Austin LLVM Meetup

Vince Bridgers

6/24/2020

Meetup Agenda

- Today
  - LLVM Walkthrough – How to build
  - Git clone llvm/clang and testing
  - Static Analysis
- Sponsorship
  - Our space is sponsored by Capital Factory
  - Meetup fees are paid by llvm.org – thank you, Arnaud de Grandmaison!

Meetup Schedule, Space

- 4th Wednesday every month
- We’re trying “virtual” for this meeting
  - Thanks for your patience!

About this tutorial

- LLVM/Clang, Z3 Introduction
- Basics of How to build – for the examples shown
- Mini introduction to LLVM IR
- Example usages of Clang
- Static Analysis
- Benefits of CTU and Z3, how to use
- A few known issues with Clang Static Analysis

What is LLVM?

- A collection of reusable compiler and toolchain libraries
- LLVM is not a compiler! Clang is built on LLVM for example
- Compilers using LLVM
  - Clang, Swift, Rust, Haskell
- LLVM's IR – Intermediate Representation – was one of it's great innovations. I'll touch upon that today
  - LLVM was "Low Level Virtual Machine" – now just LLVM
- Great for academic research into technologies related to compilers, researching new compiler passes and improvements

Comparing LLVM and GCC

- Difficult to compare, came from different origins
- Structured differently - GCC is monolithic, LLVM/Clang is modular
- GCC is difficult to extend. LLVM/Clang can be improved and extended by improving and extending the interfaces.
- GCC supports more traditional languages such as Ada, Fortran and Go.
- GCC supports less popular architectures and more assembly language features than Clang. GCC is still the only option for compiling the Linux kernel.
- Clang provides many additional useful tools, such as static analysis, clang-format, clang-tidy, refactoring tools, and IDE plugins
- Clang was designed from the very beginning to provide accurate and user friendly error messages.

Demo - Environment, Tools

- Oracle VirtualBox VM, Ubuntu 18.04 on Win10
- 16G memory, 100GB storage, 4 processors
- Bidi sharing enabled, Guest Additions added
- sudo apt install gcc g++ cmake ninja-build python

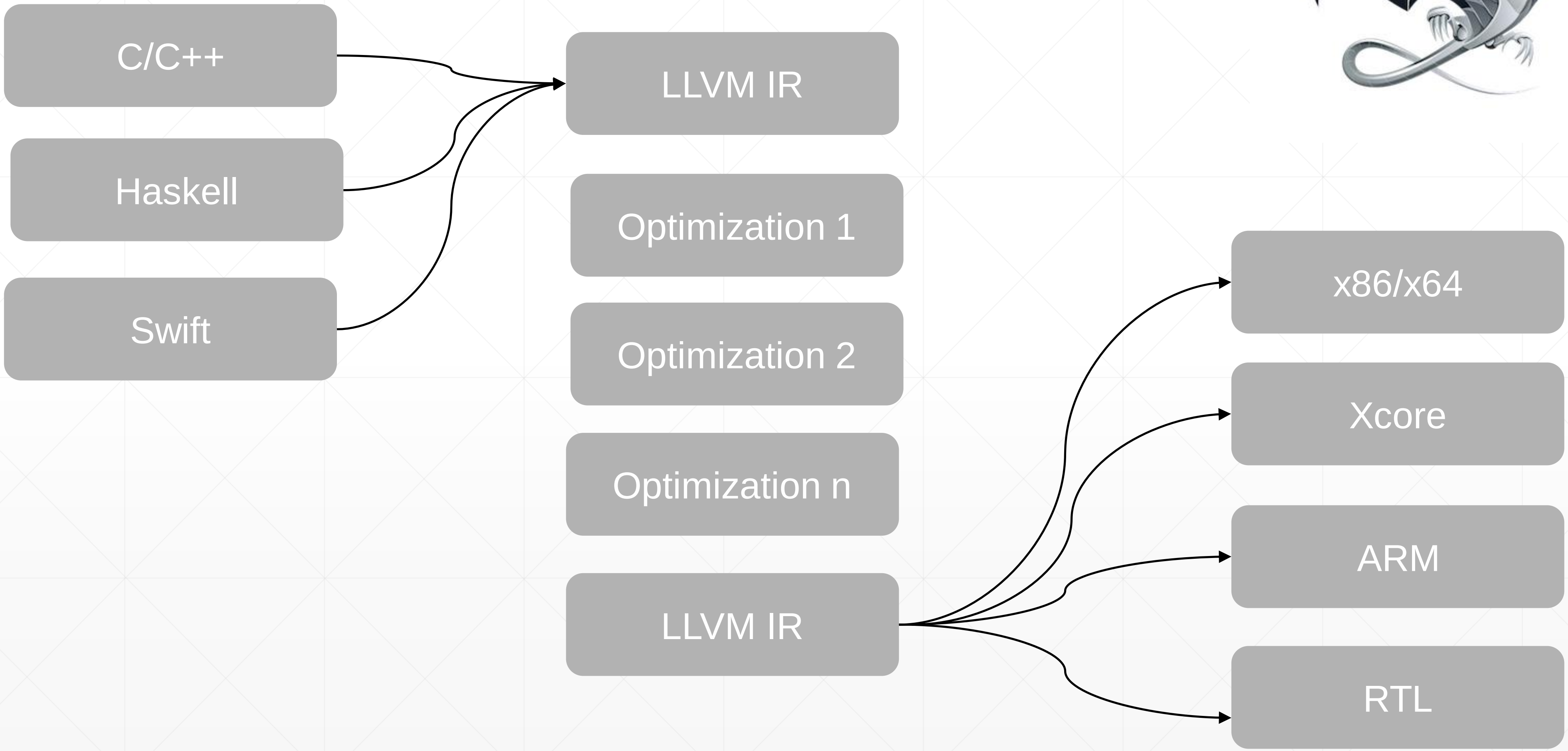
Z3 – an SMT Solver to Enhance Static Analysis

