Jan-Simon Möller
jsmoeller@linuxfoundation.org
dl9pf@gmx.de, dl9pf on #freenode
Topics
1. What is 'Static Analysis'
2. Motivations to use it
3. Open Source Tools for Static Analysis
   a. Example using gcc10
   b. Example using clang
   c. Example using cppcheck
   d. Example using CodeChecker
4. Example integration in Makefiles
5. Example integration with git hooks
6. Summary
7. Q/A
What is 'Static Analysis'
"Static Analysis" or "Static Code Analysis" in general is a method for debugging a program before it is run. It is done by analyzing the code in question and comparing it to a set of coding rules. This is in contrast to "Dynamic Analysis" which means the program does run. (covered in an upcoming webinar) In most cases it is either performed on some parsed representation of the source code or on IR.
● Static analysis identifies defects before you run a program (e.g., between coding and unit testing).
● Dynamic analysis identifies defects when you run a program (e.g., during unit testing).
● However, some coding errors might not surface during unit testing. So, there are defects that dynamic testing might miss that static code analysis can find and of course vice versa.
● Usually it is performed by an automated tool.
● This type of analysis finds weaknesses and vulnerabilities.
● It is usually done early in the development cycle.
Motivations to use it
- Static analysis simply can find bugs - early
- Static analysis can find hard-to-spot bugs
  - e.g. 30-level deep undefined/invalid access
- Static analysis can complement the peer review
- Static analysis is also used to comply with coding guidelines or industry standards
  - e.g. MISRA or ISO-26262
- It is also enforced for certain applications/industries:
  - medical
  - nuclear
  - automotive, aviation
Open Source Tools for Static Analysis
There are multiple categories of tools available:

- tools using a form of string or pattern matching
- tools analyzing the code during compilation
- tools specialized for kernel space
- tools for userspace

Of course there are also proprietary tools.
The Linux Kernel is a very large and special codebase. Currently it contains more than 20 million lines of code. This is very demanding on the tooling used. Thus there are specialized tools around the kernel:

- `scripts/checkpatch.pl` (string matching, basics&style, good for new submissions)
- `sparse`  
  ```  
  make C=1 CHECK="/usr/bin/sparse" 
  ```
- `coccinelle`  
  ```  
  make C=1 CHECK="scripts/coccicheck" 
  ```  
  (see next tuesday's webinar by Julia Lawall)
- `smatch`  
  ```  
  make C=1 CHECK="smatch -p=kernel" 
  ```  
  (see webinar in ~2 weekly by Dan Carpenter)
- `gcc / clang static analyser`
For userspace there are a large number of tools available. A selection for C/C++ is below:

- gcc
- clang
- cppcheck
- coccinelle
- splint
- rats
- flawfinder
During development you can easily use these directly within your source tree:

- **gcc** (since gcc 10)
  - gcc -fanalyzer

- **clang**
  - e.g. scan-build make

- **cppcheck**
> cppcheck nullpointer.c
Checking nullpointer.c ...
nullpointer.c:7:14: error: Null pointer dereference: pointer [nullPointer]
  int value = *pointer; /* Dereferencing happens here */
    ^
nullpointer.c:6:17: note: Assignment 'pointer=NULL', assigned value is 0
  int * pointer = NULL;
    ^
nullpointer.c:7:14: note: Null pointer dereference
int value = *pointer; /* Dereferencing happens here */
    ^
gcc -Werror -fanalyzer nullpointer.c
nullpointer.c: In function 'main':
nullpointer.c:7:5: error: dereference of NULL 'pointer' [CWE-690] [-Werror=analyzer-null-dereference]
  7 | int value = *pointer; /* Dereferencing happens here */
      ^~~~~
'main': events 1-2

