Software Testing, Quality Assurance & Maintenance—Lecture 22

Patrick Lam

March 11, 2019

Last Time

MAY-beliefs versus MUST-beliefs

Cross-checking beliefs

Today

Inferring beliefs via statistics

Part I

Inferring beliefs

Redundancy Checking

Assumption: code ought to do something

Look for identity operations, e.g.

`x = x, 1 * y, x & x, x | x.`

```
/* 2.4.5-ac8/net/appletalk/aarp.c */
da.s_node = sa.s_node;
da.s_net = da.s_net;
```

Also look for unread writes:

```
for (entry=priv->lec_arp_tables[i];
     entry != NULL; entry=next) {
  next = entry->next; // never read!
  ...
}
```

Redundancy suggests conceptual confusion.

(examples courtesy Dawson Engler)

From MUST to MAY

Preceding examples were about MUST beliefs:
violations were clearly wrong.

Let's examine MAY beliefs next:

- need more evidence of wrongdoing.

Verifying MAY beliefs

1. Record every successful MAY-belief check as “check”.
2. Record every unsucessful belief check as “error”.
3. Rank errors based on “check” : “error” ratio.

Most likely errors: “check” is large, “error” small.

Let’s find some MAY beliefs

use-after-free:

```
free(p);
print(*p);
```

That is a MUST-belief.

However, other resources are freed by custom (undocumented) free functions.

Let’s derive them behaviourally.

Finding custom free functions

Key idea:
If pointer `p` not used after calling `foo(p)`,
then derive a MAY belief that `foo(p)` frees `p`.

Just assume all functions free all arguments.

- emit "check" at every call site;
- emit "error" at every use.

(in reality, filter functions with suggestive names).

Example: finding free functions

Putting that into practice,
we might observe:

foo(p)	foo(p)	foo(p)	bar(p)	bar(p)	bar(p)
*p = x;	*p = x;	*p = x;	p = 0;	p=0;	*p = x;

Rank `bar`'s error first.
Sample results: 23 free errors, 11 false positives.

More statistical techniques: nullness checks

Situation:

Want to know which routines may return `NULL`.

Possible solution: static analysis to find out.

Problems:

- difficult to know statically (“`return p->next;`”?)
- get false positives:
  functions return `NULL` under special cases only.

Applying a statistical technique to nullness checks

Instead: let's observe what the programmer does.
Again, rank errors based on checks vs non-checks.

Just assume **all** functions can return `NULL`.

- pointer checked before use: emit "check";
- pointer used before check: emit "error".

Example: finding NULL-returning functions

This time, we might observe:

p = bar(...);
*p = x;
	p = bar(...);
	if (!p) return;
	*p = x;
		p = bar(...);
		if (!p) return;
		*p = x;
			p = bar(...);
			if (!p) return;
			*p = x;

Sort errors based on “check”:“error” ratio.
Sample results: 152 free errors, 16 false positives.

General statistical technique

"a(); . . . b();" implies MAY-belief that a() followed by b().
(is it real or fantasy? we don't know!)

Algorithm:

- assume every `a`–`b` is a valid pair;
- emit "check" for each path with "a()" and then "b()";
- emit "error" for each path with "a()" and no "b()".

(actually, prefilter functions that look paired).

Example: general technique

Consider:

foo(p, ...); bar(p, ...); // check	foo(p, ...); bar(p, ...); // check	foo(p, ...); // error: foo, no bar!

Application: course project

```
void scope1() {
 A(); B(); C(); D();
}

void scope2() {
 A(); C(); D();
}

void scope3() {
 A(); B();
}

void scope4() {
 B(); D(); scope1();
}

void scope5() {
 B(); D(); A();
}

void scope6() {
 B(); D();
}
```

"A() and B() must be paired":
either A() then B() or B() then A().

Support = # times a pair of functions appears together.

support({A,B})=3

Confidence({A,B},{A}) = support({A,B})/support({A}) = 3/4

Application: course project

```
void scope1() {
 A(); B(); C(); D();
}

void scope2() {
 A(); C(); D();
}

void scope3() {
 A(); B();
}

void scope4() {
 B(); D(); scope1();
}

void scope5() {
 B(); D(); A();
}

void scope6() {
 B(); D();
}
```

Sample output for support threshold 3, confidence threshold 65% (intra-procedural analysis):

- bug:A in scope2, pair: (A B), support: 3, confidence: 75.00%
- bug:A in scope3, pair: (A D), support: 3, confidence: 75.00%
- bug:B in scope3, pair: (B D), support: 4, confidence: 80.00%
- bug:D in scope2, pair: (B D), support: 4, confidence: 80.00%

Why are we doing this again?

```
/* 2.4.0:drivers/sound/cmpci.c:cm_midi_release: */
lock_kernel(); // [PL: GRAB THE LOCK]
if (file->f_mode & FMODE_WRITE) {
  add_wait_queue(&s->midi.owait, &wait);
  ...
  if (file->f_flags & O_NONBLOCK) {
    remove_wait_queue(&s->midi.owait, &wait);
    set_current_state(TASK_RUNNING);
    return -EBUSY; // [PL: OH NOES!!1]
  }
  ...
}
unlock_kernel();
```

Problem: lock() and unlock() must be paired!

Summary: Belief Analysis

We don't know what the right spec is.
Instead, look for contradictions.

MUST-beliefs: contradictions = errors!

MAY-beliefs: pretend they're MUST, rank by confidence.

(Key assumption: most of the code is correct.)

Further references

Dawson R. Engler, David Yu Chen, Seth Hallem, Andy Chou and Benjamin Chelf.
"Bugs as Deviant Behaviors: A general approach to inferring errors in systems code".
In SOSP '01.

Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem.
"Checking system rules using system-specific, programmer-written compiler extensions".
In OSDI '00 (best paper).
`www.stanford.edu/~engler/mc-osdi.pdf`

Junfeng Yang, Can Sar and Dawson Engler.
"eXplode: a Lightweight, General system for Finding Serious Storage System Errors".
In OSDI'06.
`www.stanford.edu/~engler/explode-osdi06.pdf`