Software Testing, Quality Assurance and Maintenance Winter 2017

Lecture 33 — March 27, 2017

Patrick Lam version 1

Facebook Infer: Deeper, but still generic, properties

We'll see one more static tool, Facebook Infer. This tool is particularly important because it is both open-source and designed to work on industrial-sized codebases in a number of languages including C, Objective-C, C++, and Java (including Android code). Facebook uses Infer on its code.

Infer's open-source nature enables others to write their own analyses, both generic and specific to particular libraries. Infer was designed to be easily extensible and applicable to multiple languages.

Try Infer in-browser (Java) here:

`https://codeboard.io/projects/11587?view=2.1-21.0-22.0`

You can find some talks about Infer here:

`https://code.facebook.com/projects/449931035189038/infer/`

Infer has linters (like PMD and FindBugs) but also more sophisticated interprocedural analyses which use separation logic (beyond the scope of this course).

Infer Eradicate: detecting null pointer dereferences. The goal of this subtool is to prevent null pointer exceptions. It does so by forcing treating Java references as not-null by default—that is, the tool ensures that the developer may not put `null` into an unannotated reference. It turns out, however, that it's quite difficult to program without `null`. So developers can mark a reference as `@Nullable`. For instance, Infer Eradicate complains that `s` might be null at the call to `length()`.

```
class C {
  int getLength(@Nullable String s) {
    return s.length();
  }
}
```

Infer can use its flow-sensitive static analysis to guarantee that there are no null dereferences here:

```
class C {
  int getLength(@Nullable String s) {
    if (s != null) { return s.length(); } else { return -1; }
  }
}
```

Eradicate checks a generic property—`null` may never be dereferenced—but uses annotations to help with the analysis.

Although the above examples are intraprocedural, Infer also knows enough to catch interprocedural errors, like this one:

```
struct Person {
  int age; // ... etc.
};
int get_age(struct Person *who) {
  return who->age;
}
int null_pointer_interproc() {
  struct Person *joe = 0;
  return get_age(joe);
}
```

Infer Analyzer: Leaks. Infer also statically detects both resource leaks in C and Java as well as memory leaks in languages like C and C++. Resources include files and sockets—entities that should generally be closed after they have been opened. For instance, Infer will complain about the following code:

```
void resource_leak_bug() {
    FILE *fp;
    fp=fopen("test.txt", "r"); // file opened and not closed.
}
```

That example is fairly straightforward for both dynamic and static approaches to find. But consider also this code:

```
  public static void foo () throws IOException {
    FileOutputStream fos = new FileOutputStream(new File("whatever.txt"));
    fos.write(7);   //DOH! What if exception?
    fos.close();
  }
```

It could happen that `fos.write()` might throw an exception (which is caught in a caller), thus leaving `fos` open. Such a leak is difficult to catch dynamically. On the other hand, static analyzers generally give you all possible leaks, including zillions that never actually happen.

By the way, you can fix the above bug in Java 7 by using try-with-resources:

```
  public static void foo() throws IOException {
    try (
      FileOutputStream fos = new FileOutputStream(new File("everwhat.txt"))
    ) {
      fos.write(7);
    }
  }
```

Note that try-without-resources works when the lifetime of the resource can be statically bounded to a lexical scope (in the above example, the single `write`) statement. Otherwise you might consider using `try`/`finally`.

Infer and tainting. Infer also supposedly contains an analysis to check for some security and privacy issues, although details are scarce. The general idea is to use a taint-based analysis. Such analyses label certain values as unsafe (e.g. data read from an untrusted source) or secret (e.g. preferences data). The analysis complains when tainted values may reach sensitive functions or the outside world.

Static versus Dynamic Analysis

We saw that Infer could detect memory and resource leaks statically. This is a good time to compare static and dynamic analyses; `valgrind` detects memory leaks dynamically. We'll do the comparison for JML and `jmlc` versus `ESC/Java` as well.

Recall that a dynamic analysis monitors program behaviour at runtime, whlie a static analysis reasons about the program text. We talked about this back in Lecture 4, but we're ready to provide more technical details now that we know more about testing and static analysis.

- dynamically: You have complete information about program state on observed executions.
- statically: You have partial information about all executions.

So how does that work out on specific examples?

Virtual method call resolution. The question here is: given a virtual method call like `m.foo()`, where the actual runtime type of `m` could vary (due to subclassing), which `foo()` method actually gets called?

Dynamically, it is trivial to answer this question. (How?)

Statically, this is quite difficult. The problem is that one needs to know the type of `m`. The easiest answer is by using Class Hierarchy Analysis: using the declared type of `m`, compute the allowed types using the class hierarchy (i.e. subclasses of the declared type). Rapid Type Analysis gives a better answer—it limits the answer to all types that are instantiated somewhere in the program. But that requires an approximation of the reachable code, which in turn requires the answer to the very question we're trying to answer. There are even more accurate algorithms which propagate type constraints through the program.

Checking data structures for cycles. Last time, we saw code to ensure that the tree was actually acyclic. That was fairly straightforward. Here again, static checks are difficult; they require the analysis to verify that the code maintains the acyclicity invariant through all possible executions.

Unreachable code. Not everything is easier to verify dynamically than statically. Consider the question of whether a line of code is reachable or not. This is relatively easy to approximate statically (that's what dead code elimination does), but quite hard to check dynamically, since it depends heavily on finding appropriate program inputs.