Software Testing, Quality Assurance and Maintenance Winter 2015

Lecture 13 — February 2, 2015

Patrick Lam *version 1*

We'll now shift to the topic of bug-finding. This is generally useful to know about, and is relevant to your course project.

A bug has got to be a violation of some specification. This might be a language-level specification (e.g. don't dereference null pointers), or it may be specific to an API that you are using.

For instance, you may have a method:

```
// callers must acquire the lock
static int reset_hardware(...) { ... }
```

and then a caller in the Linux kernel:

```
// linux/drivers/scsi/in2000.c:
static int in2000_bus_reset(...)
{
  ...
  // no lock acquisition => a bug!
  reset_hardware(...);
  ...
}
```

Examples of specifications:

- `len < 100`;
- `x = 16*y + 4*z + 3`;
- `fclose()` must be called after `fopen()`;
- and example from above: callers of `reset_hardware()` must acquire lock.

How do you get specifications? Well, programming languages come with some specifications applicable to all programs in that language. Or, you can have developers write specifications (an uphill battle). Or, you can automatically infer specifications from source code, execution traces, comments, etc. This includes static tools like Coverity, iComment, PR-Miner, etc. It also includes dynamic tools like Daikon, DIDUCE, etc.

More on Coverity. We talked a bit about Coverity last time. It is a commercial product which can find many bugs in large (millions of lines) programs; it is therefore a leading company in building bug detection tools. Clients (900+) include organizations such as Blackberry, Yahoo, Mozilla, MySQL, McAfee, ECI Telecom, Samsung, Siemens, Synopsys, NetApp, Akamai, etc. These include domains including EDA, storage, security, networking, government (NASA, JPL), embedded systems, business applications, operating systems, and open source software. We have access to Coverity for this course, but there's also a free trial:

`http://softwareintegrity.coverity.com/FreeTrialWebSite.html`

Mistaken beliefs. Coverity reported encounters with a number of mistaken beliefs about how languages work:

"No, the loop will go through once!"

```
1  for (i = 1; i < 0; i++) {
2    // ... this code is dead ...
3  }
```

"No, && is 'or'!"

```
1  void *foo(void *p, void *q) {
2    if (!p && !q)
3      return 0;
4  }
```

"No, ANSI lets you write 1 past end of the array!"

```
1  unsigned p[4]; p[4] = 1;
```

("We'll just have to agree to disagree.")

Goal. Coverity aims to find as many serious bugs as possible. The problem is: what's the definition of a bug? Is there objective truth? If there is, Coverity doesn't know it.

- Contradictions: It attempts to find lies by cross-examining; contradictions indicate errors.
- Deviance: It assumes programs are mostly-correct and tries to infer correct behaviour from that assumption. If 1 person does X, then maybe it's right, or maybe that was just a coincidence. But if 1000 people do X and 1 person does Y, the 1 person is probably wrong.

Crucially: a contradiction constitutes an error, even without knowing the correct belief.

MUST-beliefs versus MAY-beliefs. We differentiate between MUST-beliefs (related to contradictions) and MAY-beliefs (related to deviance).

MUST-beliefs are inferred from acts that imply beliefs about the code. For instance:

```
1  x = *p / z; // MUST: p not null
2              // MUST: z != 0
3  unlock(l); // MUST: l acquired
4  x++; // MUST: x not protected by l
```

MAY-beliefs, on the other hand, could be coincidental.

A(); // ... B();	A(); // ... B();	A(); // ... B();	A(); // ... B();	// MAY: A() and B() are paired.

We can check them as if they're MUST-beliefs and then rank errors by belief confidence (more on that later).

MUST-belief examples. Let's look first at a couple of MUST-beliefs about null pointers.

- If I write `*p` in a C program, I'm stating a MUST-belief that `p` had better not be `NULL`.
- If I write the check `p == NULL`, I'm implying two MUST-beliefs: 1) POST: `p` is `NULL` on true path, not-`NULL` on false path; 2) PRE: `p` was unknown before the check.

We can cross-check these for three different error types. I leave finding the actual error to you:

- check-then-use (79 errors, 26 false positives):

```
1  /* linux 2.4.1: drivers/isdn/svmb1/capidrv.c */
2  if (!card)
3    printk(KERN_ERR, "capidrv-%d: ...", card->contrnr...);
```

- use-then-check (102 errors, 4 false positives):

```
1  /* linux 2.4.7: drivers/char/mxser.c */
2  struct mxser_struct *info = tty->driver_data;
3  unsigned flags;
4  if (!tty || !info->xmit_buf)
5    return 0;
```

- contradictions/redundant checks (24 errors, 10 false positives:)

```
1  /* linux 2.4.7/drivers/video/tdfxfb.c */
2  fb_info.regbase_virt = ioremap_nocache(...);
3  if (!fb_info.regbase_virt)
4    return -ENXIO;
5  fb_info.bufbase_virt = ioremap_nocache(...);
6  /* [META: meant fb_info.bufbase_virt!] */
7  if (!fb_info.regbase_virt) {
8    iounmap(fb_info.regbase_virt);
```