C/C++ toolchain

Static and dynamic code analysis

Centre for Research on
Cryptography and Security

Karel Kubíček

Masaryk University

Brno, Czech Republic

April 20, 2018

Questions

- Who uses C/C++?

Questions

- Who uses C/C++?
- Who tried some static analysis tool? (Cpp-check, clang-tidy, Coverity, MS VS static analyzer, PVS-Studio. . .)

Questions

- Who uses C/C++?
- Who tried some static analysis tool? (Cpp-check, clang-tidy, Coverity, MS VS static analyzer, PVS-Studio. . .)
- Who tried some dynamic analysis tool? (Valgrind + Hellgrind, clang sanitizers, MS VS tool)

Questions

- Who uses C/C++?
- Who tried some static analysis tool? (Cpp-check, clang-tidy, Coverity, MS VS static analyzer, PVS-Studio. . .)
- Who tried some dynamic analysis tool? (Valgrind + Hellgrind, clang sanitizers, MS VS tool)
- Who wrote 1k lines of C/C++ code without bugs (on the first attempt)?

Questions

- Who uses C/C++?
- Who tried some static analysis tool? (Cpp-check, clang-tidy, Coverity, MS VS static analyzer, PVS-Studio. . .)
- Who tried some dynamic analysis tool? (Valgrind + Hellgrind, clang sanitizers, MS VS tool)
- Who wrote 1k lines of C/C++ code without bugs (on the first attempt)?
- Who is using clang?

Questions

- Who uses C/C++?
- Who tried some static analysis tool? (Cpp-check, clang-tidy, Coverity, MS VS static analyzer, PVS-Studio. . .)
- Who tried some dynamic analysis tool? (Valgrind + Hellgrind, clang sanitizers, MS VS tool)
- Who wrote 1k lines of C/C++ code without bugs (on the first attempt)?
- Who is using clang? Clang toolchain (clang-tidy, clang-format, clang sanitizers)?

Questions

- Who uses C/C++?
- Who tried some static analysis tool? (Cpp-check, clang-tidy, Coverity, MS VS static analyzer, PVS-Studio. . .)
- Who tried some dynamic analysis tool? (Valgrind + Hellgrind, clang sanitizers, MS VS tool)
- Who wrote 1k lines of C/C++ code without bugs (on the first attempt)?
- Who is using clang? Clang toolchain (clang-tidy, clang-format, clang sanitizers)?

Static code analysis

+ Easy to setup and run
+ Quite fast
+ Complete code coverage
– Weak (can find only fixed patterns)
– False positives

Static code analysis – tools

- cppcheck (`cpp-check -enable=all path`)

Static code analysis – tools

- cppcheck (`cpp-check -enable=all path`)
  - checks: variable scope, out-of-bounds, memory-leaks, sizeof args... (170 paterns)
  - \+ fast, simple
  - – weak, almost no support of modern C++ (11 years old project)
  - `-enable=all` overloads you with false positives

Static code analysis – tools

- cppcheck (`cpp-check -enable=all path`)
  - checks: variable scope, out-of-bounds, memory-leaks, sizeof args... (170 paterns)
  - \+ fast, simple
  - – weak, almost no support of modern C++ (11 years old project)
  - `-enable=all` overloads you with false positives
- Clang Static Analyzer

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ..
ln -s $PWD/compile_commands.json ..
clang-check -analyze
```

Static code analysis – tools

- cppcheck (`cpp-check -enable=all path`)
  - checks: variable scope, out-of-bounds, memory-leaks, sizeof args... (170 paterns)
  - \+ fast, simple
  - – weak, almost no support of modern C++ (11 years old project)
  - `-enable=all` overloads you with false positives
- Clang Static Analyzer

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ..
ln -s $PWD/compile_commands.json ..
clang-check -analyze
```

  - 40 patterns and new are still developed
  - \+ fewer false positives, modern C++ support
  - – slower, more difficult to use, unix mainly

Static code analysis – tools

- cppcheck (`cpp-check -enable=all path`)
  - checks: variable scope, out-of-bounds, memory-leaks, sizeof args... (170 paterns)
  - \+ fast, simple
  - – weak, almost no support of modern C++ (11 years old project)
  - `-enable=all` overloads you with false positives
- Clang Static Analyzer

