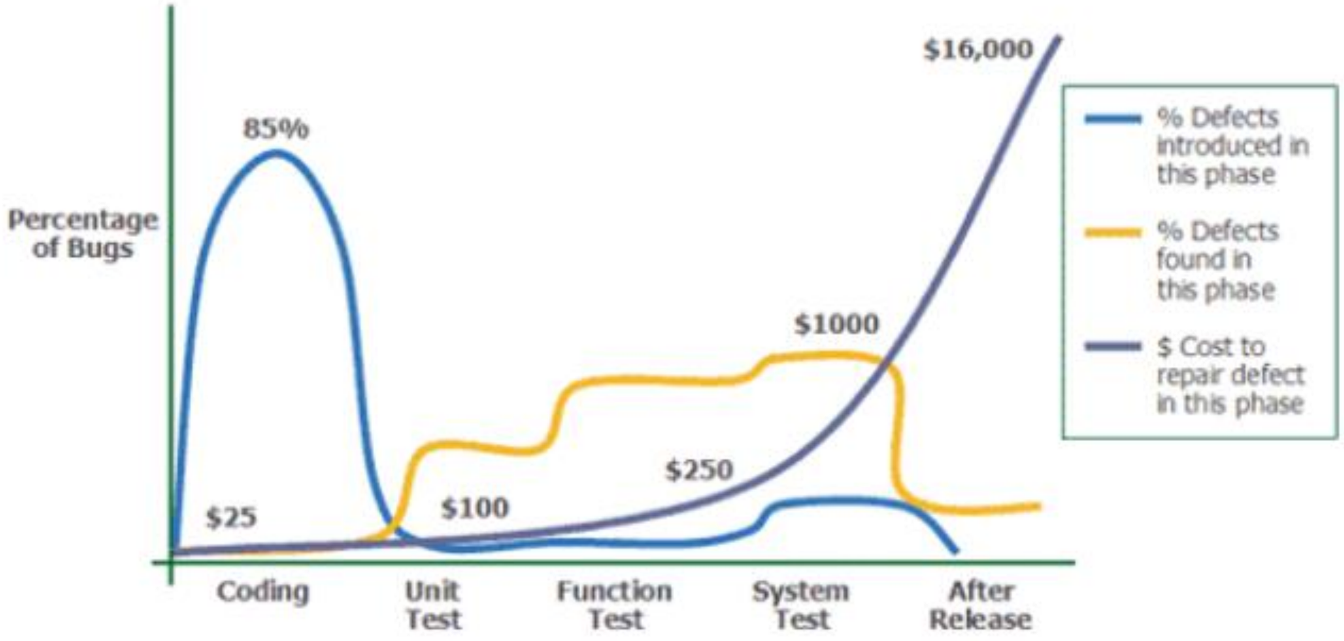
Using the Clang Static Analyzer

Vince Bridgers

About this tutorial

- “Soup to nuts” – Small amount of theory to a practical example
- Why Static Analysis?
- Static Analysis in Continuous Integration
- What is Cross Translation Unit Analysis, and how Z3 can help
- Using Clang Static Analysis on an Open Source Project

Why tools like Static Analysis? : Cost of bugs



- **Notice most bugs are introduced early in the development process, and are coding and design problems.**
- **Most bugs are found during unit test, where the cost is higher**
- **The cost of fixing bugs grow exponentially after release**
- ***Conclusion: The earlier the bugs found, and more bugs found earlier in the development process translates to less cost***

