EBA: Effect-Based Analysis of C Programs

Iago Abal*

IT University of Copenhagen, Denmark
iago@itu.dk

The analysis of 42 Linux bugs from a previous study suggests that conceptually simple resource manipulation bugs occur often in practice [1]. While simple, these bugs can be easily and inadvertently introduced when altering large or complex functions. Static code scanners (or *linters*) excel at finding shallow bugs of this class. For instance, Linux commits ca9fe158842 and 65582a7f4ce fix locking bugs found by two of these tools. However, by looking exclusively at syntax, most linters fail at finding deep bugs that span multiple functions. Semantic static analyzers do find deep bugs, but are slower and less scalable, and may not fit well into programmers' work-flow. In this paper, I shall present EBA[1], a prototype static analyzer for C, that finds deep resource manipulation bugs using a minimal amount of semantic information.

EBA employs lightweight program abstractions based on computational effects, being capable of reasoning efficiently and interproceduraly on large code bases. These abstractions are built modularly using a flow-insensitive type-and-effect analysis based on Talpin-Jouvelot's previous work [2]. This analysis infers memory *shapes* annotated with memory *regions* rather than regular C types. Effects describe computations performed on the data stored at those regions. The inferred shape and effect information is superimposed on the control-flow graph (CFG), that is model checked using a bounded depth-first search algorithm. This model checking step is also modular, since function calls are abstracted by the effect signature of the called function. However, when a potential bug is found, certain function calls must be examined to rule out false positives.

Figure 1 illustrates the overall idea using a simplified version of an actual Linux bug. Function `f` exhibits a potential deadlock by double-acquiring a non-reentrant *spin lock*: the first lock acquisition occurs in l.9, and the second occurs in l.3 after calling function `g`. The `if` conditionals in lines 2 and 10, where we use `*` to specify non-determinism, must both have taken the *else* branch. The shape-and-effect analysis records a few built-in effects: reads and writes to variables, and calls to functions. It is also possible to define arbitrary and project-specific effects that will expose any opera-

Analyzed code

```
 1 void g(struct inode *inode) {
 2   if (*) return;
 3   spin_lock(&inode->i_lock);
 4   // ...
 5   spin_unlock(&inode->i_lock);
 6 }
 7
 8 void f(struct inode *inode) {
 9   spin_lock(&inode->i_lock);
10   if (*) {
11     spin_unlock(&inode->i_lock);
12     return;
13   }
14   g(inode);
15   spin_unlock(&inode->i_lock);
16 }
```

Effect-decorated CFG for f

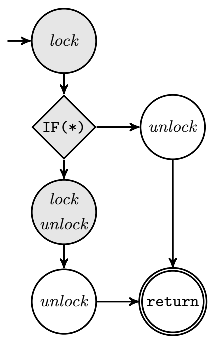

Figure 1: A simplified illustration of the bug finding technique on a double lock bug in Linux (fixed by commit d7e9711760a).

*While pursuing a PhD under the advisement of Andrzej Wąsowski and Claus Brabrand.

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

tion of interest in the program abstraction. In the effect-decorated CFG generated for `f` —on the bottom of the figure, statements are described by their locking effects on `&inode->i_lock`.

EBA is equipped with a reachability engine that matches CTL formulas of the form $\mathcal{P}$ EU $\mathcal{Q}$ against decorated CFGs. (Here, $\mathcal{P}$ and $\mathcal{Q}$ are predicates over effects.) We specify bug checkers for concrete bug types by composing reachability queries. For instance, a simple double-lock checker is specified by TRUE EU (LOCK $\wedge$ $\neg$UNLOCK EU LOCK). When the model checker finds the call to `g` in Fig. 1, it uses its effect signature as an approximation of its computational behavior: here given by the set of effects $\{lock, unlock\}$. On this CFG, the above checker reveals a potential double lock whose execution path is marked by gray colored nodes. Yet, because the analysis is flow-insensitive, it is unclear whether in `g` the acquisition of the lock happens before its release or vice-versa. In this scenario, EBA will examine the call to `g` and check that `&inode->i_lock` will be re-acquired before releasing it, thereby confirming the double lock bug. For a different example, this step may serve to rule out a potential false positive.

In a preliminary evaluation, I have run two EBA checkers on four of the major Linux subsystems, from a random configuration of the Linux-*next* kernel sources (snapshot `8babd99a86f5`), in search of potential bugs related to interrupt management. Table 1 shows the execution time, memory consumption, and bug alarms reported by the two checkers. All reported numbers are collected on *AMD Opteron 6386* CPUs running Ubuntu Linux. We stop analyzing a file when it exceeds 10 minutes time or 8 GB of virtual memory. A total of nine alarms are reported. We identified two cases in which bottom-halves are enabled after disabling interrupts, a violation of lock ordering. We also find seven cases where bottom-halves are unnecessarily re-disabled at each loop iteration, a potential performance bug. The nine alarms involved third functions, and in one case that function is called through a function pointer.

kernel subsystem	files	average time [s]	average mem. [MB]	BI	BL
`drivers`	8,598	22.93	346	1	2
`fs`	877	9.12	260	0	0
`kernel`	221	15.23	335	1	0
`net`	1215	18.18	372	0	5
total	10,911	16.37	328	2	7

Table 1: Analyzing Linux with checkers: *enable bottom-halves with interrupts disabled* (BI), and *re-disabling bottom-halves within a loop* (BL). The last two columns count the alarms raised in each subsystem

These results suggest that EBA can efficiently find cross-function bugs in large code bases. The proposed technique seems to offer a good compromise between efficiency and precision. I expect to reduce execution times and memory usage significantly in the short term, since there are several pieces of low hanging fruit for optimization. The main issue to address is the imprecision of the alias analysis. For this, more precise techniques can be used to triage potential bugs and avoid false positives.

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the Linux kernel: A qualitative analysis. ASE 2014.

[2] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 1992.