Effective Bug Finding in C Programs with Shape and Effect Abstractions

Conference Paper · January 2017

DOI: 10.1007/978-3-319-52234-0_3

Claus Brabrand
IT University of Copenhagen
60 PUBLICATIONS **972** CITATIONS
SEE PROFILE

Andrzej Wasowski
IT University of Copenhagen
138 PUBLICATIONS **3,339** CITATIONS
SEE PROFILE

Effective Bug Finding in C Programs with Shape and Effect Abstractions

Iago Abal, Claus Brabrand, and Andrzej Wąsowski

IT University of Copenhagen, Denmark
`{iago,brabrand,wasowski}@itu.dk`

Abstract. Software tends to suffer from simple resource mis-manipulation bugs, such as double-locks. Code scanners are used extensively to remove these bugs from projects like the Linux kernel. Yet, these tools are not effective when the manipulation of resources spans multiple functions. We present a *shape-and-effect* analysis for C, that enables efficient and scalable inter-procedural reasoning about resource manipulation. This analysis builds a program abstraction based on the observable side-effects of functions. Bugs are found by model checking this abstraction, matching undesirable sequences of operations. We implement this approach in the EBA tool, and evaluate it on a collection of historical double-lock bugs from the Linux kernel. Our results show that our tool is more effective at finding bugs than similar code-scanning tools. EBA analyzes nine thousand Linux files in less than half an hour, and uncovers double-lock bugs in various drivers.

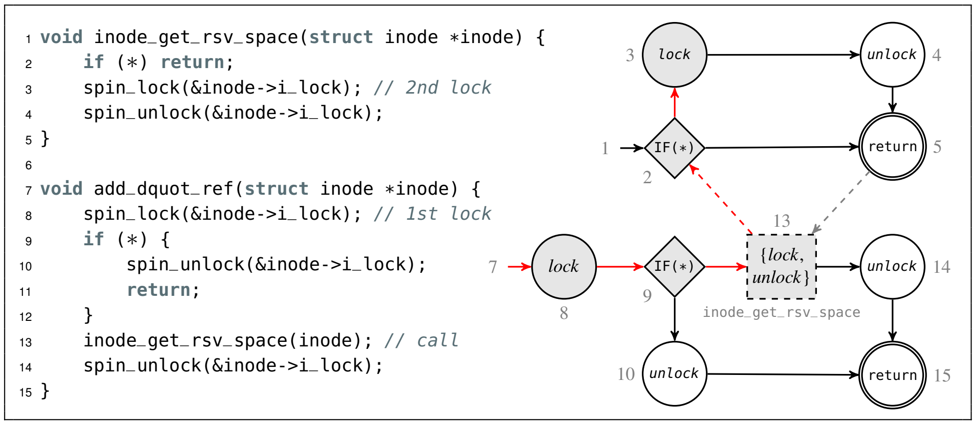
Keywords: bug finding, type and effects, model checking, C, Linux, double lock

1 Introduction

Today, the source code of the Linux kernel is continuously analyzed for bugs [12] using a handful of static code scanning tools (so-called *linters*). Code scanners find bugs by pattern matching against the structure and flow of the program. For instance, Linux commits `ca9fe15`[1] and `65582a7` fix locking bugs found by two of these tools. Linux-tailored linters like SMATCH[2] have seen adoption because they are easy to use, run fast, and are reasonably effective at finding certain classes of bugs. However, code scanners are commonly restricted to intra-procedural analysis of isolated functions; hence, they mostly find shallow bugs, and do not deal well with nested function calls.

Software bugs often cross the boundaries of a single function. The VBDb bug collection [2] documents 30 runtime bugs in Linux, 80% of which involve deeply nested function calls. Many of these bugs are conceptually simple, but will be missed by conventional linters. Examples include bugs fixed in commits: `1c17e4d` (read of uninitialized data), `6252547` (null pointer dereference), `218ad12` (memory leak), and `d7e9711` (double lock). Traditionally, static analysis of inter-procedural data-flow, or symbolic execution, could be used to find such bugs, but these analyses tend to be expensive, and have seen little adoption in practice. We argue that the above bugs can be handled by code scanners enriched with a minimal amount of semantic information.

[1] See `https://github.com/torvalds/linux/commit/`*hash* with *hash* replaced by the identifier.
[2] `http://smatch.sf.net`



```
1  void inode_get_rsv_space(struct inode *inode) {
2      if (*) return;
3      spin_lock(&inode->i_lock); // 2nd lock
4      spin_unlock(&inode->i_lock);
5  }
6
7  void add_dquot_ref(struct inode *inode) {
8      spin_lock(&inode->i_lock); // 1st lock
9      if (*) {
10         spin_unlock(&inode->i_lock);
11         return;
12     }
13     inode_get_rsv_space(inode); // call
14     spin_unlock(&inode->i_lock);
15 }
```

Fig. 1. An illustration of our bug-finding technique on a double-lock bug in Linux fixed by commit `d7e9711`. The *simplified* code is shown to the left. To the right, the associated CFG annotated with *lock* and *unlock* effects. The numbers next to the CFG nodes show corresponding line numbers. The red edges visualize the path (via the function call in line 13) to the double-lock (in line 3).

We propose a bug-finding technique consisting in model-checking a lightweight program abstraction based on the notion of *side-effect* [34, 37]. This abstraction is automatically inferred by a flow-insensitive shape-and-effect analysis, built on the work of Talpin and Jouvelot on polymorphic type-and-effect inference [47]. This analysis infers types that approximate the *shape* of data in memory—hence the term *shape*-and-effect analysis, and also computational *effects* that describe how data is manipulated by the program. The inferred effects reveal at which points the program performs operations like reading or writing variables, opening or closing files, acquiring or releasing locks, etc. The domain of effects is extensible. The inference algorithm is a small variation of the classic Damas-Milner's *Algorithm* $\mathcal{W}$ [18]. Since our goal is bug finding and not program optimization nor verification, we trade soundness for scalability [33].

The inferred shape-and-effect information is superimposed on the control-flow graph (CFG), obtaining what we call the *shape-and-effect abstraction* of the program. In this abstraction, each program expression and statement is described by a set of computational effects. The abstraction is built in a modular fashion, and each program function is given a polymorphic shape-and-effect signature, that summarizes its computational behavior. Bugs are found by matching temporal bug patterns on this abstraction, using a standard model checking algorithm. The search is inter-procedural *on demand*, and function calls can be inlined if they are of interest. Although, in practice, the majority of function calls are deemed irrelevant simply by examining their effect signature, and hence treated as opaque expressions. This prevents the path explosion associated with inter-procedural bug finding. This technique scales and finds deep resource manipulation bugs in large and complex software.

Figure 1 illustrates our approach using a simplified version of an actual Linux bug. Function `add_dquot_ref` enters a *deadlock* by recursively acquiring a non-reentrant lock. The first lock acquisition occurs in line 8, and the second occurs in line 3, after calling function `inode_get_rsv_space` in line 13; both conditionals (lines 2 and 9) must evaluate to *false* (i.e., take the *else* branch). To the right, we show a simplification of the effect-decorated CFG, annotated with locking effects on `inode->i_lock`. The red edges mark the execution path leading to the double lock. The call to `inode_get_rsv_space` is abstracted by a flow-insensitive summary of effects (the set: $\{lock, unlock\}$). These summaries are extremely cheap to compute, but can be insufficient at times. In line 13, from the effect signature of `inode_get_rsv_space` alone, it is unclear as to whether the acquisition of the lock happens *before* or *after* its release. Our bug finder needs to inline the call to `inode_get_rsv_space`, to find a path to the second lock in line 3, and finally confirm the double-lock bug. Note that if this function did not manipulate the lock at all, its effect signature would be the empty set, and the bug finder would have ignored it.