Source: Applied Software Measurement, Caspers Jones, 1996

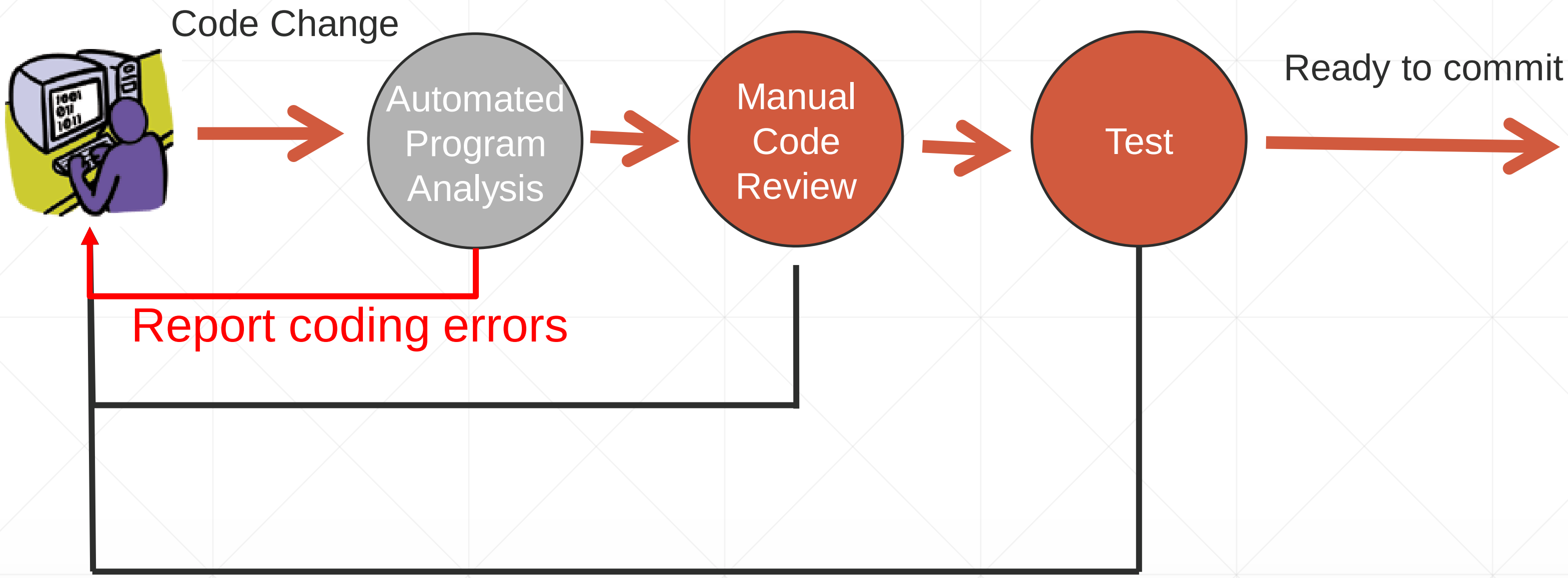
Finding Flaws in Source Code

- Compiler diagnostics
- Code reviews
- “Linting” checks, like Clang-tidy
- Static Analysis using Symbolic Execution
  - Analysis Performed executing the code symbolically through simulation
- Dynamic Analysis – Examples include UBSAN, TSAN, and ASAN
  - Analysis performed by instrumenting and running the code on a real target
  - Difficult to test the entire program, and all paths – dependent upon test cases

Four Pillars of Program Analysis

	Compiler diagnostics	Linters, style checkers	Static Analysis	Dynamic Analysis
Examples	Clang, gcc, cl	Lint, clang-tidy, Clang-format, indent, sparse	Cppcheck, gcc 10+, clang	Valgrind, gcc and clang
False positives	No	Yes	Yes	Not likely, but possible
Inner Workings	Programmatic checks	Text/AST matching	Symbolic Execution	Injection of runtime checks, library
Compile and Runtime affects	None	Extra compile step	Extra compile step	Extra compile step, extended run times

Typical CI Loop with Automated Analysis



Quick Feedback

Syntax, Semantic, and Analysis Checks:
Can analyze properties of code that cannot be tested (coding style)!
Automates and offloads portions of manual code review
Tightens up CI loop for many issues

Finding bugs with the Compiler

```
1: #include <stdio.h>
2: int main(void) {
3:     printf("%s%lb%d", "unix", 10, 20);
4:     return 0;
5: }
```

```
$ clang t.c
t.c:3:17: warning: invalid conversion specifier 'b' [-Wformat-invalid-specifier]
    printf("%s%lb%d", "unix", 10, 20);
              ~~^
t.c:3:35: warning: data argument not used by format string [-Wformat-extra-args]
    printf("%s%lb%d", "unix", 10, 20);
           ~~~~~~~~~              ^
2 warnings generated.
```

- Static analysis can find deeper bugs through program analysis techniques – like memory leaks, buffer overruns, logic errors.

