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 projects tend to suffer from conceptually simple resource manipulation bugs, such as accessing a de-allocated memory region, or acquiring a non-reentrant lock twice. Static code scanners are used extensively to remove these bugs from projects like the Linux kernel. Yet, when the manipulation of the resource spans multiple functions, efficiently finding these bugs is a challenge. We present a *shape-and-effect* inference system for C, that enables efficient and scalable inter-procedural reasoning about resource manipulation. The inference system is used to build a program abstraction: a control-flow graph decorated with alias relationships and observable effects. Bugs are found by model checking this control-flow graph, matching undesirable sequences of operations. We evaluate a prototype implementation of our approach (EBA) and run 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 files of Linux device drivers in less than half an hour, and uncovers previously unknown double-lock bugs in various drivers.

1 Introduction

Today, the source code of the Linux kernel is periodically examined using a handful of Lint-like [19] static code-scanning tools [12]. These tools 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. These Linux-tailored *linters* have seen adoption because they run fast, and are reasonably effective at finding certain classes of bugs. However, linters are fundamentally restricted by their lack of semantic understanding of the code; they mostly find shallow bugs, not involving function calls.

Bugs spanning multiple functions are common [1]. The VBDb bug collection documents 24 Linux bugs involving more than one function. 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 about resource manipulation.

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

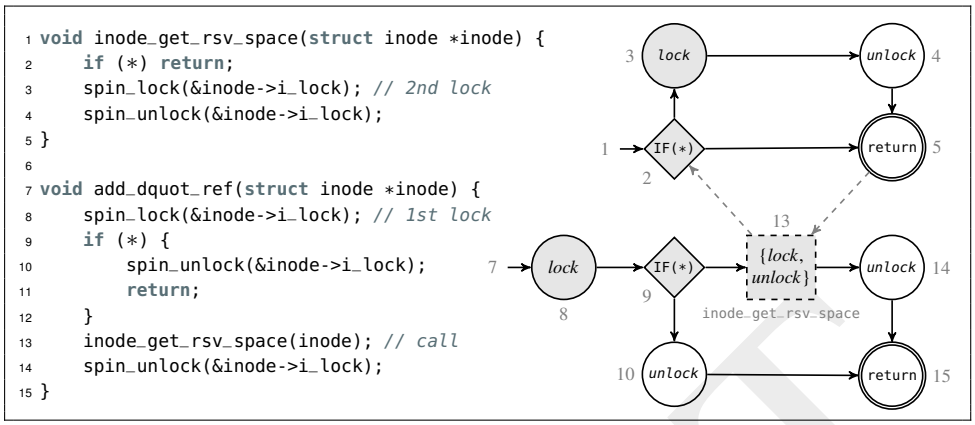


Fig. 1. An illustration of our bug-finding technique on a double-lock bug in Linux fixed by commit d7e9711. The *simplified* code is shown to the left. To the right, the associated CFG annotated with *lock* and *unlock* effects. The numbers next to the CFG nodes show corresponding line numbers. The gray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

We approach the problem by providing aliasing and side-effect information to a static code-scanning engine. This data is inferred by a flow-insensitive type-and-effect analysis [30,33]. Our types approximate the shape of the program data and computational effects; hence the name: *shape*-and-effect system. The set of effects is extensible. It describes how data is manipulated by the program, for instance: reading and writing variables, acquisition or release of locks, opening and closing of files, etc. Since our goal is bug finding and not program optimization nor verification, we trade soundness for scalability [29]. Our type analysis is based on Talpin-Jouvelot's work [42], and resembles the classic Damas-Milner's inference algorithm [18].

We use this flow-insensitive shape-and-effect system to obtain an abstraction of the program. In this abstraction, every variable is assigned a memory shape and storage regions, and expressions and statements are described by a set of computational effects over those regions. In addition, each program function is given a polymorphic shape-and-effect signature, that summarizes its computational behavior. The inferred annotations are superimposed on the control-flow graph (CFG). Then, as linters do, we match flow patterns specifying potential bugs against the effect-annotated CFG, using a standard model checking algorithm. Whenever a function call is reached, it is replaced by its effect signature. When hunting down a bug, some function calls may be inlined. 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, fixed by commit d7e9711. Function `add_dquot_ref` causes a potential deadlock by double-acquiring a non-reentrant *spin 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). For the error to occur, 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 (for the locking effects on `&inode->i_lock`). The gray nodes, and edges between them, mark an

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 of effects: $\{lock, unlock\}$). In line 13, from the set of effects alone (i.e., without looking into the function `inode_get_rsv_space`), it is unclear as to whether the acquisition of `&inode->i_lock` happens *before* or *after* its release. In such inconclusive scenarios, we perform function inlining (essentially following the gray dashed lines) which unveils the double-lock bug in line 3.