Our contributions are:

- An adaptation of Talpin-Jouvelot's [47] polymorphic type-and-effect inference system to the C language, that can be used to infer abstract discrete effects of computations, and shapes of structured values on the heap (Section 3). We use *shape inference and polymorphism* in order to add a degree of context sensitivity to our analysis, and to handle some common patterns to manipulate generic data in C.
- An inter-procedural bug-finding technique that combines shape-and-effect inference, to build lightweight *program abstractions*, with model-checking, to match *bug patterns* on those abstractions (Section 4). This technique finds several classes of bugs, even when these span multiple functions. We use function inlining as a sort of abstraction refinement, to disambiguate the ordering of operations when it is needed.
- An open-source proof-of-concept implementation of the shape-and-effect system, and the proposed bug-finding technique: EBA (Effect-Based Analyzer).[3] EBA can analyze individual Linux files for bugs in seconds and the entire x86 *allyesconfig* Linux kernel in less than an hour, and has uncovered about a dozen of previously unknown double-lock bugs in Linux 4.7–4.9 releases.
- An evaluation and comparison of our proposed analysis technique with two popular bug-finding tools within the Linux kernel community (Section 5): (1) on a collection of historical double-lock bugs in the Linux kernel, and (2) on the set of device drivers included in the 64-bit x86 *allyes* configuration of Linux 4.7.

We proceed by discussing related work (Section 2) to contextualize our main contribution, the shape-and-effect system (Section 3). We then outline our bug-finding technique (Section 4), and evaluate it (Section 5). Finally, we present our conclusions (Section 6).

2 Related Work

Types and effects. Side-effect analysis is used to compute which memory locations are accessed or updated by a function call [6, 15, 16]. Traditionally, this information is used by compilers to determine whether it is legal to perform certain code optimizations. Lucassen proposed a type-and-effect system [34] that, unlike previous analyses, correctly handles function pointers. Talpin and Jouvelot developed a complete type reconstruction

[3] `http://www.iagoabal.eu/eba/`

algorithm for such a polymorphic type-and-effect system [47]. We extend their work to the C programming language. In order to accommodate the use of type casts in C, our system infers the shape of objects in memory, rather than standard C types. KOKA [32] is a functional programming language featuring an effect system based on row types [41]. This effect system is designed to be exposed to, and understood by the programmer. Our effect system is an internal program abstraction, thus we settled on Talpin-Jouvelot's system as a basis instead. Nielson and Nielson [37] survey the development of type-and-effect systems and their applications [35, 49, 50].

Pointer analysis. Pointer analysis is used to approximate the values of pointer expressions at compile-time [43]. Side-effect analysis tracks operations on memory locations, hence it naturally embeds pointer analysis. Our shape-and-effect system implements context-sensitive alias analysis. Shapes are annotated with *regions* that abstract the locations where objects are stored in memory. Aliasing relations are recorded by unification during shape inference. This is similar to Steensgaard's points-to analysis [45], but it is significantly more precise thanks to shape and region polymorphism [20, 22, 26]. We prioritized a precise analysis of C structure types, as this is a well-known requirement to analyze real-world programs [44, 52].

Type-safe resource management. These techniques impose stricter typing disciplines that guarantee safe manipulation of resources. They are valuable for languages with restricted aliasing, but often do not work well for C. Some of these disciplines are too strict to accommodate complex resource manipulation patterns, like those used in Linux. EBA is different in that it employs type inference to build abstractions, rather than to check specifications. We discuss the simplest flow-insensitive approaches first. Kyselyoiv and Shan rely on *monadic regions* to ensure the safe use of file handles [30]. Foster et al. implemented CQUAL, a constraint-based type checker that extends the C type system with user-defined type qualifiers [23]. CQUAL could be used to enforce the correct manipulation of user- and kernel-space pointers in Linux [27].

Strom and Yemini introduce the concept of *typestate* [46], a flow-sensitive abstraction of the state of an object, where operations specify *typestate transitions*—e.g. `fclose` will turn an open file into a closed file. In [24] Foster et al. extend [23] and introduce *flow-sensitive* type qualifiers. This is essentially the same concept of typestate, but they support subtyping and consider the problem of aliasing in C. This newer version of CQUAL can be used to find double-locks and similar bugs. However, using CQUAL requires code annotations and rewrites, and otherwise reports too many false positives. Using a similar analysis to that of [24], LOCKSMITH infers the set of locks that protect shared memory locations and detects data races [40]. LOCKSMITH uses an effect system to infer which locations are thread-shared.

Code scanning. Static code scanners are mostly syntax-based bug finding tools. They are lightweight and know little about semantics: do no compute function summaries, ignore aliasing, and work intra-procedurally. Hence they are fast and scale well. For the same reasons, they are able to find relatively simple and shallow bugs. The degree of sophistication of these tools varies significantly. Some do not even fully parse the source code [10], whereas others check finite-state properties against the control-flow of programs [21]. One of the first tools of this class was LINT [19], a static checker for C created by Stephen Johnson at Bell Labs in 1970s. We discuss three tools that are used to analyze the Linux kernel source code for bugs [12], and have led to thousands of bug-fixing commits; these are SPARSE, COCCINELLE and SMATCH.

SPARSE exploits Linux-specific annotations to perform simple checks. However, some checks, like those related to locking, require too many annotations and are unpopular among developers. COCCINELLE is a program transformation tool [11], with an associated language (SmPL) to specify flow-based transformations. While originally conceived for managing collateral evolution of code [38], COCCINELLE can also encode bug-finding rules [31, 39]. SMATCH realizes the idea of *meta-level compilation* proposed by Engler et al. [21], where bug checkers are scripts run by an intra-procedural data-flow analysis engine.

These tools cannot directly find interprocedural bugs such as that of Fig. 1. For SPARSE to find this bug, the functions involved need to be properly annotated with their locking behavior—they are not. COCCINELLE and SMATCH would have to rely on ad-hoc scripts to traverse the source code and collect all functions that may perform locking. Compared to an effect system, these scripts are more difficult to extend, do not track aliasing, nor handle function pointers appropriately. SMATCH ships with such an script which, in addition, only explores one level of function calls on each run. EBA works similarly to code scanners, but effectively supports inter-procedural bug finding.

Static analysis & software model checking. Static analyzers and software model checkers have a deep understanding of program semantics, and they quite precisely track the values of expressions, and the shape of the objects in the heap. These tools can find very complex and deep bugs, even when these involve non-trivial data dependencies. In order to scale, they rely heavily on *abstraction* [3, 8, 14, 17] to model the program state. Maintaining such precise descriptions of the program state incurs in longer execution times and higher memory utilization. EBA offers a compromise solution between code scanners and these heavyweight tools, for bugs involving simple data dependencies.

ASTRÉE is a tool for analyzing safety-critical embedded C software based on abstract interpretation [17]. ASTRÉE can prove the absence of runtime errors, such as null-pointer dereferences, but only for a restricted subset of C. Reps, Horwitz and Sagiv show how an important class of inter-procedural data-flow analyses can be performed precisely and efficiently [42]. Their algorithm is popularly known as RHS. In [25] Hallem et al. extend *xgcc* [21] to perform inter-procedural analysis based on the RHS algorithm. The software model checkers BLAST [8] and SLAM [4, 5] employ path-sensitive extensions of RHS. CBMC is a bounded model checker that reduces the verification of C programs into Boolean satisfiability problems [13]. These are whole-program analyses and do not scale to the extent of being adequate for regular use by developers.

INFER is a static analyzer based on symbolic execution and separation logic [7, 9]. It is used by Facebook and others to find specific kinds of memory and resource manipulation errors in mobile apps. Similarly, SATURN is a SAT-based symbolic execution framework for checking temporal safety properties [51]. Both INFER and SATURN compute elaborated path-sensitive summaries for functions, and can model the runtime shape of complex data structures precisely. We do not model the heap as precisely as these two tools, and our function summaries are flow-insensitive; but we tackle the loss in precision by relying on heuristics and inlining, respectively. As a result, EBA is significantly simpler, and scales better.

