Update on the C++ Core Guidelines' Lifetime Safety analysis in Clang

Gábor Horváth
xazax.hun@gmail.com

Agenda

- Motivation
- Whirlwind tour of lifetime analysis
  - See the following talks for details:
    - https://youtu.be/80BZxujhY38?t=1096
    - https://youtu.be/sjnp3P9x5jA
- Highlight some implementation details
- Evaluation
- Upstreaming
- Conclusions

Motivation

- Microsoft: 70 percent of security patches are fixing memory errors
  - https://youtu.be/PjbGojjnBZQ
- C++ has many sources of errors:
  - Manual memory management, temporary objects, Pointer-like objects, …
- Dynamic tools
  - Few false positives, not every arch is supported, coverage is important
- Static tools
  - Arch independent, the earlier a bug is found the cheaper the fix
  - Works without good test coverage

Motivation #2

```
int *p;
{
    int x;
    p = &x;
}
*p = 5;
```

```
string_view sv;
{
    string s{“CoreHard"};
    sv = s;
}
sv[0] = ‘c’;
```

Many static tools warn for the left snippet but not for the right, even though they are fundamentally similar.

Motivation #3

```
std::string_view sv = "test"s;
```

```
int *p = &getIntByValue();
```

A Tour of Herb's Lifetime Analysis

- Intends to catch common errors (not a verification tool)
- Classify types into categories
  - Owners: never dangle, implementation assumed to be correct
  - Pointers: might dangle, tracking points-to sets
  - Aggregates: handled member-wise
  - Values: everything else
- Analysis is function local
- Two implementations
  - We implemented it in a Clang fork
  - Kyle Reed and Neil MacIntosh implemented the MSVC version

A Tour of Herb's Lifetime Analysis #2

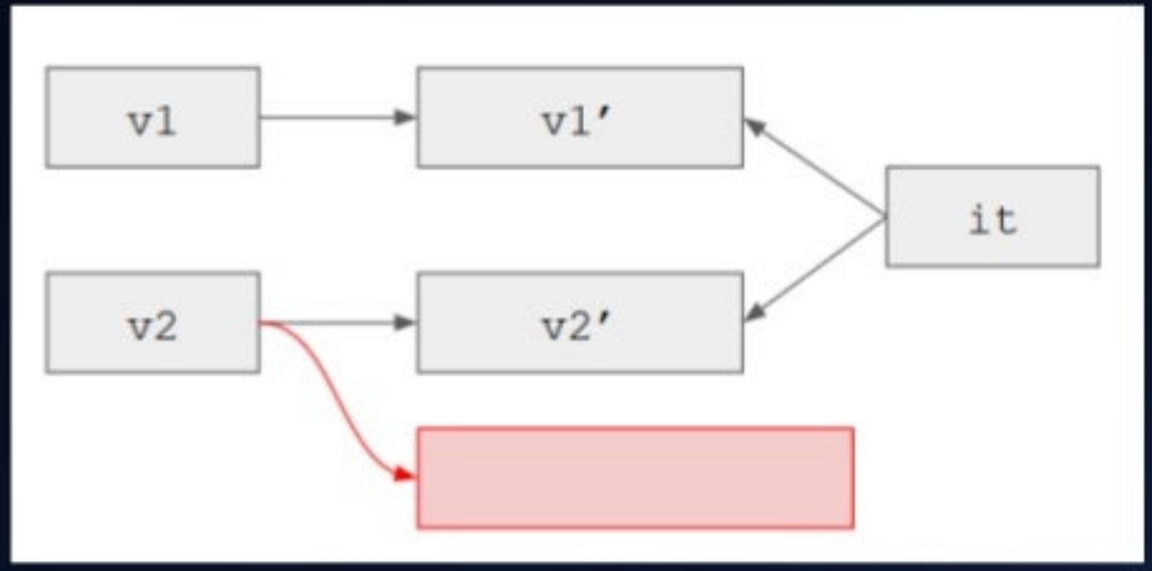
- Flow-sensitive analysis
- We only need annotations for misclassifications (rare)
- Maps each Pointer at each program point to a points-to set
- Elements of a points-to set:
  - Null
  - Invalid
  - Static (lives longer than the pointer or we cannot reason about it)
  - Local variable/parameter
  - Aggregate member
  - Owned memory of an Owner

