Faster, Stronger C++ Analysis with the Clang Static Analyzer

George Karpenkov, Apple
Artem Dergachev, Apple

Agenda

Introduction to Clang Static Analyzer

Using coverage-based iteration order

Improved C++ constructor and destructor support

Agenda

Introduction to Clang Static Analyzer

Using coverage-based iteration order

Improved C++ constructor and destructor support

Clang Static Analyzer Finds Bugs at Compile Time

- Use-after-free bugs
- Null pointer dereferences
- Uses of uninitialized values
- Memory leaks, etc…

Analyzer Visualizes Paths

- Inside IDE: Xcode, QtCreator, CodeCompass
- From command line: generate HTML
  - `$ scan-build make`
- http://clang-analyzer.llvm.org

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  extern void work(int *p);
5
6  void log_value(int *x) {
7      printf("Value of x = %d\n", *x);
8  }
9
10 void foo() {
11     int *x = (int *) malloc(sizeof(int));
           1 Memory is allocated →
12     *x = 0;
13     log_value(x);
14     work(x);
15     free(x);
           2 ← Memory is released →
16     log_value(x);
           3 ← Use of memory after it is freed
17 }
```

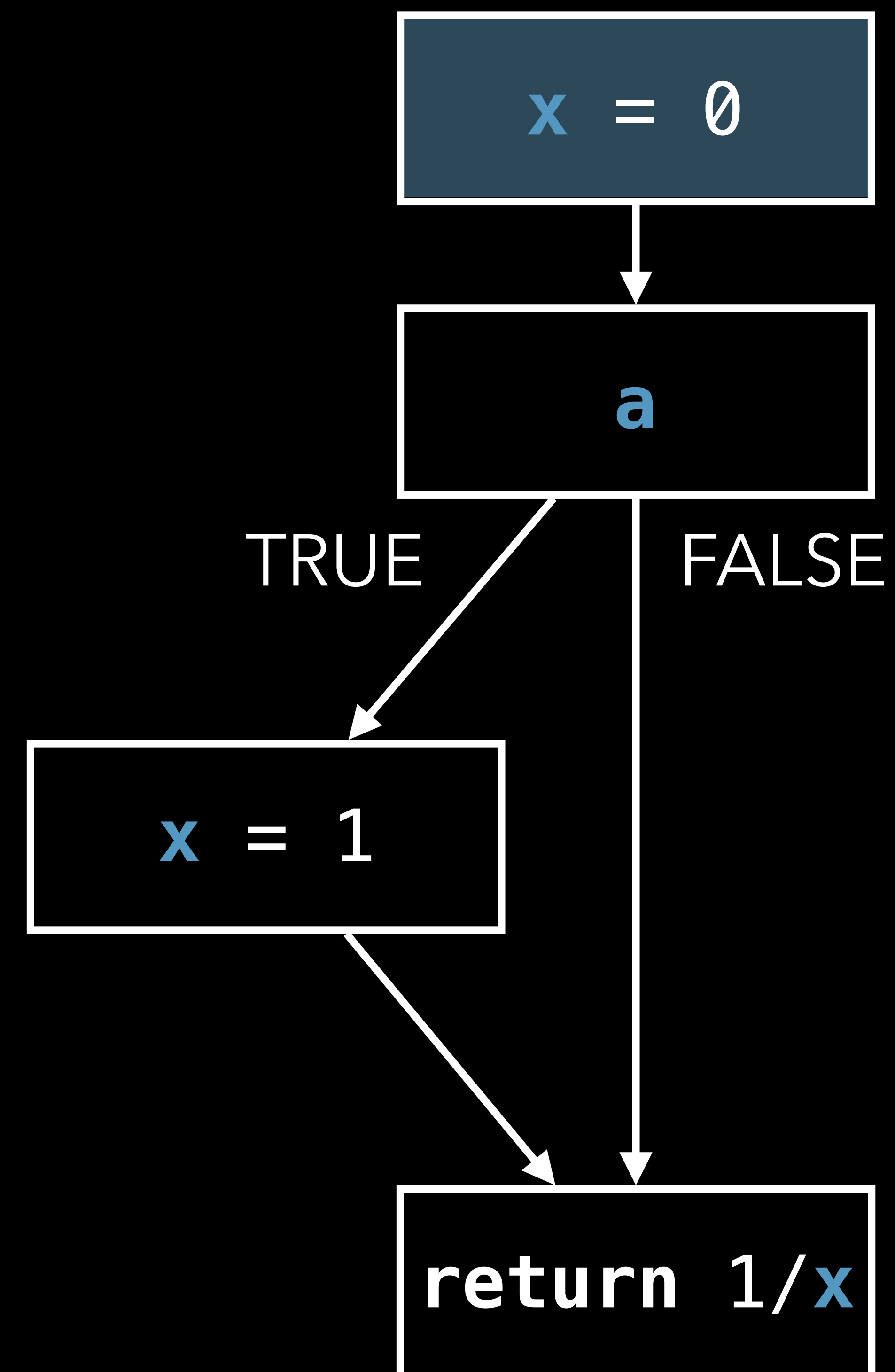
Analyzer Simulates Program Execution

- Explores paths through the program
- Uses symbols instead of concrete values
- Generates reports on errors

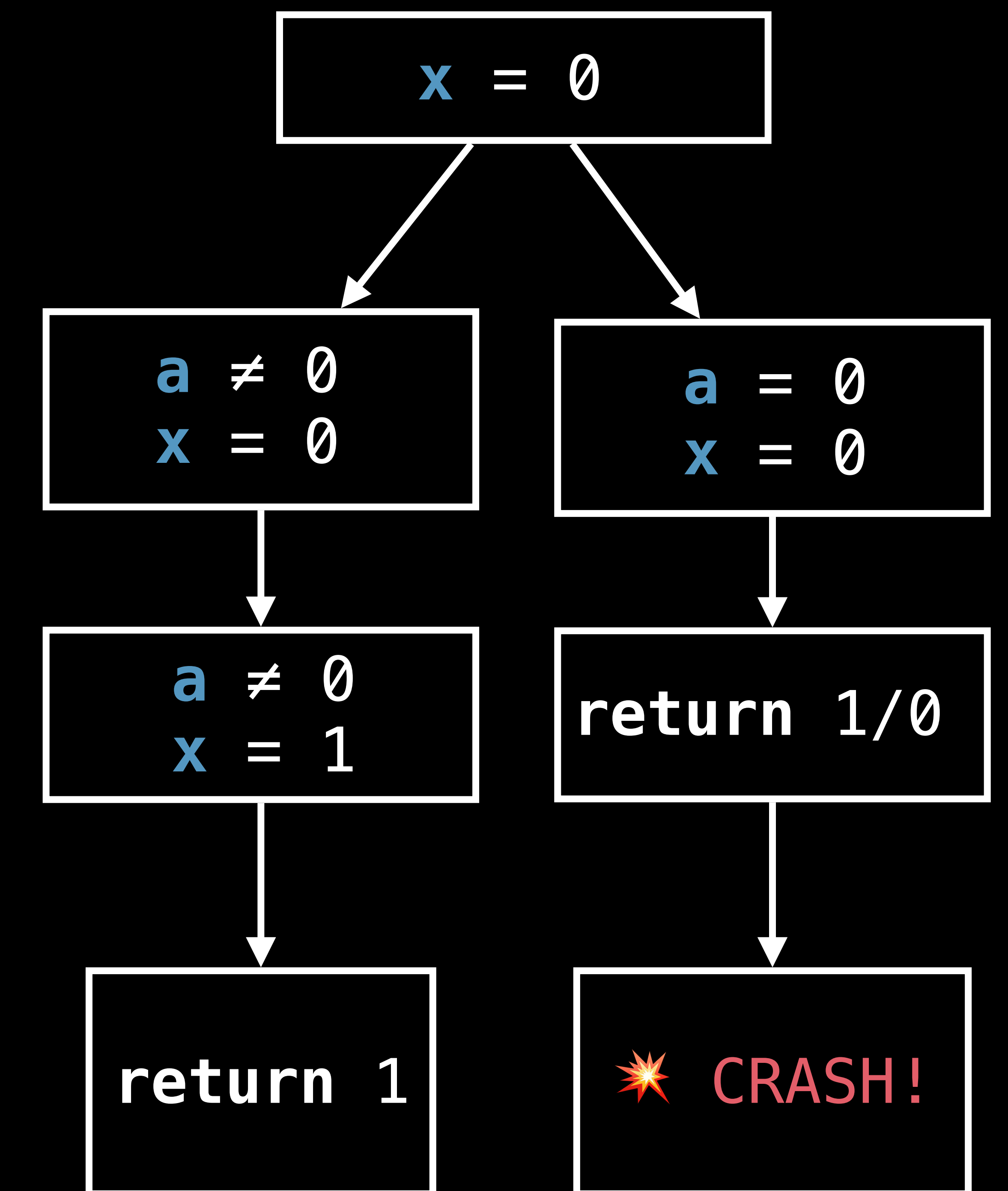
A Faster than Light Intro to the Analyzer