3 The Shape-and-Effect System

At the core of EBA there is a new type-and-effect inference system for C in the style of Talpin and Jouvelot [29, 47]. Because of unsafe casts, the standard C type system provides only a meager description of run-time objects. Thus, as known in pointer analysis [44, 45], we describe objects by their memory shape. Our system is *polymorphic* in *shapes*, *regions*, and *effects*; and it supports *sub-effecting*. We use shape and region polymorphism to add context-sensitivity to our analysis, and to handle the most common pattern of use for unsafe casts: generic data structures in C. Effect polymorphism and sub-effecting allow handling function pointers.

EBA analyzes programs in CIL (C Intermediate Language), an analysis-friendly intermediate representation of C [36]. CIL has a simpler syntax-directed type system than C, without implicit type conversions. Using CIL allows us to scaffold a tool prototype faster, while still being able to handle the entire C (via a C-to-CIL front-end). For space reasons, we present the shape-and-effect system declaratively and for a much smaller language than CIL. The declarative and algorithmic formulations for CIL [1], including support for structures, are available online.[4]

3.1 The Source Language

We assume that the analyzed programs are well-typed with respect to a C-like base type system. This is easily ensured using a compiler. We consider only the following types in the base type-system:

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

We distinguish l-value (T^L) and r-value (T^R) types, corresponding to the *left* and *right* sides of assignments, respectively. Unlike in C, reference types are explicit. A reference object, of type `ref` T, is a memory *cell* holding objects of type T. We distinguish between mutable references to r-values (data), and immutable references to function values (code). A pointer value, of type `ptr ref` T, is the *address* of a reference in memory. Like in C, functions are not first-class citizens, but function pointers are allowed. Expressions are also split into l-values (L) and r-values (E):

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

[4] http://dl.iagoabal.eu/eba/cil.pdf

L-value expressions (L) designate memory locations and are always assigned reference types T^L. We distinguish between mutable variables x and immutable function variables f. Dereferencing a pointer, `*E`, looks up the corresponding reference cell in memory; e.g., `*&`x evaluates to x.

R-value expressions (E), or simply *expressions*, denote *data* values, i.e. integers and pointers. Basic integer expressions are constants (n) and additions. The language includes `if` conditional expressions. As in C, a type cast `(`T`)`E converts the value of an expression E to type T. The expression `new` x `:` T `=` E_1`;` E_2 introduces a new local variable x, initialized to E_1 and visible in E_2; x names a memory cell of type `ref` T. (We assume that memory is automatically managed.) The bang operator, `!`L, reads an l-value. Pointer values are obtained from reference cells using the *address-of* operator `&`. L-values can be assigned a new value before evaluating another expression, as in x `:=` E_1`;` E_2. The expression `fun` T f`(`T_1 $x_1, \cdots, T_n$ x_n`)` `=` E_1`;` E_2 introduces a function variable f visible in E_2; function f binds n arguments $(x_1, \ldots, x_n)$ and evaluates E_1. Function variables name immutable reference cells holding function values. Functions may either be invoked, or passed as pointers (using `&`). We assume that fetching of and assignment to function references is forbidden by the base type system.

For clarity, we omit loops and jumps, which do not present specific challenges for our flow-insensitive inference system. Yet, they are considered in our implementation.

3.2 The Shape and Effect Language

Effects. Types describe *what* expressions compute, whereas effects describe *how* expressions compute [28]. From the type perspective, the expression y `:=` 1 `+` `!`x`;` `!`y evaluates to an integer value. From the effect perspective, it reads from locations x and y, and writes to y. Effects are a framework to reason about such and similar aspects of computations. An example set of effects is $\varphi = \{\mathit{read}_x, \mathit{read}_y, \mathit{write}_y\}$, which records reading variables x and y, and writing y. A set of effects is a *flow-insensitive abstraction* of an execution. It specifies the effects that *may* result from evaluating an expression (or statement), disregarding the flow of control.

We assume a finite number of *effect constructors* of finite arity, including nullary. A constructor ε applied to a tuple[5] $\overline{\rho}$ of memory regions (see below) defines a discrete effect $\varepsilon_{\overline{\rho}}$. Built-in effects, inherent to the C language, include reading and writing of memory locations, recorded as read_ρ and write_ρ; and calling to functions (e.g. through function pointers), recorded as call_ρ. Other effects can be introduced to capture new kinds of bugs. The example of Sect. 1 used effects lock_ρ and unlock_ρ to represent lock manipulation actions. Effects are combined into sets (φ), ordered by the usual set inclusion. We use effect variables (ξ) to stand for sets of effects, to achieve effect polymorphism.

Regions. In practice, it is not enough to track effects involving single variables. A variable is one of many possible names for a particular memory cell. Consider the C program `int x; int *y = &x;` S. Within S's scope, both x and $*y$ denote the same memory cell—they *alias*. To address aliasing, we track abstract sets of possibly-aliased memory references, *memory regions* (ρ).

[5] We use overline to denote tuples.

The shape-and-effect system performs integrated flow-insensitive alias analysis, similar to Steengaards's points-to analysis [45] but polymorphic. The analysis assigns a memory region ρ to each reference. If during pointer manipulation two regions become indistinguishable for the analysis, they are unified into a single one. For instance, if reference x belongs to region ρ_1 and y belongs to ρ_2, the effect of evaluating y `:=` 1 `+` `!`x`;` `!`y is $\{read_{\rho_1}, read_{\rho_2}, write_{\rho_2}\}$. If the analysis determines that ρ_1 and ρ_2 may alias, they will be merged —as $\rho_{\{1,2\}}$, reducing the effect set to $\{read_{\rho_{\{1,2\}}}, write_{\rho_{\{1,2\}}}\}$.

Shapes. A *shape* approximates the memory representation of an object [44, 45]. Whereas, in the base type system, an expression can be coerced to a different type, in this system, the shape of an expression is fixed and preserved across type casts (cf. Sect. 3.3). Shapes are annotated with regions, recording points-to relations between references. We use the following terms to represent shapes:

$$\begin{array}{llcl} \textit{l-value shapes} & Z^L & : & \mathsf{ref}_\rho\ Z^R \ \mid\ \mathsf{ref}_\rho\ (Z_1^L \times \cdots \times Z_n^L \xrightarrow{\varphi} Z_0^R) \\ \textit{r-value shapes} & Z^R & : & \bot \quad\mid\quad \mathsf{ptr}\ Z^L \quad\mid\quad \zeta \end{array}$$

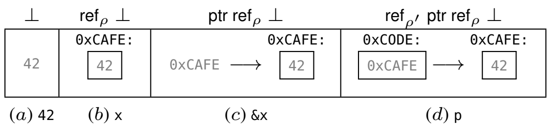
As for types, we split shapes into l-value (Z^L) and r-value (Z^R) shapes. The shape language resembles the type language, without integer type but with shape variables (ζ).

R-value shapes denote the shape of r-value objects. An *atomic* shape $\bot$ denotes objects that have no relevant structure, for instance integers, when these are not masquerading pointers to implement genericity (see below). Pointer expressions have pointer shapes, $\mathsf{ptr}\ Z^L$, where Z^L is the shape of the target reference cell of the pointer. A pointer represents the *address* of a reference cell, and therefore a pointer shape necessarily encloses a reference shape. Pointers may be cast to integers to emulate generics; such integer values will thus have a pointer shape. *Shape variables* ζ are used to make shapes polymorphic, they stand for arbitrary r-value shapes. For instance, functions manipulating a generic linked list are shape polymorphic, since they abstract from the shape of objects stored in the list.