- Z3 is a cross platform, satisfiability modulo theories (SMT) solver by Microsoft
- Helps solves problems that arise in software verification and analysis. We'll use Z3 in static analysis.
- Bindings for various programming languages include C, C++, Java, and Python among others.
- Pull – git clone https://github.com/Z3Prover/z3.git
- Build – cd z3; rm –rf build; git clean –nx src; mkdir build; cd build
- Build … - cmake –DCMAKE_INSTALL_PREFIX=<root>/z3-install –DCMAKE_EXE_LINKER_FLAGS:STRING=-Wl,-R<root>/z3-install/lib –DCMAKE_BUILD_TYPE=Release –G Ninja ../
- Build … - ninja; ninja install

Start “git clone” & build

- git clone https://github.com/llvm/llvm-project.git
  - mkdir llvm-install; mkdir llvm-build; cd llvm-build
- X86
  - cmake -DLLVM_ENABLE_ASSERTIONS=On -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" -DLLVM_TARGETS_TO_BUILD="X86" –DLLVM_Z3_INSTALL_DIR= <root>/z3-install –DLLVM_ENABLE_Z3_SOLVER=ON –DCMAKE_EXE_LINKER_FLAGS:STRING=-Wl,-R<root>/z3-install/lib -DCMAKE_INSTALL_PREFIX=<root>/llvm-install -G Ninja <root>/llvm-project/llvm
  - ninja; ninja install; ninja check-all

Compiler Flow



IR Representation

llvm-dis

Bitcode file: *.bc

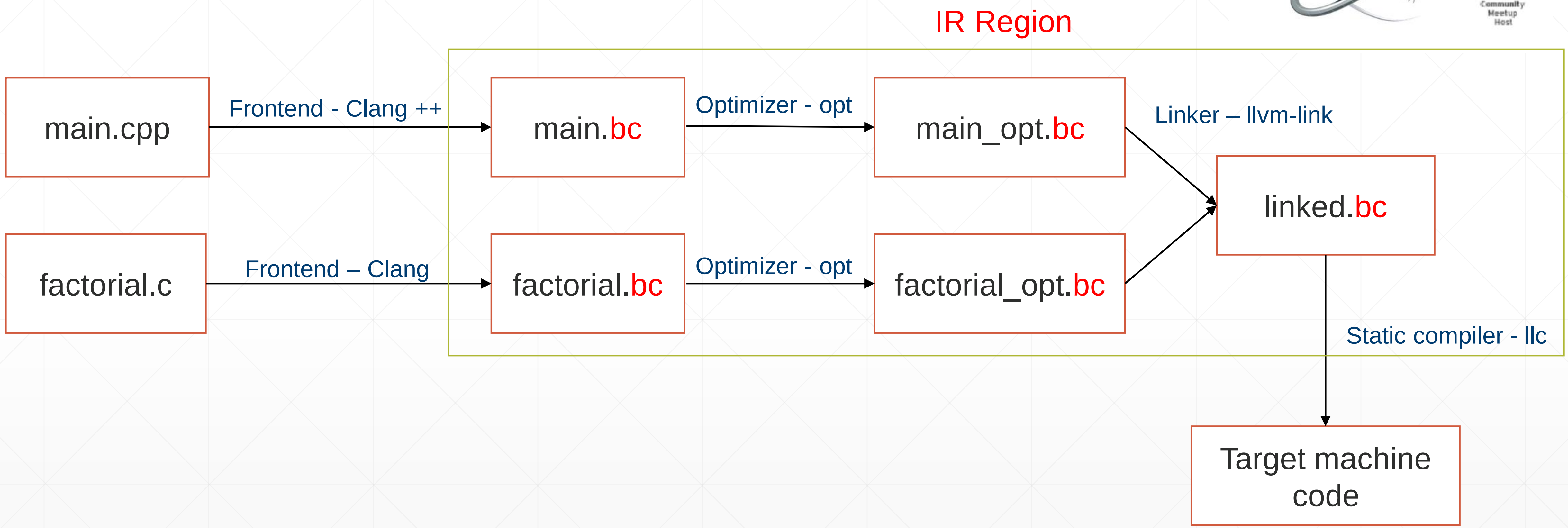
```
000101010101010000101010101010
000101010101010010111010010110
000000000000000000101010101010
000101010101010000101010101010
010111010010110000000000000000
```