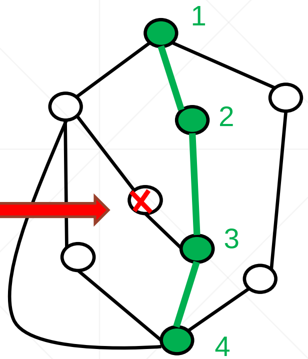
Finding bugs with the Analyzer

```
int function(int b) {
    int a, c;
    switch (b) {
        case 1: a = b / 0; break;
        case 4: c = b - 4;
                a = b/c; break;
    }
    return a;
}
```

- This example compiles fine – but there are errors here.
- Static analysis can find deeper bugs through program analysis techniques
- This one is simple, but imagine a large project – thousands of files, millions of lines of code

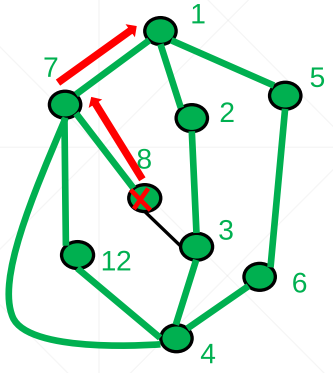
Program Analysis vs Testing

- "Ad hoc" Testing usually tests a subset of paths in the program.
  - Usually "happy paths" 
- May miss errors
- It's fast, but real coverage can be sparse
- Same is true for other testing methods such as Sanitizers
- All used together – a useful combination



Program Analysis vs Testing

- Program analysis can exhaustively explore all execution paths
- Reports errors as traces, or “chains of reasoning”
- Downside – doesn’t scale well – path explosion
- Path Explosion mitigation techniques …
  - Bounded model checking – breadth-first search approach
  - Depth-first search for symbolic execution



Clang Static Analyzer (CSA)

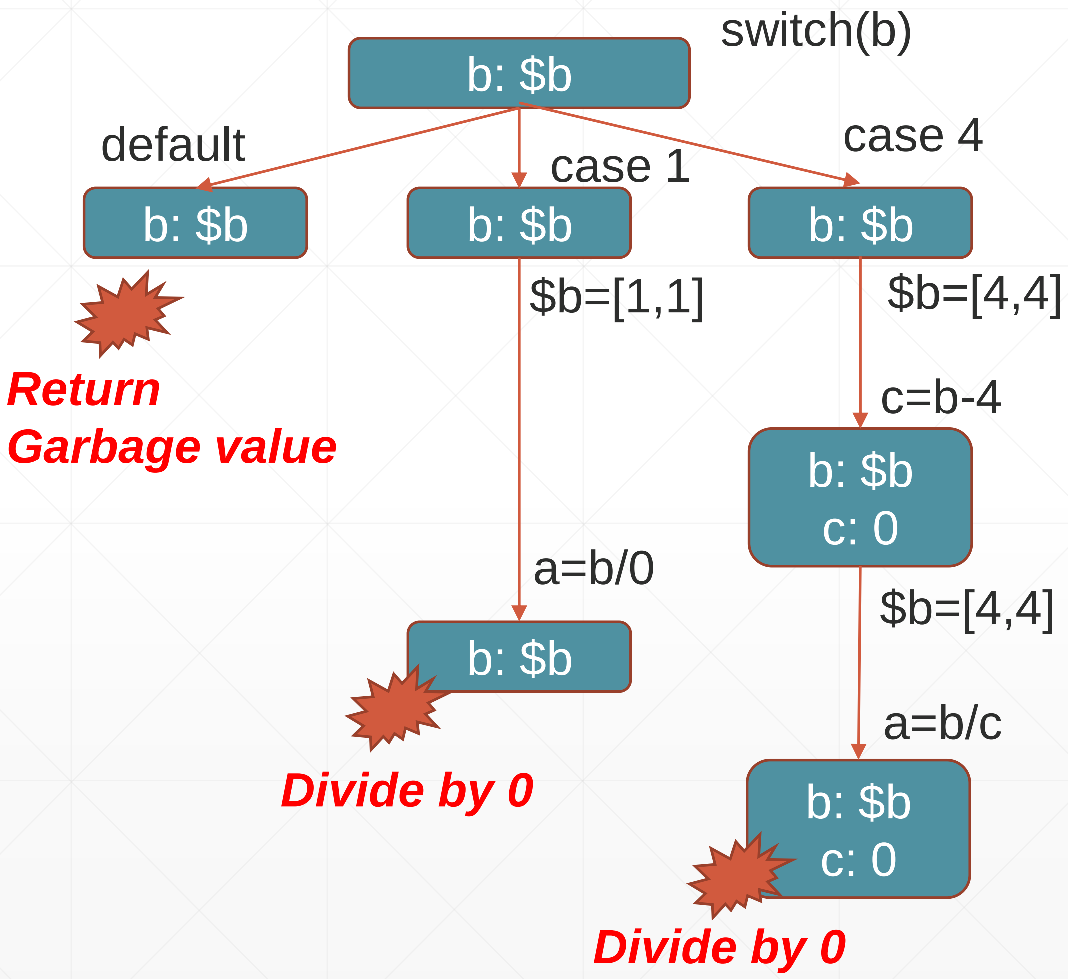
- The CSA performs context-sensitive, inter-procedural analysis
- Designed to be fast to detect common mistakes
- Speed comes at the expense of some precision
- Normally, clang static analysis works in the boundary of a single translation unit.
  - With additional steps and configuration, static analysis can use multiple translation units.

