

Finding software bugs with the Clang Static Analyzer

Ted Kremenek, Apple Inc.

Findings Bugs with Compiler Techniques

Findings Bugs with Compiler Techniques

Compile-time warnings

```
% clang t.c
```

```
t.c:38:13: warning: invalid conversion '%lb'  
printf("%s%lb%d", "unix", 10, 20);  
      ~~~~~^~~~~
```

Findings Bugs with Compiler Techniques

Compile-time warnings

```
% clang t.c
```

```
t.c:38:13: warning: invalid conversion '%lb'  
printf("%s%lb%d", "unix", 10, 20);  
      ~~~~~^~~~~
```

Static Analysis

- Checking performed by compiler warnings inherently limited
- Find path-specific bugs
- Deeper bugs: memory leaks, buffer overruns, logic errors

Benefits of Static Analysis

Benefits of Static Analysis

Early discovery of bugs

- Find bugs early, while the developer is hacking on their code
- Bugs caught early are cheaper to fix

Benefits of Static Analysis

Early discovery of bugs

- Find bugs early, while the developer is hacking on their code
- Bugs caught early are cheaper to fix

Systematic checking of all code

- Static analysis reasons about all corner cases

Benefits of Static Analysis

Early discovery of bugs

- Find bugs early, while the developer is hacking on their code
- Bugs caught early are cheaper to fix

Systematic checking of all code

- Static analysis reasons about all corner cases

Find bugs without test cases

- Useful for finding bugs in hard-to-test code
- Not a replacement for testing

This Talk: Clang “Static Analyzer”

Clang-based static analysis tool for finding bugs

- Supports C and Objective-C (C++ in the future)

Outline

- Demo
- How it works
- Design and implementation
- Looking forward

This Talk: Clang “Static Analyzer”

Clang-based static analysis tool for finding bugs

- Supports C and Objective-C (C++ in the future)

Outline

- Demo
- How it works
- Design and implementation
- Looking forward

<http://clang.lvm.org>

Demo

How does static analysis work?

How does static analysis work?

- Can catch bugs with different degrees of analysis sophistication

How does static analysis work?

- Can catch bugs with different degrees of analysis sophistication
- Per-statement, per-function, whole-program all important

How does static analysis work?

- Can catch bugs with different degrees of analysis sophistication
- Per-statement, per-function, whole-program all important

```
int f(int y) {
    int x;

    if (y)
        x = 1;

    printf("%d\n", y);

    return x;
}
```

compiler warnings (simple checks)

```
% gcc -Wall -O1 -c t.c
```

```
t.c: In function 'f':
```

```
t.c:5: warning: 'x' may be used uninitialized in
this function
```

```
% clang -warn-uninit-values t.c
```

```
t.c:13:12: warning: use of uninitialized variable
```

```
    return x;
```

```
    ^
```

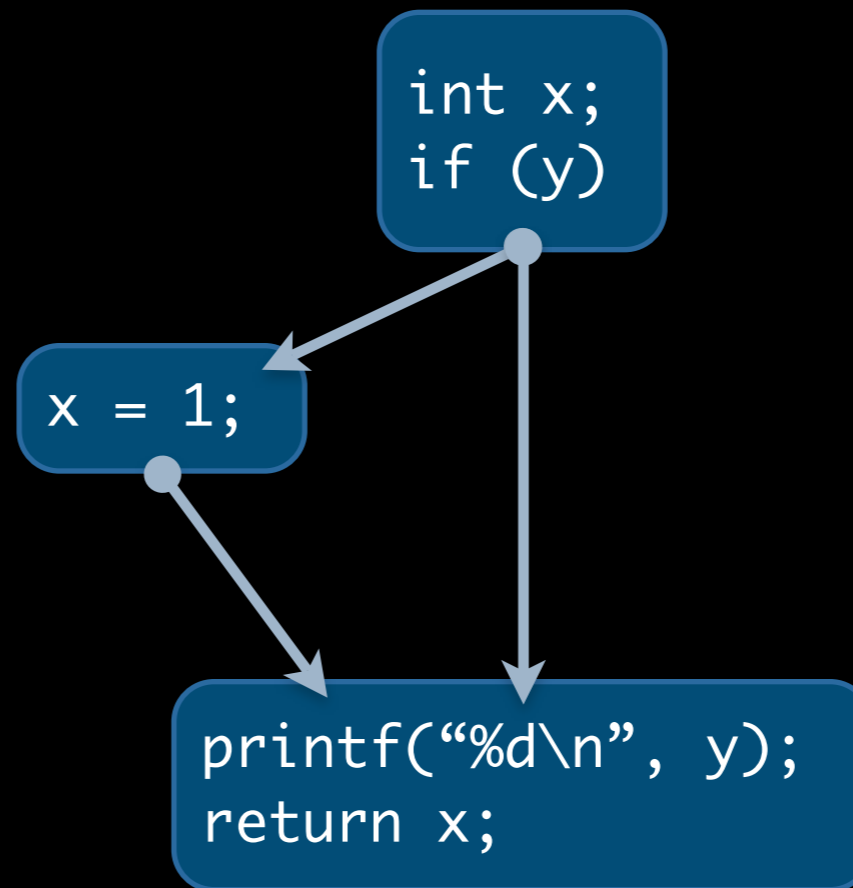
How does static analysis work?

```
int f(int y) {  
    int x;  
  
    if (y)  
        x = 1;  
  
    printf("%d\n", y);  
  
    return x;  
}
```

How does static analysis work?

control-flow graph

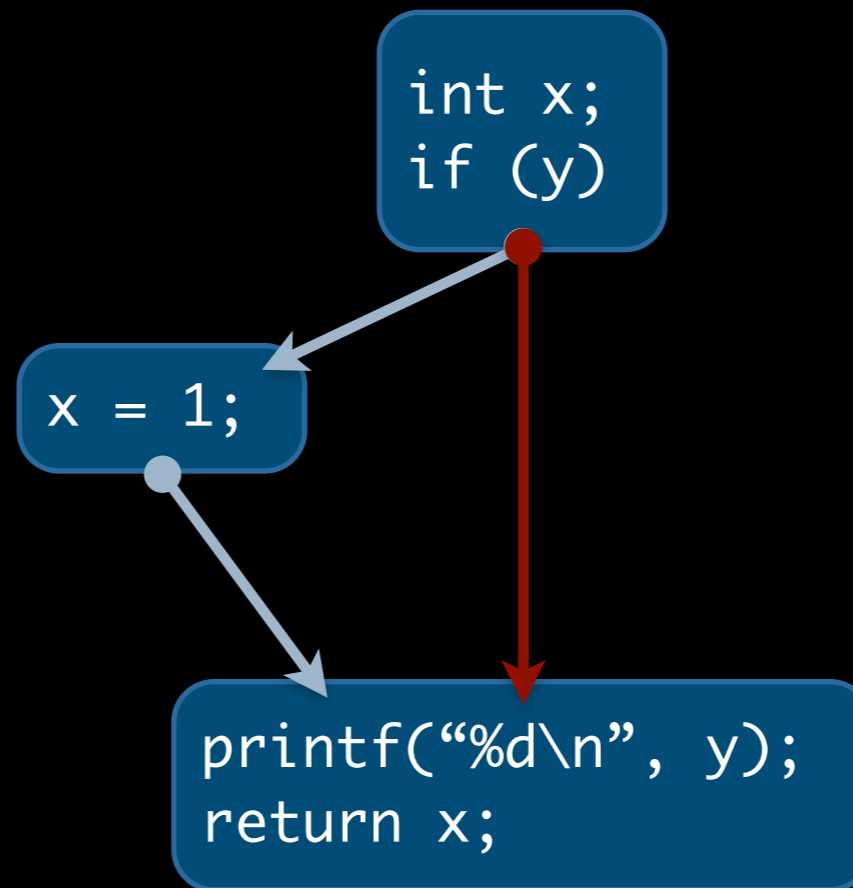
```
int f(int y) {  
    int x;  
  
    if (y)  
        x = 1;  
  
    printf("%d\n", y);  
  
    return x;  
}
```



How does static analysis work?