A Tour of Herb's Lifetime Analysis #3

- https://cppx.godbolt.org/z/IdGRsT

A Tour of Herb's Lifetime Analysis #4

```
vector v1{…};
vector v2{…};
auto it = v1.begin();
if (cond)
    it = v2.begin();
if (cond2)
    v2.push_back(…);
*it = …; // warning
```



Implementation

Analysis Within a Basic Block

```
int x;
int *p  = &x;
int *q  =  p;
```

```
2: x
3: &[B1.2]
4: int *p  = &x;
5: p
6: [B1.5] (LValToRVal)
7: int *q  = p;
```

```
2: {x}
3: {x}
4: pset(p)={x}
5: {p}
6: {x}
7: pset(q)={x}
```

- Basic blocks contain subexprs in an eval order, no AST traversal required
- End of full expression is not marked (apart from `DeclStmt`)
  - When to invalidate Pointers to temporaries?
  - Modified the CFG to include `ExprWithCleanup` AST nodes
- Clang Static Analyzer is another user

Analysis Within a Basic Block

```
int x;
int *p  = &x;
int *q  =  p;
```

```
2: x
3: &[B1.2]
4: int *p  = &x;
5: p
6: [B1.5] (LValToRVal)
7: int *q  = p;
```

```
2: {x}
3: {x}
4: pset(p)={x}
5: {p}
6: {x}
7: pset(q)={x}
```

- Basic blocks contain subexprs in an eval order, no AST traversal required
- End of full expression is not marked (apart from `DeclStmt`)
  - When to invalidate Pointers to temporaries?
  - Modified the CFG to include `ExprWithCleanup` AST nodes
- Clang Static Analyzer is another user

Analysis Within a Basic Block

```
int x;
int *p  = &x;
int *q  =  p;
```

```
2: x
3: &[B1.2]
4: int *p  = &x;
5: p
6: [B1.5] (LValToRVal)
7: int *q  = p;
```

```
2: {x}
3: {x}
4: pset(p)={x}
5: {p}
6: {x}
7: pset(q)={x}
```

- Basic blocks contain subexprs in an eval order, no AST traversal required
- End of full expression is not marked (apart from `DeclStmt`)
  - When to invalidate Pointers to temporaries?
  - Modified the CFG to include `ExprWithCleanup` AST nodes
- Clang Static Analyzer is another user

Analysis Within a Basic Block

```
int x;
int *p  = &x;
int *q  =  p;
```

```
2: x
3: &[B1.2]
4: int *p  = &x;
5: p
6: [B1.5] (LValToRVal)
7: int *q  = p;
```

```
2: {x}
3: {x}
4: pset(p)={x}
5: {p}
6: {x}
7: pset(q)={x}
```

- Basic blocks contain subexprs in an eval order, no AST traversal required
- End of full expression is not marked (apart from `DeclStmt`)
  - When to invalidate Pointers to temporaries?
  - Modified the CFG to include `ExprWithCleanup` AST nodes
- Clang Static Analyzer is another user

Analysis Within a Basic Block

```
int x;
int *p  = &x;
int *q  = p;
```

```
2: x
3: &[B1.2]
4: int *p  = &x;
5: p
6: [B1.5] (LValToRVal)
7: int *q  = p;
```

```
2: {x}
3: {x}
4: pset(p)={x}
5: {p}
6: {x}
7: pset(q)={x}
```

- Basic blocks contain subexprs in an eval order, no AST traversal required
- End of full expression is not marked (apart from `DeclStmt`)
  - When to invalidate Pointers to temporaries?
  - Modified the CFG to include `ExprWithCleanup` AST nodes
- Clang Static Analyzer is another user