L-value shapes denote *references* to either data or functions. Data (r-value) references have shape $\mathsf{ref}_\rho\ Z^R$, where ρ is a memory region, and Z^R is the shape of the objects that it holds. If a reference ρ_1 holds a pointer to another reference ρ_2, as in $\mathsf{ref}_{\rho_1}\ \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ Z$, we say that ρ_1 *points to* ρ_2. Function references have shape $\mathsf{ref}_\rho\ (Z_1^L \times \cdots \times Z_n^L \xrightarrow{\varphi} Z_0^R)$. A function shape maps a tuple of reference shapes ($Z_1^L \times \cdots \times Z_n^L$), corresponding to the formal parameters, to a value shape (Z_0^R), corresponding to the result. The shape-and-effect system describes function parameters as l-value shapes, since actual parameters are in fact stored in stack variables. The returned value is an r-value expression, hence Z^R. Function shapes carry a so-called *latent effect*, φ, which accounts for the actions that (depending on the flow of control) *may* be performed during execution of the function.

Figure 2 shows the shapes inferred for a small C program. We assign constant `42` the shape $\bot$ (Fig. 2(a)). Variable `x` holding the value `42` at location ρ, gets the shape $\mathsf{ref}_\rho\ \bot$ (Fig. 2(b)). Region ρ is an abstraction of the

Fig. 2. Shapes of expressions in the C program: `int x = 42; int *p = &x; return p;`


$\bot$
$\mathsf{ref}_\rho\ \bot$
$\mathsf{ptr}\ \mathsf{ref}_\rho\ \bot$
$\mathsf{ref}_{\rho'}\ \mathsf{ptr}\ \mathsf{ref}_\rho\ \bot$
42
0xCAFE:
42
0xCAFE
0xCAFE:
42
0xCODE:
0xCAFE
0xCAFE:
42
(a) 42
(b) x
(c) &x
(d) p

actual memory address, `0xCAFE`. Figure 2(c) shows the shape of `&x`, which is $\mathsf{ptr}\ \mathsf{ref}_\rho\ \bot$. Finally, Fig. 2($d$) shows the shape of the pointer variable `p`, $\mathsf{ref}_{\rho'}\ \mathsf{ptr}\ \mathsf{ref}_\rho\ \bot$.

Shape-type compatibility. As mentioned in the shape description above, there is often correlation between types and shapes. The compatibility of a shape Z with a type T, written $Z \leq T$, is defined as follows:

$$[\textsc{Int}]\ \frac{}{Z^R \leq \texttt{int}} \qquad [\textsc{Ptr}]\ \frac{Z \leq T}{\mathsf{ptr}\ Z \leq \texttt{ptr}\ T} \qquad [\textsc{Ref}]\ \frac{Z \leq T}{\mathsf{ref}_\rho\ Z \leq \texttt{ref}\ T}$$

$$[\textsc{Fun}]\ \frac{Z_i \leq T_i \text{ for } i \in [0,n]}{\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{\xi} Z_0 \quad \leq \quad T_1 \times \cdots \times T_n \to T_0}$$

Intuitively, shape-type compatibility requires that the given shape and type are structurally equivalent (rules [Ptr] and [Ref]), with two exceptions. First, any r-value shape is compatible with the integer type. The relations $\bot \leq \texttt{int}$, $\mathsf{ptr}\ Z \leq \texttt{int}$, and $\zeta \leq \texttt{int}$ are subsumed by rule [Int]. In other words, integer values can be used to encode arbitrary r-value objects at runtime, when they are used as pointers. Second, function shapes capture the storage location of function parameters, thus they are reference shapes, which is ignored by function types (rule [Fun]).

Environments and shape schemes. An environment Γ maps variables x to their reference shapes: $\Gamma(x) = \mathsf{ref}_\rho\ Z$; and function variables f to *function shape schemes*:

$$\Gamma(f) = \forall\, \overline{\upsilon}.\ \mathsf{ref}_{\rho_0}\ (Z_1^L \times \cdots \times Z_n^L \xrightarrow{\varphi} Z_0^R) \qquad \text{where } \rho_0 \notin \overline{\upsilon}$$

A function shape scheme is a function shape quantified over shape, region, and effect variables υ for which the function poses no constraints. We say that the function is *polymorphic* on such variables, which should occur free in the function shape (i.e. they are mentioned in $Z_1^L \times \cdots \times Z_n^L \xrightarrow{\varphi} Z_0^R$). As such, these variables are parameters that can be appropriately instantiated at each call site. If φ is of the form $\varphi' \cup \xi_0$ where $\xi_0 \in \overline{\xi}$, we say that f is effect-polymorphic: the effect of f is extended by the instantiation of ξ_0. In general, it is unsound to generalize reference types [48], but we can safely generalize function references because they are immutable. The memory region ρ_0 identifies the function; it is used to track calls to it through function pointers, and it cannot be generalized (thus $\rho_0 \notin \overline{\upsilon}$).

3.3 Shape-and-Effect Inference

L-values. Judgment $\Gamma \vdash_L L : \mathsf{ref}_\rho\ Z\ \&\ \varphi$ (Fig. 3a) specifies that, under environment Γ, the l-value expression L has shape $\mathsf{ref}_\rho\ Z$, and evaluating it results in effects φ. The shape of a variable x is obtained directly from the environment (rule [Var]). Pointer dereferencing proceeds by evaluating an expression E, obtaining a shape, from which we drop the pointer constructor obtaining a reference shape (rule [Deref]). Dereferencing has no effects by itself, but transfers the effects φ from evaluating E. The shape of a function variable f is obtained by appropriately instantiating its shape scheme (rule

(a) Inference rules for l-value expressions, $\vdash_L \subseteq \textsc{Env} \times \textsc{L-Value} \times \textsc{Shape} \times \textsc{Effect}$.

$$[\textsc{Var}]\ \frac{\mathit{\Gamma}(x) = \mathsf{ref}_\rho\ Z}{\mathit{\Gamma} \vdash_L x : \mathsf{ref}_\rho\ Z \,\&\, \emptyset} \qquad [\textsc{Deref}]\ \frac{\mathit{\Gamma} \vdash_E E : \mathsf{ptr\ ref}_\rho\ Z \,\&\, \varphi}{\mathit{\Gamma} \vdash_L *E : \mathsf{ref}_\rho\ Z \,\&\, \varphi}$$

$$[\textsc{Fun}]\ \frac{\mathit{\Gamma}(f) = \forall\, \overline{\zeta}\, \overline{\rho}\, \overline{\xi}.\ \mathsf{ref}_{\rho_0}\ Z \qquad Z = Z_1^L \times \cdots \times Z_n^L \xrightarrow{\varphi} Z_0^R}{\mathit{\Gamma} \vdash_L f : \mathsf{ref}_{\rho_0}\ (Z[\overline{\zeta \mapsto Z'}][\overline{\rho \mapsto \rho'}][\overline{\xi \mapsto \varphi'}]) \,\&\, \emptyset}$$

(b) Inference rules for r-value expressions, $\vdash_E \subseteq \textsc{Env} \times \textsc{Exp} \times \textsc{Shape} \times \textsc{Effect}$.

