Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 32 — April 3, 2019

Patrick Lam *version 1*

Dynamic Analyses

We'll continue talking about generic properties, but this time we'll talk about dynamic verification of these properties.

Memory errors

Let's start by talking about leaks and other memory errors. We saw static detection of leaks with Facebook Infer. Valgrind's Memcheck tool and Clang's Address Sanitizer detect memory errors dynamically.

This webpage compares different tools:

`https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools`

Valgrind's Memcheck detects the following errors:

- Illegal reads/writes: Memcheck complains about accesses to memory that the program should not be accessing (e.g. not returned from a `malloc`, or below the stack pointer).
- Reads of uninitialized variables: Memcheck tells you about reads of memory that has never been initialized, if the program prints out the resulting values.
- Illegal/wrong frees: Of course, you're not allowed to free memory that you didn't get from a memory allocation function, so Memcheck tells you. It also tells you when you use `free()` on something you got from `new[]`.
- Overlapping source/destination for memory moves.
- Fishy argument values: You probably don't mean to request either -3 bytes or more than 2^{63} bytes.
- Memory leaks: Memcheck tells you when you have memory that didn't get freed but that you no longer have any pointers to.

You will pay a significant performance penalty when using Valgrind; Memcheck typically comes with a 10×–50× slowdown. This is usable for bug diagnosis but obviously not for production.

Here is more information on what Valgrind can do. It can do more than just Memcheck as well.

`http://maintainablecode.logdown.com/posts/245425-valgrind-is-not-a-leak-checker`

The AddressSanitizer tool, supported by both clang and gcc, also detects memory errors. These errors are similar to those that Valgrind can detect. The technology is different, but the goals are similar. Because it uses different implementation technology, it runs much more quickly than Valgrind, with a reported typical slowdown of 2×.

Find some information about AddressSanitizer here:

`http://btorpey.github.io/blog/2014/03/27/using-clangs-address-sanitizer/`

AddressSanitizer finds the following errors:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
- Use-after-scope
- Double-free, invalid free
- Memory leaks (experimental)

Note that Valgrind may return false positives, while AddressSanitizer has a design goal of returning no false positives. In fact, ASan aborts the program whenever it encounters a memory problem. You're supposed to fix it before proceeding. Of course, ASan might miss some problems.

Implementation Techniques. Valgrind and ASan use different techniques, hence yield different results. Valgrind emulates a CPU and checks, at every memory access, whether that access is legitimate or not. ASan, on the other hand, rewrites relevant memory accesses at compile-time to call a checking library. It turns out that calling a library is cheaper than emulating the CPU.

Conceptually, ASan maintains shadow memory— metadata about where the program is allowed to access (or not). Then, it replaces the `malloc()` and `free()` calls with its own versions; these versions update shadow memory and indicate that allocated memory is OK to access, unallocated memory not OK. Finally, every memory access in the program gets replaced with a checked access:

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;  // or: ... = *address;
```

This is not so different from what Java does to check array accesses, for instance, but applies much more generally, to every memory access in the program.

More details about ASan:

`https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm`

Both of these tools are open-source. The source code is the ultimate description of the tools.

Race detectors

Valgrind also has a race detection tool, Helgrind. This tool dynamically detects memory that is concurrently accessed by two threads. Such accesses are fine as long as they are controlled by a lock. At runtime, Helgrind knows which locks are held. If the program does not hold enough locks, then Helgrind flags a memory error.

```
acquire(lock1);                 acquire(lock2);
read(x);                        write(x); // different lock!
release(lock1);                 release(lock2);
```