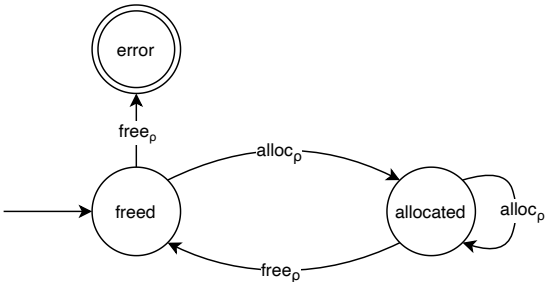


Figure 6: An illustration of a long-term double-free monitor template.

A double-free monitor detects two consecutive frees on a memory location with no allocation in between them, potentially leading to modification of unexpected memory locations. Given a region $\rho$, a double-free monitor template is defined as the tuple $(\sum, S, s_0, \delta, E, F)$ where:

- $\sum = \{free_\rho, alloc_\rho\}$, a subset of $E$
- $S = \{allocated, freed, error\}$
- $s_0 = freed$
- $\delta =$ the relation $\{(freed, alloc_\rho, allocated), (allocated, free_\rho, freed),$ $(freed, free_\rho, error), (allocated, alloc_\rho, allocated)\}$
- $E = \{error\}$
- $F = E$

An illustration of this monitor template can be seen in Figure 6.

**Short-term Use-before-init Monitor Template**

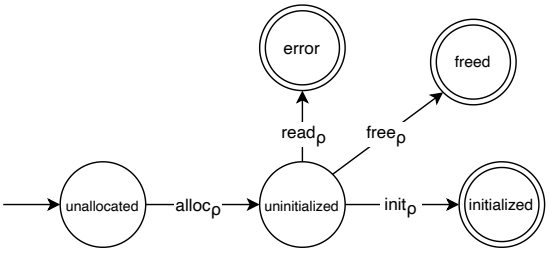


Figure 7: An illustration of a short-term use-before-init monitor template.

A use-before init monitor detects usage of a memory location before the location has been initialized, possibly leaving the resource in an unexpected state when it is accessed or used. Given a region $\rho$, a use-before-init monitor template is defined as the tuple $(\sum, S, s_0, \delta, E, F)$ where:

- $\sum = \{alloc_\rho, read_\rho, write_\rho\}$, a subset of $E$
- $S = \{unallocated, allocated, initialized, freed, error\}$
- $s_0 = unallocated$
- $\delta =$ the relation $\{(unallocated, alloc_\rho, allocated), (allocated, write_\rho, initialized), (allocated, free_\rho, freed), (allocated, read_\rho, error)\}$
- $E = \{error\}$
- $F = \{error, initialized, freed\}$

An illustration of this monitor template can be seen in Figure 7.

## 3.2 Control Flow

EBA provides a representation of the control flow of the input source files which is utilized to detect bugs. EBA generates a tree structure of the input with each path in this tree structure modeling a possible execution path containing information about the modelled statements.

The control flow graph of a program can be seen as a finite state machine $(\sum, S, s_0, \delta, F)$, where $\sum$ is the powerset of effects, $S$ is a finite non-empty set of program points, $s_0$ is an element of $S$ representing the entry point, $\delta$ is the state-transition function $\delta : S \times \sum \to S$ reflecting the edges of the control graph labeled by effects produced by computations and $F$ is the empty set of final states.

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 powerset of effects, $\sum$, 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. $S$ is the set of program points. 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 8.

- $\sum = P\{\mathcal{E}_{\rho_i} | i \in \mathbb{N} \text{ and } \mathcal{E} \in \{alloc, free, read, write, call, lock, unlock\}\}$
- $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$

The remainder of the automaton is defined according to the control flow being modelled, where the initial state, $s_0$, is the entry point. A concrete definition of an example control flow is shown below with an accompanying illustration of this in Figure 8.

- $s_0 = 1$

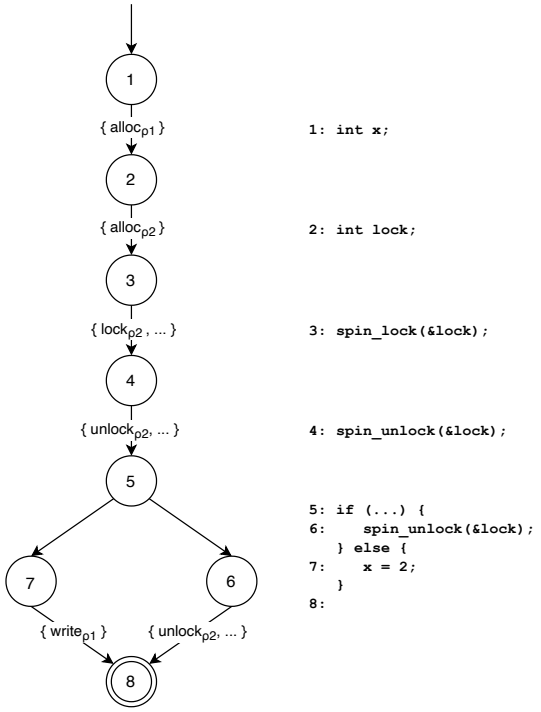


Figure 8: An illustration of a Control Flow automaton.

- $\delta =$ the relation $\{ \quad (1, \{alloc_{\rho_1}\}, 2), \quad (2, \{alloc_{\rho_2}\}, 3),$
  $(3, \{lock_{\rho_2}, ...\}, 4), \quad (4, \{unlock_{\rho_2}, ...\}, 5)$
  $(5, \emptyset, 6), \quad (5, \emptyset, 7),$
  $(6, \{unlock_{\rho_2}, ...\}, 8), \quad (7, \{write_{\rho_1}\}, 8) \quad \}$
- $F = \emptyset$

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 we find the product of the control flow example shown in Figure 8 and the monitor generated from the template.