$$[\textsc{Int}]\ \frac{}{\mathit{\Gamma} \vdash_E n : Z \,\&\, \emptyset} \qquad [\textsc{Add}]\ \frac{\mathit{\Gamma} \vdash_E E_1 : Z \,\&\, \varphi_1 \qquad \mathit{\Gamma} \vdash_E E_2 : Z \,\&\, \varphi_2}{\mathit{\Gamma} \vdash_E E_1 \texttt{ + } E_2 : Z \,\&\, \varphi_1 \cup \varphi_2}$$

$$[\textsc{If}]\ \frac{\mathit{\Gamma} \vdash_E E_0 : Z_0 \,\&\, \varphi_0 \qquad \mathit{\Gamma} \vdash_E E_1 : Z \,\&\, \varphi_1 \qquad \mathit{\Gamma} \vdash_E E_2 : Z \,\&\, \varphi_2}{\mathit{\Gamma} \vdash_E \texttt{if (}E_0\texttt{)}\ E_1\ \texttt{else}\ E_2 : Z \,\&\, \varphi_0 \cup \varphi_1 \cup \varphi_2}$$

$$[\textsc{New}]\ \frac{\mathit{\Gamma} \vdash_E E_1 : Z_1 \,\&\, \varphi_1 \qquad Z_1 \leq T \qquad \mathit{\Gamma}, x : \mathsf{ref}_\rho\ Z_1 \vdash_E E_2 : Z_2 \,\&\, \varphi_2}{\mathit{\Gamma} \vdash_E \texttt{new}\ x\ \texttt{:}\ T\ \texttt{=}\ E_1\texttt{;}\ E_2 : Z_2 \,\&\, \varphi_1 \cup \{\mathit{write}_\rho\} \cup \varphi_2}$$

$$[\textsc{Fetch}]\ \frac{\mathit{\Gamma} \vdash_L L : \mathsf{ref}_\rho\ Z \,\&\, \varphi}{\mathit{\Gamma} \vdash_E \texttt{!}L : Z \,\&\, \varphi \cup \{\mathit{read}_\rho\}} \qquad [\textsc{Addr}]\ \frac{\mathit{\Gamma} \vdash_L L : \mathsf{ref}_\rho\ Z \,\&\, \varphi}{\mathit{\Gamma} \vdash_E \texttt{\&}L : \mathsf{ptr\ ref}_\rho\ Z \,\&\, \varphi}$$

$$[\textsc{Assign}]\ \frac{\mathit{\Gamma} \vdash_L L : \mathsf{ref}_\rho\ Z \,\&\, \varphi_1 \qquad \mathit{\Gamma} \vdash_E E_1 : Z \,\&\, \varphi_2 \qquad \mathit{\Gamma} \vdash_E E_2 : Z' \,\&\, \varphi_3}{\mathit{\Gamma} \vdash_E L \texttt{ := } E_1\texttt{;}\ E_2 : Z' \,\&\, \varphi_1 \cup \varphi_2 \cup \{\mathit{write}_\rho\} \cup \varphi_3}$$

$$[\textsc{Def}]\ \frac{\begin{array}{c}\mathit{\Gamma}; x_1 : \mathsf{ref}_{\rho_1}\ Z_1; \cdots; x_n : \mathsf{ref}_{\rho_n}\ Z_n \vdash_E E_0 : Z_0 \,\&\, \varphi_0 \\ Z_f = \mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{\varphi_0} Z_0 \qquad \overline{v} = \mathrm{FreeVars}(Z_f) \setminus \mathrm{FreeVars}(\mathit{\Gamma}) \\ \mathit{\Gamma}' = \mathit{\Gamma}; f : \forall\, \overline{v}.\ \mathsf{ref}_{\rho_0}\ Z_f \qquad \mathit{\Gamma}' \vdash_E E' : Z' \,\&\, \varphi' \qquad Z_i \leq Ti / i \in [0, n]\end{array}}{\mathit{\Gamma} \vdash_E \texttt{fun}\ T_0\ f\texttt{(}T_1\ x_1, \cdots, T_n\ x_n\texttt{)}\ \texttt{=}\ E_0\texttt{;}\ E' : Z' \,\&\, \varphi'}$$

$$[\textsc{Call}]\ \frac{\begin{array}{c}\mathit{\Gamma} \vdash_L L_0 : \mathsf{ref}_{\rho_0}\ (\mathsf{ref}_{\rho_1}\ Z_1 \times \cdots \times \mathsf{ref}_{\rho_n}\ Z_n \xrightarrow{\varphi'} Z_0) \,\&\, \varphi_0 \\ \mathit{\Gamma} \vdash_E E_i : Z_i \,\&\, \varphi_i / i \in [1, n]\end{array}}{\mathit{\Gamma} \vdash_E L_0(E_1, \cdots, E_n) : Z_0 \,\&\, \varphi_0 \cup (\bigcup_{i \in [1,n]} \varphi_i) \cup \{\mathit{call}_{\rho_0}\} \cup \varphi'}$$

$$[\textsc{Sub}]\ \frac{\mathit{\Gamma} \vdash_E E : Z \,\&\, \varphi' \qquad \varphi' \sqsubseteq \varphi}{\mathit{\Gamma} \vdash_E E : Z \,\&\, \varphi} \qquad [\textsc{Cast}]\ \frac{\mathit{\Gamma} \vdash_E E : Z \,\&\, \varphi \qquad Z \leq T}{\mathit{\Gamma} \vdash_E \texttt{(}T\texttt{)}E : Z \,\&\, \varphi}$$

Fig. 3. Shape-and-effect inference system.

[Fun]). This instance is generated by substituting quantified variables with concrete shapes, regions and effects. In a typing derivation, these will depend on the calling context: the actual parameters passed to the function, and the expected shape of the function's return value in that context.

R-values. The judgment $\mathit{\Gamma} \vdash_E E : Z \,\&\, \varphi$ (Fig. 3b) specifies that, under environment $\mathit{\Gamma}$, r-value expression E has shape Z, and side effects φ.

Scalars. A constant n is given an arbitrary shape Z (rule [Int]). Each use of a constant can receive a different shape depending on the surrounding context. The shape of an integer is unknown *a priori* and depends on how this integer value is used by the program. For instance, in a expression like `ptr + 1`, where `ptr` is a pointer variable, constant `1` would be given the same shape as `ptr`. The effect of scalar addition is the combined effect of evaluating its operands (rule [ADD]). Both operands must have the same shape. Arithmetic between pointers with incompatible shapes is disallowed.

Conditionals. Both alternatives of an *if* conditional contribute to the effect of the entire expression (rule [IF]). In a particular execution, either E_1 or E_2 will be evaluated, but not both. The union of all three effects is a flow-insensitive *over*-approximation. (Loops, which we have omitted in this presentation, are treated analogously.) Both branches shall have the same shape, which is also the shape of the overall expression.

References. The `new` operator allocates a reference cell in a region ρ (rule [NEW]). The shape of the initializer expression E_1 must be compatible with the type T of x. Fetching the value stored in a reference returns an object of the expected shape, and produces a read effect on the corresponding memory region (rule [FETCH]). Given a reference cell, the computation of the memory address of such cell is side-effect free (rule [ADDR]). Assignment writes the result of evaluating an expression E_1 into a memory location denoted by L (rule [ASSIGN]). This requires evaluating both expressions, which introduces effects φ_1 and φ_2. Left- and right-hand side must be of the same shape. Hence, given a pointer assignment like `ptr = &x;`, this system considers that `*ptr` and `x` alias. In addition, we record the effect of writing to memory region ρ.

Functions. When introducing a function definition (rule [DEF]) we analyze the body E_0 under a new environment, where each parameter x_i is given a shape $\mathsf{ref}_{\rho_i}\ Z_i$. This shape Z_i should be chosen according to the use of x_i in E_0, and must be compatible with its type, T_i. The shape (Z_0) and effects (φ_0) of evaluating E_0 constitute the result shape and latent effects of f, respectively. The shape of function f is generalized over $\overline{v}$ and added to the scope of E'. Variables $\overline{v}$ must be unique to f and hence cannot occur in Γ. The C(IL) version of this type system [1] handles recursive definitions through monomorphic recursion. Function application takes a function reference L_0 and a tuple of arguments of the right shape (rule [CALL]). Calls to functions are recorded with calling call_{ρ_0} effects, where region ρ_0 identifies the callee. The latent effects φ' of the function are recorded as potential side-effects of the invocation. A particular application may perform only a subset of these effects, but cannot perform any effect outside φ'.