```
int foo(int a) {
  int x = 0;
  if (a != 0)
    x = 1;
  return 1/x;
}
```

Code



Control Flow Graph



Exploded Graph

Agenda

Introduction to Clang Static Analyzer

Using coverage-based iteration order

Improved C++ constructor and destructor support

```
5536 int
5537 nfs_dir_buf_search(
5538         struct nfsbuf *bp,
5539         struct componentname *cnp,
5540         fhandle_t *fhp,
5541         struct nfs_vattr *nvap,
5542         uint64_t *xidp,
5543         time_t *attrstampp,
5544         daddr64_t *nextlbnp,
5545         int flags)
5546 {
5547         struct direntry *dp;
5548         struct nfs_dir_buf_header *ndbhp;
5549         struct nfs_vattr *nvattrp;
5550         daddr64_t nextlbn = 0;
5551         int i, error = ESRCH;
5552         uint32_t fhlen;
5555         ndbhp = (struct nfs_dir_buf_header*)bp->nb_data;
5556         dp = NFS_DIR_BUF_FIRST_DIRENTRY(bp);
5557         for (i=0; i < ndbhp->ndbh_count; i++) {
                         13 ← Assuming the condition is false →
                 14 ← Loop condition is false. Execution continues on line 5597 →
                         26 ← Assuming the condition is true →
                 27 ← Loop condition is true.  Entering loop body →
                         29 ← Assuming the condition is true →
                 30 ← Loop condition is true.  Entering loop body →
                         32 ← Assuming the condition is true →
                 33 ← Loop condition is true.  Entering loop body →
5558                 nextlbn = dp->d_seekoff;
5559                 if ((cnp->cn_namelen == dp->d_namlen) && !strcmp(cnp->cn_nameptr, dp->d_name)) {
                         28 ← Assuming the condition is false →
                         31 ← Assuming the condition is false →
                         34 ← Assuming the condition is true →
                     35 ← Taking true branch →
5560                         fhlen = dp->d_name[dp->d_namlen+1];
5561                         nvattrp = NFS_DIR_BUF_NVATTR(bp, i);
5562                         if ((ndbhp->ndbh_ncgen != bp->nb_np->n_ncgen) || (fhp->fh_len == 0) ||
                             36 ← Assuming the condition is false →
                                                                     37 ← The left operand of '==' is a garbage value
5595                 dp = NFS_DIRENTRY_NEXT(dp);
5597         if (nextlbnp)
                 15 ← Taking true branch →
5598                 *nextlbnp = nextlbn;
5599         return (error);
                 16 ← Returning without writing to 'fhp->fh_len' →
5606 int
5607 nfs_dir_buf_cache_lookup(nfsnode_t dnp, nfsnode_t *npp, struct componentname *cnp, vfs_context_t ctx, int purge)
5608 {
5609         nfsnode_t newnp;
5610         struct nfsmount *nmp;
5611         int error = 0, i, found = 0, count = 0;
5612         u_int64_t xid;
5613         struct nfs_vattr nvattr;
5614         fhandle_t fh;
5615         time_t attrstamp = 0;
5616         thread_t thd = vfs_context_thread(ctx);
5617         struct nfsbuf *bp, *lastbp, *foundbp;
5618         struct nfsbuflists blist;
5619         daddr64_t lbn, nextlbn;
5620         int dotunder = (cnp->cn_namelen > 2) && (cnp->cn_nameptr[0] == '.') && (cnp->cn_nameptr[1] == '_');
                             1 Assuming the condition is false →
5622         nmp = NFSTONMP(dnp);
5623         if (nfs_mount_gone(nmp))
                 2 ← Assuming the condition is false →
             3 ← Taking false branch →
5625         if (!purge)
                 4 ← Assuming 'purge' is not equal to 0 →
             5 ← Taking false branch →
5629         lbn = dnp->n_lastdbl;
5630         for (i=0; i < 2; i++) {
             6 ← Loop condition is true.  Entering loop body →
             19 ← Loop condition is true.  Entering loop body →
5631                 if ((error = nfs_buf_get(dnp, lbn, NFS_DIRBLKSIZ, thd, NBLK_READ|NBLK_ONLYVALID, &bp)))
                         7 ← Assuming 'error' is zero →
                     8 ← Taking false branch →
                         20 ← Assuming 'error' is zero →
                     21 ← Taking false branch →
5633                 if (!bp)
                         9 ← Assuming 'bp' is non-null →
                     10 ← Taking false branch →
                         22 ← Assuming 'bp' is non-null →
                     23 ← Taking false branch →
5635                 count++;
5636                 error = nfs_dir_buf_search(bp, cnp, &fh, &nvattr, &xid, &attrstamp, &nextlbn, purge ? NDBS_PURGE : 0);
                                                                     11 ← '?' condition is true →
                         12 ← Calling 'nfs_dir_buf_search' →
                         17 ← Returning from 'nfs_dir_buf_search' →
                                                                     24 ← '?' condition is true →
                         25 ← Calling 'nfs_dir_buf_search' →
5637                 nfs_buf_release(bp, 0);
5638                 if (error == ESRCH) {
                     18 ← Taking true branch →
5639                         error = 0;
5644                 lbn = nextlbn;
```