Given a region $\rho$, a monitor template $A_{monitor} = (\sum, S, s_0, \delta, F)$ and an automaton $A_{CFG} = (\sum', S', s_0', \delta', \emptyset)$, the product automaton of $A_{CFG}$ and $A_{monitor}$ is an automaton $P = (\sum, Q, s_0', \delta_P, F_P)$ where

- $\mathcal{E}_{\bar{\rho}}$ is created from $\mathcal{E}$ by substituting its template parameter with an actual value $\bar{\rho}$ by instantiating the template for $\bar{\rho}$
- $F_P = S \times F'$
- $Q = S \times S'$

- $\delta_P : Q \times \sum \to Q$ which is generated for all $s_1, s_2 \in S$, $s_1', s_2' \in S'$, $e \in \sum$, $E' \in \sum'$ and $\forall q \in Q, q' \in Q'$ following two rules

$$\frac{s_1 \xrightarrow{\bar{e}} s_2 \quad s_1' \xrightarrow{E} s_2' \quad \bar{e} \in E}{(s_1, s_1') \xrightarrow{E} (s_2, s_2')} \qquad \frac{s_1' \xrightarrow{E} s_2' \quad \forall e \in E \,.\, s_1 \not\xrightarrow{\bar{e}}}{(s_1, s_1') \xrightarrow{E} (s_1, s_2')}$$

where $\bar{e} = e[\bar{\rho}/\rho]$ for some $e \in \sum$ by substitution of the template parameter.

It is necessary to define rules for the state changes within this product, given that a monitor only accepts effects on a given region. $E$ is the set of effects happening in a given program step, $s_1' \to s_2'$. The monitor state $s_1$ 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 $s_2'$, while the monitor has not and stays the same, i.e. $s_1$.

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*. The observant reader might notice that if regions are no longer present since they go out of scope in a given program, only the rightmost of the two previous inference rules is relevant. This is an opportunity for an optimization in a bug checker, which is presently not exploited. Taking scopes into account would complicate the product construction significantly and the reader might appreciate not having to reason about more convoluted definitions. Lack of scope information does lead to the question of whether these *"dangling"* regions incur a significant performance cost when implemented. I will investigate and evaluate this in later sections.

The product of the control flow example shown in Figure 8 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 9.

- $\sum = \{alloc_{\rho_1}, write_{\rho_1}, alloc_{\rho_2}, lock_{\rho_2}, unlock_{\rho_2}\}$
- $S = \{ (unlocked, 1), (unlocked, 2), (unlocked, 3), (locked, 4), (unlocked, 5), (unlocked, 6), (unlocked, 7), (error, 8), (unlocked, 8) \}$
- $s_0 = (unlocked, 1)$
- $\delta =$ the relation $\{ ((unlocked, 1), alloc_{\rho_1}, (unlocked, 2)),$
  $((unlocked, 2), alloc_{\rho_2}, (unlocked, 3)),$
  $((unlocked, 3), lock_{\rho_2}, (locked, 4)),$
  $((locked, 4), unlock_{\rho_2}(unlocked, 5)),$
  $((unlocked, 5), \emptyset, (unlocked, 6)),$
  $((unlocked, 5), \emptyset, (unlocked, 7)),$
  $((unlocked, 7), unlock_{\rho_2}, (error, 8)),$
  $((unlocked, 7), write_{\rho_1}, (unlocked, 8)) \}$
- $F = (error, 8)$

This product is illustrated in Figure 9.

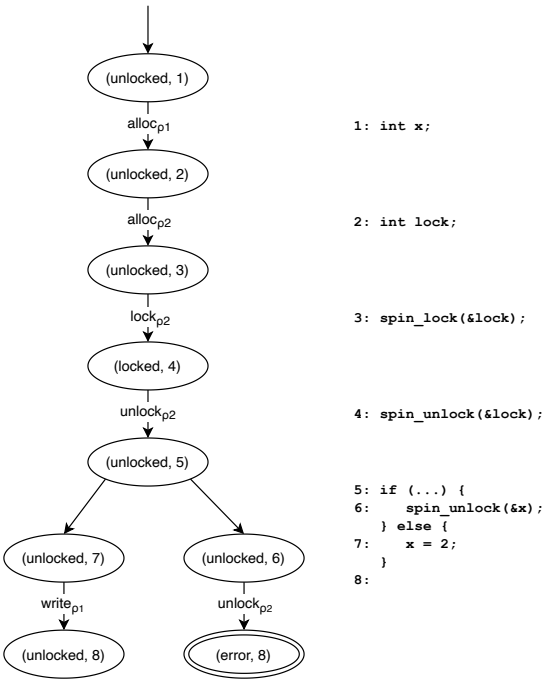


Figure 9: An illustration of the product construction of a double-unlock monitor automata and a control flow automaton.

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 this thesis in the following section.

# 4 Finding Double-Unlock Bugs in Practice

This section will detail how I have implemented the abstractions defined previously and integrated this implementation into the EBA framework in order to explore the product of the control flow graph of input programs and monitor automata to detect possible bugs.

## 4.1 Implementing Monitor Templates

Monitor automata are used to detect bugs over control flow graphs. The control flow graph is provided by the EBA framework, while the monitor automata have been defined and implemented

as part of this thesis. The framework will then instantiate a given bug checker after which an abstract representation of the input source file is passed to the instantiated checker.