Subsumption. It is always safe to enlarge the set of effects inferred for an expression (rule [SUB]). Consider two function variables: f with shape $\mathsf{ref}_{\rho_0}\ (\langle\rangle \xrightarrow{\mathit{read}_{\rho_1}} \bot)$ and g with shape $\mathsf{ref}_{\rho_0}\ (\langle\rangle \xrightarrow{\mathit{write}_{\rho_1}} \bot)$, where $\langle\rangle$ is the empty tuple. Without subsumption we could not write a program like `if (*) &f else &g`. Functions f and g perform different kinds of effects (one reads from and the other writes to ρ_1) and hence their shapes are not equal, as required by rule [IF]. With subsumption, we can enlarge the latent effects of f with write_{ρ_1} and the latent effects of g with read_{ρ_1}, so that their shapes match —now both having latent effects $\{\mathit{read}_{\rho_1}, \mathit{write}_{\rho_1}\}$.

Type casts. In this system, type casts are reduced to shape-type compatibility checks (rule [CAST]). A type cast is allowed only if the shape Z of the expression is compatible with the target type T. Type casts between integer and compatible pointer types, often used to work around type genericity, are correctly handled by this system. Casts that are not generally sensible are rejected, for instance, casting between pointers to functions with different number of arguments.

Principality. We have not proven principality of the inference system. However the original work of Talpin-Jouvelot [47] does guarantee principality, and we have followed their method closely. We have no reason to believe that the same does not hold here.

Soundiness. Our inference system for C(IL) was deliberately made unsound, to gain in simplicity and in precision. This is a necessary trade-off for a bug finding technique [33]. For instance, analyzing the Linux kernel, we have found some complex pointer usage that leads to cycles in the inferred shapes, and also casts between incompatible structure types. We approximate cyclic shapes, and accept any type cast, at the cost of missing aliasing relations. We invite the reader to look at the CIL formulation of our inference system for more details [1]. In any case, our implementation produces an effect-based abstraction for any program that the compiler accepts.

4 Effect-Based Bug Finding

We propose the following bug-finding method based on the inference system presented in Sect. 3, which we have implemented in the EBA tool and evaluated in Sect. 5.

1. Specification of effects for basic operations. We axiomatize the behavior of each relevant operation f with a signature of the form $\overline{Z_i^L} \xrightarrow{\varphi} Z_0$ (cf. Sect. 3.2). The axiom specifies shapes of the input arguments expected by the function (references $\overline{Z_i^L}$), the shape of the output produced by the function (Z_0), and the effects of executing the function (φ). For example, to find the double-lock bug of Fig. 1, we specify the operations `spin_lock` (effect lock_ρ) and `spin_unlock` (effect unlock_ρ) with the following:

$$\texttt{spin\_lock} : \qquad \mathsf{ref}_{\rho_1}\ \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ \zeta \xrightarrow{\mathit{lock}_{\rho_2}} \bot \tag{1}$$

$$\texttt{spin\_unlock} : \qquad \mathsf{ref}_{\rho_1}\ \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ \zeta \xrightarrow{\mathit{unlock}_{\rho_2}} \bot \tag{2}$$

These signatures specify that the functions `spin_lock` and `spin_unlock` receive a pointer as argument (stored as formal parameter in ρ_1) which points to some object stored in ρ_2; the effects above the function arrows indicate that the operations respectively *lock* and *unlock* the object in ρ_2. The shape of the object in question is not relevant and thus has been abstracted away by a shape variable ζ. Note that variables ρ_1, ρ_2, and ζ are local to each signature, and implicitly universally quantified.

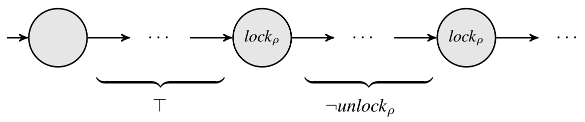
2. Shape-and-effect inference. Following Jouvelot and Talpin [29] we derived an inference algorithm from the declarative system of Sect. 3 (a classical example is the derivation of Damas-Milner's *Algorithm* $\mathcal{W}$ [18]). Essentially, we distributed the effect of the non syntax-directed rule [SUB], and replaced guesses with meta variables and constraints. We use the obtained algorithm to infer the memory shapes and aliasing relationships of all program variables, and the effects for all statements. Each function is assigned a shape-and-effect signature, which establishes aliasing relationships between inputs and outputs, and provides a flow-insensitive summary of its observable behavior.

3. Effect-CFG abstraction. We construct the *Effect-based Control-Flow Graph* (φ-CFG) of the program as in Fig. 1. We begin with a standard CFG, where nodes represent program locations and edges specify the control-flow. We distinguish branching decisions (diamond nodes), atomic operations (circles), function calls (dotted squares), and *return* statements (double-circles). A φ-CFG is an effect-abstraction of a program obtained from the standard CFG by annotating variables with their memory shapes, and nodes with the effects inferred for the corresponding locations. Function call nodes hold a flow-insensitive over-approximation of the callee's behavior. This abstraction can be refined, if needed, by inlining the callee's φ-CFG into the caller's φ-CFG (cf. Fig. 1).

4. Specification of bug patterns. We express bug patterns using existential Computational Tree Logic (CTL) formulae with effects as atomic propositions. The formulae must describe incorrect execution paths. In our example, an execution containing a double-lock bug can be matched using the following CTL formula:

$$\top \ \mathsf{EU} \ (\mathit{lock}_\rho \ \wedge \ \mathsf{EX} \ (\neg \mathit{unlock}_\rho \ \mathsf{EU} \ \mathit{lock}_\rho)) \tag{3}$$

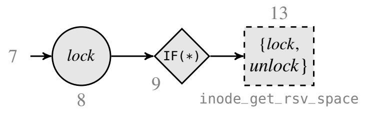
The region ρ works as a meta variable specifying that we are interested in finding a second lock on the *same* memory object, rather than two unrelated lock calls. As shown in Fig. 1, this formula reveals buggy execution paths of the form:



5. Model checking. Matching execution patterns representing bugs can be reduced to the standard CTL model-checking problem for dual safety formulae over the φ-CFG graph. A φ-CFG is interpreted as a transition system where program statements act as states, and effects act as propositions. For instance, a proposition lock_ρ holds in a state (statement) S_i iff the effects of S_i include lock_ρ. The dual property, testifying absence of double-locks, for the example above is $\mathsf{AG} \ (\mathit{lock}_\rho \ \Rightarrow \ \mathsf{AX} \ (\mathit{unlock}_\rho \ \mathsf{AR} \ \neg \mathit{lock}_\rho))$.

We analyze each function's φ-CFG separately, in a modular fashion, relying on the effect-summaries of called functions. A *match* (a counterexample of the safety property) is a bug candidate represented by an *error trace*. If no counterexample is found, we may regard the function as "correct" only if no complex use of pointers is involved (e.g., the use of `container_of` macro in Linux). In general, some complex cases are handled unsoundly, and hence bugs may be missed. This is part of a necessary trade-off [33].

6. Abstraction refinement. Our function summaries are flow-insensitive, so a match may be inconclusive. For instance, in Fig. 1 we first model-check `add_dquot_ref` independently of `inode_get_rsv_space` and obtain the following bug candidate:



Yet, in this match, the second lock acquisition happens at node 13, which is a call to `inode_get_rsv_space`. As reflected in its signature, function `inode_get_rsv_space` both acquires and releases the lock, but the order of these operations is unknown when model-checking `add_dquot_ref`. In such a case, we *refine* the effect-abstraction of `add_dquot_ref` by inlining the call to `inode_get_rsv_space`. The model-checker resumes the search on the refined φ-CFG and a new match, this time conclusive, is found (cf. Fig. 1). This inlining strategy is a simple form of Counter Example Guided Abstraction Refinement (CEGAR) [14]. It allows us to support precise inter-procedural bug finding with a very simple effect language—which otherwise would have to capture ordering.

