## Contents

Build EqMiner ... 1
= Dependencies ... 1
= Build ... 2
== Build cil, and our code chopper/wrapper ... 2
== Build the common library code for running code trunks ... 2
== Note for cil 1.3.6 and ocaml 3.09 ... 2
Execute EqMiner ... 3
= Run the code chopper and trunk wrapper ... 3
Basic use for a single .c file: ... 3
Use together with configure/make for a project ... 3
= Code Trunk Compilation & Running & Clustering ... 4
Basic steps: ... 5
Brief notes about some of the scripts: ... 6
Brief notes about "infraplacement": ... 7

Author: Lingxiao Jiang[1,2] and Zhendong Su[1]
1: University of California, Davis (UC Davis)
2: School of Information Systems, Singapore Management University (SIS/SMU)

github repo: https://github.com/skyhover/dyclone

# Build EqMiner

based on cil 1.3.6, disabled the generation of .o files, but enabled with code chopping

## = Dependencies

Ocaml 3.09
perl
autoconf
automake
make
ncurses-dev
str -> ocamlstr (cf. http://caml.inria.fr/mantis/view.php?id=5247)
if error message says "cannot foudn -lstr". need to make a symlink:

cf.
http://church.cs.virginia.edu/genprog/index.php/FAQ#I_get_the_error:_cil_file_.22is_not_a_compiled_interface.22._What_do_I_do.3F

# = Build

## == Build cil, and our code chopper/wrapper

```
cd cil
./configure EXTRASRCDIRS=../modules EXTRAFEATURES="cfgbfs cfgdfs codeinfo funchopper stmtiter"
Make
```

Our code is most in modules/. The funchopper.ml is the entry point for the code chopper and wrapper.
We also made some changes to cil/ to fix a couple of bugs, disable certain conflicting functionality, and enable the code chopper and wrapper.

## == Build the common library code for running code trunks

```
cd modules/C/
./build.sh
```

## == Note for cil 1.3.6 and ocaml 3.09

CIL 1.3.6 does not appear to build under the most recent version of OCaml (>3.10).
May be able to use an older version of OCaml as a workaround.
cf. https://code.google.com/p/crest/issues/detail?id=3

So, before the above configure&make, need to install Ocaml 3.09 on your system.
Tried building on the following systems 2014/7:

- Cygwin-64 on windows
    - ……
    - Failed to build ocaml 3.09; various errors; much work to make ocaml 3.09 be compatible with Cygwin-64.
- Ubuntu 12.04:
    - Install opam 1.1.1
        - sudo add-apt-repository ppa:avsm/ppa
        - sudo apt-get update
        - sudo apt-get install opam
        - opam init
    - Add the switch to 3.09.3 into opam's compiler switches, and switch to it:
        - cp -r docs/opam/3.09 ~/.opam/compilers/
        - opam switch 3.09
        - eval `opam config env`
    - then continue with the above configure&make

# Execute EqMiner

## = Run the code chopper and trunk wrapper

This version of modified cil doesn't produce .o files from .c files; it only chops (if any) the functions in the .c files and generate many other .c files containing the code trunks.

### == Basic use for a single .c file:

--- use cil as a replacement for gcc to compile a C program with the "funchopper" feature enabled with some parameters. must use --domakeCFG (from cil together with "funchopper"); optionally use --save-temps (tell cil to save intermediate files. c.f. cil manual)
E.g.:

```
cil/bin/cilly --save-temps --domakeCFG --dofunchopper --store-directory=trunks/ --min-stmt-number=1 --stmt-stride=1 --compilable --deckard-vector <filename>.c
```

There will be folders (one for each function names in the .c file) created in "store-directory".
Make sure "store-directory" folder exist; and empty it before hand to avoid some error messages reporting duplicated file names.
Each code trunk has these following files generated:

- <functionname>-ids-foo.c: this is the code trunk itself made compilable
- <functionname>-ids-foo.ins: a text file containing liveness info, used later in clustering scripts
- <functionname>-ids-foo.rds: a text file containing reaching definition info, used later in clustering scripts
- <functionname>-ids-foo.vec: a vector generated according to Deckard's algorithm, for comparing with syntactic clones later.
  - Each code trunk is wrapped in the function named __dyc_foo.
- <functionname>-ids-gen.c: this is "main" file that generates random inputs for the corresponding .foo.c code trunk.
  - Invocation of the code trunks is by another common main file in modules/C/dycmain.c, and based on file-specific linking

### == Use together with configure/make for a project

For projects using "configure && make", the basic idea is to first make each .c file in the project individually compilable (e.g., preprocess it to remove all includes, macros, etc.), and then apply the code chopper on each of them.
Can try the following several ways:

1. can define CC/LD to replace gcc during make. This is ok if project's configure/make process doesn't reply on generating actual .o files.
E.g.:

```
./configure
```

```
make CC=”bin/cilly” CFLAGS+=”--save-temps --domakeCFG --dofunchopper --store-directory=/path/to/trunks/” LD=”bin/cilly --save-temps --domakeCFG --dofunchopper --store-directory=/path/to/trunks/”
```

Then, all code trunks will be in /path/to/trunks/. Each folder in it is named after the corresponding function in the project; if two functions have the same name (not to happen normally), the code chopper would report some error message but continue.

2. can use “gcc -E” to convert .c files to .i files first, then continue to invoke the code chopper on each .i file (some scripts mentioned later may be useful).
E.g.,

```
./configure
make CC=”gcc” CFLAGS=”-E”
```

3. can save the intermediate .i files during normal configure/make process, then invoke the chopper on each .i files. This can be used if the compilation process needs to actually generate .o files. Used this for the Linux kernel. E.g.

```
./configure
make CC=”gcc -save-temps”
find . -name "*.i" | while read fn; do
   run codechoper on $fn
done
```

4. use **normal** version of cil to simplify .c files in a project, then use this modified cil with the code chopper for each .cil.c file.

```
./configure
make CC=”normalbin/cilly -save-temps”
```

Then do something like:

```
find . -name "*.cil.c" | while read fn; do
…...
```

We have a script "chopcode" mentioned in the following to facilitate this 4th option.

## = Code Trunk Compilation & Running & Clustering

A set of scripts for compiling, running the code chopper, submitting jobs to and running jobs on the UC Davis cluster is in the folder:

- modules/tools

The scripts may also be used by the code chopper to move around files (so as to avoid putting too many files in a same folder. see the “infraplacement” parameter).

another set of auxiliary scripts for collecting C files, getting statistics of results, etc. is in the folder:

- workscripts

## == Basic steps:

- generate a list of all available code trunks:
    - modules/tools/initcodelist-foo <store-directory> > trunklist
- compile the common dyclone/eqminer files (can be done as part of "compileallcode")
    - modules/C/build.sh
- compile all necessary common header .h/.c files for the code trunks (can be done as part of "compileallcode"
    - modules/tools/initcodelist-foo <store-directory> > trunklist
- optional but may be more efficient: compile all code first before clustering, otherwise, "clustercode" would try to compile the code trunks if it cannot find .exe during execution.
    - modules/tools/compileallcode <store-directory> trunklist
- initialize the first cluster: list all code trunks' filenames in the L* metadata file in it, and setup metadata
    - modules/tools/initcodelist <store-directory> </path/to/clusters>/0/CLSRTT.0.0/L0
        - i. "0/CLSRTT.0.0/L0" is our name convention for the first cluster
- run, compare, cluster code trunks (this also compiles the code trunks if corresponding exe don't exist)
    - modules/tools/clustercode </path/to/clusters>
        - i. clustering results should be in folders containing a CLSRTT.0.0/ within </path/to/clusters>
        - ii. how to read the files in each CLSRTT.0.0
            - the path name leading to it indicates the cluster the code trunks belong to;
            - C*: contain the name of the representative code trunk for the cluster
            - L*: contain the list of all code trunks belonging to this cluster
            - H*: contain sub-clusters belonging to this cluster
            - I* and ins/I*: contain the inputs used for executing the code trunks
            - O*: contain the outputs from the code trunks
- Logs will be in __dyc_* files in </path/to/clusters>
    - __dyc_log*: contain runtime info about the code trunks (e.g., execution failures of a code trunk)
    - __dyc_fail*: contain runtime failure info about the scripts themselves
    - __dyc_time*: contain various timing info about different parts of the scripts
- if the clustering is stopped in the middle, can start from the middle where a cluster is completed (need to identify the cluster id accordingly as follows). This only works for clustering on a local machine, not on the cluster due to likely race conditions. E.g.,
    - start the clustering from the first cluster containing all code trunks (the default)
        - i. modules/tools/clustercode </path/to/clusters> 0
    - if the clustering stopped at some first-level clustering, then can resume from there:
        - i. modules/tools/clustercode </path/to/clusters> $nextid
        - ii. the $nextid should be the number contained in the </path/to/clusters/0/nextid> file minus 1