When inferred effects happening on regions are encountered when exploring the control flow graph, a monitor is instantiated based on the monitor template. The CFG provided by the EBA framework of the given file is explored and the transition function of the monitor is applied using the effects of statements, represented as nodes in the CFG. The EBA control flow graph is represented as a tree-like structure and this tree is then explored further until the end of each path in the tree is explored, resulting in a set of monitor states. Loops in the input source code for programs are unrolled by EBA and are represented as repeating nodes on a path in the structure, allowing the control flow to be represented as a tree. This set of states can after exploration be filtered to determine if any monitors have reached accepting — Error — states. If such states are present, possible bugs have been discovered.

EBA can generate the required control flow abstraction, which is used for analysis. I have integrated 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 these control flow abstractions.

EBA generates the aforementioned tree-like structure of the input program, 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 program point.

A *step* models a program point in the input source code, and contains the inferred effects of this program point as a set. 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. The reader is encouraged to examine the example given in Figure 10 to understand why this is the case. In the example, two locks are taken on two different regions and these regions are then both unlocked in a called function. One of the regions is then unlocked again following the function call, leading to a double-unlock bug. If monitors are not instantiated for each region, an error state would occur when analyzing the effects of the function call even though the bug is in fact not found within the function, but just after the function. Recall that we use a simple effect system and a set of effects to summarize a step including function calls, as detailed in Section 2.

```
int lock1;
int lock2;

void unlock_func()
{
    _spin_unlock(&lock1);
    _spin_unlock(&lock2);
}

int main()
{
    _spin_lock(&lock1);
    _spin_lock(&lock2);
    unlock_func();
    _spin_unlock(&lock2);
}
```

Figure 10: An example of multiple locks happening on different regions, leading to a bug on one region, but not the other.

All permutations of the set of effects must therefore be found and mapped to a given region while 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 be stored for that region, as defined in the product rules previously in Section 3.2.

In order to accomplish this, the current state of the monitor monitoring a region needs to be copied and applied as the current state of the monitor for each permutation found for a given set of effects. This is done in order to find all possible states of the monitor when given all effect orderings as input. This can also be thought of as if the control flow is being interpreted as non-deterministic. Recall that the rules for computing the transition relation of a product in Section 3.2 are non-deterministic, since the syncronizing effect is picked from a set non-deterministically.

The resulting states are then stored in the map for a given region, and future effects on that region are then applied on these states. This corresponds to copying a monitor, in other words executing a subset construction. An illustration of this copying on two permutations followed by another effect happening on the same region can be seen in Figure 11, leading to a possible double-unlock bug. This provides information regarding the presence of bugs, and inlining can be used after gaining this information in order to determine whether a bug is present. This information presents itself as one of three cases:

1. All possible orderings lead to all copies of monitors being in an error state
2. All possible orderings lead to all copies of monitors not being in an error state
3. All possible orderings lead to some copies of monitors showing an error and some not, meaning that the output is inconclusive.

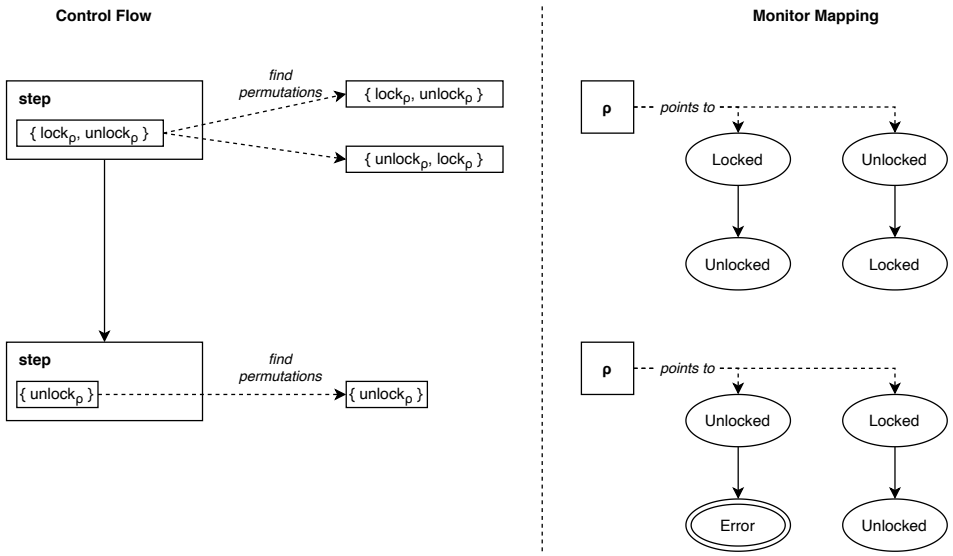


Figure 11: An illustration of the exploration of all effect orderings and how copies of an instantiated double-unlock monitor changes state given these effect orderings, leading to a possible bug being discovered in one effect ordering, but not the other. The control flow graph is shown on the left. On the right I show execution fragments of two double-unlock monitors as defined in 3.1.

If the first scenario occurs, a possible bug is definitely present, and exploration can stop. If the second scenario occurs, a possible bug is definitely not present, and exploration can continue. If the third scenario occurs, it is unknown whether a possible bug is present, and more information is needed. This extra information is obtained by inlining the given program point to gain more information regarding the order of effects and applying the effects to the transition function of the monitors once more, making it possible to determine whether a possible bug is present or not. Early use of inlining or inlining of every function leads to the Linux kernel libraries being inlined into the code under analysis, giving false positives. Inlining is therefore only used when it will benefit the precision of the analysis.