  6 | int * pointer = NULL;
      ^~~~~~~
  7 | int value = *pointer; /* Dereferencing happens here */
      ~~~~~
  (1) 'pointer' is NULL
  (2) dereference of NULL 'pointer'

cc1: all warnings being treated as errors

//see: https://developers.redhat.com/blog/2020/03/26/static-analysis-in-gcc-10/
//try: https://godbolt.org/
> clang-tidy nullpointer.c
Running without flags.
2 warnings generated.

nullpointer.c:7:5: warning: Value stored to 'value' during its initialization is never read
 [clang-analyzer-deadcode.DeadStores]
  int value = *pointer; /* Dereferencing happens here */
   ^
nullpointer.c:7:5: note: Value stored to 'value' during its initialization is never read

nullpointer.c:7:13: warning: Dereference of null pointer (loaded from variable 'pointer')
[clang-analyzer-core.NullDereference]
  int value = *pointer; /* Dereferencing happens here */
   ^

nullpointer.c:6:1: note: 'pointer' initialized to a null pointer value
  int * pointer = NULL;
   ^

nullpointer.c:7:13: note: Dereference of null pointer (loaded from variable 'pointer')
  int value = *pointer; /* Dereferencing happens here */
   ^
> scan-build make

TLDR: replaces $(CC) !!!

scan-build: Using '/usr/bin/clang-10.0.1' for static analysis
/usr/bin/ccc-analyzer  -c nullpointer.c -o nullpointer

nullpointer.c:7:5: warning: Value stored to 'value' during its initialization is never read
int value = *pointer; /* Dereferencing happens here */
    ^~~~~   ~~~~~~~~
nullpointer.c:7:13: warning: Dereference of null pointer (loaded from variable 'pointer')
int value = *pointer; /* Dereferencing happens here */
    ^~~~~~~~~
2 warnings generated.
scan-build: 2 bugs found.

> scan-view /tmp/scan-build-2020-10-15-161857-10509-1
Starting scan-view at: http://127.0.0.1:8181

(-> point browser to this)
https://github.com/Ericsson/codechecker

Collection of tools to

- intercept and log the build calls
- analyse the gathered data (using clang-tidy and clangSA)
- report (static or webui)

Extension and successor of the original clang static analyser / scan-build.
<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>.github/ISSUE_TEMPLATE</td>
<td>[GitHub] Fix minor grammatical things in the issue templates</td>
<td>7 months ago</td>
</tr>
<tr>
<td>analyzer</td>
<td>[analyzer] Fix analyzer --file option</td>
<td>20 days ago</td>
</tr>
<tr>
<td>bin</td>
<td>[license] Change license (#2729)</td>
<td>last month</td>
</tr>
<tr>
<td>codechecker_common</td>
<td>Add a missing space in a debug warning</td>
<td>last month</td>
</tr>
<tr>
<td>config</td>
<td>Adding new checkers to the profiles, setting severities</td>
<td>2 months ago</td>
</tr>
<tr>
<td>docker</td>
<td>New dockerfiles for test environments</td>
<td>2 years ago</td>
</tr>
<tr>
<td>docs</td>
<td>[tools] tu_collector get dependent source files for headers</td>
<td>21 days ago</td>
</tr>
<tr>
<td>requirements_py/docs</td>
<td>Merge pull request #1935 from gyorb/readthedocs</td>
<td>15 months ago</td>
</tr>
<tr>
<td>scripts</td>
<td>[license] Change license (#2729)</td>
<td>last month</td>
</tr>
<tr>
<td>Name</td>
<td>Number of unresolved reports</td>
<td>Detection status</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>agl-service-gps@oneshot</td>
<td>1</td>
<td><img src="1" alt="Error" /> <img src="1" alt="Error" /></td>
</tr>
<tr>
<td>cynagora@oneshot</td>
<td>17</td>
<td><img src="1" alt="Error" /> <img src="1" alt="Error" /></td>
</tr>
<tr>
<td>app-framework-binder@oneshot</td>
<td>79</td>
<td><img src="1" alt="Error" /> <img src="1" alt="Error" /></td>
</tr>
<tr>
<td>app-framework-main@oneshot</td>
<td>35</td>
<td><img src="1" alt="Error" /> <img src="1" alt="Error" /></td>
</tr>
<tr>
<td>agl-service-audiomixer</td>
<td>4</td>
<td><img src="1" alt="Error" /> <img src="1" alt="Error" /></td>
</tr>
</tbody>
</table>
Userspace tool CodeChecker is a set of python helpers