Our contributions are:

- An adaptation of Talpin-Jouvelot's [42] 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).[2] 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 a handful of previously unknown double-lock bugs in Linux 4.7.
- 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. Lucassen introduces type-and-effect systems [30]. Nielson and Nielson [33] survey a number of applications in strictness analysis [45], automatic parallelization [31], or memory management [44]. Our effect abstraction belongs to the realm of side-effect analyses using polymorphic type and effect inference [28,42].

Talpin and Jouvelot introduce a polymorphic type-and-effect system with sub-effecting for the λ-Calculus with ML-like references [42]. Polymorphism offers increased precision, and sub-effecting is relevant when manipulating function values. We extend this to a real systems-level programming language. In order to handle common patterns of generic programming in C through type casting, we assume that programs already type-check in the standard C type system, and we instead infer the shape of memory

[2] https://eba.wikit.itu.dk

objects. KOKA [28] is a functional programming language featuring effect polymorphism and sub-effecting based on row types [37,28]. It addresses important concerns when the effect system is exposed to, and needs to be understood by the programmer. Our effect system is an internal program abstraction, thus we settled on Talpin-Jouvelot's system as a basis instead.

Type-based analyses. Many works consider programming with side-effects, and type-safe resource manipulation, for instance [7,27,36,41]. These techniques impose stricter typing disciplines that prevent errors. They cannot be directly applied to finding bugs in existing system-level C programs, as they require annotating entire systems with new types.

Notably, CQUAL is a constraint-based type checker [22,23] for C that has been applied to the Linux kernel code. It extends the C type system with user-definable type qualifiers that track the state of objects in a similar fashion as *typestate* [41] does. We assume that incorrect manipulation of interrupts, which for instance have no associated C object, cannot easily be expressed in CQUAL. EBA tracks the execution of operations rather than the state of resources. Further, CQUAL may require user annotations, or even rewrite code to be accepted by the tool. We instead trade soundness for precision, and our bug finding technique requires no alterations to code.

Similar to CQUAL, LOCKSMITH is another verification tool based on constrained-based type inference [35]. LOCKSMITH detects data races in C programs by checking whether any access to the same shared memory location is consistently protected by the same set of locks. For this, it requires that any lock variable can be mapped to a unique lock object. This linearity condition is difficult to enforce when locks are part of data structures and are involved in non-trivial type conversions, like in the Linux kernel. LOCKSMITH also requires users to introduce annotations and potentially rewrite code.

Side-effect and pointer analysis. Side-effect analyses can complement intra-procedural analyses by capturing the semantics of function calls, otherwise treated as black-boxes. They have been applied to compilation, JIT, and static analysis. Work on efficient inter-procedural side-effect analysis predates type-and-effect systems [15,16]. We have preferred simpler type analyses [20,39,40] over more complex flow-sensitive ones [5,6,24], mostly for scalability. Our shape-and-effect system implements context-sensitive alias analysis via shape and region polymorphism. Studies show that context sensitivity offers a good tradeoff between precision and scalability [20,25]. We prioritized a precise analysis of C structure types, as this has been repeatedly identified as a requirement to analyze complex C software [39,47].

Code scanners. Static code scanners are mostly syntax-based tools that essentially parse the source code, construct a CFG for each program function, and analyze each function in isolation. These tools know little about semantics: do no compute function summaries, ignore aliasing, rely on syntactic pattern matching, and work intra-procedurally. Hence are fast and scale well. For the same reasons, they are often limited to relatively simple and shallow bugs. EBA works similarly, but effectively supports inter-procedural bug finding, by analyzing a lightweight abstraction based on shapes and effects.

In particular, Linux is being continuously analyzed [12] by SPARSE, COCCINELLE and SMATCH.[3] SPARSE exploits Linux-specific annotations to perform simple checks.

[3] See http://sparse.wiki.kernel.org, http://coccinelle.lip6.fr, http://smatch.sf.net.

Some annotations are more popular than others and, in particular, locking annotations are rarely used. COCCINELLE is a program transformation tool [11], with an associated DSL to specify flow-based transformations. COCCINELLE flow-patterns can help finding illegal sequences of operations, such as double lock. SMATCH is a flow-based program matcher that is tailored to finding bugs. SMATCH realizes the idea of *meta-level compilation* (MC) proposed by Engler and coauthors [21]. It is, essentially, a scriptable flow analysis engine.