Analysis Within a Basic Block

```
int x;
int *p  = &x;
int *q  = p;
```

```
2: x
3: &[B1.2]
4: int *p  = &x;
5: p
6: [B1.5] (LValToRVal)
7: int *q  = p;
```

```
2: {x}
3: {x}
4: pset(p)={x}
5: {p}
6: {x}
7: pset(q)={x}
```

- Basic blocks contain subexprs in an eval order, no AST traversal required
- End of full expression is not marked (apart from `DeclStmt`)
  - When to invalidate Pointers to temporaries?
  - Modified the CFG to include `ExprWithCleanup` AST nodes
- Clang Static Analyzer is another user

Analysis Within a Basic Block

```
int x;
int *p  = &x;
int *q  = p;
```

```
2: x
3: &[B1.2]
4: int *p  = &x;
5: p
6: [B1.5] (LValToRVal)
7: int *q  = p;
```

```
2: {x}
3: {x}
4: pset(p)={x}
5: {p}
6: {x}
7: pset(q)={x}
```

- Basic blocks contain subexprs in an eval order, no AST traversal required
- End of full expression is not marked (apart from `DeclStmt`)
  - When to invalidate Pointers to temporaries?
  - Modified the CFG to include `ExprWithCleanup` AST nodes
- Clang Static Analyzer is another user

Analysis Within a Basic Block

```
int x;
int *p  = &x;
int *q  =  p;
```

```
2: x
3: &[B1.2]
4: int *p  = &x;
5: p
6: [B1.5] (LValToRVal)
7: int *q  = p;
```

```
2: {x}
3: {x}
4: pset(p)={x}
5: {p}
6: {x}
7: pset(q)={x}
```

- Basic blocks contain subexprs in an eval order, no AST traversal required
- End of full expression is not marked (apart from `DeclStmt`)
  - When to invalidate Pointers to temporaries?
  - Modified the CFG to include `ExprWithCleanup` AST nodes
- Clang Static Analyzer is another user

Analysis Within a Basic Block

```
int x;
int *p  = &x;
int *q  =  p;
```

```
2: x
3: &[B1.2]
4: int *p  = &x;
5: p
6: [B1.5] (LValToRVal)
7: int *q  = p;
```

```
2: {x}
3: {x}
4: pset(p)={x}
5: {p}
6: {x}
7: pset(q)={x}
```

- Basic blocks contain subexprs in an eval order, no AST traversal required
- End of full expression is not marked (apart from `DeclStmt`)
  - When to invalidate Pointers to temporaries?
  - Modified the CFG to include `ExprWithCleanup` AST nodes
- Clang Static Analyzer is another user

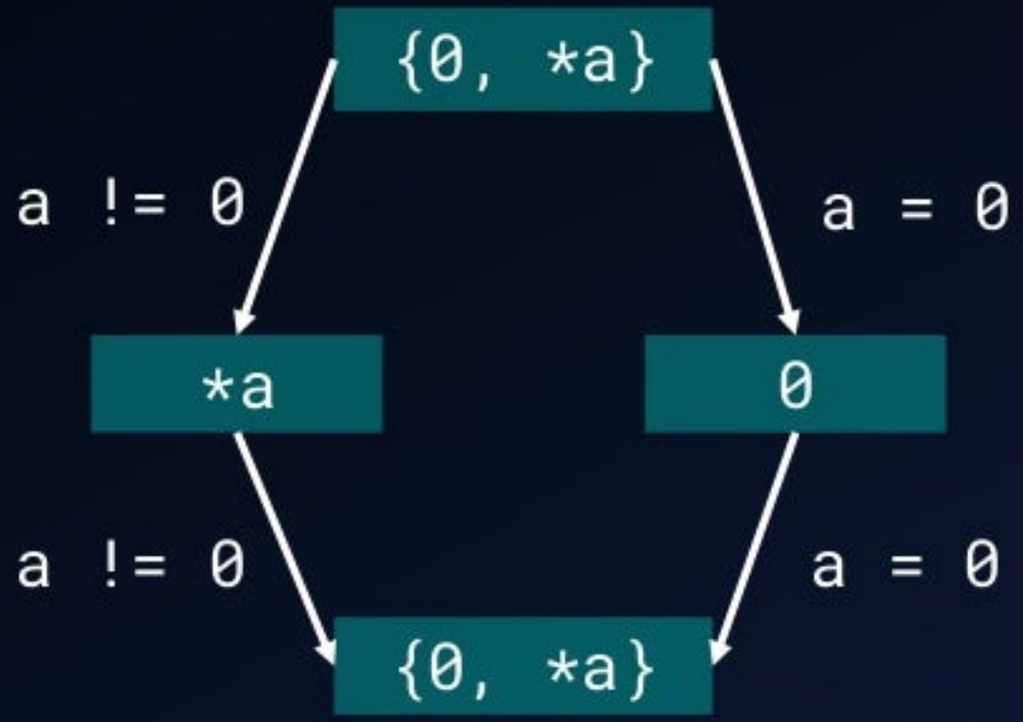
Analysis on the CFG Level – Merging Points-to Sets