Clang Static Analyzer – Symbolic Execution

- Finds bugs without running the code
- Path sensitive analysis
- CFGs used to create exploded graphs of simulated control flows

Compiler warns here

```
int function(int b) {
    int a, c;
    switch (b) {
        case 1: a = b / 0; break;
        case 4: c = b - 4;
                a = b/c; break;
    }
    return a;
}
```



Source: Clang Static Analysis - Gabor Horvath - Meeting C++ 2016

Using the Clang Static Analyzer – Example 1

- Basic example ….
- $ clang --analyze div0.c
  - Runs the analyzer, outputs text report
- $ clang --analyze -Xclang -analyzer-output=html -o <output-dir> div0.c
  - Runs the analyzer on div0.c, outputs an HTML formatted “chain of reasoning” to the output directory.
  - cd to <output-dir>, firefox report* &

Using the Clang Static Analyzer – Example 2

- Basic example ….
- $ scan-build -V clang -c div0.c
  - Runs the analyzer on div0.c, brings up an HTML report

Clang Static Analyzer – Example 1

```
void f6(int x) {
    int a[4];
    if (x==5) {
        if (a[x] == 123) {}
    }
}
```

- Intra procedural
- Array index out of bounds.

Bug Summary

File: /home/vince/examples/check.c
Warning: line 6, column 18
The left operand of '==' is a garbage value due to array index out of bounds

Annotated Source Code

Press '?' to see keyboard shortcuts

Show analyzer invocation

☐ Show only relevant lines

```
1
2
3  void f6(int x) {
4      int a[4];
5      if (x==5) {
          [1] Assuming 'x' is equal to 5 →
          [2] ← Taking true branch →
6          if (a[x] == 123) {}
              [3] ← The left operand of '==' is a garbage value due to array index out of bounds
7      }
8  }
```

```
$ clang --analyze -Xclang -analyzer-output=html -o somedir check.c
check.c:6:18: warning: The left operand of '==' is a garbage value due to array index out of bounds [core.UndefinedBinaryOperatorResult]
        if (a[x] == 123) {}
            ~~~~ ^
1 warning generated.
```

Clang Static Analyzer – Example 2

```
1:
2: int foobar() {
3:       int i;
4:       int *p = &i;
5:       return *p;
6: }
```

- Intra procedural
- ‘i’ declared without an initial value
- ‘*p’, undefined or garbage value

Bug Summary

File: /home/vince/examples/check2.c
Warning: line 5, column 5
Undefined or garbage value returned to caller

