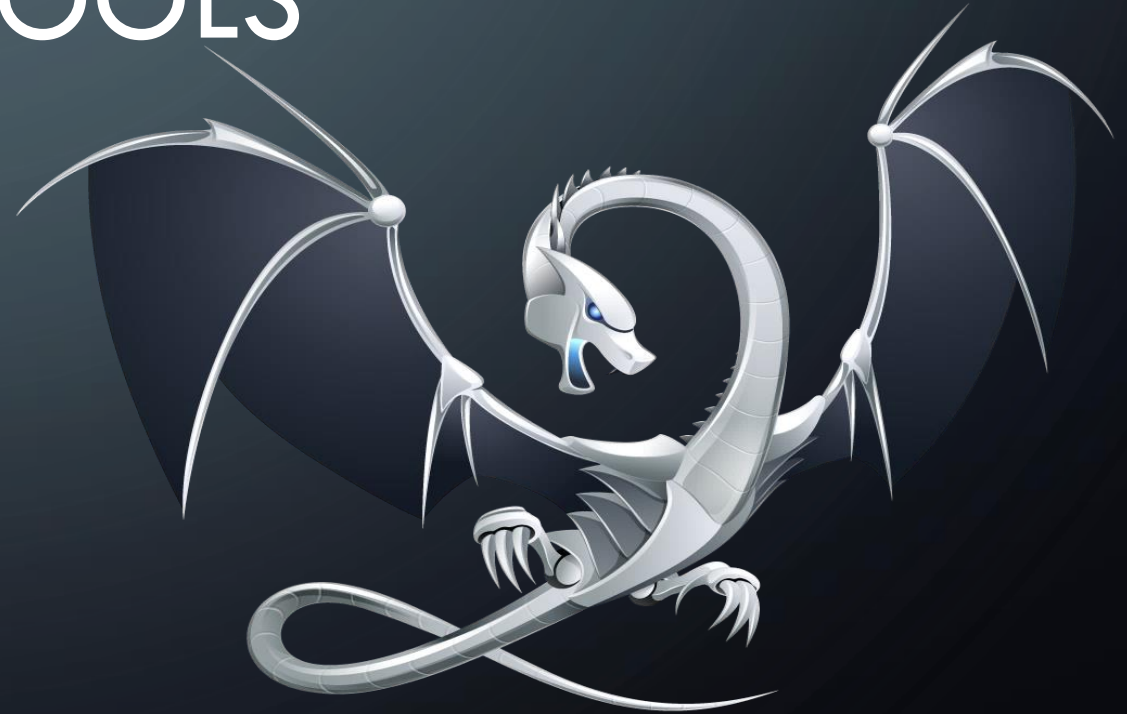



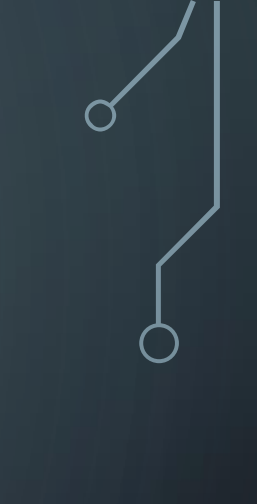
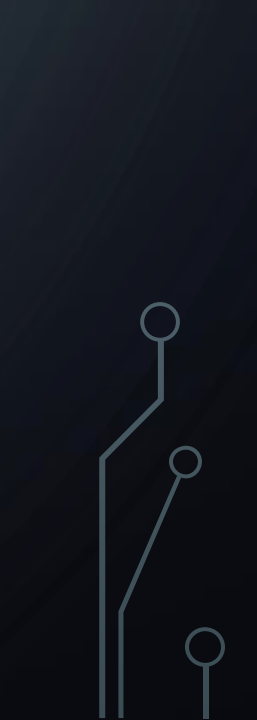
MAKE FRIENDS WITH THE CLANG STATIC ANALYSIS TOOLS

GABOR HORVATH



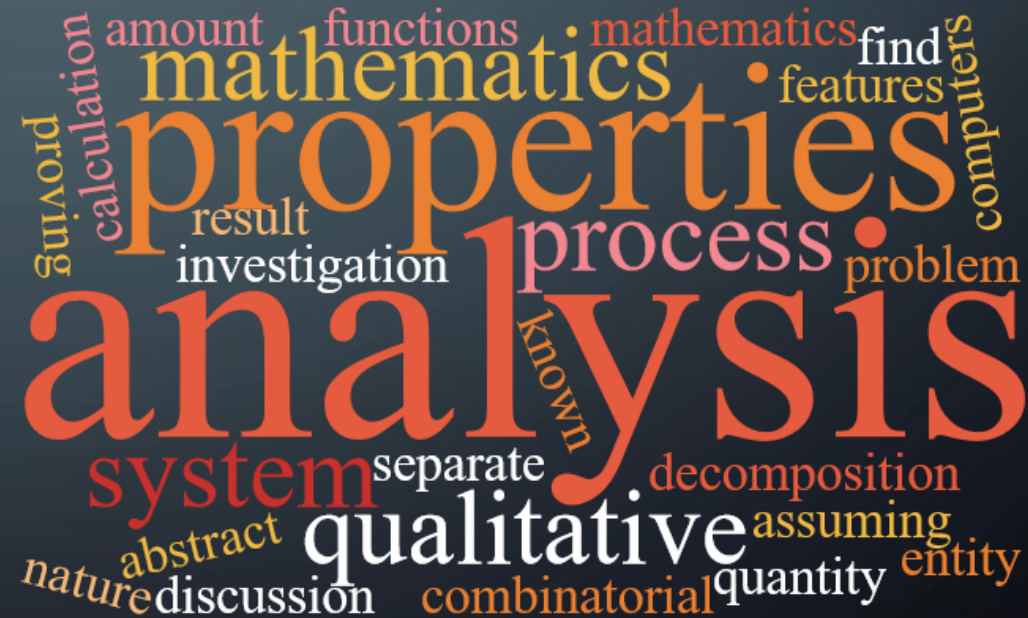


OUTLINE

- What is static analysis? What it is good for? Why is it hard?
 - How does it work?
 - What open source tools are available?
 - How to integrate them into the workflow?
 - How to write static analysis friendly code?
- 
- 
- 

STATIC ANALYSIS

- Analyze the code without execution
 - Optimization
 - Code Transformation
 - **Bug finding**
 - Verification
 - Visualization
 - Metrics
 - ...



STATIC ANALYSIS

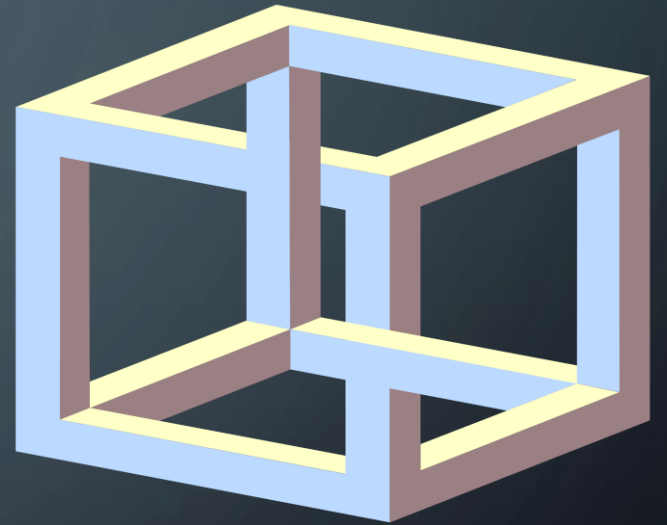
- Answer questions about the code
 - Terminates?
 - Maximum heap size?
 - What is the output for a given input?
 - Can this pointer be null?
- During development we do a fair amount of static analysis manually!



HALTING PROBLEM

- Does a program terminate on a given input?
 - Undecidable!
- Rice's theorem from '53:
 - Every interesting question about a program is undecidable!

```
int x = 5;  
if (program p halts on input i)  
    ++x;
```



APPROXIMATION

- Cannot solve undecidable problems
- We can approximate, employ heuristics
 - False positives (false reports)
 - False negatives (unreported errors)
- **Verification:** eliminate false negatives
- **Industrial use:** reduce false positives



WHY BOTHER WITH STATIC ANALYSIS?