```
544         daddr64_t *nextlbnp,
545         int flags)
546 {
547         struct direntry *dp;
548         struct nfs_dir_buf_header *ndbhp;
549         struct nfs_vattr *nvattrp;
550         daddr64_t nextlbn = 0;
551         int i, error = ESRCH;
552         uint32_t fhlen;
555         ndbhp = (struct nfs_dir_buf_header*)bp->nb_data;
556         dp = NFS_DIR_BUF_FIRST_DIRENTRY(bp);
557         for (i=0; i < ndbhp->ndbh_count; i++) {
                    13 ← Assuming the condition is false →
            14 ← Loop condition is false. Execution continues on line 5597 →
                    26 ← Assuming the condition is true →
            27 ← Loop condition is true.  Entering loop body →
                    29 ← Assuming the condition is true →
            30 ← Loop condition is true.  Entering loop body →
                    32 ← Assuming the condition is true →
            33 ← Loop condition is true.  Entering loop body →
558                 nextlbn = dp->d_seekoff;
559                 if ((cnp->cn_namelen == dp->d_namlen) && !strcmp(cnp->cn_nameptr, dp->d_name)) {
                      28 ← Assuming the condition is false →
                      31 ← Assuming the condition is false →
                      34 ← Assuming the condition is true →
                  35 ← Taking true branch →
```

Problem: Path is Too Long

- XNU (Darwin Kernel): many paths over 400 steps
- Bug can be found on the first iteration
- Aim: provide **shorter**, more **concise** diagnostics

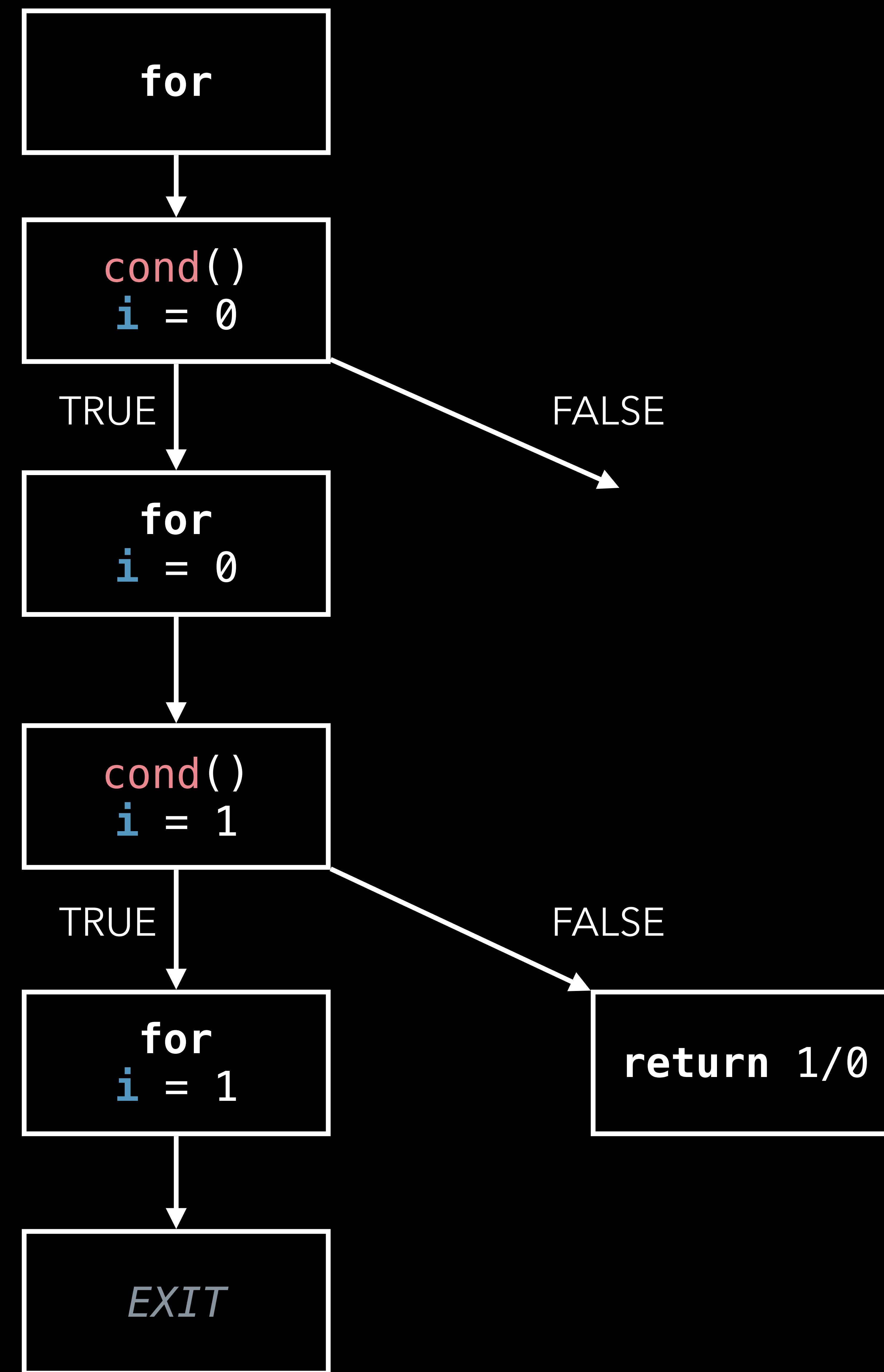
Analyzer Uses Worklist to Generate Exploded Graph