Annotated Source Code

Press '?' to see keyboard shortcuts

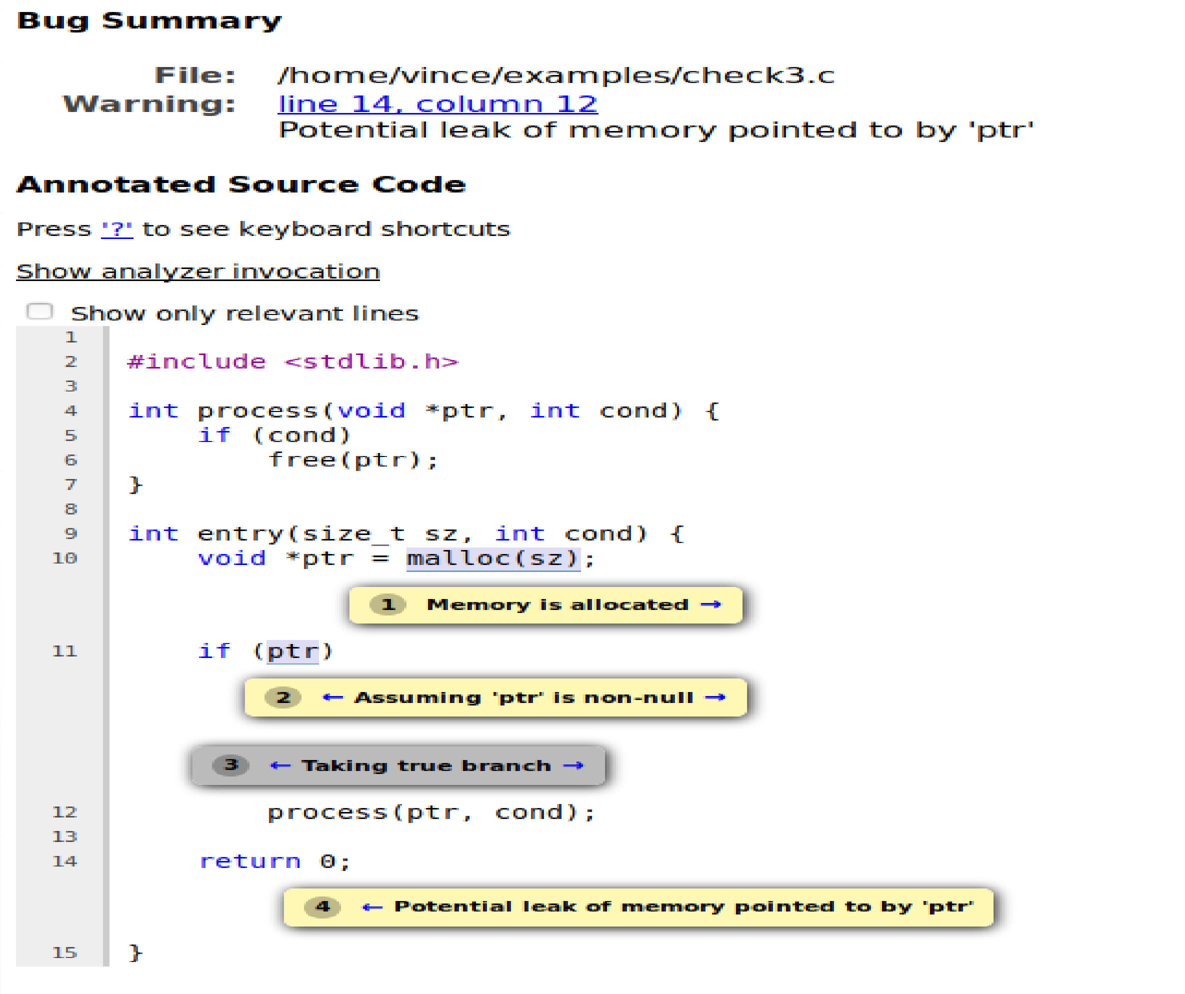
Show analyzer invocation

☐ Show only relevant lines

```
1
2  int foobar() {
3      int i;

       1  'i' declared without an initial value →

4      int *p = &i;
5      return *p;

       2  ← Undefined or garbage value returned to caller

6  }
```

Clang Static Analyzer – Example 3

```
 1:
 2: #include <stdlib.h>
 3:
 4: int process(void *ptr, int cond) {
 5:     if (cond)
 6:         free(ptr);
 7: }
 8:
 9: int entry(size_t sz, int cond) {
10:     void *ptr = malloc(sz);
11:     if (ptr)
12:         process(ptr, cond);
13:
14:     return 0;
15: }
```



- Analysis spans functions – said to be “inter-procedural”
- A Memory leak!

What about analyzing calls to external functions?

- These examples were single translation unit only.
  - In other words, in the same, single source file – “inter-procedural”, or inside of a single translation unit
- What if a function calls another function outside of it’s translation unit?
  - Referred to as “Cross translation Unit”
- Examples …