Human-readable file: *.ll

```
def void @foo(i32 %arg) {
            ; You can read me!
            ret void
}
```

llvm-as

IR and the Compilation Process



A basic main program

Hand-written IR for this program:

```c
int factorial(int val);

int main(int argc, char** argv)
{
  return factorial(2) * 7 == 42;
}
```

```llvm
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

% Virtual Registers %

Those are “local” variables.

Two flavors of names:

- Unnamed: %<number>
- Named: %<name>

“LLVM IR has infinite registers”

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

Types everywhere!

Very much a typed language.

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

The LangRef is your friend

‘call’ Instruction

Syntax: `%1 = call i32 @factorial(i32 2)`

```
<result> = [tail | musttail | notail ] call [fast-math flags] [cconv] [ret attrs] [addrspace(<num>)]
           [<ty>|<fnty> <fnptrval>(<function args>) [fn attrs] [ operand bundles ]
```

Overview:

The ‘call’ instruction represents a simple function call.

Arguments:

This instruction requires several arguments:

Basic Blocks

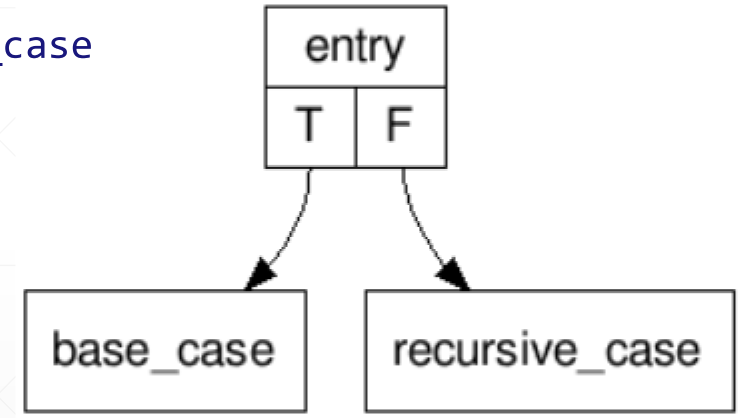
List of non-terminator instructions ending with a terminator instruction:

- **Branch - “br”**
- **Return - “ret”**
- Switch – “switch”
- Unreachable – “unreachable”
- Exception handling instructions

```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_cas
base_case:
  ret i32 1
recursive_case:
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %0)
  %3 = mul i32 %val, %1
  ret i32 %2
}
```

Control Flow Graph

```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:                  ; preds = %entry
  ret i32 1
recursive_case:             ; preds = %entry
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %0)
  %3 = mul i32 %val, %1
  ret i32 %2
}
```



CFG for 'ir_factorial' function

Automatically generated comments

Compiling a Simple Source file to IR

```
int main(void) {
      int a = 32;
      int b = 10;
      return a + b;
}
```

```
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 32, i32* %a, align 4
  store i32 10, i32* %b, align 4
  %0 = load i32, i32* %a, align 4
  %1 = load i32, i32* %b, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}
```

- $ clang -S -emit-llvm main.c
- Produces unoptimized IR

Compiling a Simple Source file to IR

```c
int main(void) {
      int a = 32;
      int b = 10;
      return a + b;
}
```

```llvm
define dso_local i32 @main()
local_unnamed_addr #0 {
entry:
  ret i32 42
}
```

Constant Folding. The compiler recognized a, b, and a+b as constants and folded the result to 42. The answer to everything!

- $ clang -S -emit-llvm –O3 main.c
- Produces optimized IR

Finding bugs with the Compiler

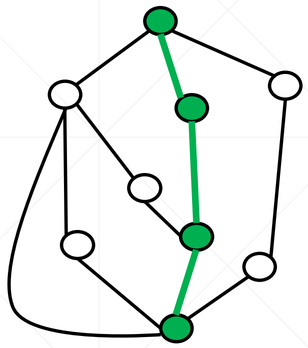
```
1: #include <stdio.h>
2: int main(void) {
3:     printf("%s%lb%d", "unix", 10, 20);
4:     return 0;
5: }
```

```
$ 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.