The region monitors are instantiated upon and the copies of these monitors are stored in a map from a region to the list of copies. This 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 are instantiated whenever a new region is encountered and are then stored in this map. If a given region is already present in the map, then the current state of the monitor(s) monitoring the region are applied on the transition function of the monitor(s). If the region is not already present in the map, a monitor is instantiated for each effect happening in the program point where this region was encountered. Given that the map maps a region to the state of monitor automata, 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, though this can be improved by filtering effects, as we will see shortly.

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 states for a given region will therefore in the worst case be $|\mathit{effects}|!$. This $N$ can be reduced by only applying effects to monitors which are of interest, that is effects which are in the alphabet $\sum$ of the monitor template definition. Effects are only applied to the monitors monitoring a region if the alphabet of the monitor in question accepts this effect, in other words utilizing the product construction inference rules defined in Chapter 3.2. Filtering effects caused by a program point to only include elements in $\sum$ greatly reduces the number of monitor states, though this is inherently dependent on the template definition. Definitions with a restricted alphabet will perform better.

The full implementation of monitor templates therefore works by:

- Taking a source file and a function declaration within that source file, both provided by EBA
- Extracting information about the provided function and verifying that the function is non-static
- Invoking the CFG-generation of EBA of the provided function
- Exploring the effect CFG provided by EBA while invoking the transition function of monitor templates
- Filtering the resulting map of monitor states
- Pretty-printing the results and returning these to EBA, in turn outputting the results

The implementation of the exploration of control flow, finding states and adding these states to the map can be seen in Figure 12. This `explore_paths` function takes a control flow graph generated by EBA and an initially empty map as input parameters and results in a map of all inferred regions and the resulting monitor states for these regions. This is accomplished by recursively exploring the tree-like control flow graph and depending on the encountered program point applying the possible effects of the program point on the transition function of the instantiated monitor. The resulting state(s) are then added to the map.

```
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 MERGE(true_branch, false_branch)
    else if tree_node is Seq(step, next) then
        effects ← step.effects
        if IS_EMPTY(effects) then return EXPLORE_PATHS(next, map)
        end if
        permutations ← FIND_PERMUTATIONS(effects)
        FOR EACH EFFECT IN EACH PERMUTATION(APPLY_TRANSITION(effect, map))
    end if
end function

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

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

Figure 12: Pseudocode illustrating an implementation of the EBA control flow while keeping track of states for regions. This function takes a control flow graph generated by EBA and an initially empty map as input parameters and results in a map of all inferred regions and the resulting monitor states for these regions.

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.

On-the-fly exploration is used since the design of EBA necessitates exploring effects when they are encountered due to performance reasons [1]. On-the-fly exploration has been used for ease-of-implementation by following the existing convention of the EBA implementation and for performance reasons. The exploration finds the product construction of the control flow graph and monitor on a step-by-step basis by application of the transition function of a monitor and the information stored in a program point in the control flow graph. The resulting state pair is the product construction of the control flow graph and monitor as decribed in Chapter 3.1.

The monitor state in the second element of this pair is, as mentioned previously, added to the region and state map in order to track monitor states for regions. The product construction

is built step-by-step, but the complete product construction is not saved for later use, as the elements of interest in the resulting pair is the found monitor state. The inference rules defined in Chapter 3.1 for constructing the product, state that a transition — the exploration of the program points — will change the state of the control flow graph, but possibly leave the monitor state in its current state if the effects of a program point are not in the transitions of the monitor. This is reflected in the implementation, where monitors remain in their current state given an input that the monitor does not operate on. Therefore, even though the product construction is not stored or presented explicitly in the implementation, it is still constructed albeit step-by-step and is in a partial state on a given program point.

Consider the illustration in Chapter 3.1, with the relevant part of this illustration shown in Figure 13. When exploring the control flow graph, on encountering the effect `unlock` and region $\rho$, a monitor is instantiated for $\rho$ in the unlocked state. If a given effect is not in the transitions of the monitor, as seen when encountering the effect `write`, the exploration continues onto the next program point, implicitly changing the state of the control flow graph by transitioning onto the next node in the graph, while leaving the monitor in its previous state as the inference rules in Chapter 3.1 state. When encountering an `unlock` effect, which is in the transitions of the monitor, the monitor will change state to an error state. This corresponds to the defined product construction, though instead of constructing the entire product, this partial product is found on-the-fly.

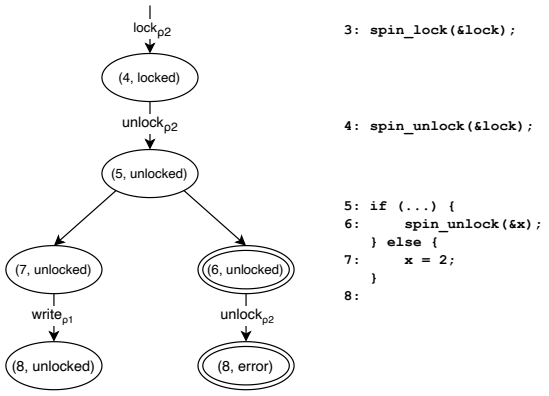


Figure 13: An illustration of the partial product construction of a double-unlock monitor automata and a control flow, based on Figure 9.

## 4.2 Implementing Monitor Templates as Extensions to EBA

In order to implement a new bug checker in EBA, the implementation has to conform to the structure set up by the framework of how these checkers should behave. Furthermore, OCaml signatures must be defined for these checkers in order to allow the framework to instantiate the checkers on the inferred regions of an input program. Specifying a signature which all automata-based checkers must conform to ensures that the automata expose the required state transition functions for them to run. This section will describe how this has been accomplished. The implementation can be seen in its entirety on GitHub at `https://github.com/andersfischernielsen/eba`.

Monitor automata are used to detect bugs over control flow graphs. The control flow graph is provided by the EBA framework, while the monitor automata have been defined and implemented

as part of this thesis. 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.

A signature which allows instantiation of monitor automata bug checkers has been defined as part of this thesis. 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. A snippet of the `Make` signature is shown in Figure 14.