Other types of bugs. Notice that this technique is fairly general. It can be instructed to find other kinds of resource manipulation bugs using different bug patterns. For example:

$$\text{double free} \qquad \top \ \mathsf{EU}\ (free_\rho \ \wedge \ \mathsf{EX}\ (\neg alloc_\rho \ \mathsf{EU}\ free_\rho)) \tag{1}$$

$$\text{memory leak} \qquad \top \ \mathsf{EU}\ (alloc_\rho \ \wedge \ \mathsf{EX}\ \mathsf{EG}\ \neg free_\rho) \tag{2}$$

$$\text{use before initialization} \qquad \neg init_\rho \ \mathsf{EU}\ use_\rho \tag{3}$$

5 Evaluation

Our objective is to assess the *effectiveness* and *scalability* of our bug-finding technique. For this purpose, we have implemented a prototype static analyzer, EBA, that realizes the bug-finding method described in Sect. 4.

Implementation. EBA is implemented in OCAML and built on top of the CIL [36] front-end infrastructure. We use CIL to generate the CFG of the program, and subsequently decorate it with the inferred shapes and effects. A custom reachability engine matches patterns, of the form P EU Q, against the decorated CFG of each function; a trace is returned upon a match. This engine can perform function inlining on demand. A *bug checker* is a small script that combines reachability queries to search for specific bug patterns. The source code of EBA is publicly available under an open-source license.[6]

Method. We measure the performance of EBA in terms of analysis time and bugs found. We compare EBA against similar bug-finding tools on (1) a benchmark of historical Linux bugs (Sect. 5.1) and (2) the set of device drivers shipped with Linux 4.7 (Sect. 5.2). For simplicity, we only target one type of bug: *double locks*. (Yet our technique is general and can find other types of bugs, see Sect. 4.) Locking bugs are a good representative of resource mis-manipulation, they are introduced regularly, and often have bad consequences for the user (e.g. a device driver hangs). Double-lock checkers are also part of many research tools that have used the Linux kernel for evaluation [24, 39, 51].

Subjects. We compare EBA against SMATCH and COCCINELLE, two tools that are popular within the Linux kernel community. SMATCH is developed and used at Oracle.

[6] https://github.com/iagoabal/eba/

COCCINELLE is a program matching and transformation tool, but it is also used as a bug finder [12] and a double-lock checker is shipped with the Linux distribution.[7]

We selected these two baseline tools for two reasons. First, they are able to run out-of-the-box on the source code of Linux, without major adaptation or further research. Second, there exist double-lock checkers tailored to the Linux kernel available for both of them. Neither CPPCHECK, CLANG STATIC ANALYZER, nor INFER ship with a double-lock checker, so they could not be used for an independent comparison. SPARSE and CQUAL both require modifications to the analyzed source code. Finally, we excluded SATURN, which we could not build against a recent version of OCAML.

Reproducibility. Evaluation artifacts and detailed instructions are available online.[8] All experiments have been conducted on a virtualized machine with a physical 8-core (16-thread) Intel Xeon E5-2660 v3 CPU, running at 2.6 GHz and with 16 GB of RAM.

5.1 Performance on a benchmark of historical Linux bugs

Setup. We evaluate our tool on a benchmark of 26 known double-lock bugs extracted from historical bug fixes in the Linux kernel. In establishing this benchmark, we first obtained a set of 77 candidates by selecting all commits containing the phrase *"double lock"* in its message.[9] We filtered out 30 cases of false positives (i.e., commits not fixing a double-lock bug), and 18 cases of bugs spanning multiple files. To avoid bias, we removed two commits (`3c13ab1` and `1d23d16`) that were fixes to bugs found by EBA. However, we kept any bug-fix derived from the other two contenders.

[7] At `scripts/coccinelle/locks/double_lock.cocci`.
[8] `https://github.com/iagoabal/2017-vmcai`
[9] Extracted from the Linux kernel's Git repository as of August 3, 2016.

Table 1. Comparison of EBA, SMATCH, and COCCINELLE on 26 historical double-lock bugs in Linux. Times in gray strikeout font indicate that the bug was *not* found by the tool.

bug		TIME (seconds)		
hash ID	depth	E	S	C
`00dfff7`	2	5.0	~~1.5~~	~~0.1~~
`5c51543`	2	2.3	~~1.5~~	~~0.3~~
`b383141`	2	~~6.1~~	~~2.9~~	~~0.3~~
`1c81557`	1	5.0	~~1.9~~	~~0.6~~
`328be39`	1	8.9	~~1.7~~	~~0.2~~
`5a276fa`	1	~~0.9~~	~~1.2~~	~~0.2~~
`80edb72`	1	~~6.3~~	~~2.1~~	~~0.7~~
`872c782`	1	1.7	~~2.8~~	~~1.9~~
`d7e9711`	1	21	~~1.3~~	~~2.7~~
`023160b`	0	1.0	2.6	~~0.1~~
`09dc3cf`	0	1.2	~~1.4~~	~~0.1~~
`0adb237`	0	1.1	1.5	0.2
`0e6f989`	0	0.4	1.0	0.3
`1173ff0`	0	0.6	1.3	0.1
`149a051`	0	~~0.7~~	~~0.6~~	~~0.3~~
`16da4b1`	0	0.4	0.8	0.1
`344e3c7`	0	0.7	1.3	~~0.1~~
`2904207`	0	5.8	2.0	~~2.8~~
`59a1264`	0	0.2	0.6	0.1
`5ad8b7d`	0	0.6	3.4	0.1
`8860168`	0	0.7	1.0	0.1
`a7eef88`	0	0.6	1.2	0.2
`b838396`	0	3.3	2.8	1.1
`ca9fe15`	0	0.4	~~0.7~~	1.8
`e1db4ce`	0	0.4	1.1	0.2
`e50fb58`	0	0.5	0.9	0.1

For the 27 remaining commits, we obtained a preprocessed version (under 64-bit x86 *allyes* configuration) of the file where each bug is located. This step excluded one file (commit `553f809`) that failed to preprocess. For COCCINELLE, we retain the original source file, since it is designed to run on unprocessed C files. We then verified that the alleged bug was indeed present in this particular configuration. Thus, we arrived at a benchmark of 26 double-lock bugs from Linux.

Results. Table 1 shows the results of running EBA, SMATCH, and COCCINELLE on this benchmark. We identify each bug by the commit that fixes it, and we group bugs by *depth*. The depth of a bug corresponds to the number of function calls involved from the first to the second acquisition of the lock, e.g. the bug of Fig. 1 involves one function call and therefore has depth one. For instance, for the first bug in the table, `00dfff7`, EBA takes five seconds (5.0) and correctly reports the bug. SMATCH and COCCINELLE take 1.5 and 0.1 seconds respectively, yet are unable to find the bug.

Regarding *effectiveness*, we observe that EBA finds 22 out of the 26 bugs. In comparison, SMATCH and COCCINELLE find 14 and 12 bugs respectively. More specifically, EBA finds six out of the nine interprocedural bugs (depth one or more), whereas SMATCH and COCCINELLE do not manage to find any at all. For the remaining 17 intraprocedural bugs (depth zero), EBA finds all but one (16 out of 17). Remarkably, any bug found by either SMATCH or COCCINELLE, is also intercepted by EBA. Thus, on this benchmark, EBA *is more effective at finding double-lock bugs* than its contenders.