control-flow graph

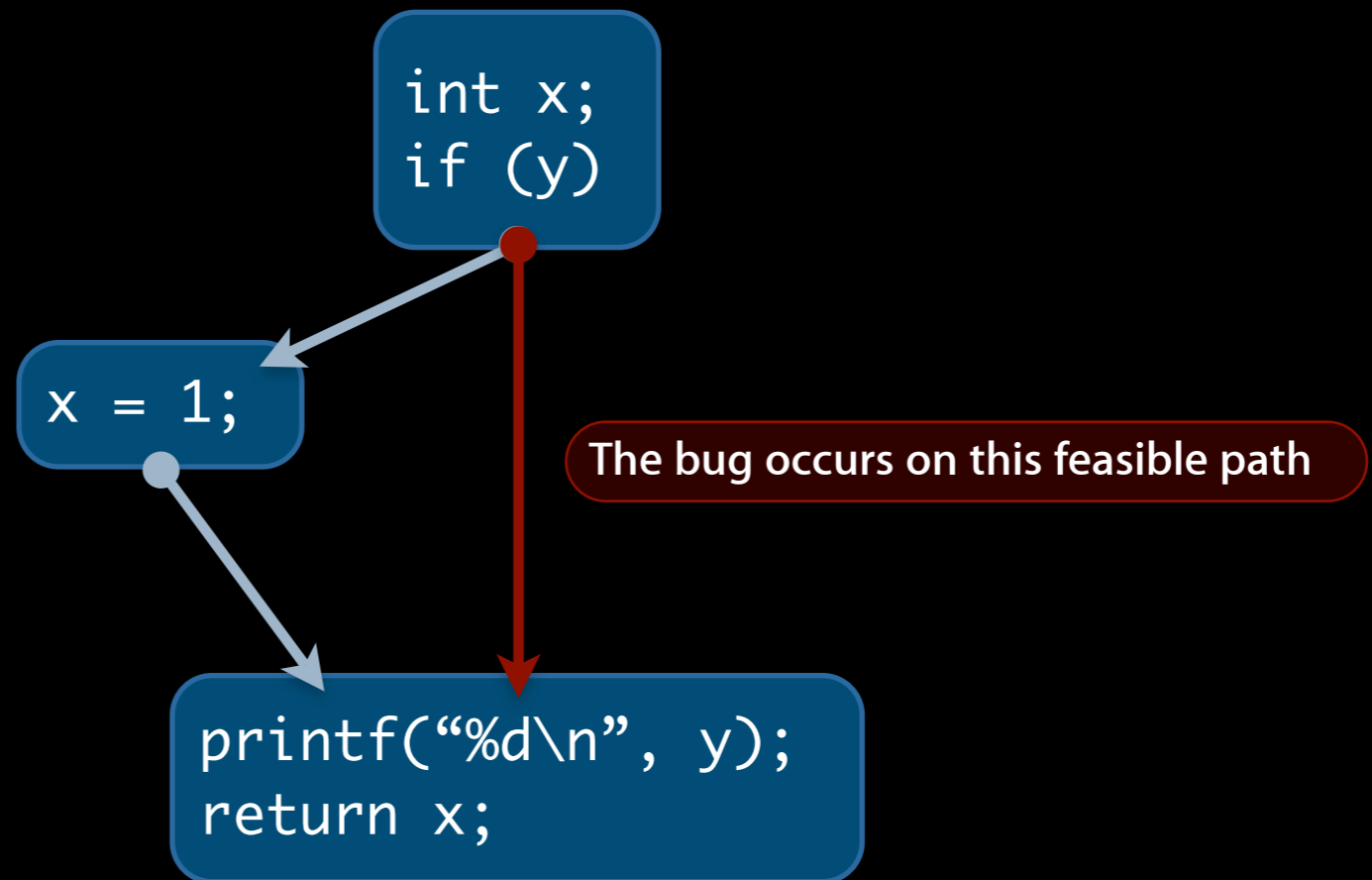
```
int f(int y) {  
    int x;  
  
    if (y)  
        x = 1;  
  
    printf("%d\n", y);  
  
    return x;  
}
```



How does static analysis work?

control-flow graph

```
int f(int y) {  
    int x;  
  
    if (y)  
        x = 1;  
  
    printf("%d\n", y);  
  
    return x;  
}
```

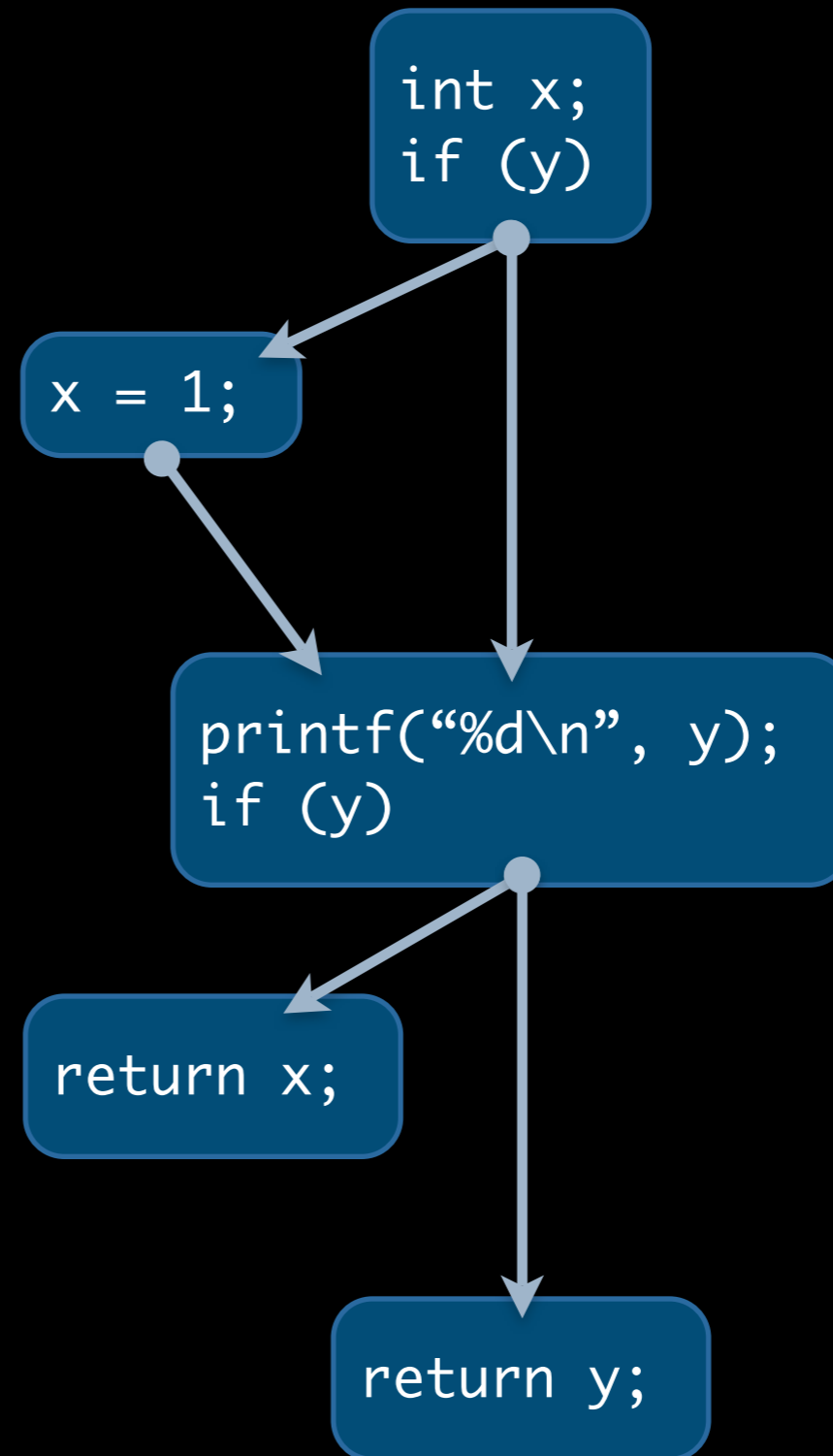


How does static analysis work?