Cross Translation Unit Analysis

Main.cpp

```
int foo();
int main() {
    return 3/foo();
}
```

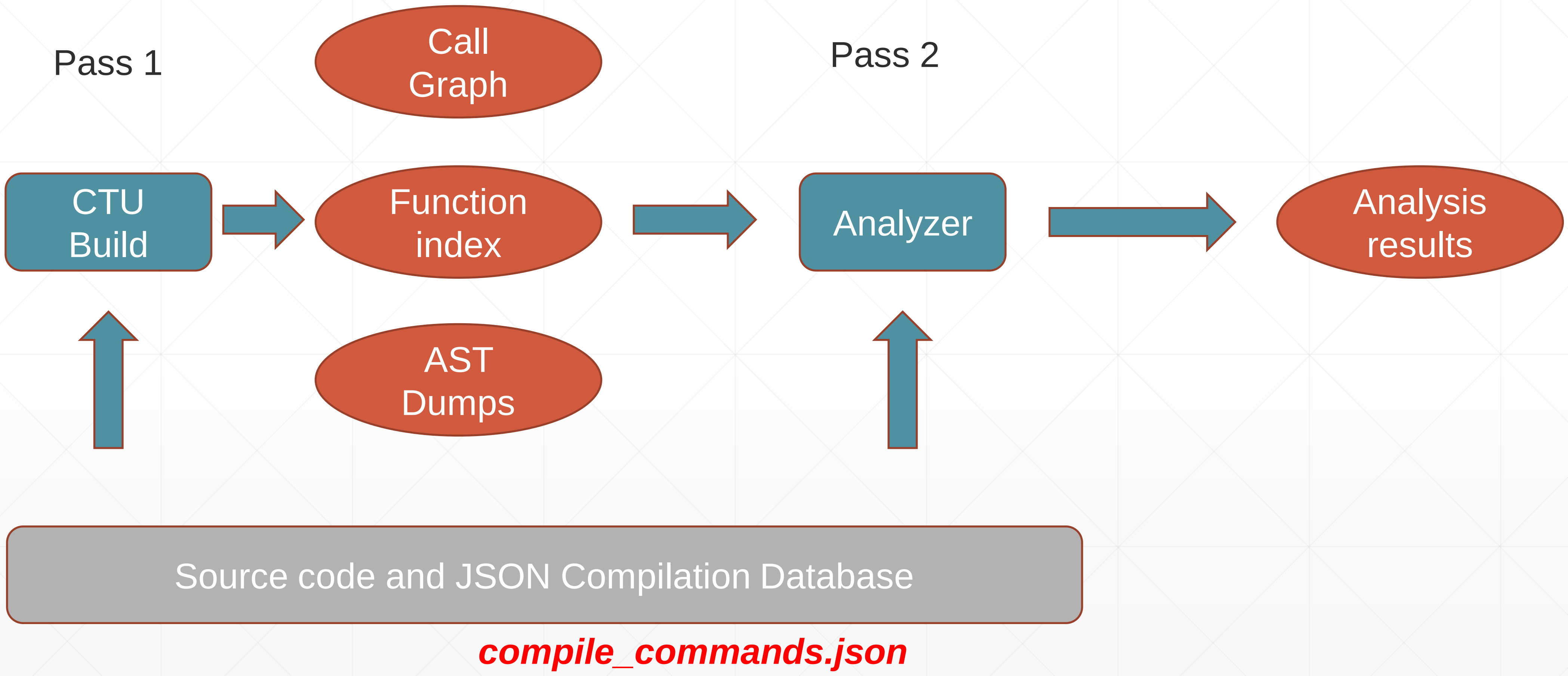
foo() is not known to be 0 without CTU

Foo.cpp

```
int foo() {
    return 0;
}
```

- CTU gives the analyzer a view across translation units
- Avoids false positives caused by lack of information
- Helps the analyzer constrain variables during analysis

How does CTU work?



Manual CTU – compile_commands.json

```
[
  {
    "directory": "<root>/examples/ctu",
    "command": "clang++ -c foo.cpp -o foo.o",
    "file": "foo.cpp"
  },
  {
    "directory": "<root>/examples/ctu",
    "command": "clang++ -c main.cpp -o main.o",
    "file": "main.cpp"
  }
]
```

- Mappings implicitly use the compile_commands.json file
- Analysis phase uses compile_command.json to locate the source files.

Source: https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html

Manual CTU - Demo

```
# Generate the AST (or the PCH)
clang++ -emit-ast -o foo.cpp.ast foo.cpp
# Generate the CTU Index file, holds external defs info
clang-extdef-mapping -p . foo.cpp > externalDefMap.txt

# Fixup for cpp -> ast, use relative paths
sed -i -e "s/.cpp/.cpp.ast/g" externalDefMap.txt
sed -i -e "s|$(pwd)/||g" externalDefMap.txt

# Do the analysis
clang++ --analyze \
    -Xclang -analyzer-config -Xclang experimental-enable-naive-ctu-analysis=true \
    -Xclang -analyzer-config -Xclang ctu-dir=. \
    -Xclang -analyzer-output=plist-multi-file \
    main.cpp
```