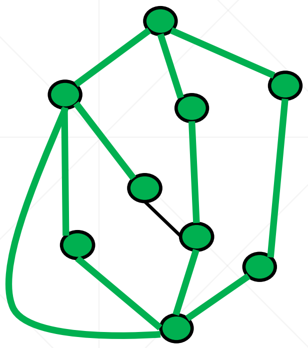
Program Analysis vs Testing

- Testing usually tests a small number of paths in the program.
- May miss errors
- It’s fast, but real coverage can be sparse
- Same is true for other useful 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
- Mitigation techniques …
  - Bounded model checking – breadth-first search approach
  - Depth-first search for symbolic execution



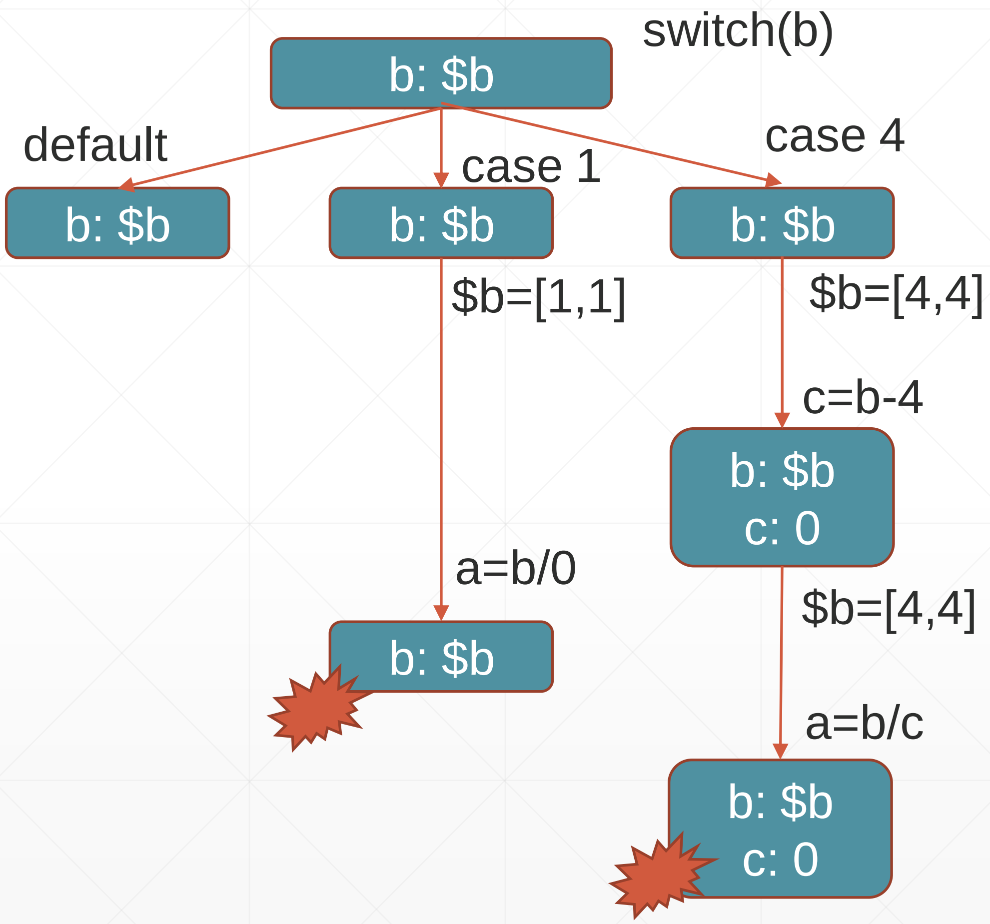
Clang Static Analyzer (CSA)

- The CSA is project built on top of clang that performs context-sensitive interprocedural analysis for programs written in the languages supported by clang.
- Designed to be fast to detect common mistakes such as divide by zero, or null pointer dereferences, even in complex projects.
- Speed comes at the expense of precision, so does not handle some arithmetic (e.g. remainder) and bitwise operations. Z3 can compensate for some of these shortcoming.
- Normally, static analysis works in the boundary of a single translation unit. With additional steps and configuration, static analysis can utilize multiple translation units.
- Memory modeling is sufficient for many issues, but also has a few incomplete areas due to the performance/function tradeoffs taken. I’ll demonstrate the opportunities for improvement.
- Early discovery of bugs, when cheaper to fix

Clang Static Analyzer – Symbolic Execution

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

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



Using the Clang Static Analyzer

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

Clang Static Analyzer – Example 1

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

- 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: }
```

- ‘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 “interprocedural”
- A Memory leak!

Bug Summary

File: /home/vince/examples/check3.c
Warning: line 14, column 12
Potential leak of memory pointed to by 'ptr'

Annotated Source Code

Press '?' to see keyboard shortcuts

Show analyzer invocation

☐ Show only relevant lines

```
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);
            1 Memory is allocated →
11      if (ptr)
            2 ← Assuming 'ptr' is non-null →
            3 ← Taking true branch →
12          process(ptr, cond);
13
14      return 0;
            4 ← Potential leak of memory pointed to by 'ptr'
15  }
```

Using the Clang Static Analyzer

- These examples were single translation unit only.
  - In other words, in the same, single source file – “interprocedural”
- 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();
}
```

