CodeChecker: A static analysis infrastructure built on the LLVM/Clang Static Analyzer tool chain

György Orbán[1], Tibor Brunner [1], Gábor Horváth[2], Réka Kovács[2]
[1]Ericsson Ltd. [2]Eötvös Loránd University, Budapest

Introduction

CodeChecker is a static analysis infrastructure which was built to run multiple static analysis tools on C/C++ projects. Integrating multiple static analysis tools into the build systems can be complex and time consuming. With CodeChecker our goal was to create a tool which can be easily integrated into various build systems, runs multiple static analyzers and helps to manage the results provided by the analyzers. CodeChecker currently supports two analyzer tools: Clang Static Analyzer and Clang-Tidy.

Problem #1
Build system integration

All the compilation flags and parameters that are used for building the project required for the analysis capturing the compilation commands can be challenging.

- Many build systems can not export the compilation commands
- Custom integration work for each build system can take a lot of time and resources
- gcc/g++ cross compilation: include path and build target needs auto detection
- Many gcc/g++ compilation flags are not compatible with Clang based tools and some build targets are not available in Clang right now
- Detect C/C++ standards where it is not set in the compilation command

Problem #2
Managing static analyzer reports

The static analyzers provide plist (xml), html or plain text output.

- Understanding is cumbersome especially for longer bug paths
- Comparing multiple analysis runs is hard to do
- Suppressing false positives cannot be done with these output formats
- Many reports can be found by the analyzers which is hard to manage

The implementation

CodeChecker is an open source project can be found on GitHub. Feel free to give us feedback or contribute to the development.

LOG

During the build process a special shared object is loaded through LD_PRELOAD (Linux only*) to capture all exec system calls and filter out the compilation commands. This solution is build system independent**.

```
__attribute__ ((visibility ("default"))) int execv(const char* filename_,
char* const argv_[]) {
CC_LOGGER_CALL_EXEC(execv, (const char*, char* const*), filename_, argv_)
;}
```

* On OSX scan-build-py can be used to collect the compile commands
** CMake can export the compilation command database json, no logging is needed

Compilation database json

ANALYZE

1. Filter out incompatible gcc/g++ compilation flags

2. Auto detect cross compilation flags and include paths

```
$/usr/bin/gcc  -E -x c  - -v

gcc version 5.4.0 20160609
COLLECT_GCC_OPTIONS='-E' '-v' '-mtune=generic' '-march=x86-64'
/usr/lib/gcc/x86_64-linux-gnu/5/cc1  -E  -quiet  -v  -imultiarch x86_64-linux-gnu - -mtune=generic -march=x86-64 -fstack-protector-strong -Wformat -Wformat-security
/usr/lib/gcc/x86_64-linux-gnu/5/include
/usr/local/include
/usr/lib/gcc/x86_64-linux-gnu/5/include-fixed
/usr/include/x86_64-linux-gnu /usr/include
…
```

3. Automatically retrieve compile target information

```
$ /usr/bin/gcc -v
…
Target: x86_64-linux-gnu
…
```

4. Detect C/C++ standard version

Compile a simple C/C++ file with the compiler used to build the project to get the macro values of:
- __STDC_VERSION__ (C standard version)
- __cplusplus (C++ standard version)

5. Configure Static Analyzers

Configure various Clang Static analyzer options:
- loop unrolling count
- analysis budget (node count)
- c++-inlining options
- …

6. Configure checker options
- unix.Malloc checker (optimistic)
- …

7. Execute the static analyzers

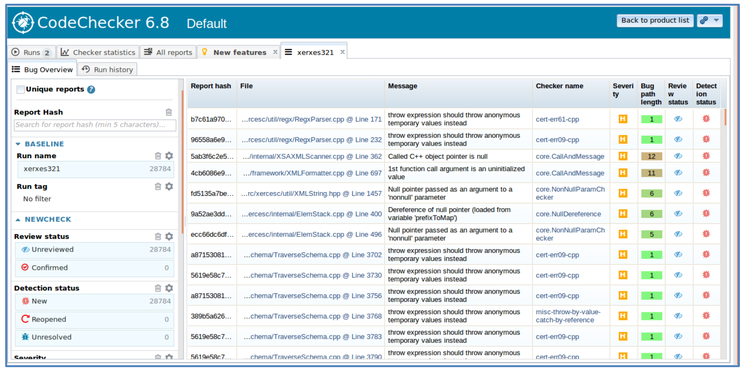
Clang Static Analyzer
- Symbolic execution
- Cross Translation Unit analysis (CTU)

Clang-Tidy
- diagnosing and fixing typical programming errors

plist (xml) report format

MANAGE AND VIEW

Process plist xml files and store them efficiently into a database (SQLite, PostgreSQL) managed by CodeChecker server. Clients (Web, command line, Eclipse) can connect to the server through a Thrift API to store, manage and query the results. Integration to the CI systems can be done easily with the provided command line client. Uniqueing of reports can be done based on the report hash, which is extremely useful in case of CTU results.



WEB UI helps to manage and understand complex analyzer reports.