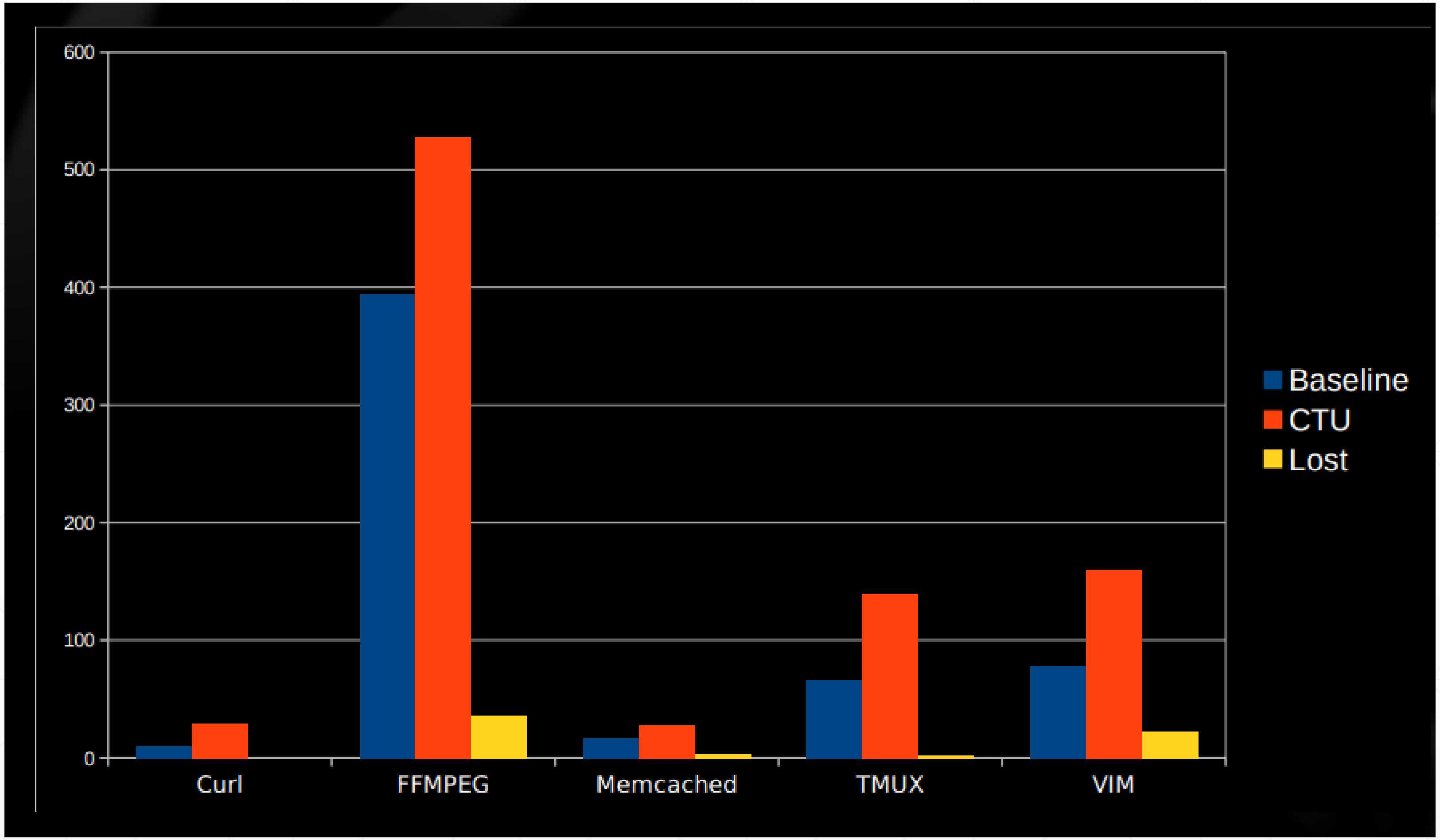
Using Cross Translation Unit Analysis

- scan-build.py within Clang can be used to drive Static Analysis on projects, scan-build is not actively maintained for Cross Translation Unit Analysis.
- Ericsson's Open Source CodeChecker tool supports CTU flows
- Let's see an example …

CodeChecker automates this process

```
# Create a compile.json
CodeChecker log –b "clang main.cpp foo.cpp" –o compile.json


# First, try without CTU
CodeChecker analyze –e default –clean compile.json –o result
CodeChecker parse result

# Add CTU
CodeChecker analyze –e default –ctu –clean compile.json –o result
CodeChecker parse result

# try with scan build
scan-build clang main.cpp foo.cpp
```

Benefits of CTU



- 2.4x Average
- 2.1x median
- 5x peak
- Note there are some lost defects when using CTU

See https://llvm.org/devmtg/2017-03//assets/slides/cross_translation_unit_analysis_in_clang_static_analyzer.pdf , https://www.youtube.com/watch?v=7AWgaqvFsgs

CSA Modeling Weaknesses

- CSA does a good job modeling program execution, but does have some weaknesses.
- CSA is built for speed, and common cases. The constraint solver gives up on some complex expressions when they appear with symbolic values.
- An example …