```
module type S = sig
    type result
    val check : AFile.t -> Cil.fundec -> result list
    val filter_results : result list -> result list
    val stringify_results : result list -> string list
end

module Make (A : AutomataSpec) : S = struct
    type result = A.checker_state
    let check file declaration = ...
    let filter_results matches = A.filter_results matches
    let stringify_results matches = ...
end
```

Figure 14: A snippet of the `Make` module and its use of dependency injection for the instantiation of a checker conforming to the `AutomataSpec` signature.

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. The `Make` module expects an implementation of this `AutomataSpec` signature.
The implementation of `check` initiating the generation of the control flow abstraction using EBA can be seen in Figure 15. This function can be summarized as:

- Taking a source file, `file` and a function declaration, `declaration` within that source file provided by EBA
- Extracting information, `variable_info`, about the provided function and verifies that the function is non-static
- Invoking the `paths_of` CFG-generation of EBA of the function
- Exploring the effect CFG provided by EBA, `explore_paths`, while invoking the transition function of a monitor template
- Filtering the resulting map of regions and monitor states
- Pretty-printing the results and returns these to EBA which in turn shows the results

```
let check file declaration =
    let variable_info = Cil.(declaration.svar) in
    match variable_info.vstorage with
    | Static -> L.of_list []
    | _ ->
        let _, global_function = Option.get(AFile.find_fun
            file variable_info) in
        let path_tree = paths_of global_function in
        let results = explore_paths global_function path_tree
            Map.empty true in
        let states = Map.values results in
        let matches = Enum.fold (fun acc m ->
            (List.filter A.is_accepting m) @ acc) [] states in
        let function_name = Cil.(variable_info.vname) in
        let print = List.map (fun m -> A.pp_checker_state m function_name)
            matches in
        let pp_list = List.map (fun m -> PP.to_string m) print in
        L.of_list pp_list
```

Figure 15: The implementation of `check` initializing the control flow generation using EBA.

Nodes in the CFG structure provided by EBA can be of 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.

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 $transition : checker\_state \rightarrow effect \rightarrow step \rightarrow checker\_state$. No references to instantiated monitor modules are stored in the map, just the current states of monitors.

```
type state =
    | Locked
    | Unlocked
    | Error of Effects.e
```

```
type checker_state = {
    current_state: state;
    trace: step list;
    matches: step list;
}
```

Figure 16: An illustration of the *state* and *checker_state* structures for a double-unlock monitor.

A concrete example of a *checker_state* structure can be seen in Figure 16. This transition function in other words operates on an effect which is part of the set of effect types and results in a new monitor state which is part of the set of possible states defined within the monitor, reflecting the definition of a monitor template seen in Section 3.1. A concrete example of an implementation of the transition function can be seen in Figure 17.