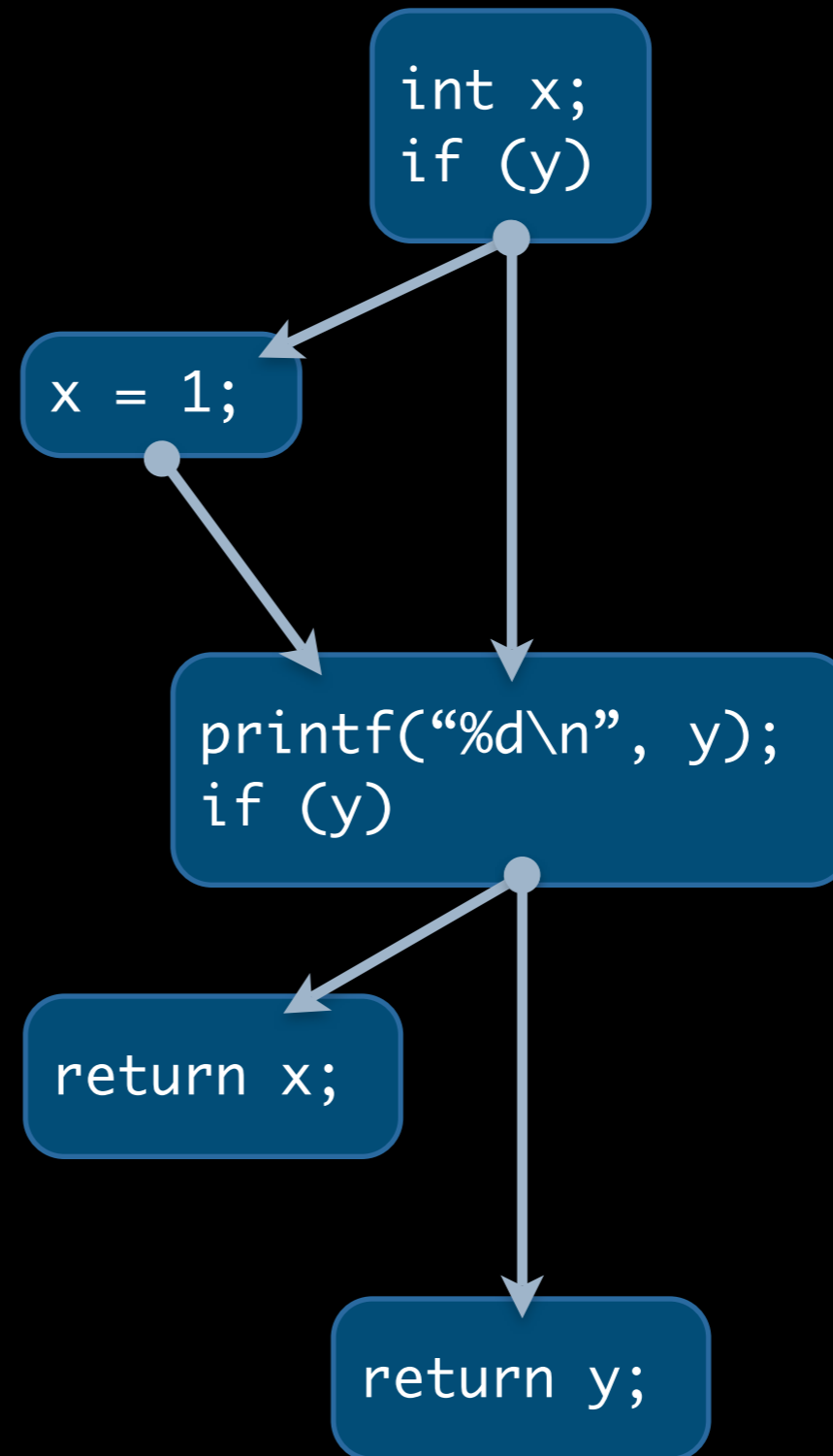
```
int f(int y) {  
    int x;  
  
    if (y)  
        x = 1;  
  
    printf("%d\n", y);  
  
    return x;  
}
```

How does static analysis work?

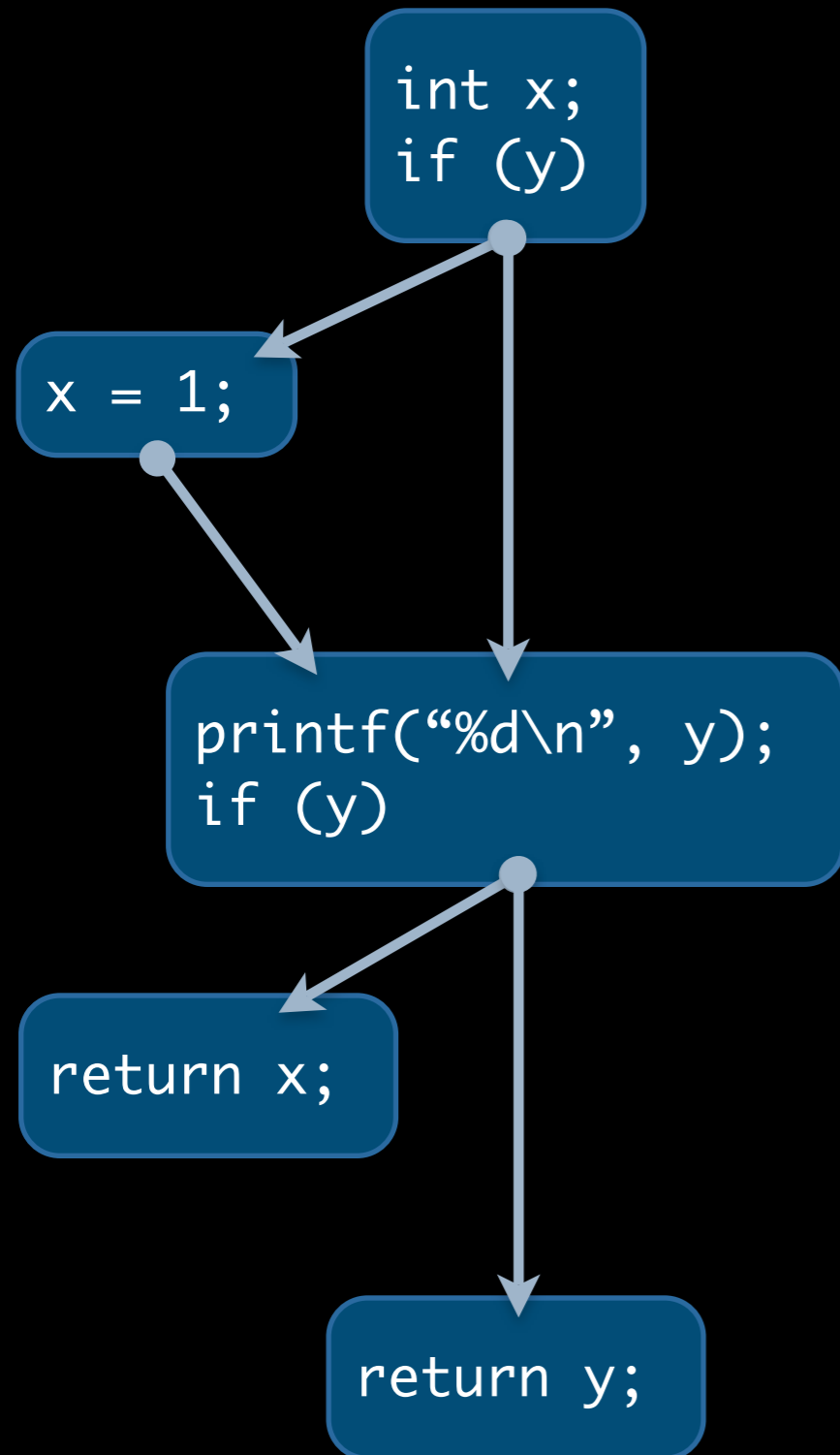
```
int f(int y) {  
    int x;  
  
    if (y)  
        x = 1;  
  
    printf("%d\n", y);  
  
    if (y)  
        return x;  
    return y;  
}
```



How does static analysis work?



How does static analysis work?



```
% gcc -Wall -O1 -c t.c
```

```
t.c: In function 'f':
```

```
t.c:5: warning: 'x' may be used uninitialized in  
this function
```

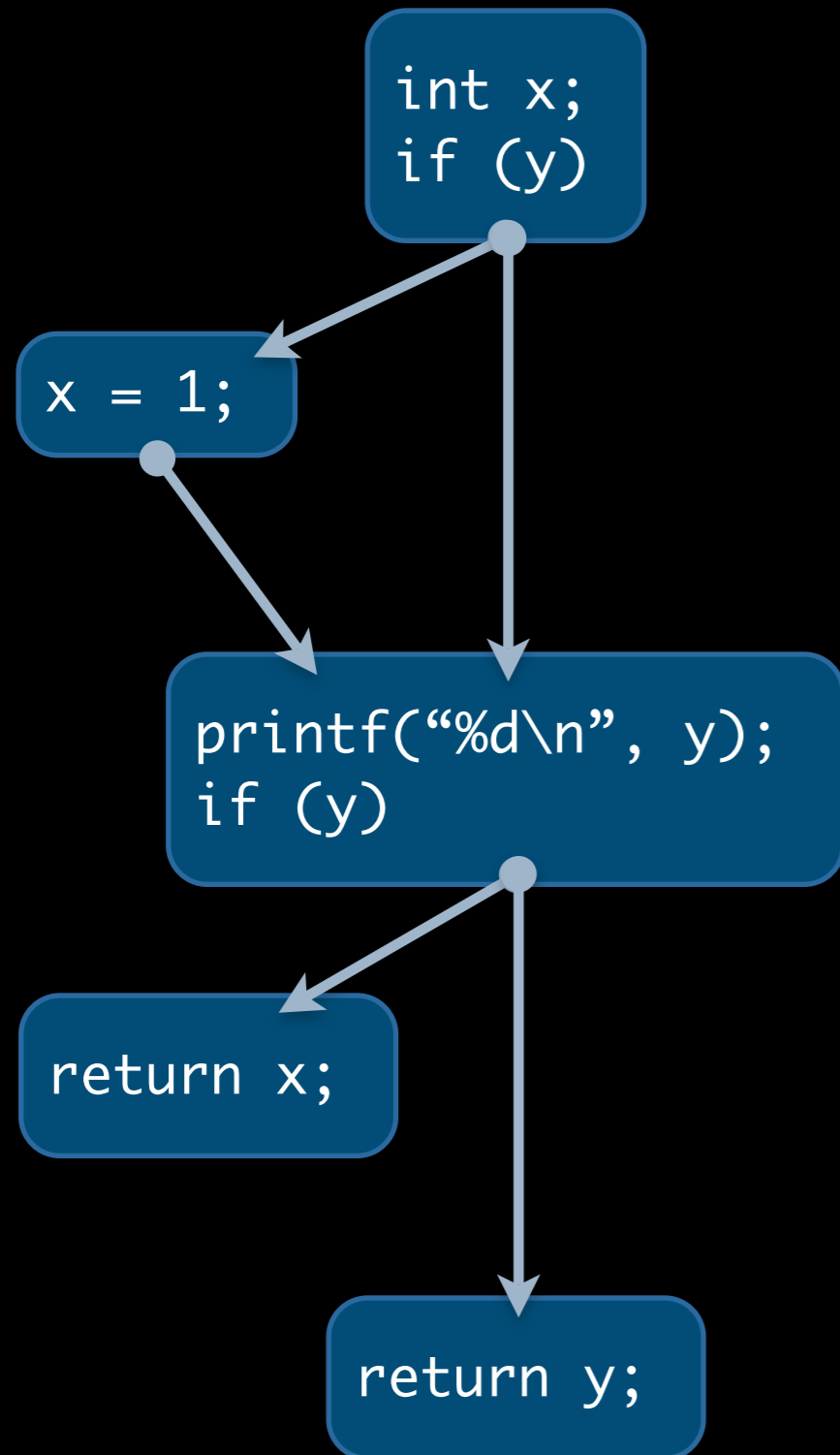
```
% clang -warn-uninit-values t.c
```

```
t.c:13:12: warning: use of uninitialized variable
```

```
return x;
```

```
^
```