Example of unhandled bitwise operations

```
1: unsigned int func(unsigned int a) {
2:      unsigned int *z = 0;
3:      if ((a & 1) && ((a & 1) ^1))
4:           return *z; // unreachable
5:      return 0;
6: }
```

- This program is safe, albeit brittle

```
$ clang --analyze test.cpp
test.cpp:5:16: warning: Dereference of null pointer (loaded from variable 'z') [core.NullDereference]
        return *z;
               ^~
1 warning generated.
```

Z3 Refutation, preferred

```
$ clang --analyze -Xclang -analyzer-config -Xclang crosscheck-with-z3=true test.cpp
```

```
$ clang --analyze  -Xclang -analyzer-constraints=z3 func.c
```

Z3 constraint manager, slower

Source: Refuting false bugs in the clang static analyzer, Gadelha… https://www.youtube.com/watch?v=SO84AmbWiLA

Refuting False Positives with Z3

- CSA sometimes detects false positives because of limitations in the CSA constraint manager.
- Speed comes at the expense of precision -- symbolic analysis does not handle some arithmetic and bitwise operations. Z3 can compensate for some of these shortcoming.
- CodeChecker enables Z3 by default, if found.
- See https://github.com/Z3Prover/z3. Clang can be compiled to use Z3.

Why not just replace the CSA solver?

- First SMT backend solver (Z3) implemented in late 2017. It aimed to replace the CSA constraint solver.
- This solver was 20 times slower than the built in solver.
- A refutation approach gives us best of both worlds
  - Clang Static Analyzer’s Speed for common cases
  - A chance for a Z3 solver to refute bugs
- So, this is the approach for now

Putting it all together …

- How do we use everything we've learned to find some real bugs?
- Using LLVM/Clang "tip of tree", compiled with Z3 "tip of tree"
- Let's look at the "bitcoin curve" library https://github.com/bitcoin-core/secp256k1.git.
  - It's small enough to demonstrate, and does have some bugs CSA can find
- I'll demonstrate how to run Static Analysis on this code, and the differences in analysis results using Z3 and Cross Translation Unit Analysis
- I'll also demonstrate using Clang Static Analyzer on a well developed project, gzip

Results & Conclusion

- We found some real bugs in the “bit coin curve” library.
- Demonstrated how more bugs can be found, or refuted, using CTU and Z3
- Shown you how to make use of Clang tools to find real bugs

References

- Using scan-build https://clang-analyzer.llvm.org/scan-build.html
- Cross Translation Unit Analysis https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html
- CodeChecker https://github.com/Ericsson/codechecker
- Z3 Refutation in Clang - https://arxiv.org/pdf/1810.12041.pdf
- Implementation of CTU in Clang - https://dl.acm.org/doi/pdf/10.1145/3183440.3195041
- https://llvm.org/devmtg/2017-03//assets/slides/cross_translation_unit_analysis_in_clang_static_analyzer.pdf
- SMT based refutation of spurious bug reports in CSA - https://www.youtube.com/watch?v=WxzC_kprgP0
- “Bit coin curve” library - https://github.com/bitcoin-core/secp256k1.git
- Compile command JSON Specification https://clang.llvm.org/docs/JSONCompilationDatabase.html
- Z3 https://github.com/Z3Prover/z3
- Tutorial Source - https://github.com/vabridgers/LLVM-Virtual-Tutorial-2020.git

Thank you for attending!

Demo notes

- git clone https://github.com/Z3Prover/z3.git
- cd z3; mkdir build; cd build
- cmake -G Ninja ../ ; ninja ; sudo ninja install # assumes installed at /usr/local/lib/libz3.so
- CodeChecker pulled/installed from https://github.com/Ericsson/CodeChecker.git
  - Be sure to set “CC_ANALYZERS_FROM_PATH=1”, set PATH to your clang
- Bit coin curve library git clone https://github.com/bitcoin-core/secp256k1.git
- Gzip https://git.savannah.gnu.org/git/gzip.git

- Run scan-build -> “scan-build make”
- CodeChecker command notes …
  - CodeChecker log –b “make” –o compile_commands.json
  - CodeChecker analyze –e default –clean –j 16 compile_commands.json –o outputdir
  - CodeChecker analyze –e default –ctu –clean –j 16 compile_commands.json –o outputdir
  - CodeChecker analyze –e default –ctu –z3-refutation off –clean –j 16 compile_commands.json –o outputdir
  - CodeChecker parse –e html –o html-output-dir outputdir