```
let transition previous input step =
    let next new_state = with_previous previous new_state step in
    let previous_state = previous.current_state in
    match previous_state with
    | Unlocked ->
        (match input with
        | Mem(Lock, _)      -> next Locked
        | Mem(Unlock, _)    -> next (Error input)
        | _                 -> next previous_state
        )
    | Locked ->
        (match input with
        | Mem(Unlock, _)    -> next Unlocked
        | _                 -> next previous_state
        )
    | Error _               -> next previous_state
```

Figure 17: The implementation of the transition function of the double-unlock monitor template definition.

This implementation has been chosen for performance reasons and simplicity, since the transition function of a monitor does not change after it has been defined and can therefore be implemented as a performant static function. Storing only previous states and taking this previous state into consideration in the implementation of transition functions results in a lower memory footprint and possibly higher readability of the implementation since an alternative implementation, e.g. based on dynamically assigning transitions to an internal map within the monitor, would increase the memory footprint and possible make it hard to track what the possible states for a given monitor are for the reader.

It became apparent during testing of the implementation that the loop unrolling by EBA resulted in false positives being detected. The unrolling of loops in EBA causes false positives since monitor templates register loops as multiple repetitions of the same effect in a path due to the way unrolled loops are represented. An effect $e$ of a program point in a loop will be unrolled to

a path of $N$ length with each program point in that path having the effect $e$. Figure 18 shows an illustration of this problem. In the case of unlocks, any unlock happening within a loop is therefore reported as a double unlock, since the path under examination will have $N$ unlocks in sequence due to the structure of the effect-CFG. This $N$ is configurable in EBA, though only with an $N > 1$. Filtering has been implemented so such paths are detected and filtered, improving the quality of the output, in turn making the implementation more desirable to use [14]. This filtering examines the trace of all positives resulting of checking a file, and will remove any positive which has a trace consisting of the same program point, in effect detecting the original unrolled loop path. Furthermore, kill-region detection has been implemented in line with the kill-region detection implementation found in the existing CTL checkers of EBA [1]. Kill-region detection as well as detection and filtering of undesirable effects of the loop unrolling found in EBA has reduced the amount of false positives significantly, in turn improving the output of bug checkers significantly.

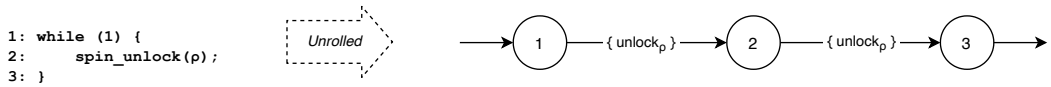


Figure 18: An illustration of the unrolling of loops found in EBA, resulting in a path containing the same effect of a program point in a loop repeated in sequence. An unlock effect of a program point in a loop will be unrolled to a path of length $N = 2$ with each program point in that path having the same effect.

# 5 Evaluation

## 5.1 Expressiveness

In this section I want to answer part of my research question: *"How can bug checkers be defined using finite automata with* ***greater expressibility****?"*. I show how monitor templates are more expressive compared to the the CTL subset template formula found in EBA by providing examples of finite automata which cannot be expressed using the CTL template formula. It is worth noting that I am not comparing the entirety of CTL and monitor templates, but the subset of CTL used to define bug checkers found in EBA.

In the implementation of the CTL template bug checker specifications by Abal et. al., the checkers implement four predicates, forming the CTL template formula of the shape $a$ EU ($b \wedge EX$ ($c$ EU $d$)) [1] [9]. This formula can also be written as $E\ a$ U ($b \wedge EX$ ($E\ c$ U $d$)). This limits the expressiveness of checks which can be defined, given that the logic for detecting bugs must be implemented within these four functions $a$, $b$, $c$ and $d$. Specifying the checkers as monitor templates therefore allows for greater expressiveness in the checker definition. To illustrate this, Figure 19 below shows a monitor template definition which cannot be defined in the existing CTL implementation of the EBA framework because there is no way to express the circular alternation.

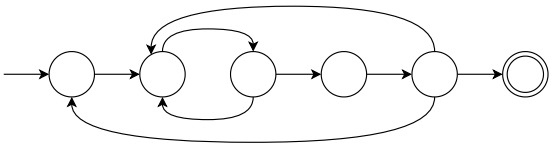

Figure 19: An illustration of a monitor template which cannot be expressed by the CTL template formula $a$ EU $(b \ \wedge \ EX \ (c$ EU $d))$.

Another example of a structure that the CTL template formulation cannot model is a very long sequence of given effects, e.g. an unlock and a lock repeatedly, followed by a final effect. This can be modelled using a monitor template defined as a loop between effects, going to the accepting state on the final effect. This is illustrated in Figure 20.

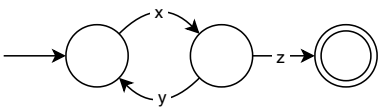


Figure 20: An illustration of a monitor template which cannot be expressed by the CTL template formula $a$ EU $(b \ \wedge \ EX \ (c$ EU $d))$.

I want to argue why this figure cannot be expressed in the CTL template. The CTL template has a shape with two EU operators, $a$ EU (...) and $c$ EU $d$. Each of these operators enforce that the rightmost argument appears, but not necessarily the left, e.g. in $c$ EU $d$, $d$ must appear while $c$ may or may not appear. Moreover, the left element can appear more than once. In the CTL template we only have two positions where we can enforce appearance and prevent repetition, on the righthand side of each EU. The monitor template in Figure 20 requires strict alternation between any number of $x$ and $y$, so before any $y$ there is an $x$. Without a looping construct in the CTL template and only two positions where we force appearance and not allow repetition, we can not express any alternations between repeating symbols of a length greater than two.

On the other hand, the CTL template formula $a$ EU $(b \ \wedge \ EX \ (c$ EU $d))$ *can* be expressed as a monitor template, as seen in Figure 21.

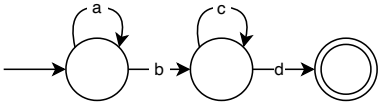


Figure 21: An illustration of a monitor template equivalent to the CTL formula $a$ EU $b$ X$(c$ EU $d)$.

This expressiveness allows specifying the CTL template checkers defined in the work by Abal et. al and also possible future more granular checkers, in other words increasing the posibilities of extending the EBA framework. Certain bug types can show themselves in different code constructions, such as a double-unlock bug being defined as both an unlock happening before

any locks happening and explicit double-unlocks. The expressiveness of monitor templates allows specifying multiple types of behaviour in the code under analysis for a single bug type as more complex transition function in order to increase the precision of checks.

