

Using Programmer-Written Compiler Extensions to Catch Security Holes

Ken Ashcraft and Dawson Engler

Presented by Kelvin Ku

Some material adapted from Engler's PASTE
2002 slides

Context

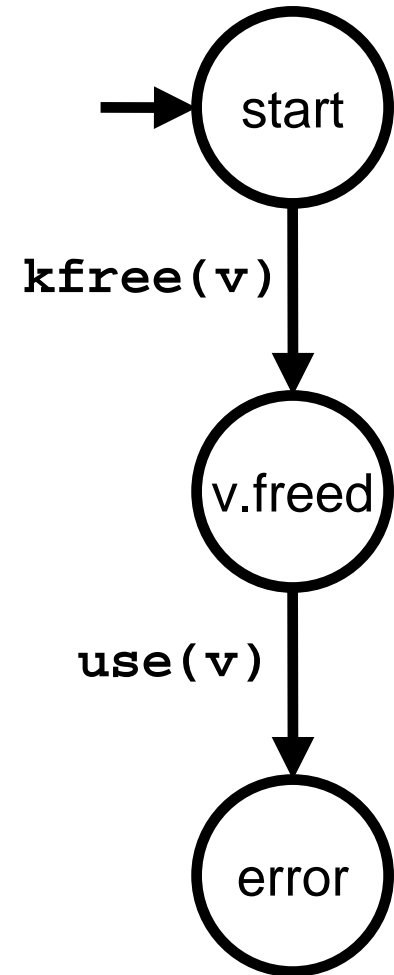
- Systems must obey ad hoc rules
 - User data must be bounds-checked before certain uses
 - Do not dereference user pointers
- Rules often require expert knowledge to specify
 - Sources of user data, dangerous uses
- Rules are typically expressible in terms of program constructs
 - u refers to user data in copyin (u, k, sz)
 - $x < y$ is an upper-bound check on x
- Rules are difficult to specify completely
 - Some sources of user data may not be known initially

Proposed Technique

- Specify a rule as a finite state machine
 - Transitions defined in terms of program constructs
 - States associate high-level property with a variable
- Check rule on program's AST
 - Program statements trigger transitions in FSM
 - Checker emits error if FSM enters error state
- Augment specification by “belief inference”
 - Heuristics for finding missing transitions

Specification Method: Metal

```
sm free_checker {  
state decl any_pointer v;  
decl any_pointer x;  
start: { kfree(v); } ==> v.freed;  
v.freed:  
{ v != x } || { v == x }  
==> { /* do nothing */ }  
| { v }  
==> { err("Use after free!"); };  
}
```



Verification Method: Metacompilation

- Traverse each path through program, matched constructs trigger FSM transitions

```
foo(int *x) {                                freeit(int *z) {
    freeit(x); {2:x.freed}                    kfree(z) {1:z.start->z.freed}
    if (y) {                                  }
        ... {3:x.freed}
    } else {
        ...
    }
    *x; {4:x.freed->x.error}
}
```

Applications

- Mainly memory safety
- Range-checking of tainted data
 - includes checking for misuse of signed integer as unsigned; ignoring bounds-checks on potentially overflowed expressions
- User-pointer dereference
- Memory leak
- Double-free
- Deadlock (special case)
- Extension: marking user-controllable paths

Refinement: Belief Inference

```
mystery(&i);    case A:
               copy_from_user
if (i < MAX)   (&ti, arg, sizeof (ti));
               ...
               case B:
               tbl[arg] = ...;
```

- `mystery` is probably an untrusted source since `i` is bounds-checked
- `arg` is probably tainted in case B since it's tainted in case A

Other Refinements

- State propagation
 - Transitive tainting: $p.\text{tainted}; q = p; q.\text{tainted}$
- State inheritance
 - Struct fields inherit struct state
- Inter-procedural analysis (context-sensitivity)
- Function summaries
 - $\text{source} \rightarrow f1 \rightarrow f2 \rightarrow \dots$
 - $f1 \rightarrow f2 \rightarrow \dots \rightarrow \text{sink}$
- Error ranking
 - Generic: e.g., short over long
 - Checker-specific: e.g., demote traces w. `kmalloc`

Summarized Results

- Linux kernel: found 125 bugs, 52 fixed
 - 109 local; inter-procedural analysis small payoff
 - 24 false positives; verified rest of unfixed bugs
 - 12 inferred user ints, 4 sinks; relatively small inference payoff
- OpenBSD kernel: found 12 bugs, 12 fixed
 - All bugs local
 - Four false positives
 - Zero inferred user ints, sinks
- No performance or labour metrics

Further Discussion

- Mainly an application paper, architecture and details are discussed elsewhere:
- Metal: *How to Write System-specific, Static Checkers in Metal*
- Metacompilation: *A System and Language for Building System-Specific, Static Analyses*
- Belief inference: *Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code*

Discussion: Pros

- Successful analysis tool requires flexible, easy specification method
 - Simple analysis is sufficient
- FSM: intuitive method for specifying programmer knowledge (obscure rules)
- metal: C-based = gentle learning curve, programmers will use it
- Most results verified by kernel developers
- Fair discussion of limitations (false pos/negs)

Discussion: Pros

- Lessons learned (useful for tool developers)
 - Belief inference isn't too beneficial (all but one inferred sink harmless)
 - Refining a technique → small gains
 - Severe bugs are as common as minor ones
 - Result filtering is essential
 - Bugs are mostly local (intraprocedural)
 - Less scalable/more precise methods still worth pursuing

Discussion: Cons

- Range checker only ensures that programmer thought about sanitization
 - No guarantee that the selected bounds were correct
 - EXE paper is able to make stronger guarantees
- State info. doesn't persist across code paths
 - EXE also doesn't have this problem
- Adjustment of rules to lower false positives
 - Not so easy to write after all?
- “ad hoc knowledge” exploitation worked well for Linux
 - Will it work so well for other projects?

Discussion: Cons

- Overly dismissive of runtime systems
 - Claims that OSs have too many code paths for testing
 - EXE automates the generation of these test cases
- Overly dismissive of type systems
 - Belief Inference type detection could probably exist in some form, despite opposite claim
 - Annotation process could be automated using templates that exploit project-specific knowledge
- No real comparison to model checking
 - Technique with similar aims and tradeoffs