The bug of Fig. 1 is not found by SPARSE since the functions involved have no locking annotations. It is also missed by SMATCH, and by the COCCINELLE double-lock script shipped with Linux. This is because the second acquisition of the lock happens through a nested call to function `inode_get_rsv_space`, something that EBA knows looking at its effect signature. SMATCH has limited support for inter-procedural bug finding. As in [21], SMATCH provides a script to traverse the whole program and collect all functions that, for instance, may perform locking. On a regular laptop, this script takes a few hours to run on the entire Linux kernel, and must be run n times to examine n levels of the call graph. Compared to a proper effect system such a script is difficult to extend; and does not track regions and aliasing, nor handles function pointers appropriately.

Static analysis. Static analyzers have a deep understanding of program semantics. They can precisely track the values of expressions and the shape of the objects in the heap. These tools can find very complex bugs including divisions by zero, arithmetic and buffer overflows. In order to scale, they rely heavily on abstraction [2,9,14,17]; and to compute abstractions, they make extensive use of automated theorem provers. This power comes at the cost of longer execution times and higher memory usage. Our shape-and-effect abstraction is lightweight but powerful enough to find our target class of bugs. On a regular laptop, EBA analyzes the entire Linux kernel in about an hour.

ASTRÉE is a scalable abstract interpretation tool for static analysis of safety-critical systems [17]. ASTRÉE can prove the absence of errors such as null-pointer dereferences, but only for a restricted subset of C. Reps et al. show that an important class of inter-procedural data-flow analyses can be performed precisely and efficiently by reduction to a graph reachability problem [38]. Their formulation supports partial function summaries. Based on this work, Ball and Rajamani developed BEBOP [3], a path-sensitive data-flow engine that is used to model-check Boolean programs in the SLAM Toolkit [4]. CBMC is a model checker that reduces the verification of statically bounded 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 [8,10]. 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 [46]. Both INFER and SATURN compute elaborated path-sensitive summaries for functions, and can model the runtime shape of complex data structures precisely. Our technique differs in that both our shape analysis and function summaries are flow-insensitive, and we tackle the lost 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 [26,42]. Because of unsafe casts, the standard C type system provides only a meager description of run-time objects. Thus, as known in pointer analysis [39,40], 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 [32]. 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, 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:

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

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 $\texttt{ref}\ T$, 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 $\texttt{ptr}\ \texttt{ref}\ T$, 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):

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

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, $*\texttt{E}$, 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

[4] https://eba.wikit.itu.dk/file/view/EBAinfer.pdf

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 values, while effects describe other properties of evaluation. From the type perspective, the expression y `:=` 1 `+` `!`x`;` `!`y evaluates to an integer value. From the memory 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 an evaluation, 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 $\varepsilon_{\overline{\rho}}$. Specific constructors read_ρ and write_ρ are reserved to represent reading and writing of memory locations. The example of Sect. 1 used effects lock_ρ and unlock_ρ to represent lock manipulation actions. Other effects can be introduced to capture new kinds of bugs. 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).

The shape-and-effect system performs integrated flow-insensitive alias analysis, similar to Steengaards's points-to analysis [40] 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 $\{\mathit{read}_{\rho_1}, \mathit{read}_{\rho_2}, \mathit{write}_{\rho_2}\}$. 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 $\{\mathit{read}_{\rho_{\{1,2\}}}, \mathit{write}_{\rho_{\{1,2\}}}\}$.

[5] We use overline to denote tuples.

Shapes. A *shape* approximates the memory representation of an object [39,40]. 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:

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

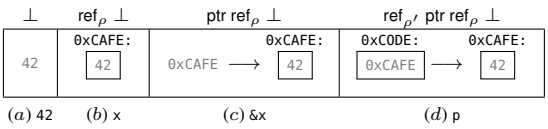
As for types, we split shapes into l-value (Z^L) and r-value (Z^R) shapes. The shape language resembles the type language, without integer type but with shape variables (ζ).

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 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`, $\mathsf{ref}_{\rho'}\ \mathsf{ptr}\ \mathsf{ref}_\rho\ \bot$.

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



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:

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

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

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 $\bot \le \texttt{int}$, $\mathsf{ptr}\ Z \le \texttt{int}$, and $\zeta \le \texttt{int}$ 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 [43], 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 [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 $\Gamma \vdash_E E : Z\ \&\ \varphi$ (Fig. 3b) specifies that, under environment Γ, r-value expression E has shape Z, and side effects φ.

(a) Inference rules for l-value expressions, $\vdash_L \subseteq$ ENV $\times$ L-VALUE $\times$ SHAPE $\times$ EFFECT.