## 5.2 Accuracy

In this section I want to answer the part of my research question: *"How effective are such monitor template-based checker definitions?"*. I accomplish this by evaluating the implementation of a double-unlock bug checker using a monitor template on Linux kernel files known to contain double-unlock bugs. Effectiveness in this case means *recall* since I am trying to detect actual positives in a set exclusively containing positives. Furthermore, the false positive rate is also a measure of effectiveness, since many false positives effectively reduce the overall quality of the output. Finding the precision of the double-unlock bug checker would require analysing all files in the Linux kernel and validating whether the reported positives are true or false positives by reading through and gaining an understanding for all files in the kernel — a considerable undertaking consuming too much time.

Experimental static analysis has recently been added to the GCC compiler chain [11], allowing developers to run check for double-free bugs in their code. David Malcolm — the developer of the static analysis released in GCC 10 — has described his challenges in reducing the amount of false positives during the development of the analysis [16], showing that reducing these proves to be a difficult problem. Reducing false positives is important, since developers might avoid using a tool if they see false output too often. The developer of the popular `curl` command-line tool confirms this, noting that the addition of the analysis in GCC 10 is appreciated, but that it still produces too many false positives to be usable. [18]. Reducing the amount of false positives in the implementation of monitor templates has been difficult, since a tradeoff between the ability to detect more bugs and reducing the amount of false positives has to be made. In other words, increasing precision is hard. We see that the actual bugs are present, but false positives are also reported.

The implementation has been evaluated by assembling a set of patched double-unlock bugs found in the Linux kernel source control, *git*. Kernel developers submit patches as *commits*, containing an message explaining what the patch addresses. These messages have been explored in search of the phrase *"double unlock"*, resulting in 16 matching patches. These patches can be seen in Table 2. These patches can be *checked out* using git, in turn reverting the source code repository to the state where the patch was submitted. The source code repository can then be rolled back further to the commit just prior to the patch, in turn reverting the repository to a state where the bug is present. Once this has been done, the file under analysis can be compiled and analysed by the EBA framework with the implementation of the double-unlock checker.

This process has been automated using the Docker tool in order to evaluate tests in a reproducable manner [5]. A base Linux Docker *image* is used to provide the required Linux libraries for compilation and files are then compiled and extracted from this Docker image for analysis by EBA with and without the extensions described in this thesis. The results of analysis using the previous approach and my approach are then stored in text files for easy comparison for each file under analysis.

| Linux Kernel File | Lines | Patched in | Present in | Time | Mine | EBA | FP |
|---|---|---|---|---|---|---|---|
| fs/ubifs/orphan.c | 1042 | 4dd75b33 | 7542c6de | 1.03s | + | + | 0 |
| drivers/block/drbd/drbd_main.c | 3928 | 8e9c5230 | b0814361 | 25.93s | + | - | 6 |
| drivers/gpu/drm/nouveau/nouveau_svm.c | 875 | de4ee728 | 5fbcf501 | 0.97s | - | - | 0 |
| fs/btrfs/file.c | 3358 | f49aa1de | 78e03651 | 2.93s | - | - | 0 |
| drivers/staging/wilc1000/wilc_wlan.c | 1346 | fea69916 | ca641bae | 1.26s | + | - | 1 |
| drivers/staging/kpc2000/kpc_dma/fileops.c | 420 | c85aa326 | d4c596eb | 0.74s | - | - | 0 |
| fs/nfs/client.c | 1311 | c260121a | a46126cc | 1.79s | - | - | 1 |
| fs/btrfs/file.c | 3358 | 8fca9550 | 2b90883c | 2.93s | - | - | 0 |
| drivers/media/dvb-core/dvbdev.c | 1076 | 122d0e8d | ded71626 | 0.55s | % | % | 0 |
| mm/memory_hotplug.c | 1867 | e3df4c6e | 6376360e | 1.51s | - | - | 0 |
| sound/soc/codecs/pcm512x.c | 1721 | 28b698b7 | fd270fca | 1.06s | - | - | 0 |
| drivers/target/target_core_user.c | 2735 | f0e89aae | 807cf197 | 0.82s | % | % | 0 |
| drivers/rpmsg/qcom_smd.c | 1532 | c3388a07 | fb416f69 | 0.40s | % | % | 0 |
| drivers/scsi/aacraid/commsup.c | 2615 | d844752e | 09624645 | 5.51s | - | - | 2 |
| drivers/staging/rtl8188eu/os_dep/usb_intf.c | 531 | 23bf4042 | 612e1c94 | - | ? | ? | - |
| block/blk-cgroup.c | 1422 | bbb427e3 | e0223003 | - | ? | ? | - |

Table 2: The results of comparing the previous CTL-based approach to my monitor-template-based approach. The symbol + indicates that the bug was detected, **-** indicates that the bug was not detected, **%** indicates an internal error in EBA and **?** indicates a compilation error. The column named FP indicates the number of false positives by the implementation using monitor templates reported in a file.

The files `drbd_main.c`, `orphan.c` and `wilc_wlan.c` are successfully detected. These files have a double-unlock in an if-statement which, when true, results in a double-unlock bug. In the case of the file `drbd_main.c` by the use of mutexes, by a spin-unlock in the file `orphan.c`.

The files `nouveau_svm.c`, `fileops.c`, `file.c` and `memory_hotplug.c` are not detected by either approach. These files exhibit a double-unlock bug when certain error codes are reported during execution, but unlocking happens by the use of an external function and is therefore not detected as an unlock by EBA.

The files `pcm512x.c` and `target_core_user.c` contain double mutex unlocks similar to `drbd_main.c`, but these bugs are within static functions and are therefore ignored. The files `qcom_smd.c` and `commsup.c` unlock using goto-statements, either resulting in EBA giving an internal error or not detecting the unlocks.

EBA raises a stack overflow error on the file `dvbdev.c` and the remainder of the files `usb_intf.c` and `blk-cgroup.c` fail to compile. A few reported positives in e.g. the file `commsup.c` seem like actual undiscovered double-unlock bugs, but given the complexity of the code under analysis I have been unable to verify whether these are in fact false positives or actual bugs.

We see that a total of 10 false positives are reported. Based on the variance in the amount of false positives in each file under analysis, it becomes apparent that the number of false positives is dependent on the file under analysis. We only see three false positives reported in files with no positives being detected, while the remaining files with no detections show no false positives. While the worst case of the file `drbd_main.c` shows 6 false positives, the confirmed bug in this file is also detected. This file is also the largest of all files at nearly 4000 lines, increasing the rate of false positives. The number of false positives is not prohibitively high, although in an ideal scenario these false positives would of course be eliminated completely.

The false positives are mainly reported in when an unlock is detected before the region the unlock happens on has been locked, as per the POSIX defintion of a double-unlock [12]. This is an unfortunate effect of the single-file analysis of EBA. Memory regions can have been unlocked elsewhere than the code under analysis if the file under analysis is part of a module consisting of several files.