- Calculate points-to sets within each basic block
- Merge incoming points-to sets on basic block entry
- Fixed-point iteration
  - Loops

```
int* p;
// pset(p) = {(invalid)}
if (cond) {
    p = &i;
    // pset(p) = {i}
} else {
    p = nullptr;
    // pset(p) = {(null)}
}
// pset(p) = {i, (null)}
```

Analysis on the CFG Level – Dealing with Forks

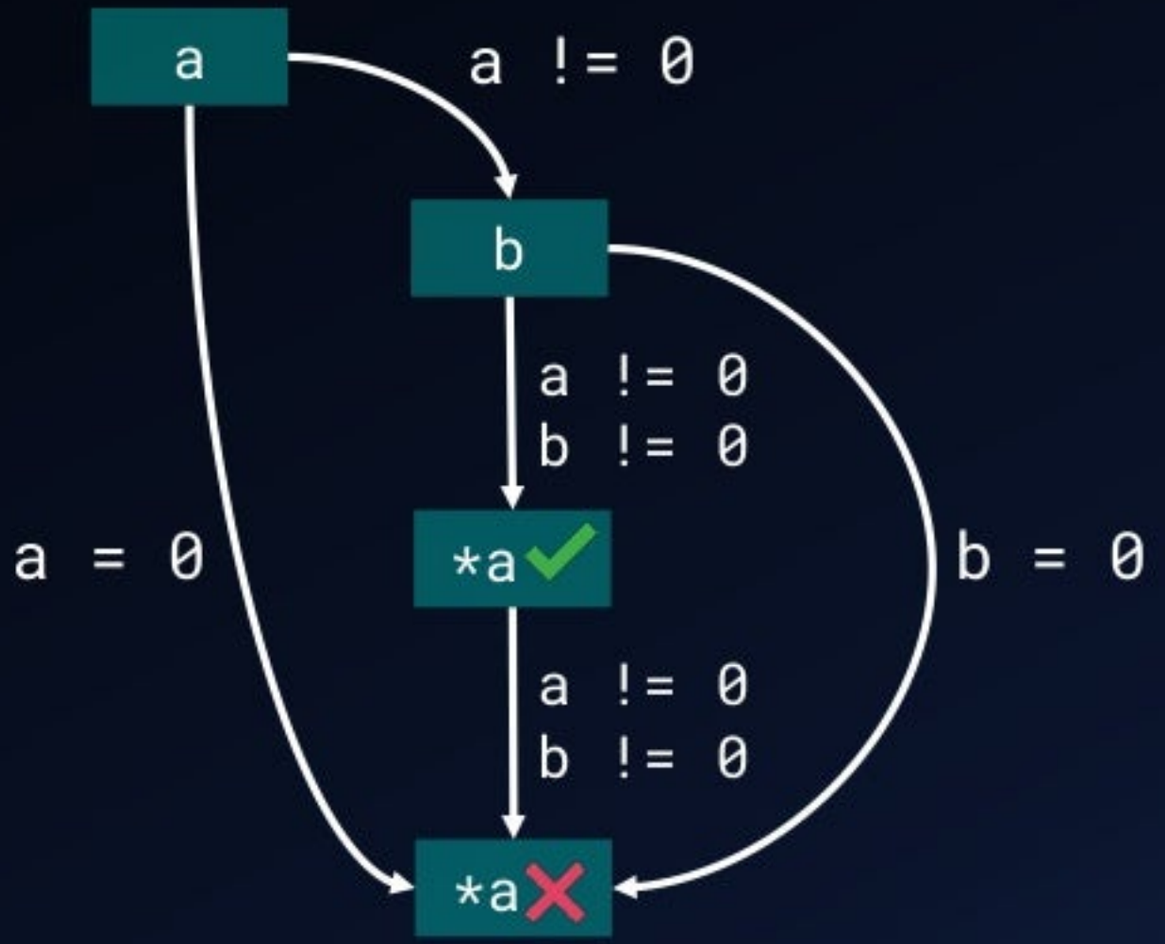
```
void f(int* a) {
  // pset(a) = {(null), *a)
  if (a) {
    // pset(a) = {*a}
  } else {
    // pset(a) = {(null)}
  }
  // pset(a) = {(null), *a)
}
```