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

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

(b) Inference rules for r-value expressions, $\vdash_E \subseteq$ ENV $\times$ EXP $\times$ SHAPE $\times$ EFFECT.

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

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

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

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

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

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

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

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

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

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. Uses of `new` are recorded with initialization $init_\rho$ effects. 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. 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 $call_{\rho_0}$ 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 $\mathsf{ref}_{\rho_0}\ (\langle\rangle \xrightarrow{read_{\rho_1}} \bot)$ and g with shape $\mathsf{ref}_{\rho_0}\ (\langle\rangle \xrightarrow{write_{\rho_1}} \bot)$, 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 $write_{\rho_1}$ and the latent effects of g with $read_{\rho_1}$, so that their shapes match —now both having latent effects $\{read_{\rho_1}, write_{\rho_1}\}$.

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.

Soundiness. When analyzing real C programs we cannot afford the luxury of rejecting those that do not satisfy our typing rules. Systems-level programming often exploits compiler and architecture specific behaviors, that we cannot deal with easily. Yet, our goal is to build an effect-based abstraction of any program that the compiler accepts. In practice, our implementation handles common uses of pointer arithmetic and type

casting gracefully, as we described it here. But, to be pragmatic, it relaxes the application of rules, and is deliberately unsound if needed. We do not know of any verification tool for C that is actually sound [29], and we find this to be a necessary trade-off for a bug finding technique.

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 $lock_\rho$) and `spin_unlock` (effect $unlock_\rho$) with the following:

$$\texttt{spin\_lock}: \quad \mathsf{ref}_{\rho_1}\ \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ \zeta \xrightarrow{lock_{\rho_2}} \bot \tag{1}$$

$$\texttt{spin\_unlock}: \quad \mathsf{ref}_{\rho_1}\ \mathsf{ptr}\ \mathsf{ref}_{\rho_2}\ \zeta \xrightarrow{unlock_{\rho_2}} \bot \tag{2}$$

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.

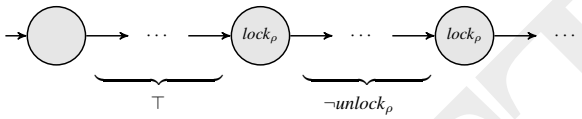
2. Shape-and-effect inference. Following Jouvelot and Talpin [26] we derived an inference algorithm from the declarative system of Sect. 3 (a classical example is the derivation of Damas-Milner's *Algorithm* $\mathcal{W}$ [18]). Essentially, we distributed the effect of the non syntax-directed rule [SUB], 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 (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:

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

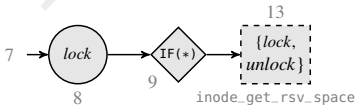
The region ρ works as a meta variable specifying that we are interested in finding a second lock on the *same* memory object, rather than two unrelated lock calls. As shown in Fig. 1, this formula reveals buggy execution paths of the form:



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 [29].

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* `add_dquot_ref`'s effect-abstraction 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 Sections 3 and 4. The source code of EBA is publicy available under an open-source license.[6] EBA is implemented in OCAML and built on top of the CIL [32] front-end infrastructure.

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. the OS hangs). Double-lock checkers are also part of many research tools that have used the Linux kernel for evaluation [23,34,46].

Subjects. We compare EBA against SMATCH and COCCINELLE, two tools popular within the Linux kernel community. SMATCH is developed and used at Oracle to find security bugs in Linux. COCCINELLE is a program matching and transformation tool, but it is also used as a bug finder [12] and a COCCINELLE-based double-lock checker is shipped with the Linux distribution (at `scripts/coccinelle/locks/double_lock.cocci`).

We selected these two baseline tools for two reasons. First, they are able to run out-of-the-box on Linux's source code, 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. Also, CLANG-based tools cannot analyze the Linux kernel directly due to unsupported GCC extensions. 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.[7] 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.

[6] https://github.com/models-team/eba/

[7] https://github.com/iagoabal/2017-vmcai

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 commit containing the phrase *"double lock"* in its message.[8] 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 keep any bug-fix derived from the other two contenders.