- No execution required
 - Works when code coverage is suboptimal
 - Works when the runtime environment is not available
- Enforce coding guidelines more rigorously
- Cheap supplement to testing
- Find bugs early in the development cycle → cheaper to fix

ANALYZING C++ IS HARD

- The compilation model implies a lot of reparsing!
 - Modules to the rescue?
- Type information required for parsing
 - Parsing and semantic analysis cannot be split
- Macros make it hard to do semantics preserving rewrites or caching representations of code in header files!

```
T * t;  
S (x) ;  
f ( (R) *x) ;
```


STATIC ANALYSIS METHODS

- Textual pattern matching – CppCheck
- Matching the Syntax Tree – CppCheck, Clang Tidy
- Flow sensitive algorithms – Some compiler warnings
- Path sensitive algorithms – Clang Static Analyzer

TEXTUAL PATTERN MATCHING

- Transform code to canonical form
- Match patterns on the transformed code

`Token::Match(tok, " / 0");`

```
#define M 0
int a = 5 / M;
int b = getValue();
if (b == 4) {
    ...
    int c = 5 / (b - 4);
}
```

TEXTUAL PATTERN MATCHING

- Transform code to canonical form
- Match patterns on the transformed code

`Token::Match(tok, " / 0");`

```
#define M 0
int a = 5 / M;
int b = getValue();
if (b == 4) {
    ...
    int c = 5 / (b - 4);
}
```

TEXTUAL PATTERN MATCHING

- Transform code to canonical form
- Match patterns on the transformed code

`Token::Match(tok, " / 0");`

```
#define M 0
int a = 5 / M;
int b = getValue();
if (b == 4) {
    ...
    int c = 5 / (b - 4);
}
```

MATCHING THE SYNTAX TREE

FunctionDecl main

-CompoundStmt

-DeclStmt

-VarDecl d 'double *'

-BinaryOperator '='

-DeclRefExpr 'double *' 'd'

-CStyleCastExpr 'double *'

-CallExpr

-ImplicitCastExpr

-DeclRefExpr 'malloc'

-ImplicitCastExpr 'int'

-UnaryExprOrTypeTraitExpr sizeof

-ParenExpr

-DeclRefExpr 'double *' 'd'

```
int main() {  
    double *d;  
    d = (double*) malloc(sizeof(d));  
}
```

- Type information is available
- Typedefs, overloads, templates are resolved

MATCHING THE SYNTAX TREE

FunctionDecl main

 -CompoundStmt

 -DeclStmt

 -VarDecl d 'double *'

 -BinaryOperator '='

 -DeclRefExpr 'double *' 'd'

 -CStyleCastExpr 'double *'

 -CallExpr

 -ImplicitCastExpr

 -DeclRefExpr 'malloc'

 -ImplicitCastExpr 'int'

 -UnaryExprOrTypeTraitExpr sizeof

 -ParenExpr

 -DeclRefExpr 'double *' 'd'

```
int main() {  
    double *d;  
    d = (double*) malloc(sizeof(d));  
}
```

- Type information is available
- Typedefs, overloads, templates are resolved

FLOW SENSITIVE ALGORITHMS

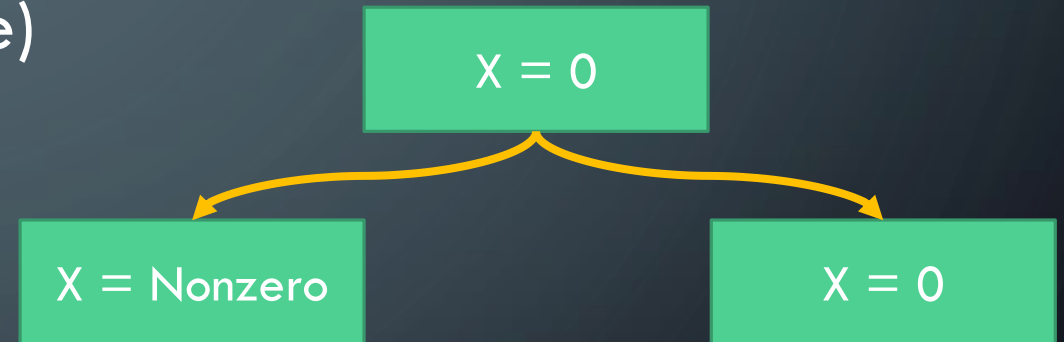
- Walk on the control flow graph
- What to do on merge points? (Lattice)

```
int x = 0;  
if (z) {  
    x = 5;  
}  
if (z) {  
    y = 10 / x;  
}
```


FLOW SENSITIVE ALGORITHMS

- Walk on the control flow graph
- What to do on merge points? (Lattice)

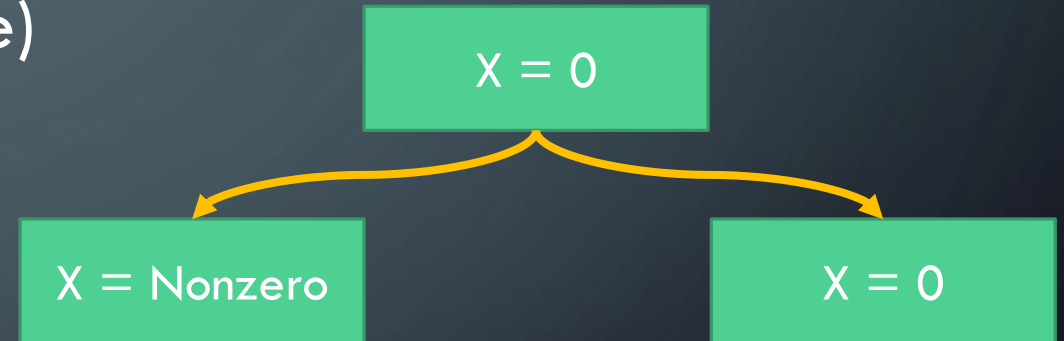
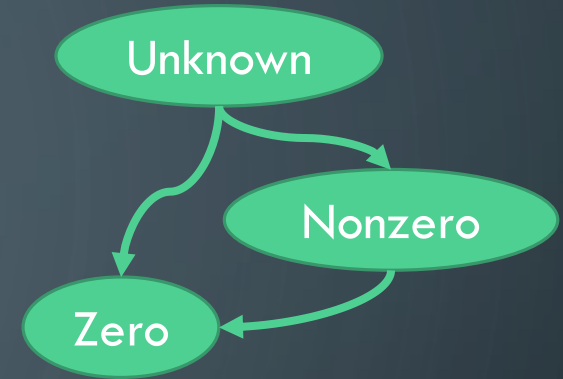
```
int x = 0;  
if (z) {  
    x = 5;  
}  
if (z) {  
    y = 10 / x;  
}
```



FLOW SENSITIVE ALGORITHMS

- Walk on the control flow graph
- What to do on merge points? (Lattice)

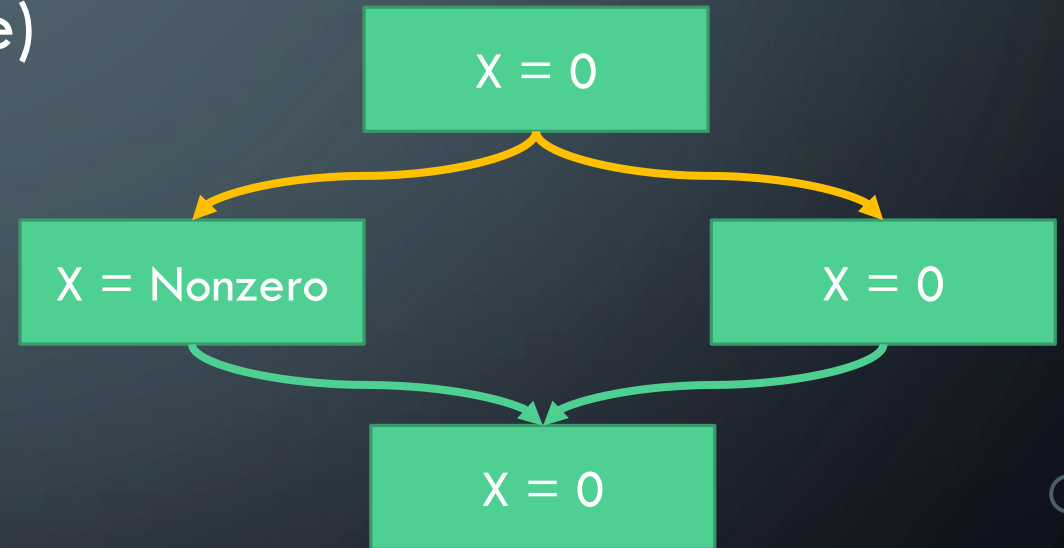
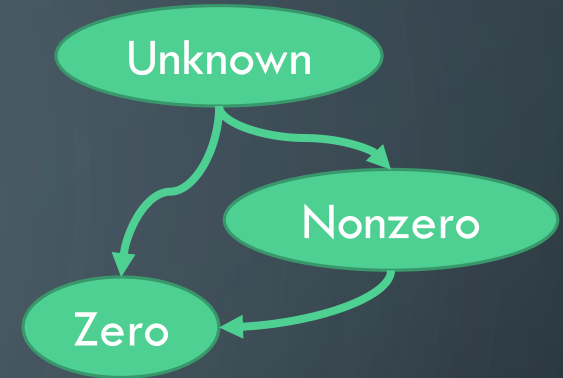
```
int x = 0;  
if (z) {  
    x = 5;  
}  
if (z) {  
    y = 10 / x;  
}
```