- main feature is that you wrap your build commands like so
  
  CodeChecker log -b "make" -o compilation.json

- This will preload a logger and store the compiler commands

- With the exact commands logged, we can replay the compilation using clang and its tools clang-tidy and clangSA

  CodeChecker analyze compilation.json -o ./reports

TLDR: does not need to change $(CC)
● From there you can 'parse' into reports

CodeChecker parse ./reports

CodeChecker parse ./reports -e html -o reports_html

● or 'store' online in webui/frontend

CodeChecker store ./reports --name mypkg@v0.9 \--url http://localhost:8001/Default
<table>
<thead>
<tr>
<th>Name</th>
<th>Number of unresolved reports</th>
<th>Detection status</th>
<th>Analyzer statistics</th>
<th>Storage date</th>
<th>Analysis duration</th>
<th>Check command</th>
<th>Version tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agl-service-gps@oneshot</td>
<td>1</td>
<td><img src="red" alt="Red (1)" /></td>
<td>clangs: <img src="green" alt="Green" /> (1)</td>
<td>2020-07-02</td>
<td>06:41:01</td>
<td>Show</td>
<td>6.13 (dfb5618c00  b2641197d8 fa2f1599a37 58909924)</td>
<td></td>
</tr>
<tr>
<td>cynagora@oneshot</td>
<td>17</td>
<td><img src="red" alt="Red (17)" /></td>
<td>clangs: <img src="green" alt="Green" /> (30)</td>
<td>2020-07-02</td>
<td>08:00:16</td>
<td>Show</td>
<td>6.13 (dfb5618c00  b2641197d8 fa2f1599a37 58909924)</td>
<td></td>
</tr>
<tr>
<td>app-framework-binder@oneshot</td>
<td>79</td>
<td><img src="red" alt="Red (79)" /></td>
<td>clangs: <img src="green" alt="Green" /> (92) <img src="red" alt="Red" /> (3)</td>
<td>2020-07-02</td>
<td>07:50:44</td>
<td>Show</td>
<td>6.13 (dfb5618c00  b2641197d8 fa2f1599a37 58909924)</td>
<td></td>
</tr>
<tr>
<td>app-framework-main@oneshot</td>
<td>35</td>
<td><img src="red" alt="Red (35)" /></td>
<td>clangs: <img src="green" alt="Green" /> (34)</td>
<td>2020-07-01</td>
<td>22:04:52</td>
<td>Show</td>
<td>6.13 (dfb5618c00  b2641197d8 fa2f1599a37 58909924)</td>
<td></td>
</tr>
<tr>
<td>agl-service-audiomixer</td>
<td>4</td>
<td><img src="red" alt="Red (4)" /></td>
<td>clangs: <img src="green" alt="Green" /> (2)</td>
<td>2020-07-01</td>
<td>21:36:00</td>
<td>Show</td>
<td>6.13 (dfb5618c00  b2641197d8 fa2f1599a37 58909924)</td>
<td></td>
</tr>
</tbody>
</table>
Example integration in Makefiles
Integrating gcc's -fanalyzer into your Makefiles is easy: just add it to the CFLAGS!

Similar for cmake. Add it to the CFLAGS.