How does static analysis work?



```
% gcc -Wall -O1 -c t.c
```

```
t.c: In function 'f':
```

```
t.c:5: warning: 'x' may be used uninitialized in  
this function
```

```
% clang -warn-uninit-values t.c
```

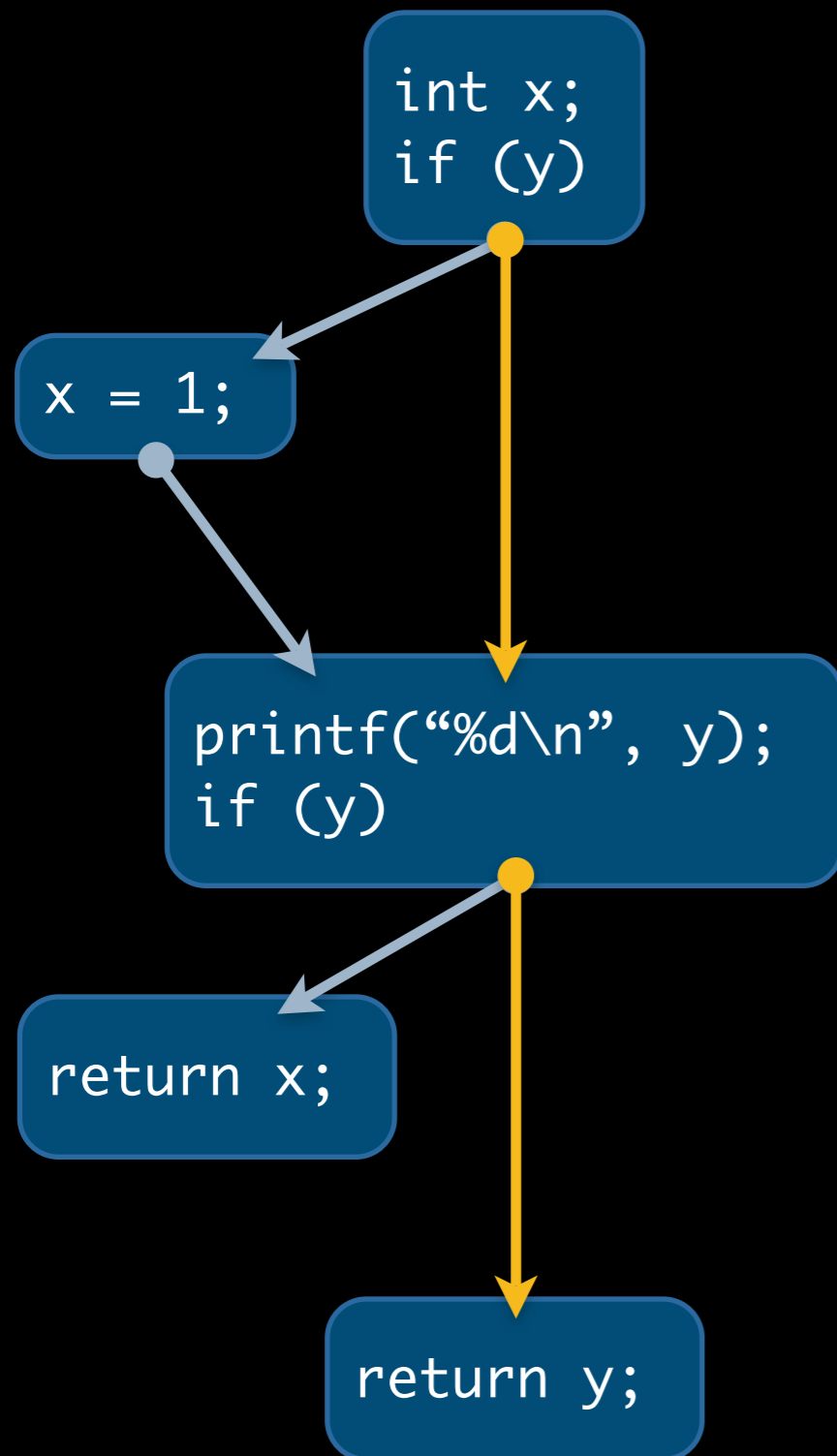
```
t.c:13:12: warning: use of uninitialized variable
```

```
return x;
```

```
^
```

Two feasible paths:

How does static analysis work?



```
% gcc -Wall -O1 -c t.c
```

```
t.c: In function 'f':
```

```
t.c:5: warning: 'x' may be used uninitialized in  
this function
```

```
% clang -warn-uninit-values t.c
```

```
t.c:13:12: warning: use of uninitialized variable
```

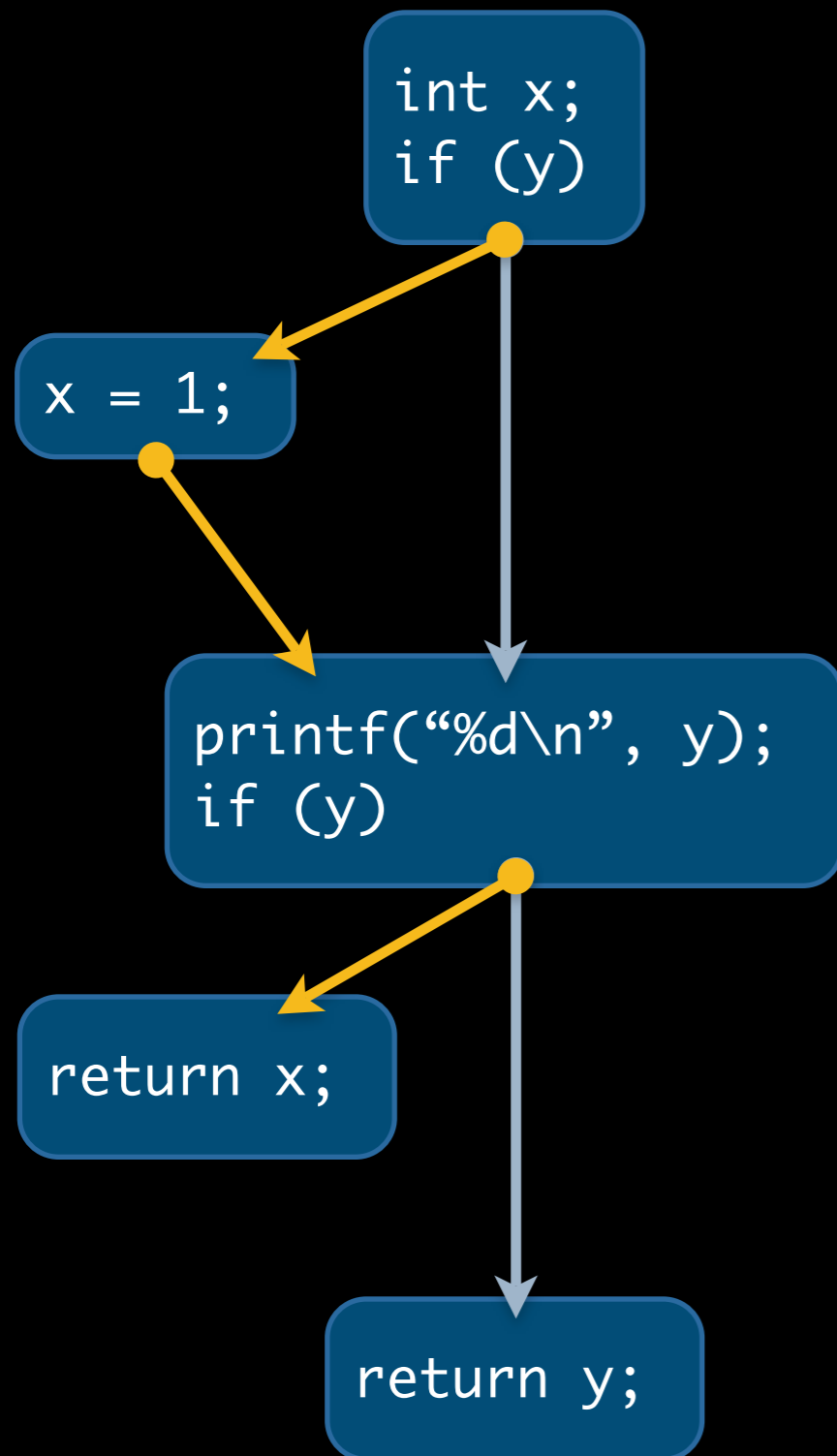
```
return x;
```

```
^
```

Two feasible paths:

- Neither branch taken ($y == 0$)

How does static analysis work?



```
% gcc -Wall -O1 -c t.c
```

```
t.c: In function 'f':
```

```
t.c:5: warning: 'x' may be used uninitialized in  
this function
```

```
% clang -warn-uninit-values t.c
```

```
t.c:13:12: warning: use of uninitialized variable
```

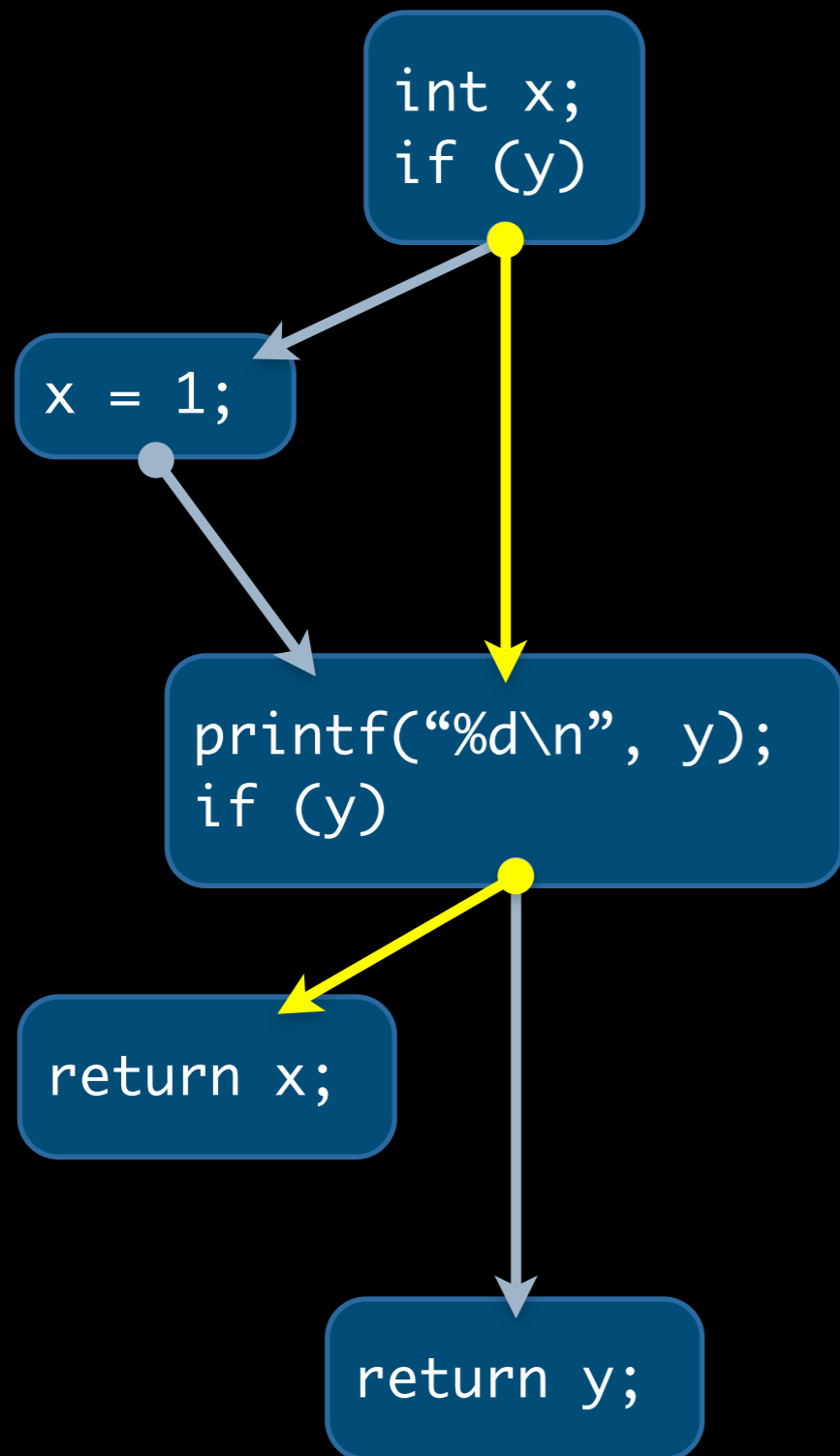
```
return x;
```

```
^
```

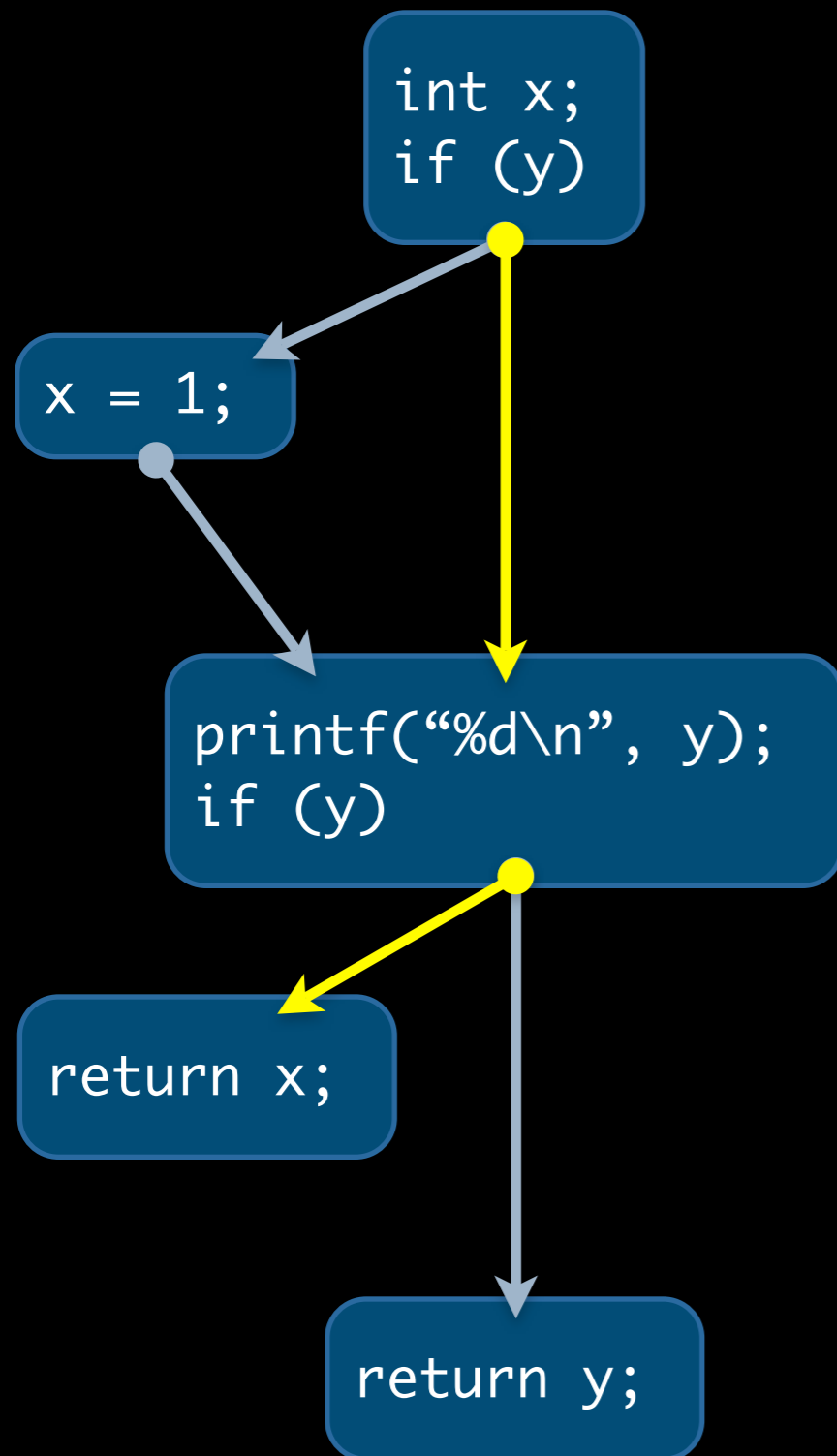
Two feasible paths:

- Neither branch taken ($y == 0$)
- Both branches taken ($y != 0$)

How does static analysis work?



