Detecting Incorrect Build Rules

Introduction

The full implementation of our method is available on GitHub at https://github.com/nandor/mkcheck, released under the MIT License. The automated evaluation script used to obtain the numbers reported in the paper is available at https://github.com/nandor/mkcheck-eval, released under the same license. We provide an artifact to easily reproduce all the published results. The projects are also available under the following DOIs:

- mkcheck: https://doi.org/10.5281/zenodo.2576574
- mkcheck-eval: https://doi.org/10.5281/zenodo.2576787

The artifact is provided as a VirtualBox appliance, running Ubuntu. The image includes the source code of the project, the evaluation script and all the dependencies required by the projects we evaluated. To run the evaluation script, a host machine with at least 15Gb of free space and network access is required.

More information about the tools is available in the `README.md` of their repositories.

Getting Started

The artifact is provided as a VirtualBox appliance. We tested it using VirtualBox version 5.2.12 r122591 on macOS High Sierra (10.13.6) but other modern versions of VirtualBox on other platforms should also work, as long as Hyper-V is disabled. A single user `nand` with no password has been created. To boot the machine:

1. **Install VirtualBox** by following the instructions at https://www.virtualbox.org/wiki/Downloads to install a version of VirtualBox for your operating system.
2. **Import the appliance** using the option from the File menu in VirtualBox
3. **Start the VM** once it has finished importing

After booting, a shell should be available, pointed to the home directory of the `nand` user, containing the source code of `mkcheck` and the automated evaluation script.

Building and Running the Tool

The `mkcheck` executable should already be available at `~/mkcheck/build/mkcheck`. To rebuild it, follow these instructions, which are also available in the `README.md` of the project.

1. Remove the old build directory: `rm -f ~/mkcheck/build`
2. Create a build directory: `mkdir ~/mkcheck/build`
3. Enter the directory: `cd ~/mkcheck/build`

4. Configure the project: `cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_COMPILER=clang++`
5. Build the tool: `make`

The fuzz testing script can be invoked as `python ~/mkcheck/tools/fuzz_test [command]`. The script must be invoked in the directory of the project being tested and it is fully documented in the README.md of the `mkcheck` project.

The script supports a one of the following commands:

- **fuzz:** perform fuzz testing on the project. If only a few files are to be tested, their paths can be passed as optional arguments. If no files are specified, all unfiltered files are considered
- **list:** list the files which are considered by default
- **query:** *query* query the dependencies of a file from the graph
- **build:** run a build and generate the graph, overwriting it
- **race:** perform race testing, same arguments as the fuzz command

We next provide two small examples that use the script and explain how to fuzz a project.

Running Small Tests

The project includes two tests, one to verify if simple missing dependencies are detecting and one to test for race conditions. Both tests rely on GNU Make and invoke `gcc`. To run the tests:

1. `make -C ~/mkcheck/test/make test`

   Expected output:

```
[1/12] /home/nand/mkcheck/test/make/a.c:
[2/12] /home/nand/mkcheck/test/make/a.h:
   -  /home/nand/mkcheck/test/make/out/c.o (x86_64-linux-gnu-as)
   -  /home/nand/mkcheck/test/make/out/main.o (x86_64-linux-gnu-as)
[3/12] /home/nand/mkcheck/test/make/b.c:
[4/12] /home/nand/mkcheck/test/make/b.h:
   -  /home/nand/mkcheck/test/make/out/main.o (x86_64-linux-gnu-as)
[5/12] /home/nand/mkcheck/test/make/c.c:
[6/12] /home/nand/mkcheck/test/make/c.h:
   +  /home/nand/mkcheck/test/make/out/b.o (x86_64-linux-gnu-as)
   -  /home/nand/mkcheck/test/make/out/main.o (x86_64-linux-gnu-as)
[7/12] /home/nand/mkcheck/test/make/d.h:
[8/12] /home/nand/mkcheck/test/make/lib_a/lib_a.c:
[9/12] /home/nand/mkcheck/test/make/lib_a/lib_a.h:
   -  /home/nand/mkcheck/test/make/out/main.o (x86_64-linux-gnu-as)
[10/12] /home/nand/mkcheck/test/make/lib_b/lib_b.c:
[11/12] /home/nand/mkcheck/test/make/lib_b/lib_b.h:
   -  /home/nand/mkcheck/test/make/out/a.o (x86_64-linux-gnu-as)
   -  /home/nand/mkcheck/test/make/out/main.o (x86_64-linux-gnu-as)
[12/12] /home/nand/mkcheck/test/make/main.c:
```

The 'test' target of the Makefile triggers the building of the dependency graph and then runs the fuzz testing script. In this project, each object file has a single `.c` and `.h` file as a dependency, with matching names, however the `.c` files include additional headers. Fuzzing detects these missing dependencies. In the Makefile, there is also a dependency on `c.h` for `b.o`, but `b.c` does not include `c.h` and the dependency is identified as redundant. Files which were unnecessarily rebuilt are prefixed with '+', whereas files which were expected to be regenerated, but were not touched, are prefixed with '-'.

