Presentation of

“Bugs as deviant behavior: A general approach to inferring errors in systems code”

Lucas Panjer

October 12, 2006

Problem

- Finding programming errors is difficult
- Defining the rules that describe programming errors is difficult

Solution

- Attempt to automatically find good programming error rules
- Detect flaws in belief sets
- Assume that the majority is correct and minority is likely to be incorrect
- Process code for rules, flag instances that don’t match the rules

Method

- Define a set of rule templates
- Parse code to find instances that fit the rules, developing rules dynamically
- Order the output based on relevance
- Evaluate the identified errors

Consistency

- Checker defined by
  1. Rule Template
  2. Valid slot instances
  3. Code actions that imply beliefs
  4. Rules for belief combination, contradiction
  5. Rules for belief propagation
  - Examples:
    - function <f> must be checked for failure
    - In context <x>, do <b> after <a>
- Develop a belief set as you work through a piece of code
- When you find a contradiction you mark it as an error

Statistical analysis

- Example:
  - <a> MAY be paired with <b>
- Observe a behaviour that happens frequently
- Mark as a possible error when it doesn't happen (with confidence rating)
- Filter results based on system specific rules

```
lock l;            // Lock
int a, b;          // Variables potentially
                   // protected by l
void foo() {
    lock(l);       // Enter critical section
    a = a + b;     // MAY: a,b protected by l
    unlock(l);     // Exit critical section
    b = b + 1;     // MUST: b not protected by l
}
void bar() {
    lock(l);
    a = a + 1;     // MAY: a protected by l
    unlock(l);
}
void baz() {
    a = a + 1;     // MAY: a protected by l
    unlock(l);
    b = b - 1;     // MUST: b not protected by l
    a = a / 5;     // MUST: a not protected by l
}
```

Implementation

- *metal* a high level state machine language for compiler extensions
- creates *xgcc* extensions
- Tested against OpenBSD, Linux

Usage

- Four case studies

Internal Null Consistency

Check-then-use
  - A pointer thought to be null is dereferenced
- Use-then-check
  - A pointer is dereferenced the checked to be null
- Redundant checks
  - A Pointer known to be (!)null checked to be (!)null

```
/* 2.4.1:drivers/isdn/avmb1/capidrv.c: */
if (card == NULL) {
        printk(KERN_ERR "capidrv-%d: ... %d!\n",
                card->contrnr, id);
}
```

Checker	Bug	False
check-then-use	79	26
use-then-check	102	4
redundant-checks	24	10

Table 3: Results of running the internal null checker on Linux 2.4.7.

Security Backdoors

- Looks for unsafe dereferencing of pointers in system code
- Need to define a significant number of routine and variable names to ignore to suppress false positives

OS	Errors	False	Applied
OpenBSD 2.8	18	3	1645
Linux 2.4.1	12 (3)	16 (1)	4905
Linux 2.3.99	5	n/a	n/a

Inferring Failure

- Looks for unchecked or incorrectly checked routine failures
- Count number of times the function was checked in a certain manner
- Count minority as a errors
- Rank

Version	Bug	False
2.4.1	52 + 102	16
OpenBSD	27 + 14	21
Total	195	37

Deriving Temporal Rules

- Freed memory should not be used
- If a function arg is not used after the call, programmer may believe it is deallocated
  - Check all function argument pairs where function contains a dealloc function “free”, “dealloc”, etc
  - Collect stats on number of times checked vs failed
- Linux kernel checking found 23 free errors 11 false positives

```
 /* fs/proc/generic.c:proc_symlink */
 ent->data = kmalloc(...);
 if (!ent->data) {
     kfree(ent);
     goto out;
 }
out:
 return ent;
```

Contributions

- Finds bugs without knowing the correctness rules of the system.
- Previous work manually specified rules to check against a system. This work improves by:
  - Templating the rules (Consistency)
  - Automatically finding rules (Statistical)
- Found lots of errors in real systems code. Resulted in many kernel patches.

Positive

- Lets the rules be highly targeted to the code in question
- Automation far superior to human code review for error cases
- Could translate easily into error checking in compilers
- Allow domain specific knowledge to be applied at a compiler level.
- Allows checking of non-runnable code (drivers)

Negative

- Must know what classes of errors to check
- Need to write compiler extensions
- Relatively high false positive rate
- Often need to add domain specific knowledge to suppress false positives and assist ranking algorithms
- Previous work has impacted current study