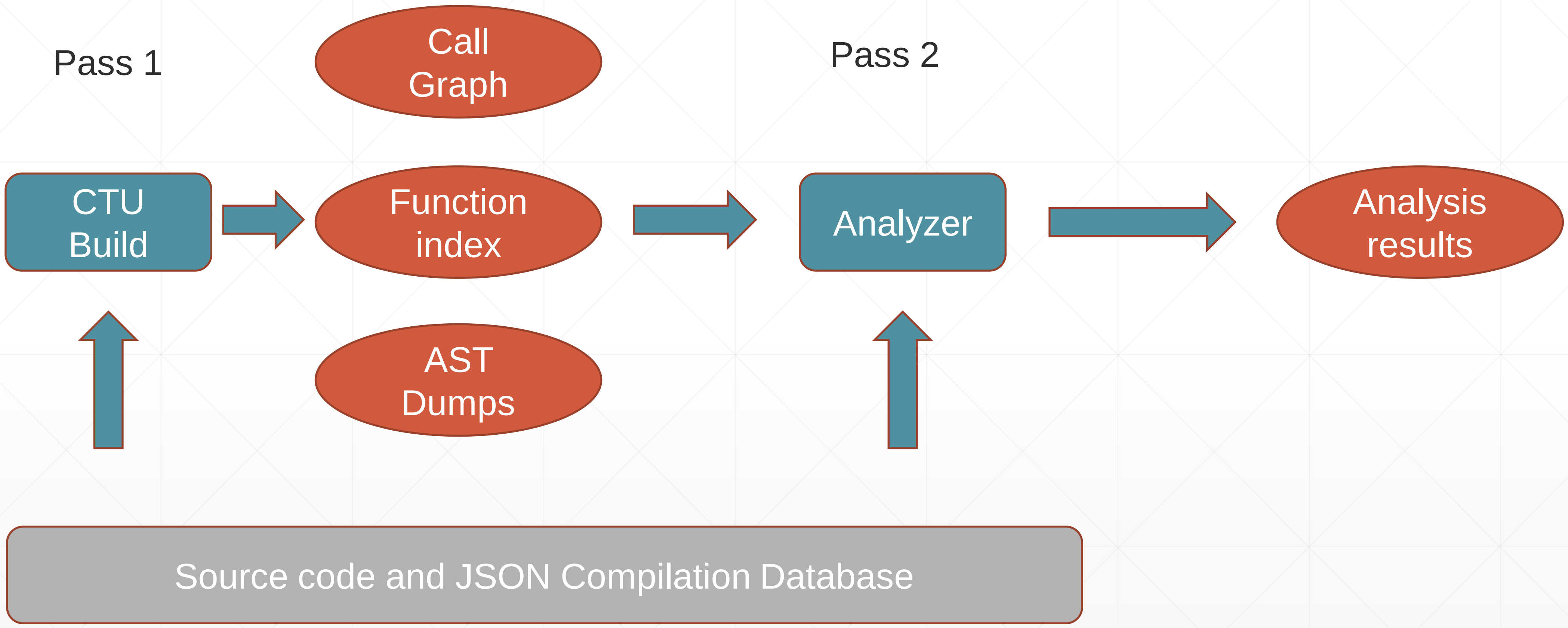
foo() is not known to be 0 without CTU

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.

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
```

Cross Translation Unit Analysis – Example 2

Main.cpp