2. `make -C ~/mkcheck/test/parallel test`

   Expected output:

```
[1/4] /home/nand/mkcheck/test/parallel/a.in:
   -  /home/nand/mkcheck/test/parallel/out/c.o (x86_64-linux-gnu-as)
[2/4] /home/nand/mkcheck/test/parallel/b.in:
   -  /home/nand/mkcheck/test/parallel/out/c.o (x86_64-linux-gnu-as)
[3/4] /home/nand/mkcheck/test/parallel/c.in:
[4/4] /home/nand/mkcheck/test/parallel/main.cpp:
Race testing parallel
Races:
/home/nand/mkcheck/test/parallel/out/c.o x86_64-linux-gnu-as
Missing edges:
/home/nand/mkcheck/test/parallel/out/a.h ->
/home/nand/mkcheck/test/parallel/out/c.o
/home/nand/mkcheck/test/parallel/out/b.h ->
/home/nand/mkcheck/test/parallel/out/c.o
```

   In this example, internal edges are missing between the generated headers and the generated sources, which originate from the `.in` files. Since these edges are missing, the build older or `c.o` is not constrained and the build system can schedule `c.o` to be built before the headers on which it depends are generated.

Evaluating a Custom Project

To evaluate a different project, the dependency graph must be inferred using `mkcheck` and the project must be fuzzed using the python fuzzing script. Detailed instructions are available in the `README.md` of the mkcheck repository. For example, the parallel test included in the project is evaluated using the following sequence of commands, located in the Makefile of the project at `~/mkcheck/test/parallel/Makefile`.

1. `make clean`
   Cleans the project - `mkcheck` should trace a clean build after it was configured.
2. `../../build/mkcheck -o /tmp/graph -- make j1`
   Trace the clean build, executed using `make -j1`. Due to potential race conditions, projects should be fuzzed with parallelism disabled. The tools does support forking processes though, so tracing can be accelerated if race conditions are not an issue.
3. `python ../../tools/fuzz_test --graph-path=/tmp/graph fuzz`
   Fuzz testing - as the tool runs, it displays the file being tested and the missing redundant dependencies of that file: files which were not rebuilt are introduced with '-' in the output,

whereas files which should have been rebuilt are introduced by a '+'.

4. `python ../../tools/fuzz_test --graph-path=/tmp/graph race`
   Race testing - displays a list of files which might be built out of order, along with the edges that are missing which might cause those files to be built out of order.

Reproducing Results (4-6 hours)

Our results are reproducible through the `eval.py` script, which automatically downloads open-source projects and runs our analysis on them, outputting the results. More information about this script can be found in the `README.md` file of the mkcheck-eval project (or `~/README.txt` in the VM). To run the script, execute the following commands:

```
1. cd ~/eval
2. ./eval.py
```

Results should be available in 4-6 hours, displayed in the terminal. Running the script again will not re-evaluate projects, but will display the results obtained previously. To re-run evaluation, `~/eval/tmp` must be deleted first. The script accepts an optional project name, in which case it only evaluates that project. The expected output is:

Name of project	*#files*	*Missing*	*Redundant*	*Races*	*Fixed*
linux	*27*	*6*	*1*	*True*	*False*
redis	*545*	*87*	*4*	*True*	*False*
tcc	*26*	*6*	*1*	*True*	*False*
namespaced_parser	*4*	*2*	*0*	*True*	*True*
cboy	*67*	*42*	*0*	*True*	*False*
tinyvm	*21*	*7*	*0*	*False*	*True*
x86-thing	*52*	*20*	*52*	*False*	*False*
CacheSimulator	*7*	*2*	*0*	*False*	*True*
lec	*74*	*1*	*0*	*False*	*True*
Generic-C-Project	*3*	*1*	*0*	*False*	*False*
hindsight-is-8080	*19*	*8*	*0*	*False*	*False*
reon	*15*	*1*	*2*	*False*	*False*
apron	*190*	*147*	*69*	*True*	*False*
mysql-server	*21*	*1*	*20*	*False*	*False*
anbox	*24*	*9*	*23*	*False*	*False*
PiFox	*41*	*23*	*0*	*True*	*False*
gr-ieee802-11	*7*	*2*	*0*	*False*	*True*
grappa	*89*	*0*	*9*	*False*	*False*
automate	*86*	*23*	*0*	*True*	*False*
ALang	*3*	*2*	*0*	*False*	*True*
specfem3d_geotech	*25*	*20*	*0*	*True*	*False*
tiny3Dloader	*26*	*20*	*6*	*False*	*False*
decaf	*2*	*2*	*0*	*False*	*False*
sppl	*5*	*0*	*1*	*False*	*False*
Pixslam	*20*	*4*	*2*	*False*	*True*
tetris	*6*	*2*	*0*	*False*	*True*
libcalrom	*4*	*2*	*0*	*False*	*True*
FreeNOS	*54*	*54*	*54*	*False*	*False*
baresifter	*27*	*0*	*12*	*False*	*False*

steppinrazor	*15*	*13*	*0*	*False*	*False*
nonpareil	*18*	*1*	*0*	*False*	*False*
fsp	*40*	*40*	*40*	*False*	*False*

The results are analysed in more detail in the paper itself. Some builds are not fully deterministic, so a slight deviation in the number of files is acceptable.