FLOW SENSITIVE ALGORITHMS

- Walk on the control flow graph
- What to do on merge points? (Lattice)

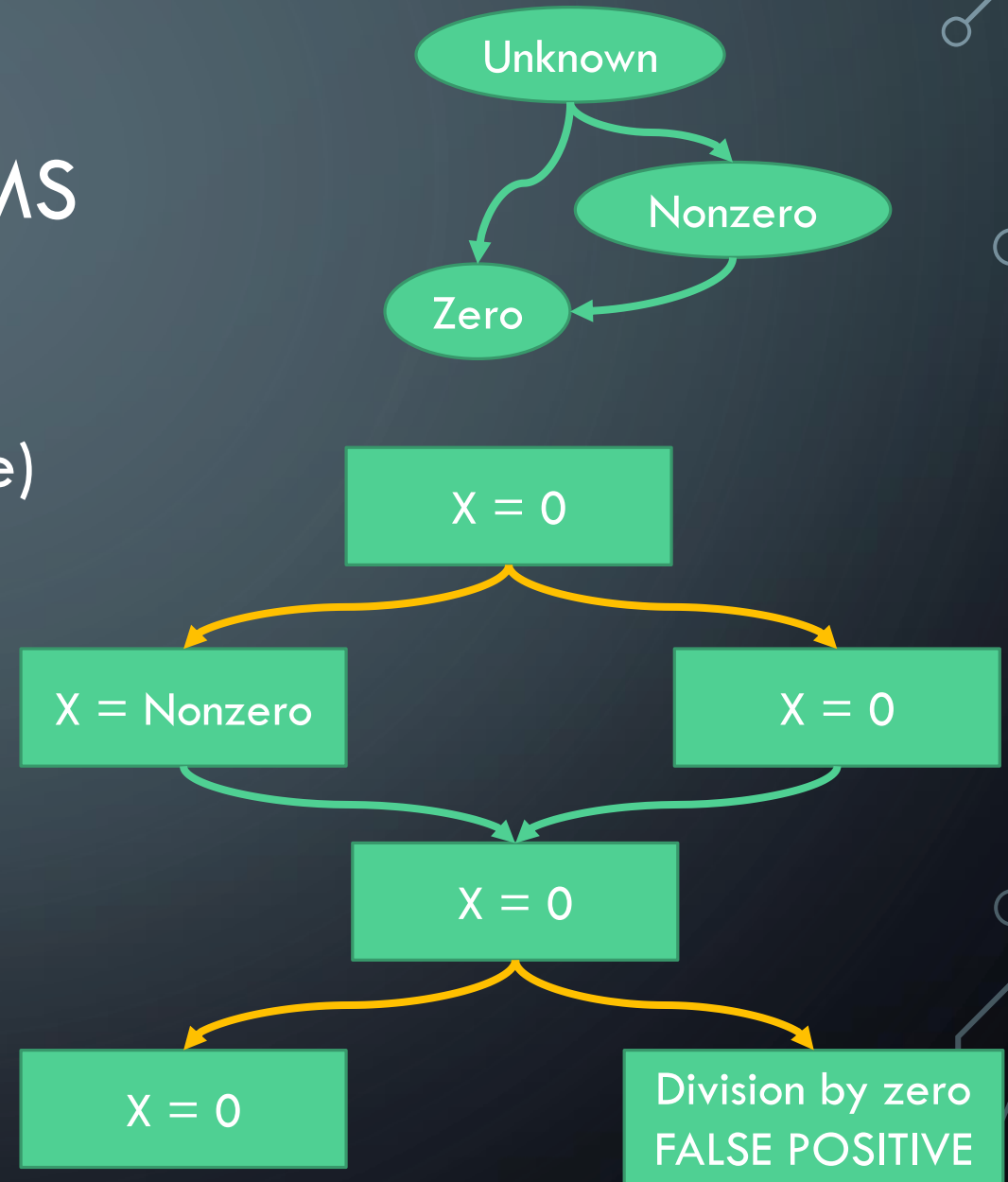
```
int x = 0;  
if (z) {  
    x = 5;  
}  
if (z) {  
    y = 10 / x;  
}
```



FLOW SENSITIVE ALGORITHMS

- Walk on the control flow graph
- What to do on merge points? (Lattice)

```
int x = 0;  
if (z) {  
    x = 5;  
}  
if (z) {  
    y = 10 / x;  
}
```

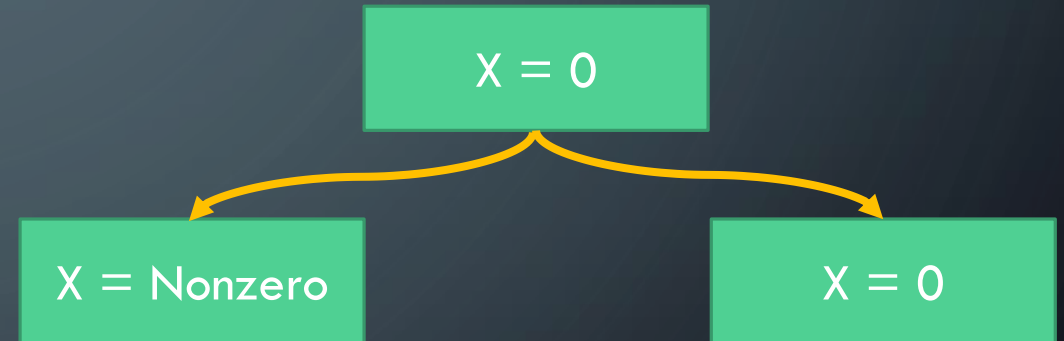


FLOW SENSITIVE ALGORITHMS

```
int x = 0;  
if (z) {  
    x = 5;  
}  
if (!z) {  
    y = 10 / x;  
}
```

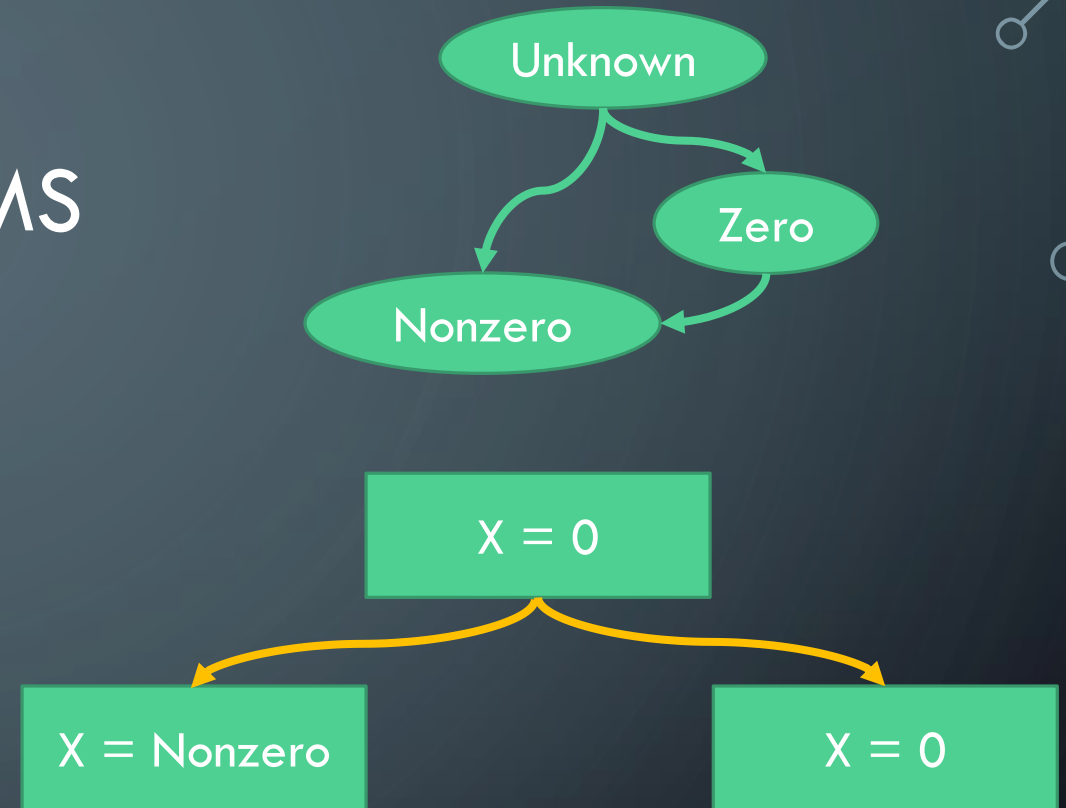
FLOW SENSITIVE ALGORITHMS

```
int x = 0;  
if (z) {  
    x = 5;  
}  
if (!z) {  
    y = 10 / x;  
}
```



