CHERITON SCHOOL OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO

Fact Extractor Manual

Bryan J Muscedere
Professor Joanne Atlee
Professor Michael Godfrey
Dr. Ian Davis

Updated by: Sunjay Varma (April 2019)

Contents

July 2019

1 Installing Rex

Rex is a C++ fact extractor that utilizes the clang AST to generate ”facts” from C++ source files. This section will provide information that installs the required libraries and tools used to install the Rex extractor. Before Rex can be built, it requires the following to be installed on the target system: **CMake** 3.0.0 or greater, **Boost** 1.69 or greater, **LibSSL**, and **Clang** 8.0.0. CMake is used to build Rex, Boost is a collection of C++ libraries that is used to process command line arguments and directory information, and Clang is used to obtain AST information about the source code currently being processed. The Clang API provides methods for operating on C++ language features and carries out the brunt of the C++ analysis.

The remainder of this section provides information on how to install these required libraries and how to build Rex from source. These instructions are for Linux-based systems. Section 1.1 describes how to install CMake, Boost, and Clang. If any of these are already installed on the target system, it can be skipped. Section 1.2 describes how to build Rex from source.

If you find any errors in this document or if the steps do not work, please report what you find to someone working on the project. If you yourself work on the project, please fix whatever you find. The source code for these instructions can be found in the `docs/` directory of the Rex repository.

1.1 Prerequisites

CMake If using a Ubuntu or Debian-based system, installing CMake is as simple as using `apt-get`. This is done using the following two commands:

```
$ sudo apt-get install cmake
$ cmake --version
```

Using `apt-get` might not install the latest version of CMake. However, as long as it installs a version of CMake greater than `3.0.0`, it can be used to build ClangEx and Rex. If this is the case, proceed to the Boost installation instructions.

If this method did not work, CMake must be built from source. To do this, the latest version of CMake source needs to be downloaded from the CMake website, compiled, and then installed. This guide provides instructions on how to build CMake `3.7.0` from source. The first step is to download the CMake source code and unzip it. This can be from the command line using the following commands:

```
$ wget https://cmake.org/files/v3.7/cmake-3.7.0.tar.gz
$ tar xvzf cmake-3.7.0.tar.gz
$ cd cmake-3.7.0
```

Once in the `cmake-3.7.0` directory, CMake can be configured and install on the target system. This process may take several minutes. To do this, use the following commands:

```
$ ./configure
$ make
$ make install
$ cmake --version
```

If these steps completed successfully, CMake is now installed and ready for use.

Boost Do **not** install Boost using the `apt-get` package manager. The version of Boost provided is not guaranteed to be up to date and can otherwise cause you many problems. Instead, visit the Boost downloads website and download the Boost 1.69 source code.

Boost Downloads Page: `https://www.boost.org/users/download/`

Then, extract the files into a `boost_1_69_0` directory and run the following commands:

```
$ cd path/to/boost_1_69_0
$ ./bootstrap.sh
$ ./b2
```

This will build Boost 1.69 in that directory, and otherwise leave your system untouched.

Clang Although Clang exists as a package that can be installed using `apt-get`, it needs to be installed from source to take advantage of Clang's API. To do this, source code needs to be checked out from the official Clang repository, compiled, and then installed. The first step is to checkout the Clang source code. This can be done by executing the following:

```
$ wget releases.llvm.org/8.0.0/llvm-8.0.0.src.tar.xz
$ tar xf llvm-8.0.0.src.tar.xz
$ wget releases.llvm.org/8.0.0/cfe-8.0.0.src.tar.xz
$ tar xf cfe-8.0.0.src.tar.xz && mv cfe-8.0.0.src llvm-8.0.0.src/tools/clang
$ wget releases.llvm.org/8.0.0/clang-tools-extra-8.0.0.src.tar.xz
$ tar xf clang-tools-extra-8.0.0.src.tar.xz
$ mv clang-tools-extra-8.0.0.src llvm-8.0.0.src/tools/clang/tools/extra
```

These commands will download the LLVM and Clang source code to a directory called llvm-8.0.0.src in the current working directory. Clang can now be built. This process can take up to several hours and uses a large amount of RAM and disk space (make sure you have enough swap space). This guide shows how to build Clang in a directory called Clang-Build that is adjacent to the llvm-8.0.0.src directory. To use another directory, simply replace the Clang-Build string in the following commands with another directory name.

The following commands build Clang in the Clang-Build directory:

```
$ mkdir Clang-Build
$ cd Clang-Build
$ cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_EH=ON -
    DLLVM_ENABLE_RTTI=ON ../llvm-8.0.0.src
$ make
$ sudo make install
```

Once these steps have been completed, Clang and LLVM have been installed. By typing clang –version, Clang should report that it is version 8.0.

1.2 Building Rex

To build Rex, the source code needs to be checked out from the Rex git repository and then built using CMake. If Rex fails to build, first ensure that all tools and libraries installed in the previous section are present on the target system.

To download Rex, you first need to checkout the source code. Run the `git clone` command to download the repository into a directory called `Rex`. Before checking out the repository, ensure that you are in a directory where you want to install Rex.

