Software Testing, Quality Assurance and Maintenance	Winter 2015

Lecture 14 — February 4, 2015

Patrick Lam *version 1*

Recall that we've been discussing beliefs. Here are a couple of beliefs that are worthwhile to check. (examples courtesy Dawson Engler.)

Redundancy Checking. 1) Code ought to do something. So, when you have code that doesn't do anything, that's suspicious. Look for identity operations, e.g.

$$\texttt{x} = \texttt{x},\ \texttt{1} * \texttt{y},\ \texttt{x\&x},\ \texttt{x}|\texttt{x}.$$

Or, a longer example:

```
/* 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.

So far, we've talked about MUST-beliefs; violations are clearly wrong (in some sense). Let's examine MAY beliefs next. For such beliefs, we need more evidence to convict the program.

Process for verifying MAY beliefs. We proceed as follows:

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 occur when "check" is large, "error" small.

Example. One example of a belief is use-after-free:

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

That particular case is a MUST-belief. However, other resources are freed by custom (undocumented) free functions. It's hard to get a list of what is a free function and what isn't. So, let's derive them behaviourally.

Inferring beliefs: finding custom free functions. The key idea is: if pointer `p` is not used after calling `foo(p)`, then derive a MAY belief that `foo(p)` frees `p`.

OK, so which functions are free functions? Well, 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).

Putting that into practice, we might observe:

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

We would then rank `bar`'s error first. Plausible results might be: 23 free errors, 11 false positives.

Inferring beliefs: finding routines that may return `NULL`. The situation: we want to know which routines may return `NULL`. Can we use static analysis to find out?

- sadly, this is difficult to know statically ("`return p->next;`"?) and,
- we get false positives: some functions return `NULL` under special cases only.

Instead, let's observe what the programmer does. Again, rank errors based on checks vs non-checks. As a first approximation, assume **all** functions can return `NULL`.

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

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;

Again, sort errors based on the "check":"error" ratio.

Plausible results: 152 free errors, 16 false positives.

General statistical technique

When we write "a(); ... b();", we mean a MAY-belief that a() is followed by b(). We don't actually know that this is a valid belief. It's a hypothesis, and we'll try it out. 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).

Consider:

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

This applies to the course project as well.

```
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

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%

The point is to find examples like the one from `cmpci.c` where there's a `lock_kernel()` call, but, on an exceptional path, no `unlock_kernel()` call.

Summary: Belief Analysis. We don't know what the right spec is. So, look for contradictions.

- MUST-beliefs: contradictions = errors!
- MAY-beliefs: pretend they're MUST, rank by confidence.

(A key assumption behind this belief analysis technique: 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`

Using Linters

We will also talk about linters in this lecture, based on Jamie Wong's blog post `jamie-wong.com/2015/02/02/linters-as-invariants/`.

First there was C. In statically-typed languages, like C,

```c
#include <stdio.h>

int main() {
  printf("%d\n", num);
  return 0;
}
```

the compiler saves you from yourself. The guaranteed invariant:

"if code compiles, all symbols resolve."

Less-nice languages. OK, so you try to run that in JavaScript and it crashes right away. Invariant?

"if code runs, all symbols resolve?"

But what about this:

```javascript
function main(x) {
  if (x) {
    console.log("Yay");
  } else {
    console.log(num);
  }
}

main(true);
```

Nope! The above invariant doesn't work.

OK, what about this invariant:

> "if code runs without crashing, all symbols referenced in the code path executed resolve?"

Nope!

```
function main() {
  try {
    console.log(num);
  } catch (err) {
    console.log("nothing to see here");
  }
}

main();
```

So, when you're working in JavaScript and maintaining old code, you always have to deduce:

- is this variable defined?
- is this variable always defined?
- do I need to load a script to define that variable?

We have computers. They're powerful. Why is this the developer's problem?!

```
1 function main(x) {
2   if (x) {
3     console.log("Yay");
4   } else {
5     console.log(num);
6   }
7 }
8
9 main(true);
```

Now:

```
$ nodejs /usr/local/lib/node_modules/jshint/bin/jshint --config jshintrc foo.js
foo.js: line 5, col 17, 'num' is not defined.

1 error
```

Invariant:

> "If code passes JSHint, all top-level symbols resolve."

Strengthening the Invariant. Can we do better? How about adding a pre-commit hook?

"If code is checked-in and commit hook ran,
all top-level symbols resolve."

Of course, sometimes the commit hook didn't run. Better yet:

- Block deploys on test failures.

Better invariant.

"If code is deployed,
all top-level symbols resolve."

Even better yet. It is hard to tell whether code is deployed or not. Use git feature branches, merge when deployed.

"If code is in master,
all top-level symbols resolve."