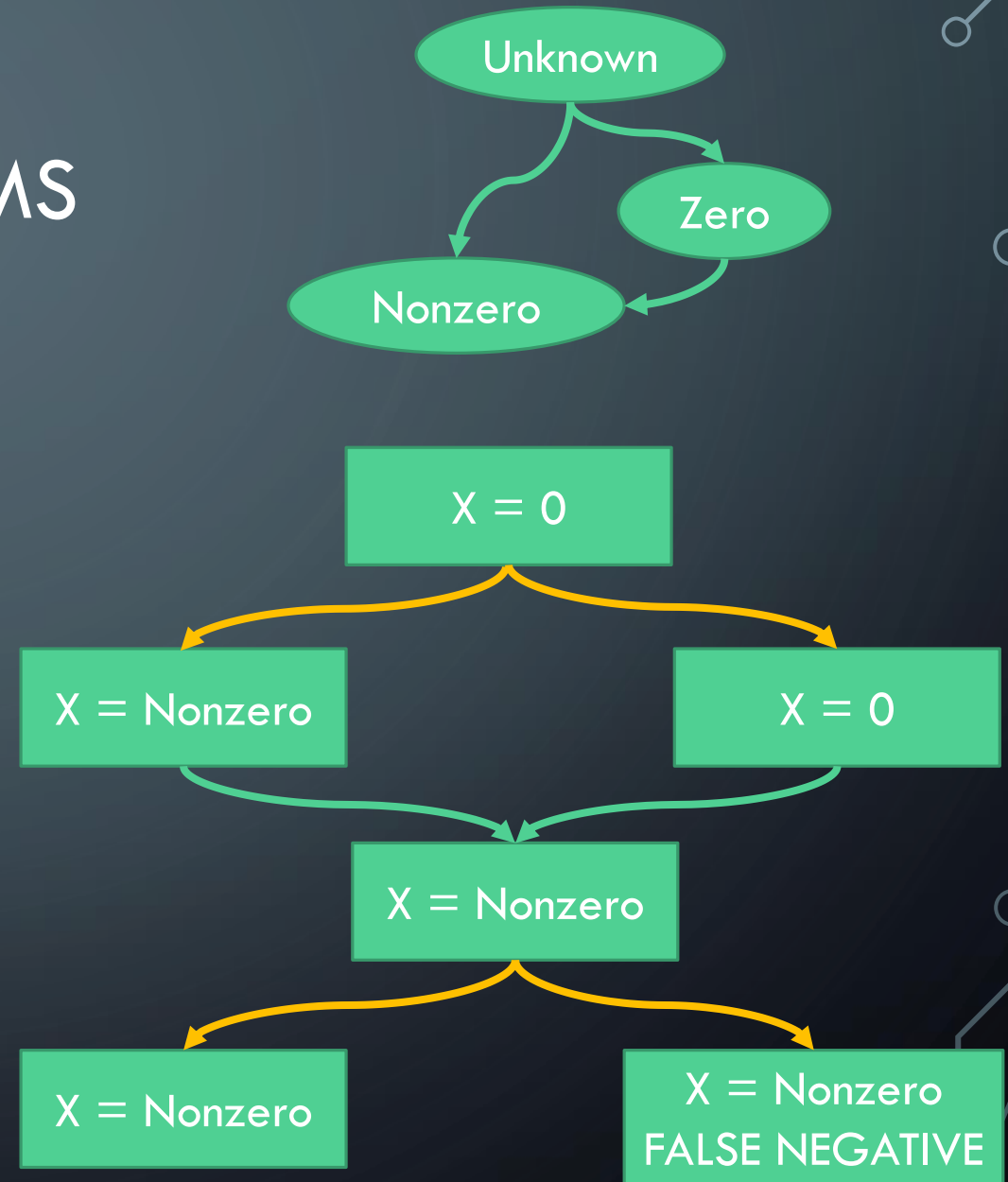
FLOW SENSITIVE ALGORITHMS

```
int x = 0;  
if (z) {  
    x = 5;  
}  
if (!z) {  
    y = 10 / x;  
}
```



FLOW SENSITIVE ALGORITHMS

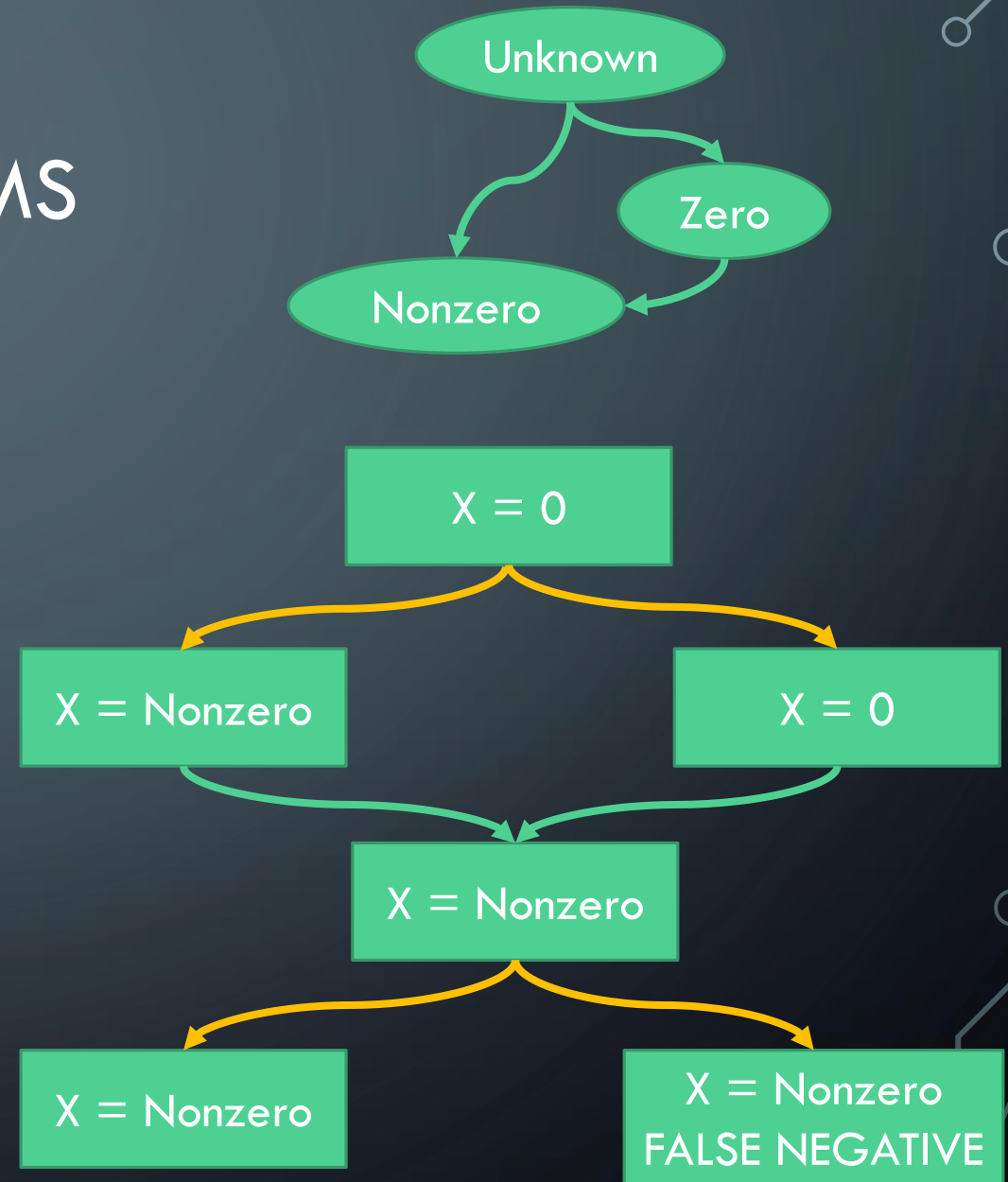
```
int x = 0;  
if (z) {  
    x = 5;  
}  
if (!z) {  
    y = 10 / x;  
}
```