Tracking Null Pointers – Logical operators

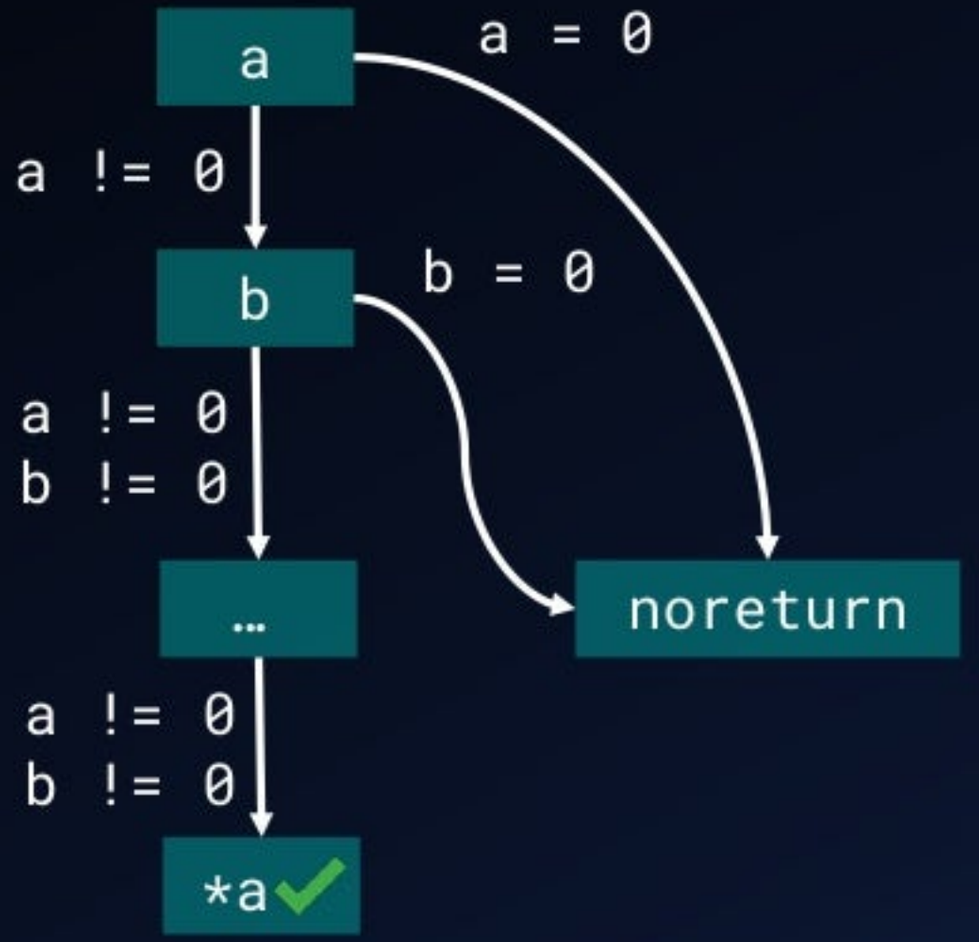
```
if (a && b) {
    *a;
}
*a;
```

```
if (a) {
    if (b) {
        *a; // OK
    }
}
*a; // warning
```