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ..
ln -s $PWD/compile_commands.json ..
clang-check -analyze
```

  - 40 patterns and new are still developed
  - \+ fewer false positives, modern C++ support
  - – slower, more difficult to use, unix mainly
- MS VS static analyzer, PVS-Studio, Splint, OCLint, Coverity scan...

Static code analysis – tools 2

- Clang tidy `clang-tidy -checks='*' src-file.c -`

Static code analysis – tools 2

- Clang tidy `clang-tidy -checks='*' src-file.c -`
  - Clang Static Analyzer (`clang-analyze-*`) + much more (`modernize-*`) + C++ Core guidelines
  - \+ integrates static analysis with C++1x suggestions, easier to run on single file, integration in CLion
  - – running `-checks='*'` might be overhelming (although the number of false positives is not high, severity is often low)

Static code analysis – demo

- Working: everything in static examples
- Failing:
  - zune bug
  - `clang-tidy -checks=* asan1.cc - -std=c++17`
  - large codebase `clang-tidy -help`
- Motivation $\rightarrow$ we want modern C++
- clang-tidy examples
- clang-format examples

Dynamic code analysis

+ Easy to set up and run
+ Super strong (none false negatives)
+ (Almost) none false positives
– Slows your code (you cannot use it when debugging real-time issue [embedded SW with interrupts, GUI])
– Covers only code that is executed

Dynamic code analysis – tools

- Valgrind

Dynamic code analysis – tools

- Valgrind
  - Hellgrind, Cachegrind (profiler for cache access), Callgrind (records call history), Massif (heap profiler)
  - Virtual machine with just-in-time compilation (sandboxing, heap reference counting)
  - \+ no limitation for compiler (can run executable)
  - − only dynamic memory, 5-50+ slow-down

Dynamic code analysis – tools

- Valgrind
  - Hellgrind, Cachegrind (profiler for cache access), Callgrind (records call history), Massif (heap profiler)
  - Virtual machine with just-in-time compilation (sandboxing, heap reference counting)
  - \+ no limitation for compiler (can run executable)
  - − only dynamic memory, 5-50+ slow-down
- Clang sanitizers

Dynamic code analysis – tools

- Valgrind
  - Hellgrind, Cachegrind (profiler for cache access), Callgrind (records call history), Massif (heap profiler)
  - Virtual machine with just-in-time compilation (sandboxing, heap reference counting)
  - \+ no limitation for compiler (can run executable)
  - – only dynamic memory, 5-50+ slow-down
- Clang sanitizers
  - Address sanitizer ASan, LeakSanitizer, Thread sanitizers TSan, Memory sanitizer MSan, Undefined Behaviour sanitizer UBSan, thin LTO
  - creates map of memory + around alocated blocks is shadow memory
  - \+ both stack and heap checks
  - – dependent on compiler (MSan requires building all code with MSan)
  - – 2x slow-down, 4x memory consumption, less "googlable" error messages than Valgrind (but good github wiki)
  - – crashes on first error (but `__attribute__((no_sanitize("address")))` or continue-after-error mode)

Dynamic code analysis – tools

- Valgrind
  - Hellgrind, Cachegrind (profiler for cache access), Callgrind (records call history), Massif (heap profiler)
  - Virtual machine with just-in-time compilation (sandboxing, heap reference counting)
  - \+ no limitation for compiler (can run executable)
  - – only dynamic memory, 5-50+ slow-down
- Clang sanitizers
  - Address sanitizer ASan, LeakSanitizer, Thread sanitizers TSan, Memory sanitizer MSan, Undefined Behaviour sanitizer UBSan, thin LTO
  - creates map of memory + around alocated blocks is shadow memory
  - \+ both stack and heap checks
  - – dependent on compiler (MSan requires building all code with MSan)
  - – 2x slow-down, 4x memory consumption, less "googlable" error messages than Valgrind (but good github wiki)
  - – crashes on first error (but `__attribute__((no_sanitize("address")))` or continue-after-error mode)

Dynamic code analysis – demo

	ASan	Valgrind
Heap out-of-bounds	Y	Y
Stack out-of-bounds	Y	N
Global out-of-bounds	Y	N
Use after free	Y	Y
Use after return	Y	N
Uninitialised memory read	N	Y
Initialisation order issues	Y	N
Leaks	N	Y
Slowdown	2x	>10x

Dynamic code analysis – demo

- Folder `./dynamic` – small examples
- CryptoStreams (single execution, tests)
- Monero? On demand projects

Summary

- Write tests (allows the project to survive longer)
  - you can refactor code (and use new language features)
  - allows you to use dynamic code analysis
- set up continuous integration (Travis)
  - run tests under dynamic analysis (clang sanitizers)
- use llvm toolchain (clang, clang-format, clang-tidy)
- use static analysis localy (set it up in your IDE)

Sources

- Pacific++ 2017: Chandler Carruth "LLVM: A Modern, Open C++ Toolchain":
  `https://www.youtube.com/watch?v=uZI_Qla4pNA`
- Tools from the C++ Ecosystem to save a leg - Anastasia Kazakova - Meeting C++ 2017:
  `https://www.youtube.com/watch?v=Hlmp-zTyrxM`
- CppCon 2017: Kostya Serebryany "Fuzz or lose: why and how to make fuzzing a standard practice for C++":
  `https://www.youtube.com/watch?v=k-Cv8Q3zWNQ`
  - Fuzzing is really cool – together with clang sanitizers, it can catch much more than normal tests
  - Another topic for OpenLab?