```
void neg(int *x);
Void g(int *x) {
    if (*x>0){
        neg(x);
    if (*x>0){
        *x/0;
    neg(NULL);
}
```

*x is positive

*x is unknown

false positive

API misuse

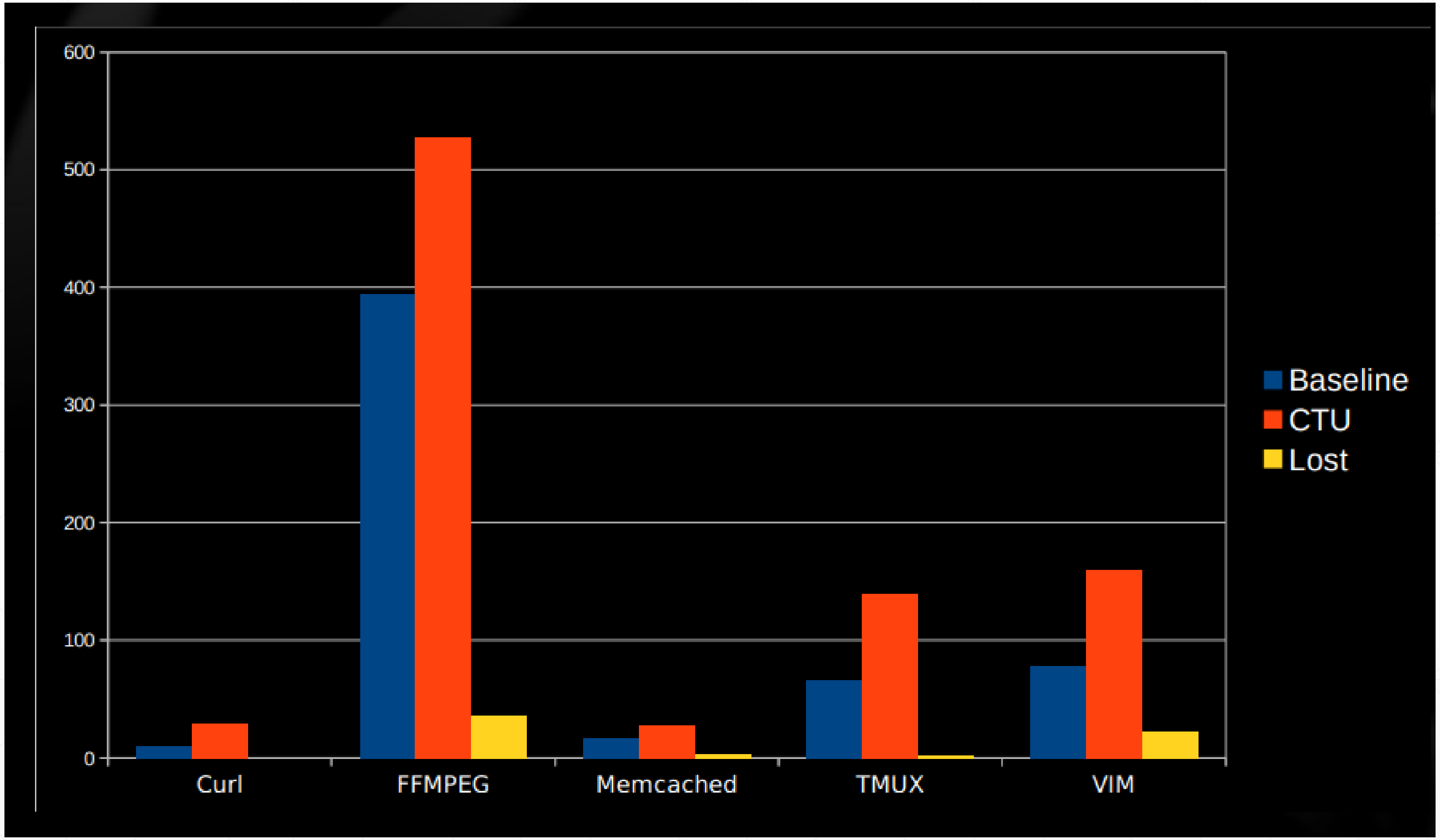
Foo.cpp

```
void neg(int *) {
    *x = -(*x);
}
```

Using Cross Translation Unit Analysis

- scan-build.py within Clang can be used to drive Static Analysis on projects, scan-build is NOT actively developed for CTU.
  - Don't try it, you'll probably encounter a lot of pain ☺
- Ericsson's CodeChecker tool supports CTU flows
- It's possible to develop your own – but why? CodeChecker is open source ☺
- One could always try to upstream fixes to scan-build.py ☺

Some statistical benefits of CTU



- 2.4x Average
- 2.1x median
- 5x peak
- Note there are some lost defects when using CTU
- For this reason, w/ and w/o CTU runs are recommended

See https://llvm.org/devmtg/2017-03//assets/slides/cross_translation_unit_analysis_in_clang_static_analyzer.pdf

Refuting some False Positives with Z3

- CSA sometimes detects false positives because of limitations in the CSA constraint manager.
- Speed comes at the expense of precision, so does not handle some arithmetic (e.g. remainder) and bitwise operations. Z3 can compensate for some of these shortcoming.
- See https://github.com/Z3Prover/z3. Clang can be compiled to use Z3. They are separated because of licensing issues.

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;
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.
```

