Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code

Dawson Engle, David Yu Chen, Seth Hallem, Andy Chou, Benjamin Chelf

Miroslav Cupak

10/01/2012

Introduction

- systems obey rules for correctness and performance
- verification, testing, manual and automatic inspection
- problem with static analysis: what rules to check?
- solution: automatically extract from source

Techniques

- problem: finding what is incorrect without knowing what is correct
- tools: contradictions & common behaviour
- approach: internal consistency & statistical analysis
- implement checkers and apply them to Linux & OpenBSD

Terminology

- beliefs
    - MUST
    - MAY
- templates
- slots

Internal Consistency

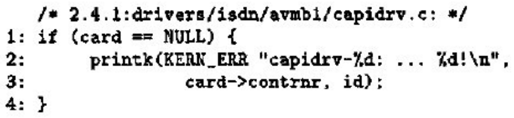
- checkers defined by:
    - the rule template T
    - the valid slot instances for T
    - the code actions implying beliefs
    - the rules for how beliefs combine & the rules for contradiction
    - the rules for belief propagation
- issues:
    - inferring beliefs (direct observation, pre/post-conditions)
    - relating code (implementation, abstraction)

Statistical Analysis

- internal consistency checkers with modifications:
    - assume all slot-instance combinations are MUST beliefs
    - indicate checks and failures
    - rank the errors and order the results to get most relevant results
- issues:
    - large set of cases
    - pre-processing
    - noise (large samples, ranking, human-level operations)
- extensible, makes use of empty templates
- latent specifications (naming conventions, error codes)

Checker: Null Pointer Consistency

- static null pointer detection
- based on internal consistency
- associates pointers with belief sets and flags contradictions
  - check-then-use
  - use-then-check
  - redundant checks



```
   /* 2.4.1:drivers/isdn/avmbi/capidrv.c: */
if (card == NULL) {
      printk(KERN_ERR "capidrv-%d: ... %d!\n",
                  card->contrnr, id);
}
```

Checker: Statistical Lock Inference

- detection of accesses to shared variables without their locks
- problem: finding variable-lock bindings
- test MAY beliefs, rank errors
- forward and backward propagation of locks

Checker: Security

- determine whether the pointer is a kernel (safe) or a user (tainted) pointer, report intersection
- based on dereference counts
- problems:
    - false positives problem due to kernel backdoors checking if they are called from user or kernel code
    - manual inspection

Checker: Failure Checking

- find routines that are not checked or are incorrectly checked for failures
    - routines returning null pointers are checked before use
    - unnecessary checking of routines that cannot fail

Checker: Temporal Rules

- no A after B (free memory)
- B must follow A (lock-unlock)
- important preprocessing of traces

Results

- finding bugs without special knowledge of the correctness of the program
- many interesting bugs
- new surprising bugs
- real bug reports and patches
- significant portion of false positives (bug/false positives ratio: null pointer 205/40, security 35/19)

Related work

- extension of Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions (Using Meta-level Compilation to Check FLASH Protocol Code)
- type systems
- specifications
- dynamic invariant inference

Questions?

What else can we check? Any ideas for other checkers or situations?

What else can we check?

- check permissions before writing to data structures
- reenable interrupts after disabling them
- size limit on variables
- hold read locks if variable is not modified
- memory allocation before its use
- protocol headers
- double if statements
- waiting on synchronous sends
- certain references not allowed in parts of code

How can we use this tool to prevent deadlocks?

How can we use this tool to prevent deadlocks?

- lock releasing after acquiring it
- kernel cannot call blocking functions with interrupts disabled
- thread holding spin lock cannot block
- temporal ordering

What do you see as the biggest problem of the proposed tools? How would you tackle it?

What do you see as the biggest problem of the proposed tools?

- setup cost
- manual inspection
- false positives (ranking, thresholds)
- performance

If you were a developer in a real SW company trying to extend their QE processes, would you consider adoption of this tool a good idea? What could make it more practical?

Do you consider the tool usable in real SW companies?

- still a lot of manual work, but it can help
- relatively easily extensible
- flexibility
- scalability
- + IDE integration
- + improve performance (storing history, running on parts of code, ignore certain paths)

Are the metrics in use objective?

Are the metrics in use objective?

- exhaustive all-paths search shifting the rating
- preprocesssing step
- heuristics
- relying on code convention