Next, before Rex can be built, three environment variables must be set:

- `LLVM_PATH` should be set to the `clang_llvm_8.0.0` directory
- `CLANG_VER` should be set to the string `8.0.0`
- `BOOST_ROOT` should be set to the `boost_1_69_0` directory

To set these variables, open up `.bashrc` located in the home directory and add the following lines to the bottom of the file:

```
    $ export LLVM_PATH=<PATH_TO_clang_llvm_8.0.0>
    $ export CLANG_VER=8.0.0
    $ export BOOST_ROOT=<PATH_TO_boost_1_69_0>
```

Restart the terminal to ensure these variables have been exported.

The next step is to build Rex. These steps install Rex in a directory called `Rex-Build` adjacent to the `Rex` directory. To install it somewhere else, simply replace the `Rex-Build` string in the following commands with a desired directory name. The following steps install Rex in the `Rex-Build` directory:

```
    $ cd Rex-Build
    $ cmake -G "Unix Makefiles" ../Rex
    $ make -j 8
```

By running these commands, CMake will build the source code and an executable called *Rex* will be created inside the `Rex-Build` directory. To verify that Rex built correctly, run:

```
    $ ./Rex --help
```

If Rex built, this command will run the Rex executable and print out some help information about the kinds of arguments you can pass to Rex. Now, Rex is ready to process C++ source files and generate tuple-attribute models!

2 Using Rex

Rex is a command line tool. This section describes the various ways you can configure it using arguments passed to the `Rex` executable built in the previous section.

2.1 Separate Compilation

To run Rex, pass in one or more source files or directories that you would like it to analyze. For directories, Rex will recursively search for both CPP source files and header files. All common C++ file extensions are supported. Here's an example:

```
    $ ./Rex foo.cpp foo2.cpp bar/
```

Rex will run clang on both `foo.cpp` and `foo2.cpp`. It will also walk the `bar` directory recursively looking for both source files and header files. All files found will be analyzed as well.

To run multiple analyses in parallel, pass in the `--jobs` flag (abbreviated `-j`) as shown below:

```
    $ ./Rex foo.cpp foo2.cpp bar/ -j 8
```

This will usually result in a faster analysis because Rex can analyse multiple source files at once.

By default, Rex will analyse these files, but not attempt to link together the results in any way. This enables us to do separate compilation and optimize re-running Rex when only a few files have changed. The results of the analysis of each file are stored in "TA Object Files". You can find those files in the directory where you ran Rex. Each file will have the `.tao` file extension. This is a custom, Rex-specific file format that is specially designed to make the linking process fast.

One caveat to be aware of: Most files require at least some compiler flags in order for clang to be able to compile them. The way these flags are discovered is with a special file called a compilation database. If the project being processed uses any compiler flags, a compilation database will need to be generated for that project and placed in the root directory of all the project source files. All files being analyzed by Rex must either be in the same directory as the database or in some descendant directory. A compilation database is a JSON file that has an entry for **each** file that Rex will analyze. If a file is not present in that database, Rex may fail to analyze that file at all. If a compilation database does not exist, Rex will show a warning and attempt to proceed without it.

2.2 Linking

To produce a TA file from the generated `.tao` files, pass the `--output` flag (abbreviated `-o`) as shown below:

```
    $ ./Rex foo.cpp foo2.cpp bar/ -j 8 -o bar.ta
```

This will analyze each file, generate `.tao` files for each, and link them together to produce a TA file with the filename `bar.ta`. If you have already generated the `.tao` files and do not want to re-run the analysis, you can pass those directly as well:

```
$ ./Rex *.tao -o bar.ta
```

Rex even supports a mix of source file/directory arguments and `.tao` file arguments:

```
$ ./Rex foo.cpp.tao foo2.cpp bar/ -j 8 -o bar.ta
```

In this case, the `.tao` file arguments will only be used during linking and the other provided files/directories will be analyzed and linked as would be normally expected.

You can mix and match these styles of passing arguments to optimize your run of Rex so that it only ever analyses the source files you want it to. This style of command line arguments is similar to what you may be used to with most compilers.

2.3 Including Files in the Analysis

Rex will not analyze any files that you do not provide as arguments. That means that if you are passing individual files into Rex, you will often need to provide **both** the CPP source file and the associated header file in order to get any results:

```
$ ./Rex foo.cpp foo.h -j 8 -o bar.ta
```

If you omit the header file, you may find that the generated TA file is empty or far smaller than you expected. The reason for this is because most declarations are in header files and Rex needs to analyze those header files in order to be able to link their declarations.

If you use a directory instead, Rex will take care of picking up both types of files automatically. That is usually the preferred method of using Rex unless you are trying to diagnose a specific problem or otherwise need to narrow down the analysis to specific files.

It's not that the source file and header file need to be run in the same command. The only requirement is that when linking into a TA file, the `.tao` file for both the source and the header are included.

```
$ ./Rex foo.cpp
$ ./Rex foo.h
$ ./Rex foo.cpp.tao foo.h.tao -o bar.ta
```

A great consequence of this slightly unconventional setup is that only the files we intend to be included are included in our analysis. That means that system files are completely ignored unless we explicitly pass their files/directories into Rex. Automatically detecting the header files associated with a given source file is non-trivial in C++. This solution gives us flexibility without too much added complexity.