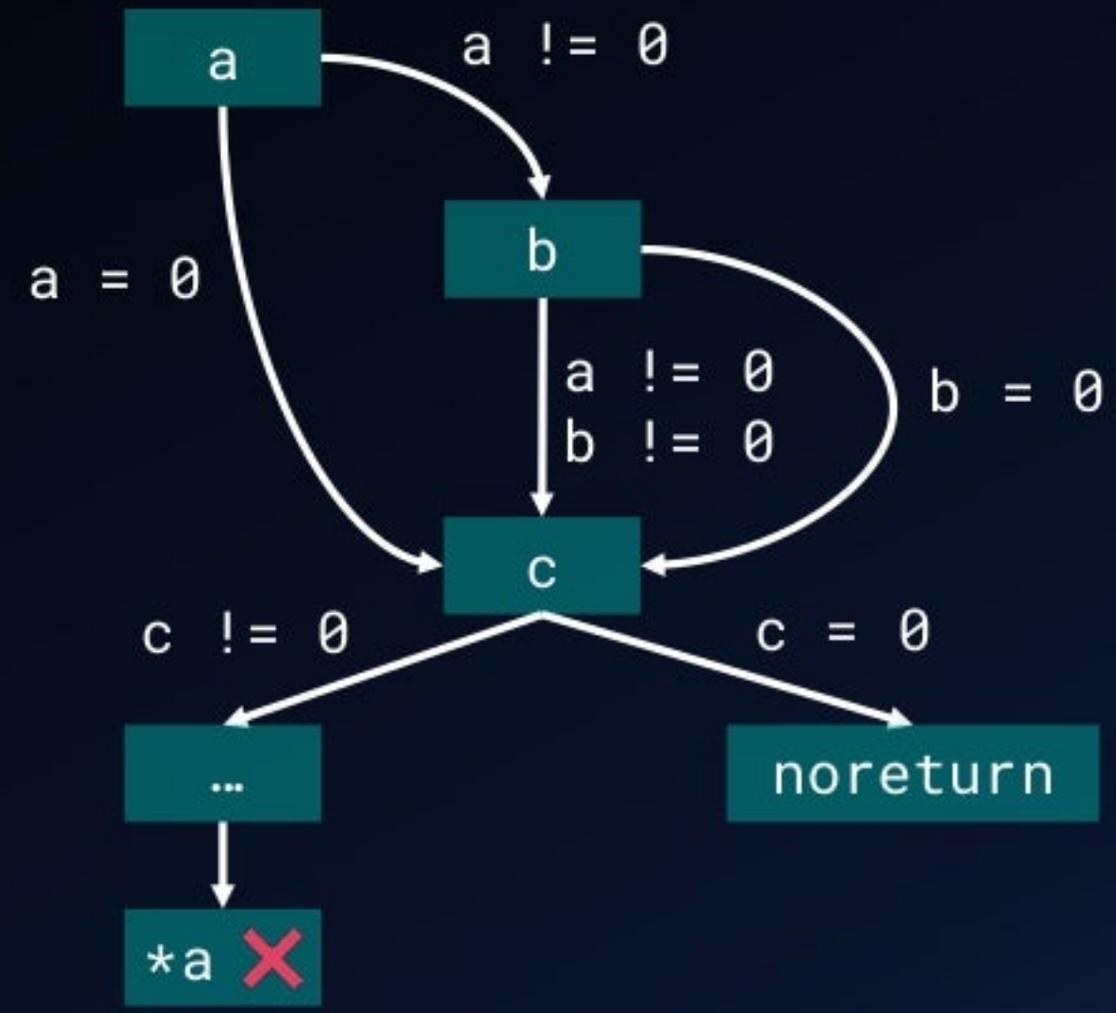
Tracking Null Pointers – The Role of noreturn

```
(a && b)? … : noreturn();
*a;
```