FLOW SENSITIVE ALGORITHMS

- Polynomial but imprecise

```
int x = 0;  
if (z) {  
    x = 5;  
}  
if (!z) {  
    y = 10 / x;  
}
```



PATH SENSITIVE ALGORITHMS

- Path sensitive walk over the control flow graph
- Simulated execution
 - Try to enumerate every possible execution path
- Symbols to represent unknown values
 - Record constraints on symbols for a path
 - Prune impossible paths
- Exponential in the branching factor

SYMBOLIC EXECUTION

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

Nodes are immutable states,
arrows are transitions.

SYMBOLIC EXECUTION

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

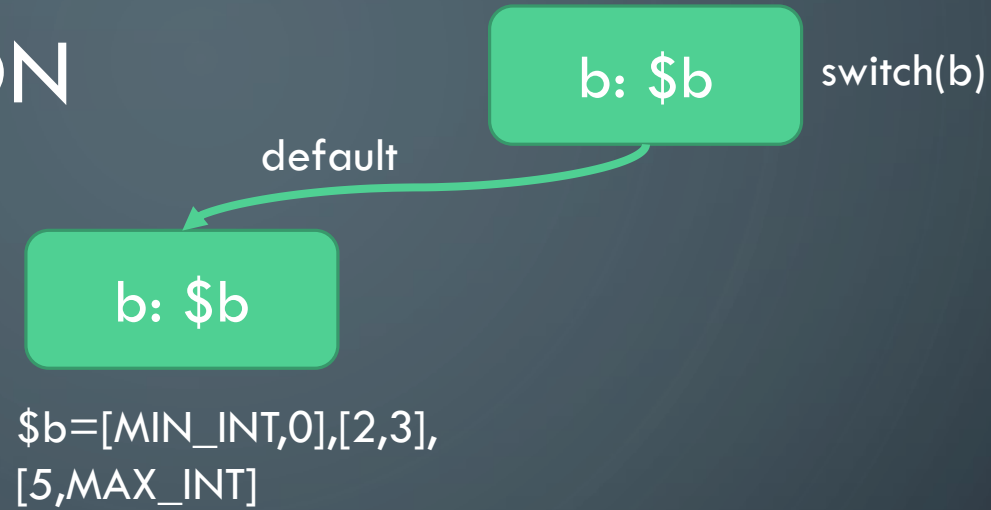
Nodes are immutable states,
arrows are transitions.

b: \$b

switch(b)

SYMBOLIC EXECUTION

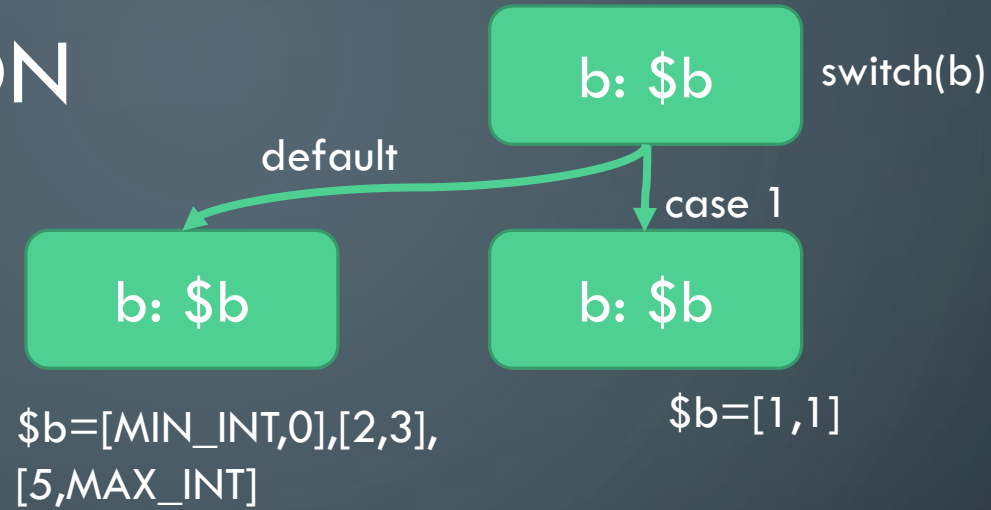
```
void test(int b) {  
    int a,c;  
    switch (b) {  
        case 1: a = b / 0; break;  
        case 4: c = b - 4;  
                a = b / c; break;  
    }  
}
```



Nodes are immutable states,
arrows are transitions.

SYMBOLIC EXECUTION

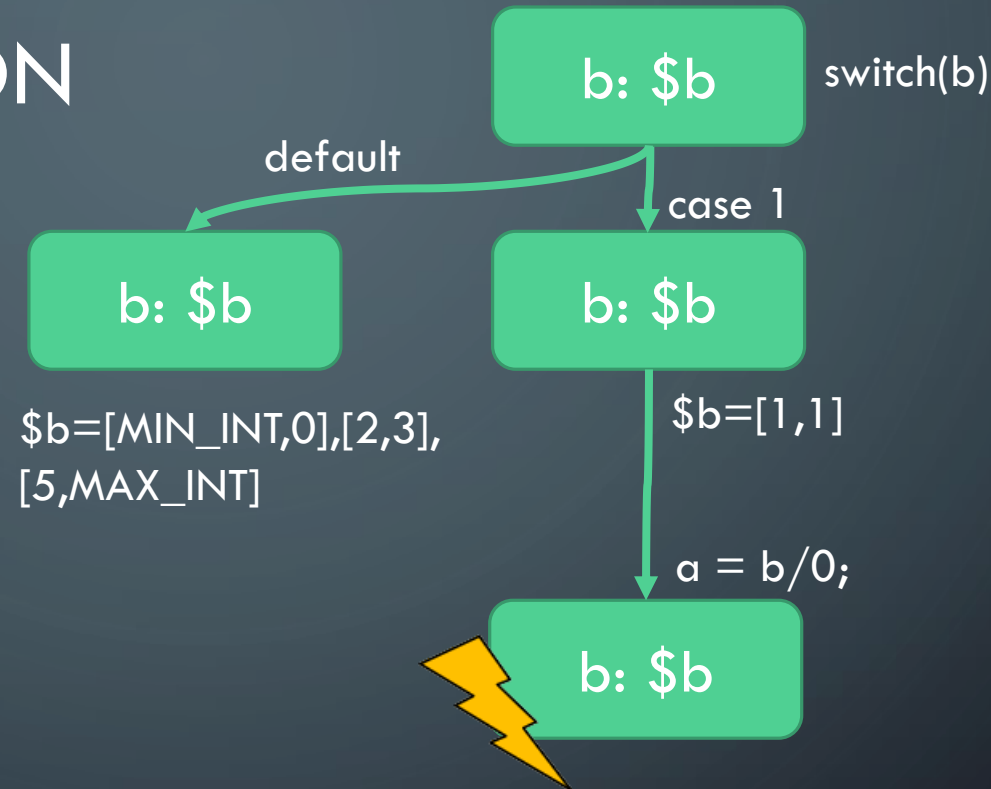
```
void test(int b) {  
    int a,c;  
    switch (b) {  
        case 1: a = b / 0; break;  
        case 4: c = b - 4;  
                a = b / c; break;  
    }  
}
```



Nodes are immutable states,
arrows are transitions.