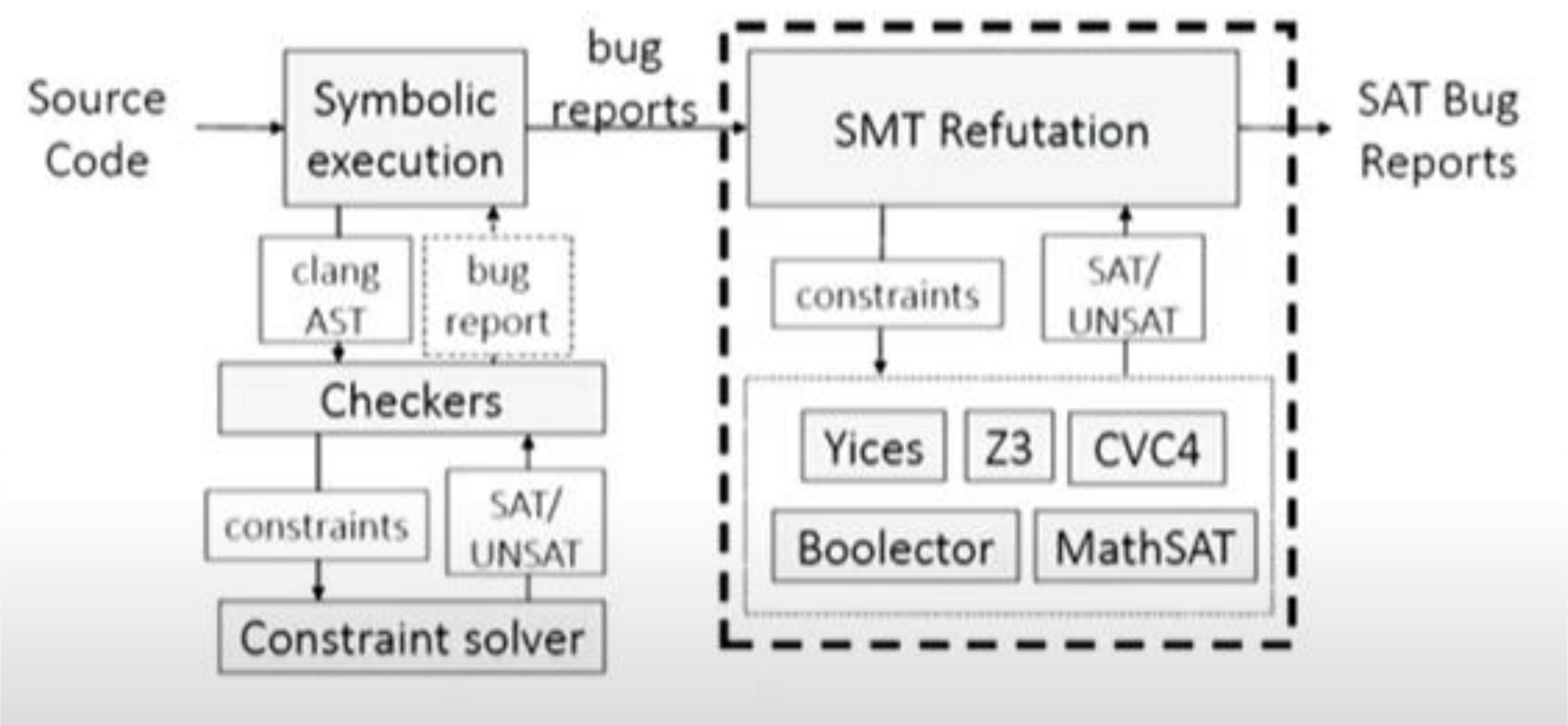
Z3 Refutation, preferred

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

```
$ clang --analyze  -Xclang -analyzer-constraints=z3 test.cpp
```

Z3 constraint manager, slower

False positives – Z3 Refutation



Benefits of Z3

Project	Analysis time (no refutation)	Analysis time (+ refutation)	Number of bugs (no refutation)	Number of refuted bugs
tmux	90.35	96.28	19	0
redis	328.85	316.22	86	1
openSSL	128.67	119.12	36	2
twin	205.86	207.87	52	0
git	384.00	356.75	69	11
postgreSQL	1024.68	1074.73	184	5
SQLite3	1016.38	1057.99	82	14
curl	80.68	81.56	39	0
libWebM	41.15	42.10	6	0
memCached	89.41	100.22	25	0
xerces-C++	408.91	387.17	58	2
XNU	3564.91	3464.46	441	49

See https://arxiv.org/pdf/1810.12041.pdf

Why not just replace the CSA solver?

- First SMT backend solver (Z3) implemented in late 2017 by Dominic Chan. 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
  - Speed for common cases
  - A chance for a Z3 solver to refute bugs
- So, this is the approach for now