Tracking Null Pointers – Merging Too Early

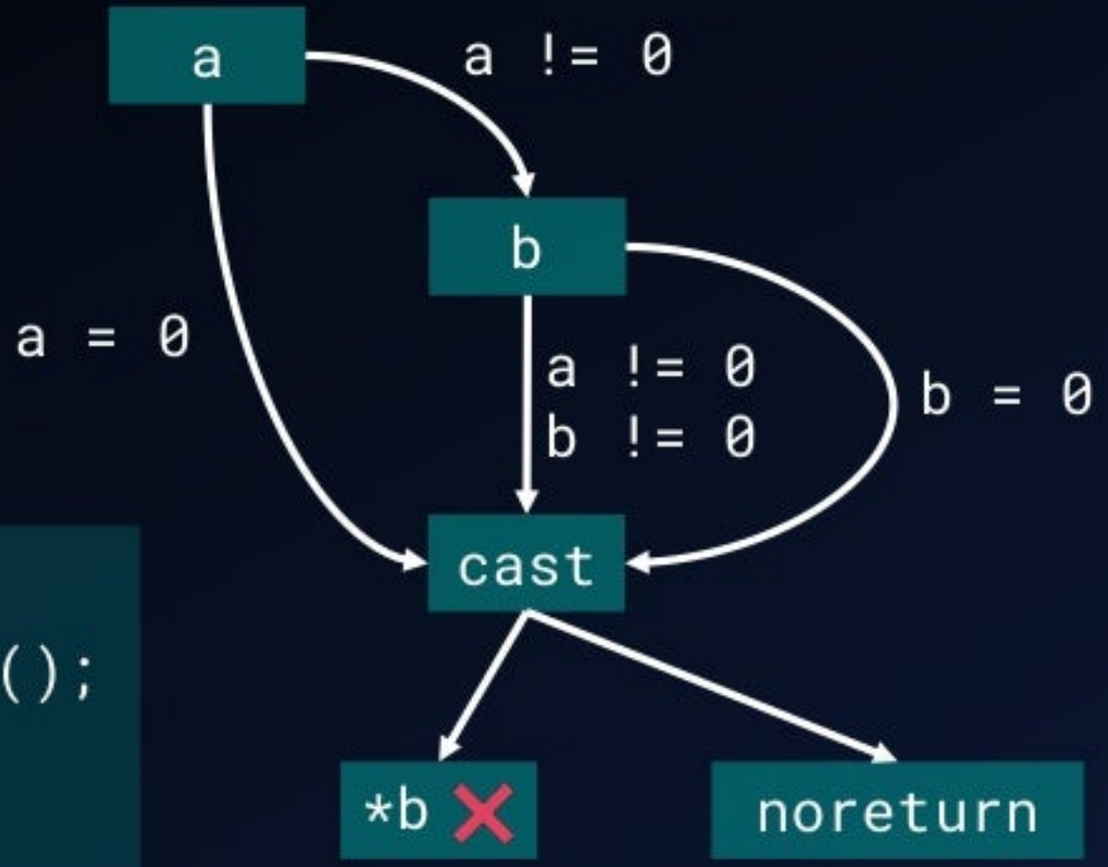
```
bool c = a && b;
c ? … : noreturn();
*a; // false positive
```



Tracking Null Pointers – Challenges with Assertions

```
void f(int* a, int *b) {
  assert(a && b);
  *b;
}
```

```
void f(int* a, int *b) {
  (bool)(a && b)? … : noreturn();
  *b; // false positive
}
```