SYMBOLIC EXECUTION

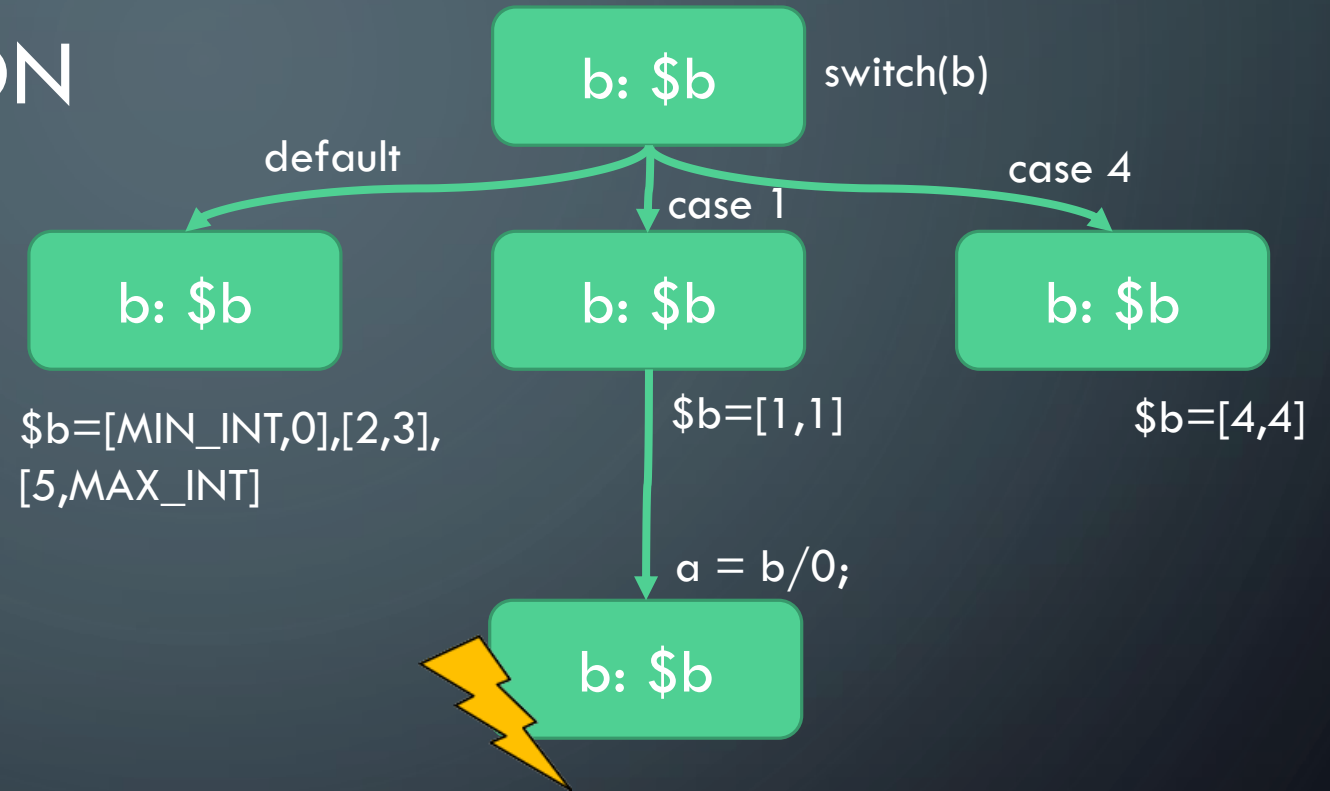
```
void test(int b) {  
    int a,c;  
    switch (b) {  
        case 1: a = b / 0; break;  
        case 4: c = b - 4;  
                a = b / c; break;  
    }  
}
```



Nodes are immutable states,
arrows are transitions.

SYMBOLIC EXECUTION

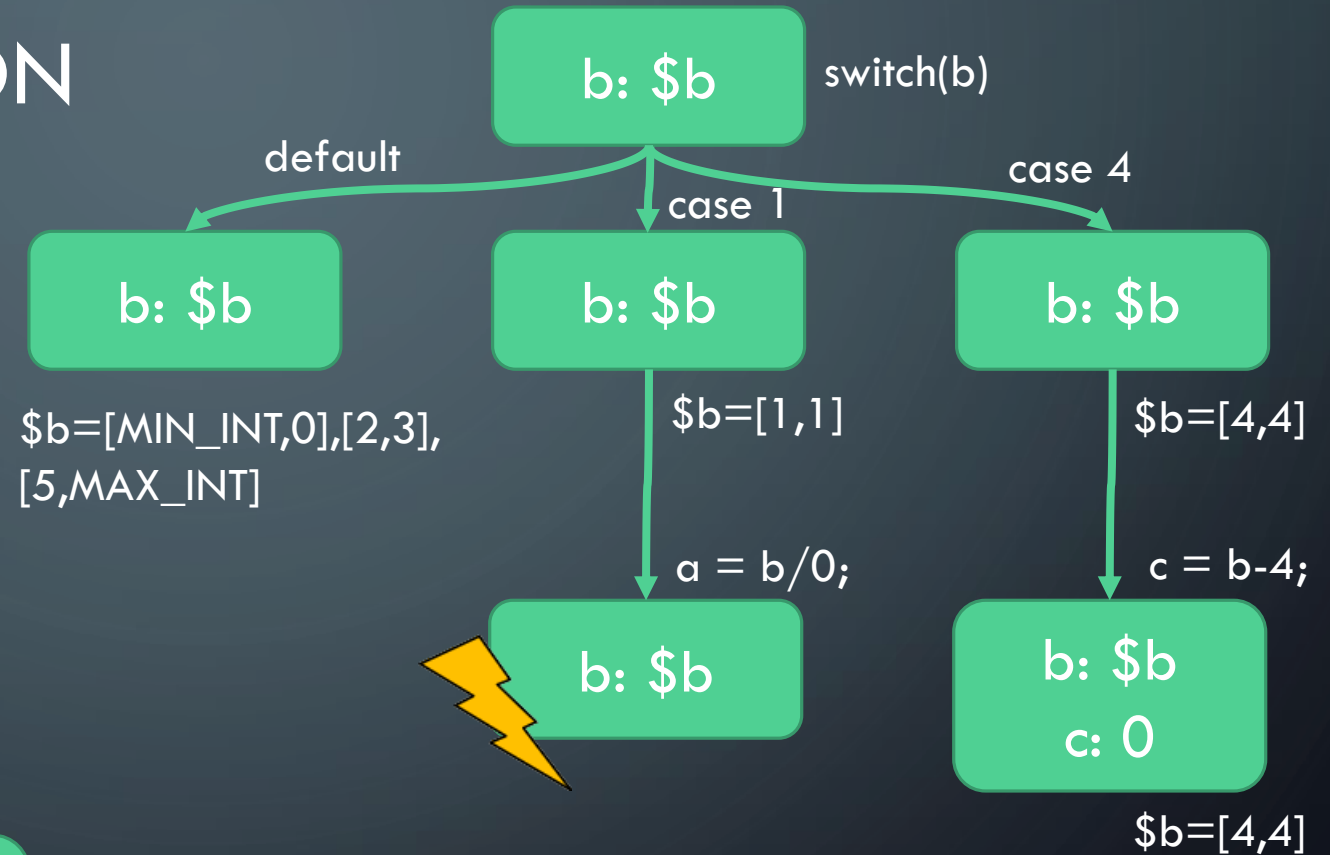
```
void test(int b) {  
    int a,c;  
    switch (b) {  
        case 1: a = b / 0; break;  
        case 4: c = b - 4;  
                a = b / c; break;  
    }  
}
```



Nodes are immutable states,
arrows are transitions.

