Detecting Incorrect Build Rules

Nandor Licker and Andrew Rice

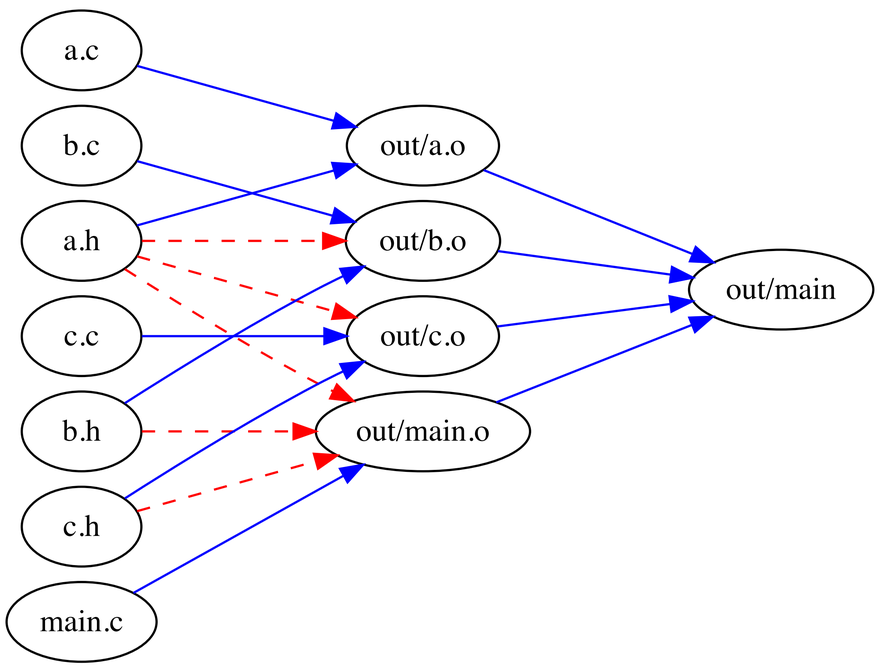
University of Cambridge, United Kingdom

Problem #1: Missing Dependencies

Makefiles encode the dependencies between the files an the modules of a project in order to enable build systems to run parallel and incremental builds. Unfortunately, the definitions are not always in sync with the dependencies actually involved between the files compiled or generated by the build tool. Consider the following GNU Makefile, which is broken:

```
out/a.o: a.c a.h
  gcc -c -o out/a.o a.c

out/b.o: b.c b.h a.h c.h
  gcc -c -o out/b.o b.c

out/c.o: c.c c.h
  gcc -c -o out/c.o c.c

out/main.o: main.c
  gcc -c -o out/main.o main.c

out/main: out/main.o out/a.o out/b.o out/c.o
  gcc -o out/main $^
```

The issue is not obvious unless we identify and overlay the missing dependencies from the object files onto the dependency graph with the defined dependencies from the Makefile, as shown below. When `c.h` changes, `main.o` will not be rebuilt incrementally since the build system is unaware of the dependency and the final executable will be built from stale objects.

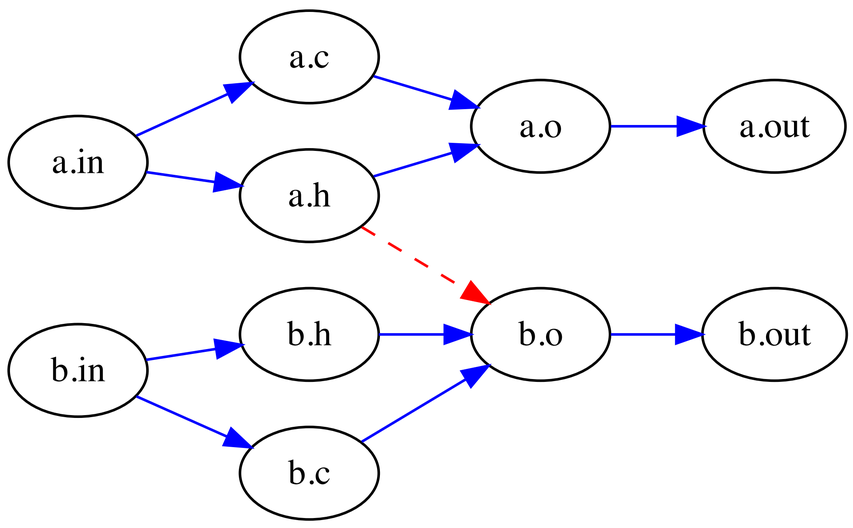


Problem #2: Race Conditions

The dependency graphs can also lack internal edges, typically introduced by code generators such as `lex` or `thrift`. The following is a typical example of a Makefile written by developers to invoke a C++ code generator and compile the outputs:

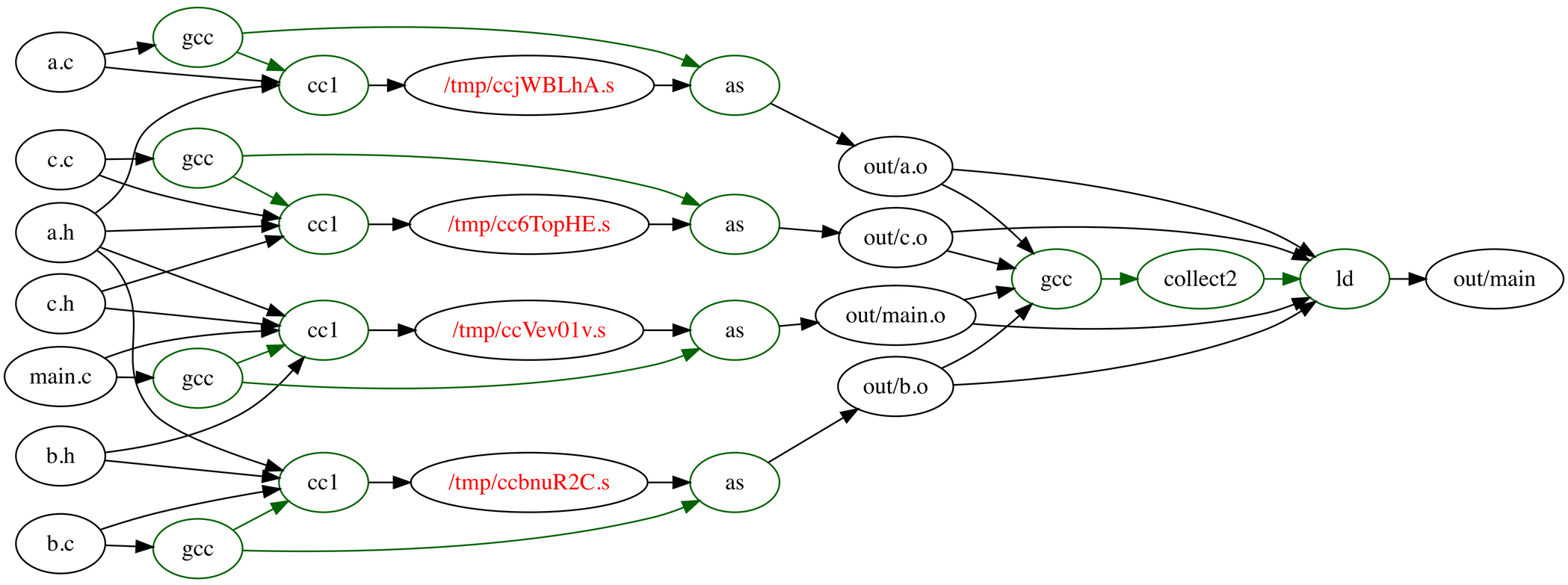
```
a.h : a.in
  generate a.in a.h a.cpp

a.o : a.h a.cpp
  gcc -c -o a.o a.cpp

b.h : b.in
  generate b.in b.h b.cpp

b.o : b.h b.cpp
  gcc -c -o b.o b.cpp
```

Even though `a.in` is not read while `b.in` is processed, the generated `b.h` depends on the generated `a.h`. As shown in the following dependency graph, the process compiling `b.o` can be invoked before the process generating `a.h` is triggered since the build system is unaware of the missing constraint between the two files, causing the build to fail spuriously.



These issues are hard to identify and can potentially cause major inconvenience, as they randomly occur in non-deterministic parallel builds or when the build system schedules files in an unexpected order caused by an unstable topological sorting method. Instead of actually fixing the problems, developers are tempted to re-run builds until they finally succeed.

Solution: Build Fuzzing



Our method automatically recovers dependency graphs by tracing the system calls executed by the processes launched during a correct, single-threaded, clean build. This results in a graph containing **files** and processes reading to or writing from them (including temporaries not persisted after the build). In turn, all files involved in the build are touched and an incremental build is triggered to identify the set of dependants known to the build tool. The edges between nodes in the transitive closure of the input in the inferred graph and nodes which were regenerated by the build tool are considered to be missing.

After identifying missing edges, our tool outputs diagnostic messages identifying files and processes involved in potentially incorrect build rules. Relying on topological sorting, we identify the earliest time a file can be generated, assuming a linear order: if it is before any of the dependencies, we warn for a potential race condition.

Try it out, it's all open-source and available on GitHub:

`https://github.com/nandor/mkcheck`