Summary of Flow-Sensitive Lifetime Analysis

- The performance overhead of the prototype is less than 10% of `-fsyntax-only`
- 3 sources of false positives:
  - Infeasible paths
  - Miscategorizations
  - Function modelling

Lifetime analysis for the rest of us

Typical Lifetime Issues

```
reference_wrapper<int> data() {
  int i = 3;
  return {i};
}
```

```
S& V = *get();
```

```
auto add(int a) {
  return [&a](int b) {
    return a + b;
  };
}
```

```
return o->name().c_str();
```

```
string_view sv = "test"s;
```

```cpp
#include <vector>
#include <optional>

std::vector<int> getVec();
std::optional<std::vector<int>> getOptVec();

void sj3() {
  int sum = 0;
  for (int value : getVec()) {
      sum += value;
  }

  for (int value : *getOptVec()) {
      sum += value;
  }
}
```

```
<source>:13:18:
warning: passing a
dangling pointer
as argument [-
Wlifetime]
  for (int value :
*getOptVec()) {
                ^
<source>:13:20:
note: temporary
was destroyed at
the end of the
full expression
  for (int value :
*getOptVec()) {
^
1 warning
generated.
Compiler returned:
0
```

Goal: Enable a Subset of Lifetime Warnings with No False Positives

Clang warnings exist for:

```
struct Y {
    int *p;
    Y(int i) : p(&i) {}
};
```

```
int *data() {
  int i = 3;
  return &i;
}
```

```
new initializer_list<int>{1, 2, 3};
```

Let’s generalize them!

Evaluation of the Statement Local Analysis

- No false positives or true positives for LLVM and Clang head
  - Few FPs if we categorize every user defined type
  - FPs could be fixed with annotating `llvm::ValueHandleBase`
- Sample of 22 lifetime related fixes
  - Faulty commits passed the reviews
  - 11 would have been caught before breaking the bots
  - 1 false negative due to `Path` not being automatically categorized as owner
  - 3 are missed due to assignments not being checked
- Less than 1% performance overhead

What is the Issue Here?

- Faulty:

```
StringRef Prefix = is_abs(dir)
    ? SysRoot : "";
```

- Fixed:

```
StringRef Prefix = is_abs(dir)
    ? StringRef(SysRoot) : "";
```

- Contextual information is required to catch the problem

Other True Positive Findings

- `cplusplus.InnerPointer` check of the Clang Static Analyzer found 3 true positives in Ceph, Facebook's RocksDB, GPGME
  - GSoC 2018 project by Réka Kovács
  - Problems were reported and fixed promptly
- The true positives were all statement local problems
- The same true positives can also be found with our statement-local analysis
- How many true positives would we expect from the original warnings?

Plans for upstreaming

- Annotations
  - Other analyses can start to adopt to type
- Generalize warnings
  - On by default for STL and explicitly annotated types
- Type category inference is controversial
  - Plans for a tool that inserts inferred annotations
- Add flow sensitive analysis
  - First handle function calls conservatively
  - Add further annotations
  - Infer annotations for functions
  - Implement use-after-move checks, add exception support

Conclusions

- Herb's analysis is useful for new projects, not always applicable to old
- Type categories are useful for other analyses
  - Generalizing Clang warnings
  - Generalizing CSA checks
  - Generalizing Tidy checks
- Generalized warnings has low performance impact, all sources of false positives can be addressed
  - Infeasible paths → statement local analysis
  - Miscategorization → only trigger for STL and annotated types
  - Function modelling → only rely on known functions

Thank you!

- Clang implementation
  - https://github.com/mgehre/clang
- Lifetime paper
  - https://herbsutter.com/2018/09/20/lifetime-profile-v1-0-posted/
- MSVC implementation
  - https://devblogs.microsoft.com/cppblog/lifetime-profile-update-in-visual-studio-2019-preview-2/