Regarding *false negatives*, we observe that EBA misses five bugs due to limitations in our pointer analysis. This happens, for instance, when the lock object is obtained through the Linux `container_of` macro (defined in `include/linux/kernel.h`). For SMATCH, false negatives seem to be due to path-insensitivity and lack of inter-procedural support. COCCINELLE lacks inter-procedural support and, in addition, its double-lock checker does not recognize some common locking functions. All three bug finders make the assumption that the formal parameters of a function do not alias one another—as indeed mostly the case; and thus, all three tools missed bug `149a051`.

Regarding *analysis time*, we observe that, for the bugs that all three tools find, EBA is on average about 1.4 times faster than SMATCH, yet COCCINELLE is about five times faster than EBA. Note, however, that SMATCH is checking for more bugs than double locks. Also, EBA and SMATCH analyze a total of 665 KLOC of *preprocessed* C code, whereas COCCINELLE analyzes only 27 KLOC of *unprocessed* C files. In this benchmark, all bugs can be found without including headers, which is an advantage for COCCINELLE. We also observe that variance of execution times is higher for EBA, with six files taking more than five seconds to analyze, and one file taking 21 seconds. These files contain large functions that manipulate multiple locks and, as of now, EBA will check one lock object at a time. We foresee that EBA will speed up considerably with some optimization work, e.g. by performing multiple checks in a single traversal.

5.2 Performance of analyzing device drivers in Linux 4.7

Setup. We use EBA to analyze widely the entire `drivers/` directory of Linux in search of double-lock bugs. EBA was run on the Linux 4.7-rc1 kernel in the 64-bit x86 *allyes* configuration, invoked by Kbuild during a parallel build process with 16 jobs (i.e., `make -j16`). About nine thousand files in `drivers/` were analyzed, and we manually classified each one of the bugs reported, as either a true or a false positive. We repeated this process for SMATCH and COCCINELLE, to confront analysis times and number of false positives. All tools were given 30 seconds to analyze each file.

Results. EBA reported nine bugs in nine different files (i.e., 0.1% of the files analyzed). *Five* of these bug reports have been reported and *confirmed* by the respective Linux maintainers, and *three are now fixed* in Linux 4.9 (see commits `1d23d16`, `e50525b` and `bea6403`).[10] These bugs affected some TTY, SCSI, USB, Intel IOMMU, and Atheros wireless drivers. The five bugs had depth one or more, and required an inter-procedural analyzer. (SMATCH and COCCINELLE found no bugs, but that is somewhat expected because, presumably, any bugs would have already been reported and fixed.)

EBA analyzes all Linux drivers in *less than half an hour* (23 minutes) and is only slightly slower than SMATCH which does the same in 16 minutes (1.4 times faster than EBA). COCCINELLE is significantly faster and completes the analysis in only two minutes, as it scans much smaller unprocessed files.

We classified four of the nine bugs reported by EBA as false positives. Three cases were due to limitations in our pointer analysis, and in one case the reported error trace was not a feasible execution path. One of the false positives reported by EBA still led to a cosmetic fix (see `3e70af8`). Both SMATCH and COCCINELLE report more false positives (eight and six, respectively). It is worth noting that, at least in eight of these cases, there was an *unlock* operation being performed through a nested function call—that these tools were not aware of.

6 Conclusion

We have presented a two-step bug-finding technique that uncovers deep resource manipulation bugs in systems-level software. This technique is lightweight and easily scales up to large code bases, such as the Linux kernel. First, a *shape-and-effect* inference system is used to build an abstraction of the program to analyze (Section 3). In this abstraction, objects are described by memory *shapes*, and expressions and statements by their operational *effects*. Second, bugs are found by matching temporal bug-patterns against the control-flow graph of this program abstraction (Section 4).

We have implemented our technique in a prototype bug finder, EBA, and demonstrated the effectiveness and scalability of our approach (Section 5). We have compared the performance of EBA with respect to two bug-finders popular within the Linux community: COCCINELLE and SMATCH. On a benchmark of 26 historical Linux bugs, EBA was able to detect strictly more bugs, and more complex, than the other two tools. EBA is able to analyze nine thousand files of Linux device drivers in less than half an hour, in which time it uncovers five previously unknown bugs. So far, EBA has found *more than a dozen double-lock bugs* in Linux 4.7–4.9 releases, eight of which have already been confirmed, and six are fixed upstream.

[10] Bug `e50525b` was independently found and fixed during beta testing, but that bug-fix was unknown to us.

References

[1] I. Abal. Shape-region and effect inference for C(IL). Technical report, 2016.

[2] I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the Linux kernel: A qualitative analysis. ASE 2014.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. PLDI '01, 2001.

[4] T. Ball and S. K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. PASTE 2001.

[5] T. Ball and S. K. Rajamani. The SLAM toolkit. CAV 2001.

[6] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. POPL 1979.

[7] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. FMCO 2005.

[8] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, Oct. 2007.

[9] L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. POPL 2004.

[10] F. Brown, A. Nötzli, and D. Engler. How to build static checking systems using orders of magnitude less code. ASPLOS '16, 2016.

[11] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, and G. Muller. A foundation for flow-based program matching: Using temporal logic and model checking. POPL 2009.

[12] Y. Chen, F. Wu, K. Yu, L. Zhang, Y. Chen, Y. Yang, and J. Mao. Instant bug testing service for Linux kernel. HPCC/EUC 2013.

[13] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs, 2004.

[14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. CAV '00, 2000.

[15] K. D. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. SIGPLAN 1984.

[16] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. PLDI 1988.

[17] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does Astrée scale up? *Form. Methods Syst. Des.*, 35(3), Dec. 2009.

[18] L. Damas and R. Milner. Principal type-schemes for functional programs. POPL 1982.

[19] I. F. Darwin. *Checking C Programs with Lint*. O'Reilly, 1986.

[20] M. Das. Unification-based pointer analysis with directional assignments. PLDI 2000.

[21] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. OSDI, 2000.

[22] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *Proceedings of the 7th International Symposium on Static Analysis*, SAS '00, pages 175–198, London, UK, UK, 2000. Springer-Verlag.

[23] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 2006.

[24] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. PLDI 2002.

[25] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. PLDI '02. ACM, 2002.

[26] M. Hind. Pointer analysis: Haven't we solved this problem yet? PASTE 2001.

[27] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[28] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. POPL '91, 1991.
[29] P. Jouvelot and J.-P. Talpin. The type and effect discipline, 1993.
[30] O. Kiselyov and C.-c. Shan. Lightweight monadic regions. Haskell 2008.
[31] J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller. Finding error handling bugs in openssl using coccinelle. EDCC 2010.
[32] D. Leijen. Koka: Programming with Row Polymorphic Effect Types. MSFP 2014.
[33] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2), Jan. 2015.
[34] J. M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD thesis, 1987.
[35] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. POPL 1988.
[36] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. CC 2002.
[37] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances*, volume 1710 of *LNCS*. Springer, 1999.
[38] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. EuroSys 2006.
[39] N. Palix, G. Thomas, S. Saha, C. Calvès, G. Muller, and J. Lawall. Faults in Linux 2.6. volume 32, pages 4:1–4:40. ACM, June 2014.
[40] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. PLDI 2006.
[41] D. Remy. Type inference for records in a natural extension of ML. In *Theoretical Aspects Of Object-Oriented Programming*. MIT Press, 1993.
[42] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. POPL 1995.
[43] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, Apr. 2015.
[44] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. CC 1996.
[45] B. Steensgaard. Points-to analysis in almost linear time. POPL 1996.
[46] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 1986.
[47] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2, 7 1992.
[48] M. Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1), 1990.
[49] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. POPL 1994.
[50] D. A. Wright. A new technique for strictness analysis. TAPSOFT 1991.
[51] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. POPL 2005.
[52] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. PLDI 1999.