How does static analysis work?



Bogus warning occurs on infeasible path:

- Don't take first branch (`y == 0`)
- Take second branch (`y != 0`)

How does static analysis work?

False Positives (Bogus Errors)

False Positives (Bogus Errors)

- False positives can occur due to analysis imprecision
 - False paths
 - Insufficient knowledge about the program

False Positives (Bogus Errors)

- False positives can occur due to analysis imprecision
 - False paths
 - Insufficient knowledge about the program
- Many ways to reduce false positives
 - More precise analysis
 - Difficult to eliminate false positives completely

Flow-Sensitive Analyses

Flow-Sensitive Analyses

- Flow-sensitive analyses reason about flow of values

```
y = 1;  
x = y + 2; // x == 3
```

Flow-Sensitive Analyses

- Flow-sensitive analyses reason about flow of values

```
y = 1;  
x = y + 2; // x == 3
```

- No path-specific information

```
if (x == 0)  
    ++x; // x == ?  
else  
    x = 2; // x == 2  
y = x; // x == ?, y == ?
```

Flow-Sensitive Analyses

- Flow-sensitive analyses reason about flow of values

```
y = 1;  
x = y + 2; // x == 3
```

- No path-specific information

```
if (x == 0)  
    ++x; // x == ?  
else  
    x = 2; // x == 2  
y = x; // x == ?, y == ?
```

- LLVM's SSA form designed for flow-sensitive algorithms

Flow-Sensitive Analyses

- Flow-sensitive analyses reason about flow of values

```
y = 1;  
x = y + 2; // x == 3
```

- No path-specific information

```
if (x == 0)  
    ++x; // x == ?  
else  
    x = 2; // x == 2  
y = x; // x == ?, y == ?
```

- LLVM's SSA form designed for flow-sensitive algorithms
- Linear-time algorithms
 - Used by optimization algorithms and compiler warnings

Path-Sensitive Analyses

Path-Sensitive Analyses

- Reason about individual paths and guards on branches

```
if (x == 0)
  ++x;          // x == 1
else
  x = 2;       // x == 2
y = x;        // (x == 1, y == 1) or (x == 2, y == 2)
```

Path-Sensitive Analyses

- Reason about individual paths and guards on branches

```
if (x == 0)
  ++x;          // x == 1
else
  x = 2;       // x == 2
y = x;        // (x == 1, y == 1) or (x == 2, y == 2)
```

- Uninitialized variables example:
 - Path-sensitive analysis picks up only 2 paths
 - No false positive

Path-Sensitive Analyses

- Reason about individual paths and guards on branches

```
if (x == 0)
  ++x;          // x == 1
else
  x = 2;        // x == 2
y = x;          // (x == 1, y == 1) or (x == 2, y == 2)
```

- Uninitialized variables example:
 - Path-sensitive analysis picks up only 2 paths
 - No false positive
- Worst-case exponential-time
 - Complexity explodes with branches and loops
 - Lots of clever tricks to reduce complexity in practice

Path-Sensitive Analyses

- Reason about individual paths and guards on branches

```
if (x == 0)
    ++x;           // x == 1
else
    x = 2;        // x == 2
y = x;           // (x == 1, y == 1) or (x == 2, y == 2)
```

- Uninitialized variables example:
 - Path-sensitive analysis picks up only 2 paths
 - No false positive
- Worst-case exponential-time
 - Complexity explodes with branches and loops
 - Lots of clever tricks to reduce complexity in practice
- Clang static analyzer uses flow- and path-sensitive analyses

Finding leaks in Objective-C code

Memory Management in Objective-C

Objective-C in a Nutshell

- Used to develop Mac/iPhone apps
- C with object-oriented programming extensions

Memory management

- Objective-C objects have embedded reference counts
- Reference counts obey strict ownership idiom
- Garbage collection also available... but there are subtle rules

Ownership Idiom

Ownership Idiom

```
// Allocate an NSString. Since the object is newly allocated,  
// 'str' is an owning reference (+1 retain count).  
NSString* str = [[NSString alloc] initWithCString:"hello world"  
               encoding:NSUTF8StringEncoding];
```

Ownership Idiom

```
// Allocate an NSString. Since the object is newly allocated,  
// 'str' is an owning reference (+1 retain count).  
NSString* str = [[NSString alloc] initWithCString:"hello world"  
               encoding:NSUTF8StringEncoding];  
  
// Pass 'str' to 'foo'. 'foo' may increment the retain  
// count, but we are still obligated to decrement the +1  
// count we have because 'str' is an owning reference.  
foo(str);
```

Ownership Idiom

```
// Allocate an NSString. Since the object is newly allocated,  
// 'str' is an owning reference (+1 retain count).  
NSString* str = [[NSString alloc] initWithCString:"hello world"  
               encoding:NSUTF8StringEncoding];  
  
// Pass 'str' to 'foo'. 'foo' may increment the retain  
// count, but we are still obligated to decrement the +1  
// count we have because 'str' is an owning reference.  
foo(str);  
  
// We're done using str. Decrement our ownership count.  
[str release];
```

Ownership Idiom

```
// Allocate an NSString. Since the object is newly allocated,  
// 'str' is an owning reference (+1 retain count).  
NSString* str = [[NSString alloc] initWithCString:"hello world"  
               encoding:NSUTF8StringEncoding];  
  
// Pass 'str' to 'foo'. 'foo' may increment the retain  
// count, but we are still obligated to decrement the +1  
// count we have because 'str' is an owning reference.  
foo(str);  
  
// We're done using str. Decrement our ownership count.  
// LEAK!
```

Memory Leak: Colloquy

```
32  
33 - (void) interpretKeyEvents:(NSArray *) eventArray {
```

[1] Method returns an object with a +1 retain count (owning reference).

```
34     NSMutableArray *newArray = [[NSMutableArray allocWithZone:nil] initWithObjects:];  
35     NSEnumerator *e = [eventArray objectEnumerator];  
36     NSEvent *anEvent = nil;  
37
```

[2] Taking true branch.

```
38     if( ! [self isEditable] ) {
```

[3] Object allocated on line 34 and stored into 'newArray' is no longer referenced after this point and has a retain count of +1 (object leaked).

```
39         [super interpretKeyEvents:eventArray];  
40         return;  
41     }  
42
```