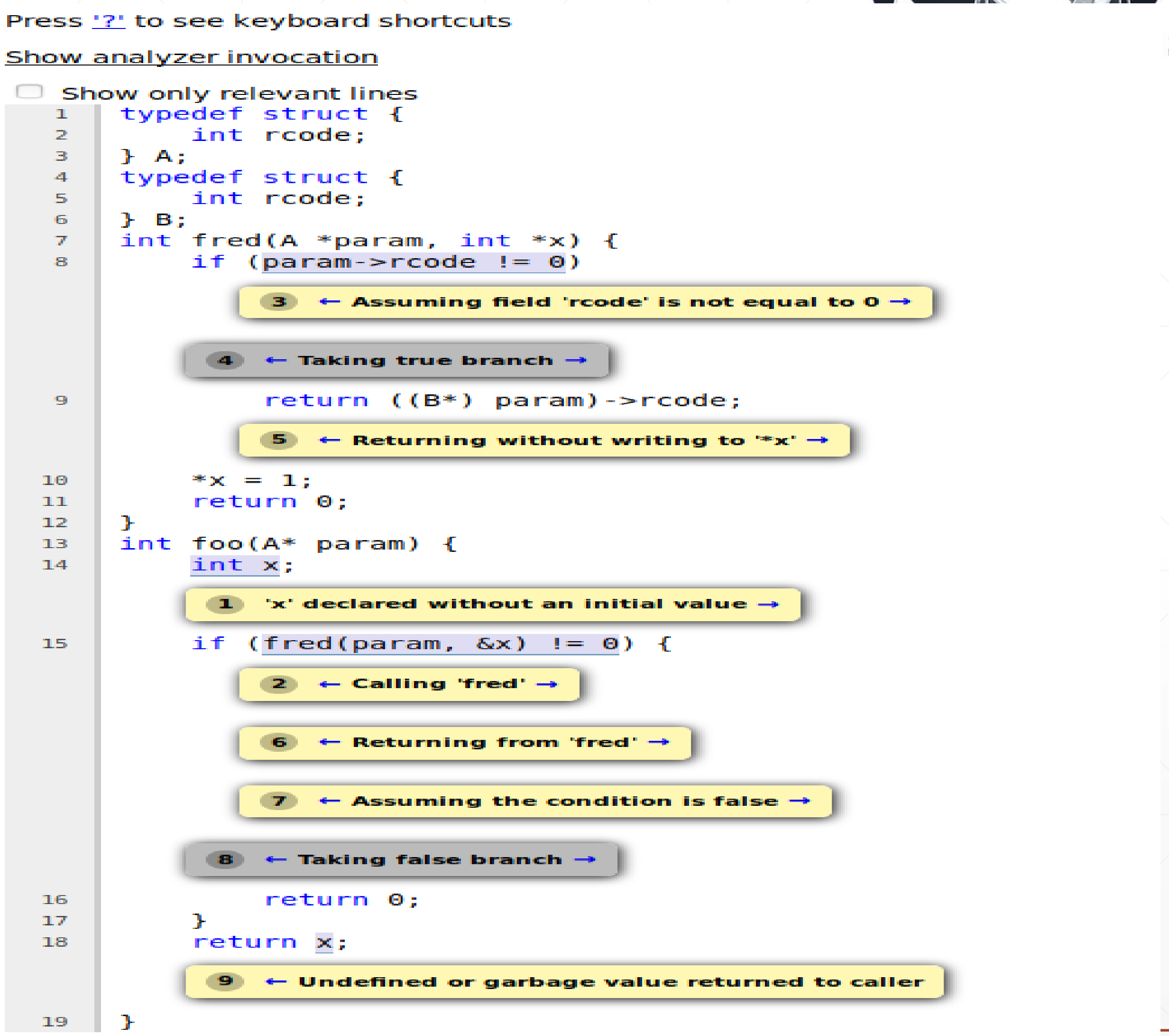
CSA Memory Modeling Weaknesses

- CSA does a good job addressing many memory modeling problems during static analysis, but does suffer from some technical debt in it's implementation.
- It's difficult to know for certain if this was a mindful design choice, or an oversight – at any rate – doesn't matter ☺ . The term "technical debt" implies this was an oversight, but performance is sometimes quoted as the reason. Meh ?
- For a list of technical debt in the CSA, see https://clang-analyzer.llvm.org/open_projects.html.
- A list of Bugzilla bugs related to the Clang Static Analyzer
- Some examples …

Bugzilla link - https://bugs.llvm.org/buglist.cgi?component=Static%20Analyzer&list_id=147756&product=clang&resolution=---

CSA Weakness – Example 1

```c
typedef struct {
      int rcode;
} A;
typedef struct {
      int rcode;
} B;
int fred(A *param, int *x) {
      if (param->rcode != 0)
            return ((B*) param)->rcode;
      *x = 1;
      return 0;
}
int foo(A* param) {
      int x;
      if (fred(param, &x) != 0) {
            return 0;
      }
      return x;
}
```



- Casts are not recognized in some cases.

CSA Weakness – Example 2

```
 1: struct s1 {
 2:     unsigned short u16;
 3: };
 4: int main(void) {
 5:     struct s1 s1var = { 0x1122 };
 6:     char *p = &s1var;
 7:     int x = 0;
 8:     if (p[1])
 9:         x++;
10:     return x;
11: }
```

- p[1] is not recognized by the CSA as defined.

Bug Summary

File: /home/vince/examples/false-positives/case1.c
Warning: line 8, column 9
Branch condition evaluates to a garbage value

Annotated Source Code

Press '?' to see keyboard shortcuts

Show analyzer invocation

☐ Show only relevant lines

```
1  struct s1 {
2      unsigned short u16;
3  };
4  int main(void) {
5      struct s1 s1var = { 0x1122 };
6      char *p = &s1var;
7      int x = 0;
8      if (p[1])
           Branch condition evaluates to a garbage value
9          x++;
10     return x;
11 }
```

CSA Weakness – Example 3

```
 1: unsigned long getV();
 2: void foo() {
 3:      unsigned long len = getV();
 4:      void *ptrs[len*2];
 5:
 6:      for (int i=0; i<len*2; i++) {
 7:           ptrs[i] = 0;
 8:      }
 9:
10:      for (int i=0; i<len; i++)
11:           if (ptrs[i] == ptrs[i+len])
12:                return;
13: }
```

- Evaluation of Symbolic Values in conditionals is not handled in all cases.

References

- 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

Links

- https://www.meetup.com/Austin-LLVM-Meetup/
- https://www.capitalfactory.com/ Capital Factory
- https://docs.google.com/document/d/1bqms0I4u9-sNdHWfK5c76-iu1Q7DvZTHXIxmq74-8aA/edit LLVM Meetup suggestions
- https://berlincodeofconduct.org/ Berlin Code of Conduct
- https://llvm.org/pubs/2002-12-LattnerMSThesis.pdf

Thank you for attending!