TARGET_EXEC ?= myprog
BUILD_DIR ?= ./build
SRC_DIRS ?= ./src

SRCS := $(shell find $(SRC_DIRS) -name *.c)
OBJS := $(SRCS:%=$(BUILD_DIR)/%.o)
DEPS := $(OBJS:.o=.d)
INC_DIRS := $(shell find $(SRC_DIRS) -type d)
INC_FLAGS := $(addprefix -I,$(INC_DIRS))
CFLAGS := $(INC_FLAGS) -Wall -Werror -fanalyzer

$(BUILD_DIR)/$(TARGET_EXEC): $(OBJS)
  $(CC) $(OBJS) -o $@ $(LDFLAGS)

# c source
$(BUILD_DIR)/%.c.o: %.c
  $(MKDIR_P) $(dir $@)
  $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean

clean:
  $(RM) -r $(BUILD_DIR)

-include $(DEPS)

MKDIR_P ?= mkdir -p
If you use clang, you can run scan-build like so:

- `scan-build make <make options>`

It will add the flags on the fly.
(If your Makefile uses `$\mathrm{(CC)}$` !!)

As shown, CodeChecker will record/reply the compilation without this need.
You can add cppcheck like so:

```bash
SOURCES = main.cpp
CPPCHECK = cppcheck
CHECKFLAGS = -q --error-exitcode=1

default: cppcheck.out.xml hellomake
.PHONY: default clean

cppcheck.out.xml: $(SOURCES)
    $(CPPCHECK) $(CHECKFLAGS) $(^) -xml >$@

hellomake: $(OBJ)
    $(LINK.c) -o $@ $(^)
```
Example integration with git hooks
Git hooks are a mechanism that allows arbitrary code to be run before, or after, certain Git lifecycle events occur. For example, one could have a hook into the commit-msg event to validate that the commit message structure follows the recommended format.

The hooks can be any sort of executable code, including shell, PowerShell, Python, or any other scripts. Or they may be a binary executable. Anything goes! The only criteria is that hooks must be stored in the .git/hooks folder in the repo root, and that they must be named to match the corresponding events (as of Git 2.x):
- applypatch-msg
- pre-applypatch
- post-applypatch
- **pre-commit**
- prepare-commit-msg
- commit-msg
- post-commit
- pre-rebase

- post-checkout
- post-merge
- pre-receive
- update
- post-receive
- post-update
- post-rewrite
- pre-push
#!/bin/sh

echo "Running static analysis..."
# Inspect code using scan-build, will exit 1 when bug is found
scan-build make -j2

status=$?

if [ "$status" = 0 ] ; then
    echo "Static analysis found no problems."
    exit 0
else
    echo 1>&2 "Static analysis found violations."
    exit 1
fi
Example from: 

[...]
# We should pass only added or modified C/C++ source files to cppcheck.
changed_files=$(git diff-index --cached $against | \
    grep -E '[MA] .*(c|cpp|cc|cxx)$' | cut -f 2)

if [ -n "$changed_files" ]; then
    cppcheck --error-exitcode=1 $changed_files
    exit $?
fi

exit 0
• Static analysis
  – can help you improve your projects codebase early during coding
  – is one requirement in various standards / industries
  – can be easily added to your automation / CI
References:

- https://github.com/dl9pf/staticanalysis-webinar
- https://developers.redhat.com/blog/2020/03/26/static-analysis-in-gcc-10/
- https://godbolt.org/
- https://clang.llvm.org/extra/clang-tidy/
- https://github.com/Ericsson/codechecker
- http://cppcheck.sourceforge.net/
We hope it will be helpful in your journey to learning more about effective and productive participation in open source projects. We will leave you with a few additional resources for your continued learning:

- The **LF Mentoring Program** is designed to help new developers with necessary skills and resources to experiment, learn and contribute effectively to open source communities.
- **Outreachy remote internships program** supports diversity in open source and free software.
- **Linux Foundation Training** offers a wide range of **free courses**, webinars, tutorials and publications to help you explore the open source technology landscape.
- **Linux Foundation Events** also provide educational content across a range of skill levels and topics, as well as the chance to meet others in the community, to collaborate, exchange ideas, expand job opportunities and more. You can find all events at [events.linuxfoundation.org](http://events.linuxfoundation.org).