For the 27 remaining commits, we obtained a preprocessed version (under 64-bit x86 *allyes* configuration) of the file where each bug is located. For COCCINELLE, we retain the original source file, since it is designed to run on unpreprocessed C files. We then verified that the alleged bug was indeed present in this particular configuration. This process excluded one commit (`553f809`) that failed to preprocess on our machine, presumably due to hardware-specific code for the ARM architecture. 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

[8] 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 hash ID	depth	TIME (seconds) 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

bug hash ID	depth	TIME (seconds) E	S	C
`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

finds six out of the nine inter-procedural bugs (depth one or more), whereas SMATCH and COCCINELLE do not manage to find any at all. For the remaining 17 intra-procedural 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*. For SMATCH, false negatives seem to be due to path-insensitivity and lack of inter-procedural support. COCCINELLE lacks inter-procedural support and its double-lock checker does not recognize some common locking functions. The false negatives in EBA's reports can be attributed to limitations in the alias analysis. 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. EBA and SMATCH analyze a total of 665 KLOC of *preprocessed* C code, whereas COCCINELLE analyzes only 27 KLOC of *unpreprocessed* 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 (Recall that EBA is a proof-of-concept prototype that did not undergo much optimization).

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 (`make -j16`). About nine thousand files in `drivers/` were analyzed, and each bug reported was classified as either a true or a false positive. This process was repeated for SMATCH and COCCINELLE.

Results. Table 2 shows how many bugs (true positives) were found by each tool along with the number of false positives and the total analysis time (in minutes). EBA found *four bugs* all of which span multiple functions. We have submitted EBA's bug reports and two of these have already been confirmed by a Linux maintainer and fixed in Linux; one in a USB gadget driver and another in a wireless card driver (see Linux commits `1d23d16` and `e50525b`). In turn, SMATCH and COCCINELLE found no bugs, but that is somewhat expected, given that these tools are actively used by Linux community (presumably, any bugs would have already been reported and fixed). EBA reports five false positives. Both SMATCH and COCCINELLE report more false positives (eight and six, respectively). One

Table 2. Analyzing the entire `drivers/` subsystem of the Linux kernel.

	EBA	SMATCH	COCCINELLE
Bugs found	**4**	0	0
False positives	**5**	8	6
TIME (minutes)	23	16	**2**

of the false positives reported by EBA still led to a cosmetic fix (see `3e70af8`). In total, 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 (but it scans unpreprocessed files without expanding macros nor including header files).

Analyzing variations of the *allyes* configuration, EBA has found another *six bugs* in other Linux files not included in *allyesconfig*. Two of these bugs are confirmed and fixed, one in the CMT speech protocol used by Nokia modems, another one in the CPDMA driver for Texas Instruments ethernet cards (cf. `3c13ab1` and `fccd5ba`). Another bug was confirmed by a senior Linux developer in the `net/` sub-system. So far, EBA has found a total of *ten* double-lock bugs in Linux.

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 *four* previously unknown bugs. So far, EBA has found a total of *ten* double-lock bugs in Linux 4.7, five of which have already been confirmed and fixed.

References

1. I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the Linux kernel: A qualitative analysis. ASE 2014.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. PLDI '01, 2001.
3. T. Ball and S. K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. PASTE 2001.
4. T. Ball and S. K. Rajamani. The slam toolkit. CAV 2001.
5. J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. POPL 1979.
6. J. P. Banning. *A Method for Determining the Side Effects of Procedure Calls.* PhD thesis, Stanford, CA, USA, 1978. AAI7905815.
7. A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012.

8. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. FMCO 2005.
9. 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.
10. L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. POPL 2004.
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, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 2006.
23. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. PLDI 2002.
24. M. Geffken, H. Saffrich, and P. Thiemann. Precise interprocedural side-effect analysis. ICTAC 2014.
25. M. Hind. Pointer analysis: Haven't we solved this problem yet? PASTE 2001.
26. P. Jouvelot and J.-P. Talpin. The type and effect discipline, 1993.
27. O. Kiselyov and C.-c. Shan. Lightweight monadic regions. Haskell 2008.
28. D. Leijen. Koka: Programming with Row Polymorphic Effect Types. MSFP 2014.
29. 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.
30. J. M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD thesis, 1987.
31. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. POPL 1988.
32. 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.
33. F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances*, volume 1710 of *LNCS*. Springer, 1999.
34. N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: Ten years later. ASPLOS XVI, 2011.
35. P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. PLDI 2006.
36. M. Pretnar. Inferring algebraic effects. *Logical Methods in Computer Science*, 10(3), 2014.
37. D. Remy. Type inference for records in a natural extension of ML. In *Theoretical Aspects Of Object-Oriented Programming*. MIT Press, 1993.

38. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. POPL 1995.
39. B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. CC 1996.
40. B. Steensgaard. Points-to analysis in almost linear time. POPL 1996.
41. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 1986.
42. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2, 7 1992.
43. M. Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1), 1990.
44. M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. POPL 1994.
45. D. A. Wright. A new technique for strictness analysis. TAPSOFT 1991.
46. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. POPL 2005.
47. S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. PLDI 1999.