- if the clustering stopped after the first-level clustering finishes, and at some 2nd-level cluster, then can resume from the 2nd-level cluster:
  1. modules/tools/clustercode </path/to/clusters> 0/$clusterid/$nextid
  2. the $clusterid is the id from the first-level cluster in which the 2nd-level clustering stopped
  3. the $nextid is similar to the above, just that it should come from the "nextid" file contained in the first-level cluster in which the 2nd-level clustering stopped
- similarly, if the clustering stopped after the m-level clustering finishes, and at some m-level cluster, then can resume from it by running:
  1. modules/tools/clustercode </path/to/clusters> 0/$FirstLevelClusterid/$SencondLevelClusterid/.../$MminusOneLevelClusterid/$nextid

## == Brief notes about some of the scripts:

- utils.sh: common functions to move around files due to file number limits on ext3
- **chopcode**: run the code chopper for each .cil.c file in a folder
- cluster_chopcode: similar to chopcode, but is for the UCD cluster
- initcodelist-foo: output a list of available code trunks (*.foo.c files) with absolute path names.
- **compileallcode**: compile all code trunks in a folder according to a given file list
- cluster_compileallcode: similar to compileallcode, but for the cluster
- compone*/comptwo*: different ways to compare similarity between two sets of outputs
- shuffle*/permute*: shuffle a list of inputs
- killproc: to kill a proc if it's long-running
- **initcodelist**: initialize the first cluster and metadata for later code clustering
- initcodelinks-infra: create symbolic links with absolute pathname/filename to code trunks, so as to make compile the code trunks easier even when the code trunks are moved around
- **clustercode**: compile, run, compare, and cluster code trunks on a local machine; clustering up to MAXDEPTH=10 levels; the compilation may not work
- cluster_clustercode: to compile, run, compare, and cluster code trunks on the cluster
- cluster_clustercodestart: starting point for compiling, running, comparing, and clustering code trunks on the cluster
- submitsubjobs: submit clustercode jobs to the cluster
- reinfraplace: move files into a subfolder when the folder contains too many files; we limit to 2 levels of infraplacement. See "notes about infraplacement" below.
- collectvectors: collect .vec vector files for syntactical clone detection
- codefix: cil cannot handle certain c code features; so this script is to fix such code

## == Brief notes about "infraplacement":

Due to EXT3_LINK_MAX and performance issues, we cannot put too many files/folders into the same folder. So, when there are a lot of code trunks generated (sometimes, hundreds of thousands potential code trunks for just one BIG function), we need to reorganize the folder structure to hold the trunks along the way. So we basically use "infraplacement" (moving downward), i.e., creating a new subfolder for a folder if the folder now contains too many files, and moving exiting files in the folder into its subfolder.

Thus, we needed a bunch of scripts to help track/manage infraplaced files.