This evaluation has allowed me to answer part of my research question *"How effective are such monitor template-based checker definitions?"*. The accuracy of the implementation of a double-unlock bug checker using monitor templates is higher than the previous approach. It is furthermore possible to express more complex bug checkers using monitor templates compared to the CTL template-based checker implementation, giving greater expressibility. In summary; though not all confirmed bugs are detected, a higher accuracy is observed when using monitor templates and monitor templates allow for greater expressibility when defining bug checkers.

# 6 Future Work

Implementing other bug checkers using monitor templates would allow EBA to detect these bug types. An implementation of a use-after-free bug checker could specifically prove very useful in detecting security risks in programs. This type of memory bug has been deemed as the cause for 70% of the critical security vulnerabilities found in the popular Chromium browser, which is the base for Google's Chrome browser [7]. This goes to show that this type of bug is prevalent in the wild and detecting such bugs in the Linux kernel would benefit the community surrounding the Linux kernel. In connection to this, the precision of bug checkers using monitor templates should also be measured by sampling files in the Linux kernel and evaluating the results of running bug checkers on this sample. This would allow determining the precision and false positive rate of the checkers.

The addition of more effect types to the EBA framework would allow defining a greater range of bug checkers using monitor templates, in turn enabling the detection of more bug types. Such effects could show effects of multi-threaded code, allowing the definition of bug checkers for concurrency errors.

The effect inference of the EBA framework could be improved upon in order to reduce the imprecision in the inferred effects. Currently, program points can have a wide range of effects after inference, leading to false positives. The inference of especially *may* effects could possibly be improved in order to limit the exploration of program points which have none of the inferred possible effects when executed in practice.

The GCC compiler supports compiler extensions and the Linux kernel project utilizes some of these, specifically *Static Assertions*, added in C11 which have been implemented since since GCC 4.6[2] and *Assembler Instructions with C Expression Operands*, an extension available since GCC 3.1[3]. The use of these produces compiler output which EBA does not support and the analysis will therefore fail. Support for the output of recent versions of GCC would fix this issue.

Symbolic execution [15] could be utilized in order to check the feasability of paths found in errors. Verifying that an execution path leading to a bug is actually reachable would increase the precision of bug checkers.

[2]See *Programming Languages - C* [13]
[3]See *Assembler Instructions with C Expression Operands* [10]

The number of false positives reported by the implementation of a double-unlock monitor template in this thesis should be improved upon. Johnson, Song, Murphy-Hill and Bowdidge [14] have concluded in their research that false positives is a key factor in developers choosing to not use static analysis tools. They note that future static analysis should strive to improve upon this problem in order to see increased use. The number of false positives in loops for the implementation presented in this thesis could be implemented through extensions to the output of EBA to expose information on loops in the effect-CFG or by implementing sophisticated filtering of bug checker output.

# 7 Conclusion

This thesis has described *shape-and-effect-inference* which allows the defintion of *monitor templates* operate on *effects* of program points in an *effect-CFG*. I have defined monitor templates as state machines and defined several bug checkers as such monitor templates and presented the product construction of an effect-CFG and a monitor template as reasoning for the correctness of monitor templates.

Furthermore, I have described how monitor templates are implementated as well as shown how to implement monitor templates as an extension to the EBA framework in OCaml. Lastly, this implementation has been evaluated by comparing the implementation to previous work which has shown greater expressibility in which checkers can be defined and has shown a higher detection rate of an implemented of a bug checkers defined as a monitor template compared to the existing CTL template implementation which is based on a subset of Computation Tree Logic, in turn answering my research question: *"How can bug checkers utilizing shape-and-effect inference be defined using finite automata with greater expressibility, how can such bug checkers be implemented as an extension to the EBA framework operating on the Linux kernel and how effective are such checker definitions?"*.

Monitor templates have been shown to provide greater expressibility and they allow defining a multitude of bug checkers with an implementation performing better in testing on Linux kernel source files. Finally, I have described possible future work which would improve upon this thesis by extensions to EBA and to the implementation of the monitor template implementation. The definition of monitor templates shows promise as a foundation for future bug checkers, extending the capabilities of the EBA framework.

# References


[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. Effective bug finding in c programs with shape and effect abstractions. In *VMCAI*, 2017.

[2] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[3] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, April 2006.

[4] Thomas Ball and Sriram K. Rajamani. The slam project: Debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, January 2002.

[5] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[6] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 289–300, New York, NY, USA, 2009. Association for Computing Machinery.

[7] The Chromium Project. Memory safety. `https://www.chromium.org/Home/chromium-security/memory-safety`, May 2020. Accessed: 2020-05-24.

[8] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*. USENIX Association, 2000.

[9] Anders Fischer-Nielsen. Finding double-unlock bugs with shape-and-effect analysis. Project Report available at: `https://github.com/andersfischernielsen/Finding-Double-Unlock-Bugs-with-Shape-and-Effect-Analysis/raw/master/report.pdf`, 2019.

[10] Inc. Free Software Foundation. Assembler instructions with c expression operands. `https://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Extended-Asm.html`, 2002. Accessed: 2019-11-25.

[11] Inc. Free Software Foundation. Gcc 10 release series. `https://gcc.gnu.org/gcc-10/changes.html`, May 2020. Accessed: 2020-05-11.

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

[13] ISO. *ISO/IEC 9899:1999(E) - Programming Languages - C*. Geneva, Switzerland, 1999.

[14] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 672–681. IEEE Press, 2013.

[15] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[16] David Malcolm. Static analysis in gcc 10. https://developers.redhat.com/blog/2020/03/26/static-analysis-in-gcc-10/, May 2020. Accessed: 2020-05-13.

[17] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228. Springer, 2002.

[18] Daniel Stenberg. Daniel stenberg. https://twitter.com/bagder/status/1252900440305537025, May 2020. Accessed: 2020-05-13.