Ownership DFA

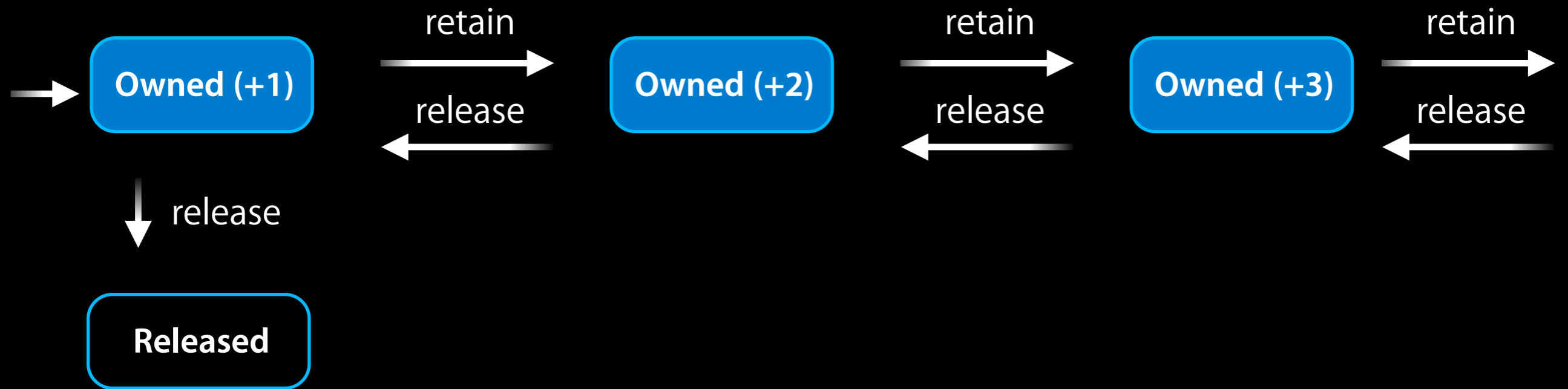
Ownership DFA



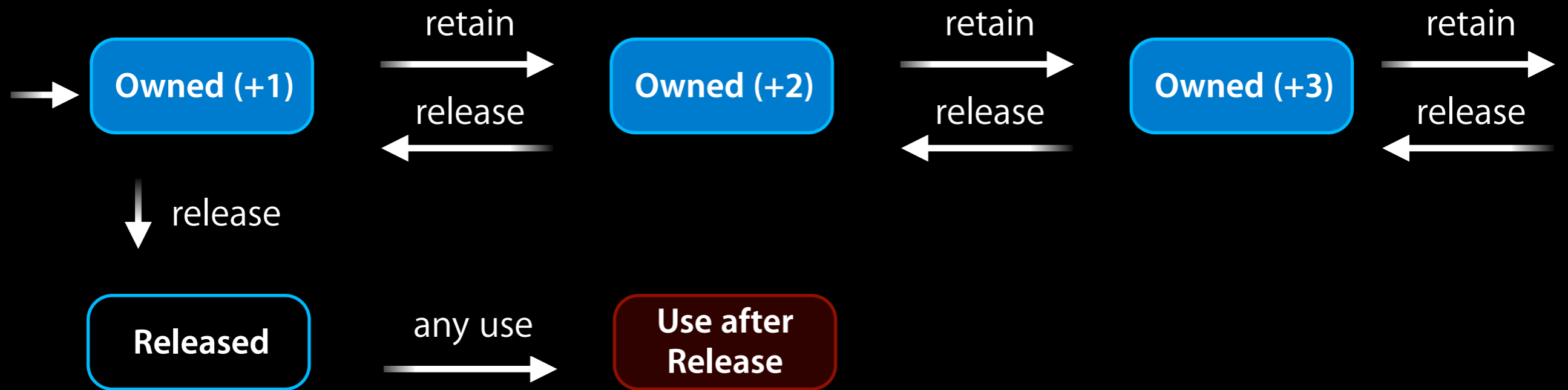
Ownership DFA



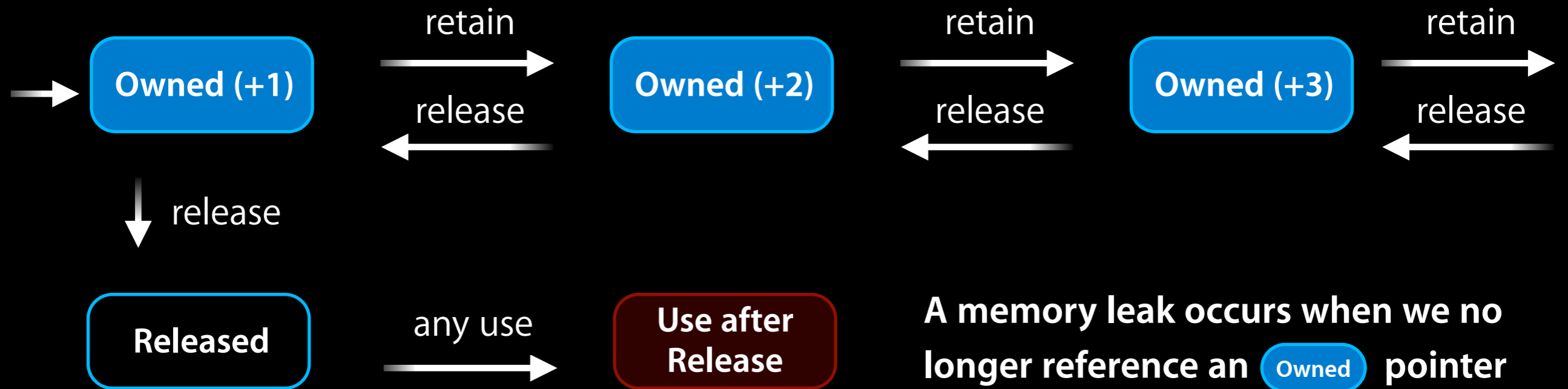
Ownership DFA



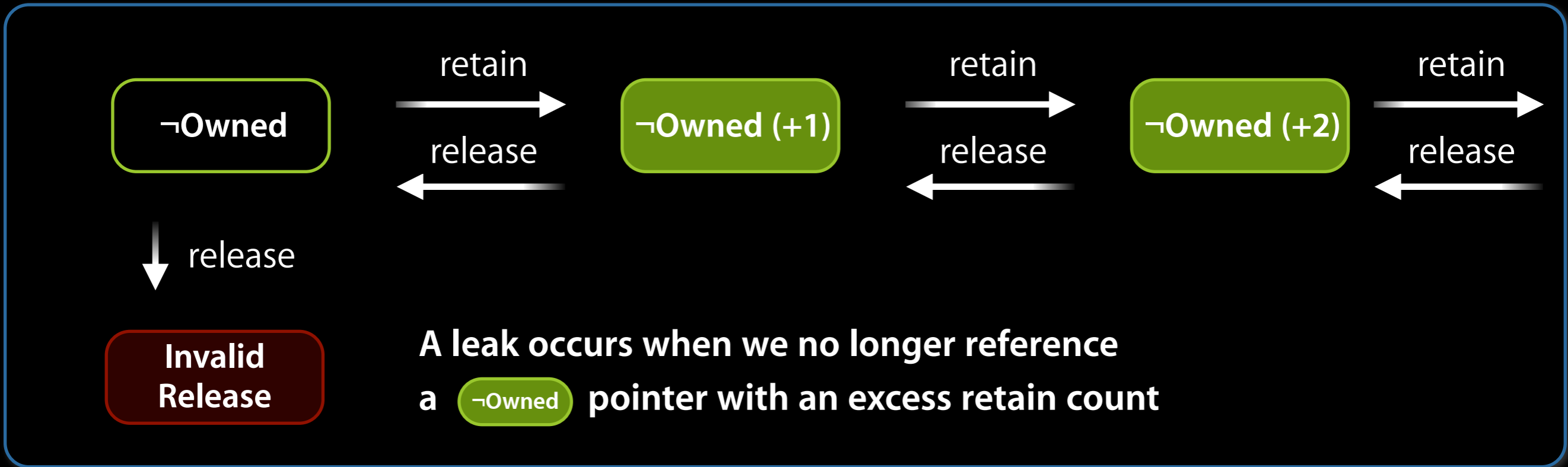
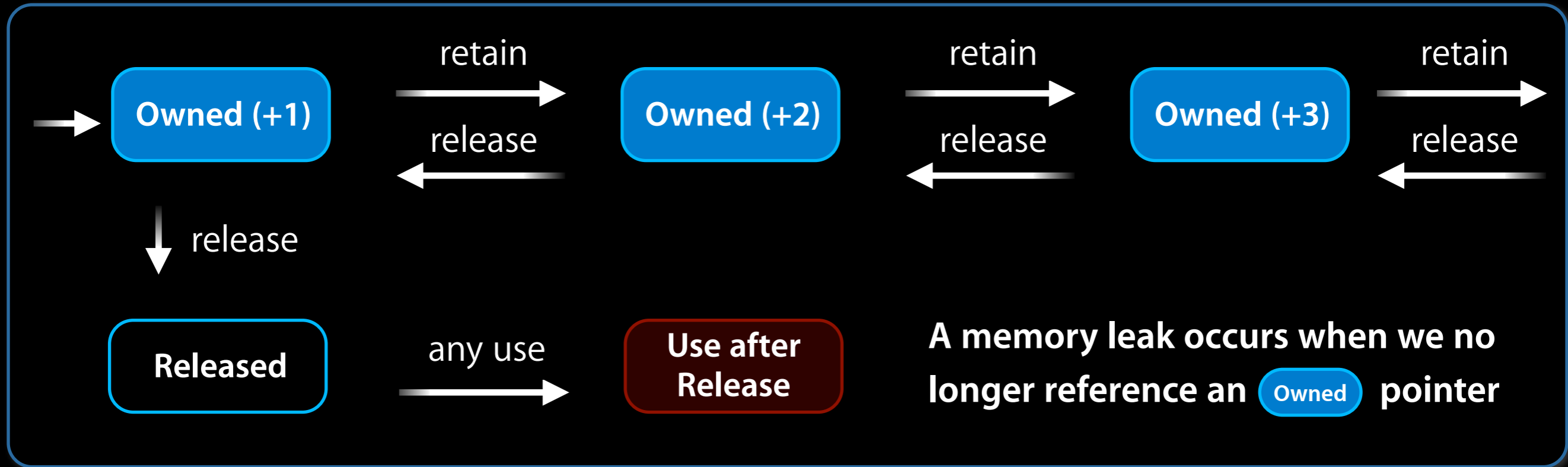
Ownership DFA



Ownership DFA



Ownership DFA



Miscellanea

Checker-specific issues

- Autorelease pools
- Objective-C 2.0 Garbage Collection
- API-specific ownership rules
- Educational diagnostics

Analysis issues

- Aliasing
- Plenty of room for improvement

Checker Results

- Used internally at Apple
- Announced in June 2008 (WWDC)
 - Hundreds of downloads of the static analyzer
 - Thousands of bugs found

Some Implementation Details

Why Analyze Source Code?