SYMBOLIC EXECUTION

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

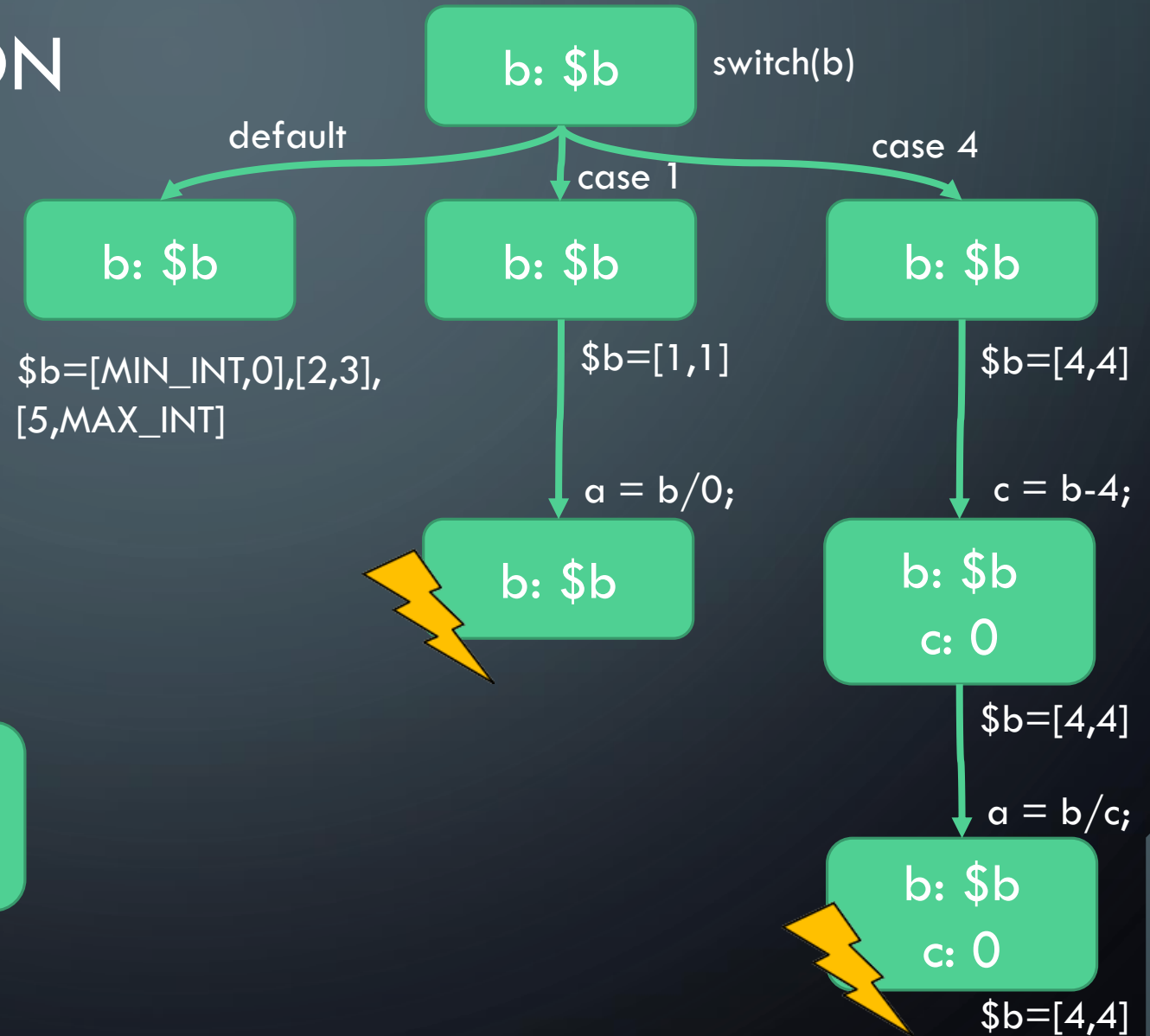


Nodes are immutable states,
arrows are transitions.

SYMBOLIC EXECUTION

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

Nodes are immutable states,
arrows are transitions.

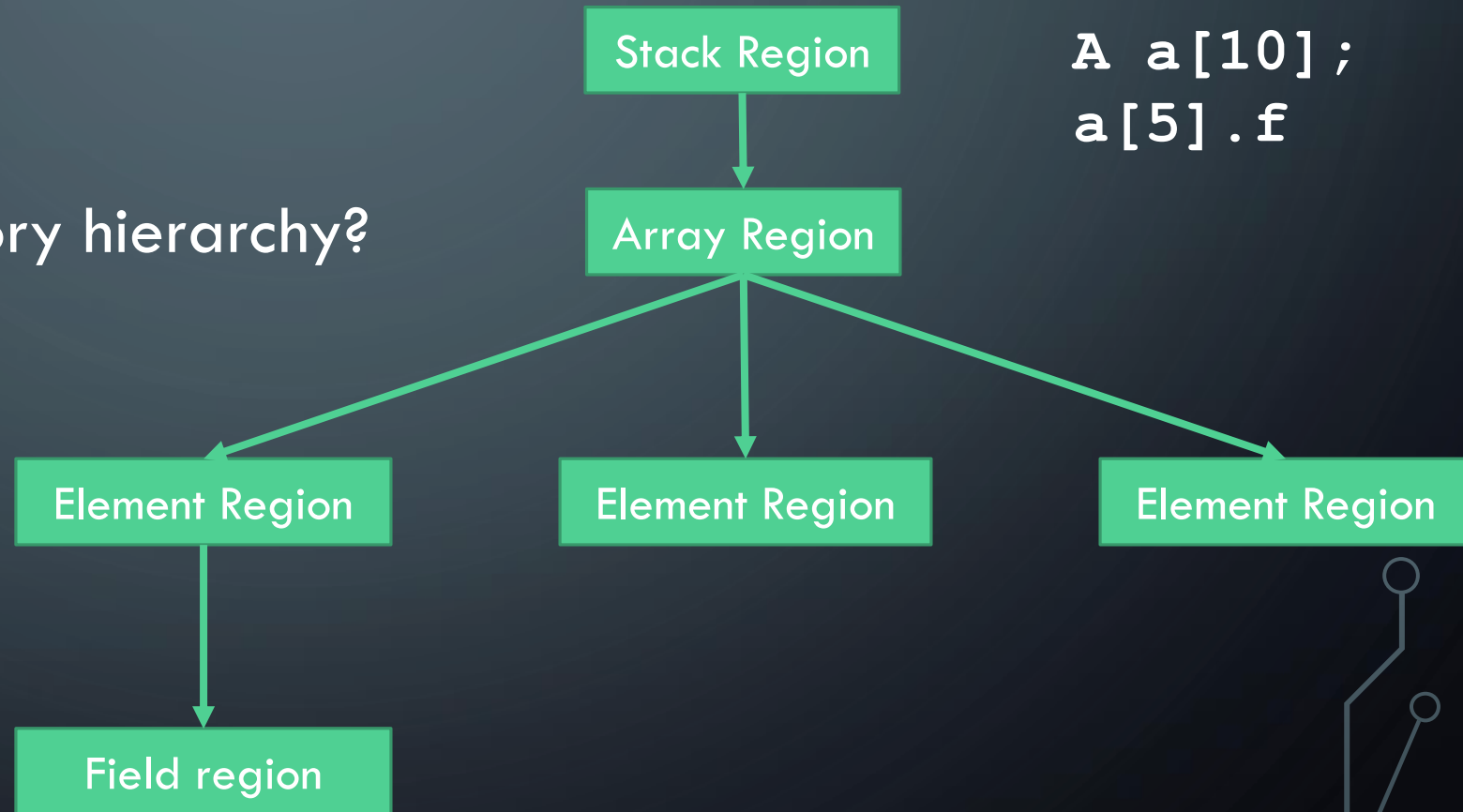


MODELING MEMORY

- What about aliasing?
- What about the memory hierarchy?
- What about locality?

MODELING MEMORY

- What about aliasing?
- What about the memory hierarchy?
- What about locality?



CONTEXT SENSITIVITY (INTRA/INTERPROCEDURAL)

```
int getNull(int a) {  
    return a?0:1;  
}
```

```
void test(int b)  
{  
    int a;  
    switch (b) {  
        ...  
        case 5:  
            a = b / getNull(b);  
            break;  
        ...  
    }  
}
```


CONTEXT SENSITIVITY (INTRA/INTERPROCEDURAL)

```
int getNull(int a) {  
    return a?0:1;  
}
```

Called with a == 5, returns 0

```
void test(int b)  
{  
    int a;  
    switch (b) {  
        ...  
        case 5:  
            a = b / getNull(b);  
            break;  
        ...  
    }  
}
```

CONTEXT SENSITIVITY (INTRA/INTERPROCEDURAL)

```
int getNull(int a) {  
    return a?0:1;  
}
```

Called with a == 5, returns 0

```
void test(int b)  
{  
    int a;  
    switch (b) {  
        ...  
        case 5:  
            a = b / getNull(b);  
            break;  
        ...  
    }  
}
```

Division by zero

CROSS TRANSLATION UNIT ANALYSIS

A.cpp

```
void f(int *x);  
void g(int *x) {  
    if (*x > 0) {  
        f(x);  
        f(x);  
    }  
}
```

*x is positive

*x is unknown

B.cpp

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

GLOBAL ANALYSIS

- If cross translation unit analysis works, what about other modules?
- What about other projects/libraries?
- What about the closed source libraries?
- It is likely to have a boundary to unknown

SUMMARY BASED ANALYSIS

- One way to implement interprocedural analysis
- Summarize some information about a function
 - What summarization means depends on the actual analysis
- Instead of “inlining” the function call, use a summary
- Alternatively, use summary when the function body is not available

OPEN SOURCE TOOLS

- CppCheck
- CppLint
- **Clang Static Analyzer**
- **Clang Tidy**
- ...

OPEN SOURCE TOOLS

- CppCheck
- CppLint
- **Clang Static Analyzer**
- **Clang Tidy**
- ...

CLANG STATIC ANALYZER

- Path sensitive
- Symbolic execution
- Good memory modeling capabilities
- Context sensitive
- Lacks cross translation unit analysis support

CLANG TIDY

- Supports code rewriting
- Large amount of checks (and growing fast)
- Most checkers are based on syntax tree matching
- Lacks cross translation unit analysis support
- C++ Core Guideline checks

HOW TO INTEGRATE INTO THE WORKFLOW?

- Old and huge projects
 - New commits should not introduce new reports
 - Most severe reports should be fixed gradually
- Green field projects
 - Keep the number of reports minimal (ideally 0)
- Integrate into the CI, commit hooks

HOW TO INTEGRATE INTO THE WORKFLOW?

- Incremental analysis locally
 - Build system support?
- Full analysis every night
- Triaging (critical, non-critical, false positive, fixed, ...)
- False positive suppression
- Skip uninteresting files (maintained by other team, etc...)
- Easy to use

CODECHECKER

- Open source tool to support the workflow using static analyzers
 - Clang Static Analyzer and Clang Tidy supported
- Developed by Ericsson, might end up under the LLVM umbrella
- Actively developed
- Supports Mac and Linux
- How does this work? See the EuroLLVM 2015 talk!
 - <http://llvm.org/devmtg/2015-04/>

CODECHECKER

- Web viewer for defects
 - Filtering, false positive suppression
 - Differential view (what are the new reports compared to a baseline?)
 - Incremental analysis
- Command line client for CI system integration
- Eclipse integration
- CodeCompass integration (Lightning talk tomorrow!)

HOW TO USE CODECHECKER?

- Install instructions are detailed on the GitHub page
- Run the check:

```
$> CodeChecker check -n FirstCheck -b "make"
```

- Start the webserver (previous command prints this):

```
$> CodeChecker server -w ~/.codechecker
```

CodeChecker

List of runs FirstRun x

Run Overview - hits : 82

Path filter

Unsuppressed

All (82)

All (82)

High (42)

core.NullDereference (29)
core.CallAndMessage (6)
core.NonNullParamChecker (6)
core.DivideZero (1)

Medium (16)

core.UndefinedBinaryOperatorResult (8)
core.uninitialized.Assign (6)
unix.cstring.NullArg (1)
unix.Malloc (1)

Low (22)

deadcode.DeadStores (22)

Unspecified (2)

(2)

File	Message	Category	Severity	Suppress Comment
/home/xazax/Random/vim/src/ex_docmd.c @ Line 10742	Function call argument is an uninitialized value	core.NullDereference	high	---
/home/xazax/Random/vim/src/ops.c @ Line 5164	Function call argument is an uninitialized value	core.NullDereference	high	---
/home/xazax/Random/vim/src/os_unix.c @ Line 4306	Function call argument is an uninitialized value	core.NullDereference	high	---
/home/xazax/Random/vim/src/os_unix.c @ Line 4419	Function call argument is an uninitialized value	core.NullDereference	high	---
/home/xazax/Random/vim/src/os_unix.c @ Line 4499	Function call argument is an uninitialized value	core.NullDereference	high	---
/home/xazax/Random/vim/src/screen.c @ Line 399	Function call argument is an uninitialized value	core.CallAndMessage	high	---
/home/xazax/Random/vim/src/screen.c @ Line 10372	Division by zero	core.DivideZero	high	---
/home/xazax/Random/vim/src/ops.c @ Line 4502	Null pointer passed as an argument to a 'nonnull' parameter	core.NonNullParamChecker	high	---
/home/xazax/Random/vim/src/regexp.c	Null pointer passed as an argument to a 'nonnull' parameter	core.NonNullParamChecker	high	---

CodeChecker

List of runs

FirstRun x

Run Overview - hits : 82

buffer.c @ Line 628 x

- Unspecified
- Critical
- High
 - Line 628 : core.NullDereferen
- Medium
- Low
- Style

Result : Access to field 'w

Line 454 : Assuming 'curwin'

Line 459 : Assuming 'action'

Line 494 : Assuming 'win' is e

Line 583 : Assuming 'buf' is r

Line 619 : Assuming 'buf' is r

Line 628 : Access to field 'w

Suppress bug

Show documentation

buffer.c : /home/xazax/Random/vim/src/buffer.c

Search: (Use /re/ syntax for regexp search) < prev next >

```
608     aborting() /* autocmds may abort script processing */
609     return;
610 #endif
611
612 /*
613  * It's possible that autocommands change curbuf to the one being deleted.
614  * This might cause the previous curbuf to be deleted unexpectedly. But
615  * in some cases it's OK to delete the curbuf, because a new one is
616  * obtained anyway. Therefore only return if curbuf changed to the
617  * deleted buffer.
618  */
619     if (buf == curbuf && !is_curbuf)
        Assuming 'buf' is not equal to 'curbuf'
        return;
620
621     if (
622 #ifdef FEAT_WINDOWS
623         win_valid_any_tab(win) &&
624 #else
625         win != NULL &&
626 #endif
627         win->w_buffer == buf)
        Access to field 'w_buffer' results in a dereference of a null pointer (loaded
628         win->w_buffer = NULL; /* make sure we don't use the buffer now */
629
630
631     /* Autocommands may have opened or closed windows for this buffer.
632     * Decrement the count for the close we do here. */
633     if (buf->b_nwindows > 0)
634
```


HOW TO USE CODECHECKER?

- The analysis run name should be:
 - New and unique for clean build (e.g. nightly on CI)
 - Name of an existing run for incremental check (e.g.: on local machine)
- The web viewer is localhost only by default
- For differential view, the baseline should be selected first

WRITING STATIC ANALYSIS FRIENDLY CODE

- Minimize macro use
- Avoid nonstandard extensions, ensure generated code is warning free
- Use plenty of types (type based alias analysis, ...)
 - Guarantees from the type system are always better than relying on 3rd party static analysis tools
- Do not overuse references, pointers

WRITING STATIC ANALYSIS FRIENDLY CODE

- Use annotations! (noreturn, deprecated, nullability, ...)
 - Very important to annotate custom assertion functions!
 - Users benefit from annotations in library headers
- Always analyze the debug build!
 - Asserts help the analyzer to understand the code
- Avoid arcane build systems!

WRITING STATIC ANALYSIS FRIENDLY CODE

```
void customAssert();  
int foo(int *b) {  
    if (!b)
```

1 Assuming 'b' is null

2 Taking true branch

```
        customAssert();  
    return *b;  
}
```

3 Dereference of null pointer (loaded from variable 'b')

HANDLING FALSE POSITIVES

- False positives?
 - Impossible path
 - Invariant unknown to the analyzer
- Help the analyzer understanding your code
 - Add asserts, rewrite code snippets (e.g.: making an implicit cast explicit)
 - Prefer widely used APIs to your custom solution (Standard C API, Posix, STL)
 - Increase the coverage, a report can halt the analysis on a path
 - Helps readability

HANDLING FALSE POSITIVES

- Do **NOT** turn off core checkers to suppress warnings!
 - Preprocess instead (or use CodeChecker)
- Exclude problematic code using macros
- Use false positive suppression only as a final resort
 - Comment! Why is it false positive? Why is it not possible to change the code? What are the requirements to remove this suppression? In case of an analyzer bug, link to the bug report!
- Do **NOT** mark 3rd party headers as system headers
 - Use post processing to remove these reports (or use CodeChecker)

CONCLUSION

- Static analysis is useful to have in the workflow
- There are several powerful tools available for free
- Writing static analysis friendly code requires some effort, but worth it
- Library devs should pay attention to static analysis

REFERENCES

- CodeChecker: <https://github.com/Ericsson/codechecker>
- CodeChecker Eclipse plugin:
<https://github.com/Ericsson/CodeCheckerEclipsePlugin>
- CodeCompass: <https://github.com/Ericsson/CodeCompass>
- C++ Core Guidelines: <https://github.com/isocpp/CppCoreGuidelines>