Bug-finding requires excellent diagnostics

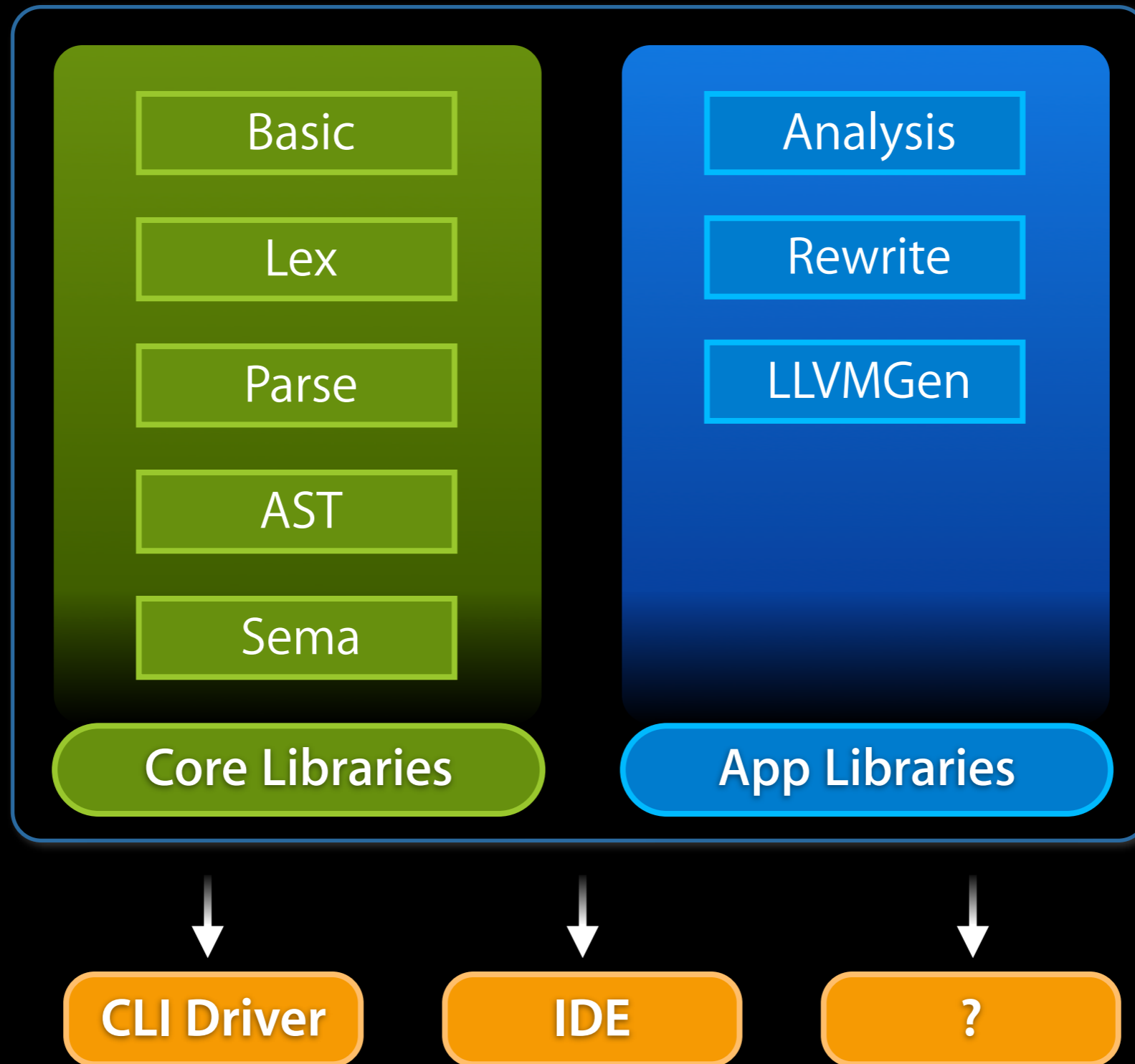
- Tool must **explain** a bug to the user
- Users cannot fix bugs they don't understand
- Need rich source and type information

What about analyzing LLVM IR?

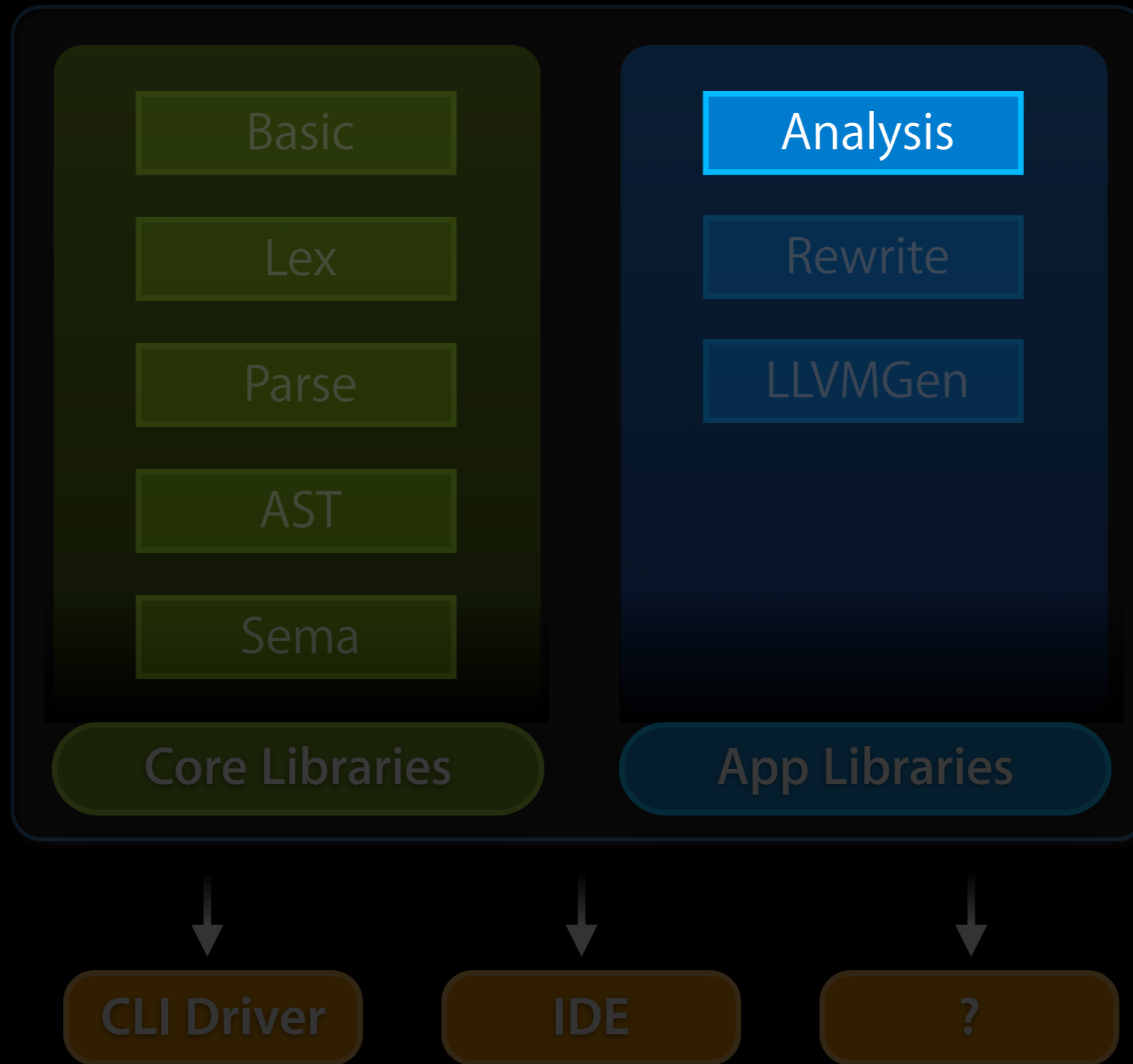
- Loss of source information
- High-level types discarded
- Compiler lowers language constructs
- Compiler makes assumptions (e.g., order of evaluation)

Clang Libraries

Clang Libraries



Clang Libraries



libAnalysis

libAnalysis

Intra-Procedural Analysis

- Source-level Control-Flow Graphs (CFGs)
- Flow-sensitive dataflow solver
 - Live Variables
 - Uninitialized Values
- Path-sensitive dataflow engine
 - Retain/Release checker
 - Logic bugs (e.g., null dereferences)
- Various checks and analyses
 - Dead stores
 - API checks

libAnalysis

libAnalysis

Path Diagnostics (Bug-Reporting)

- PathDiagnosticClient
 - Abstract interface to implement a “view” of bug reports
 - Separates report visualization from generation
 - HTMLDiagnostics (renders HTML, uses libRewrite)
- BugReporter
 - Helper class to generate diagnostics for PathDiagnosticClient

Looking Forward

- Richer Diagnostics
- Inter-procedural Analysis (IPA)
- Lots of Checks
- Scriptability
- Multiple Analysis Engines

Looking Forward

- Richer Diagnostics
- Inter-procedural Analysis (IPA)
- Lots of Checks
- Scriptability
- Multiple Analysis Engines

<http://clang.llvm.org>