```
worklist = {start}
while worklist:
  node = worklist.pop()
  successors = execute(node)
  for successor in successors:
    worklist.push(successor)
```

- Start: entry point
- Successors:
  - Simulated execution of a statement
- Allows different exploration strategies
  - Previously: DFS by default

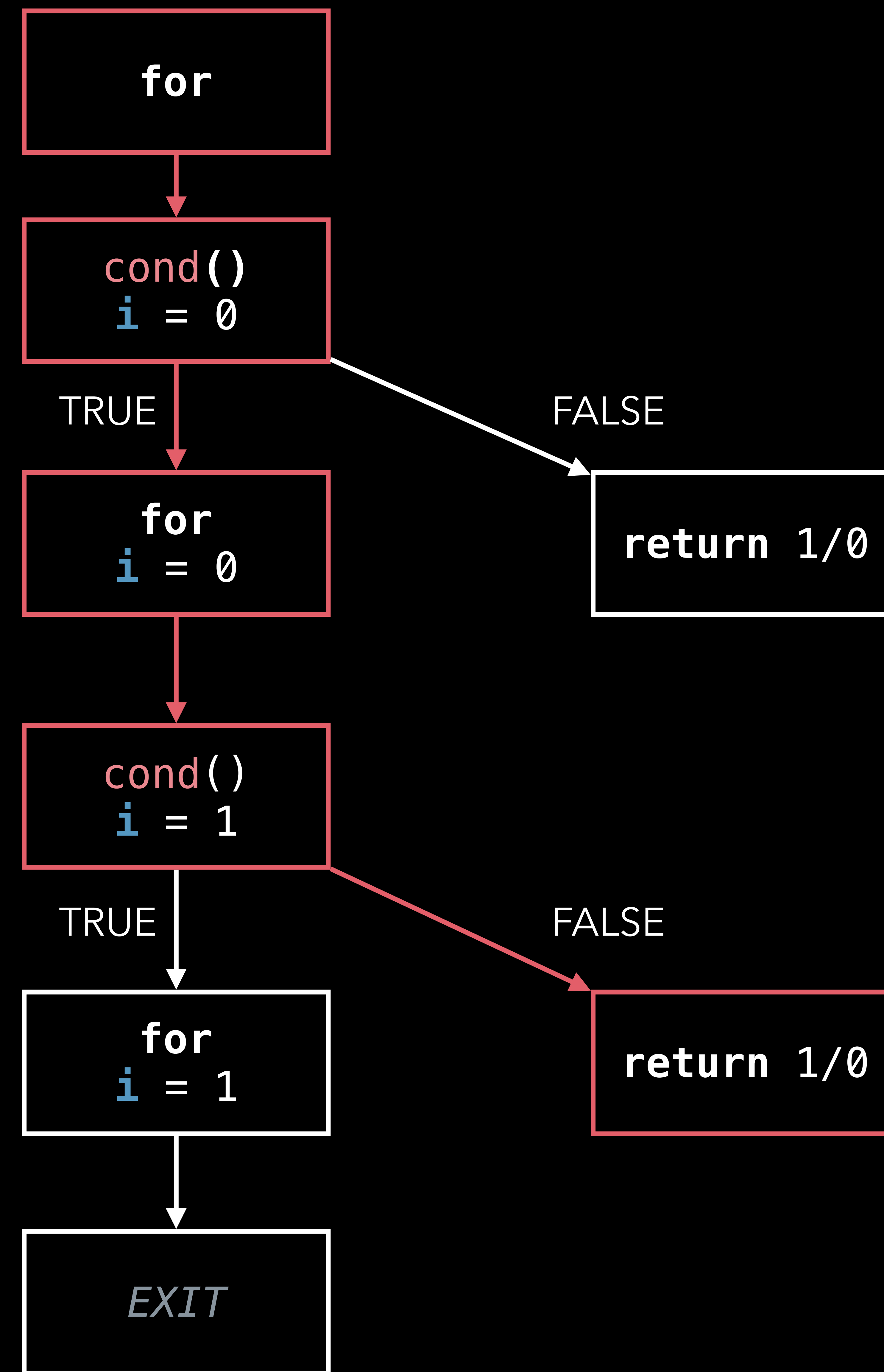
DFS Exploration Order Leads to Wasted Effort

```
int main() {
  for (int i = 0; i < 2; ++i) {
    if (cond())
      continue;
    return 1/0; // 💥 crash
  }
}
```



DFS Exploration Order Leads to Wasted Effort

```
int main() {
  for (int i = 0; i < 2; ++i) {
    if (cond())
      continue;
    return 1/0; // 💥 crash
  }
}
```



Problem Often Mitigated by Analyzer Heuristics

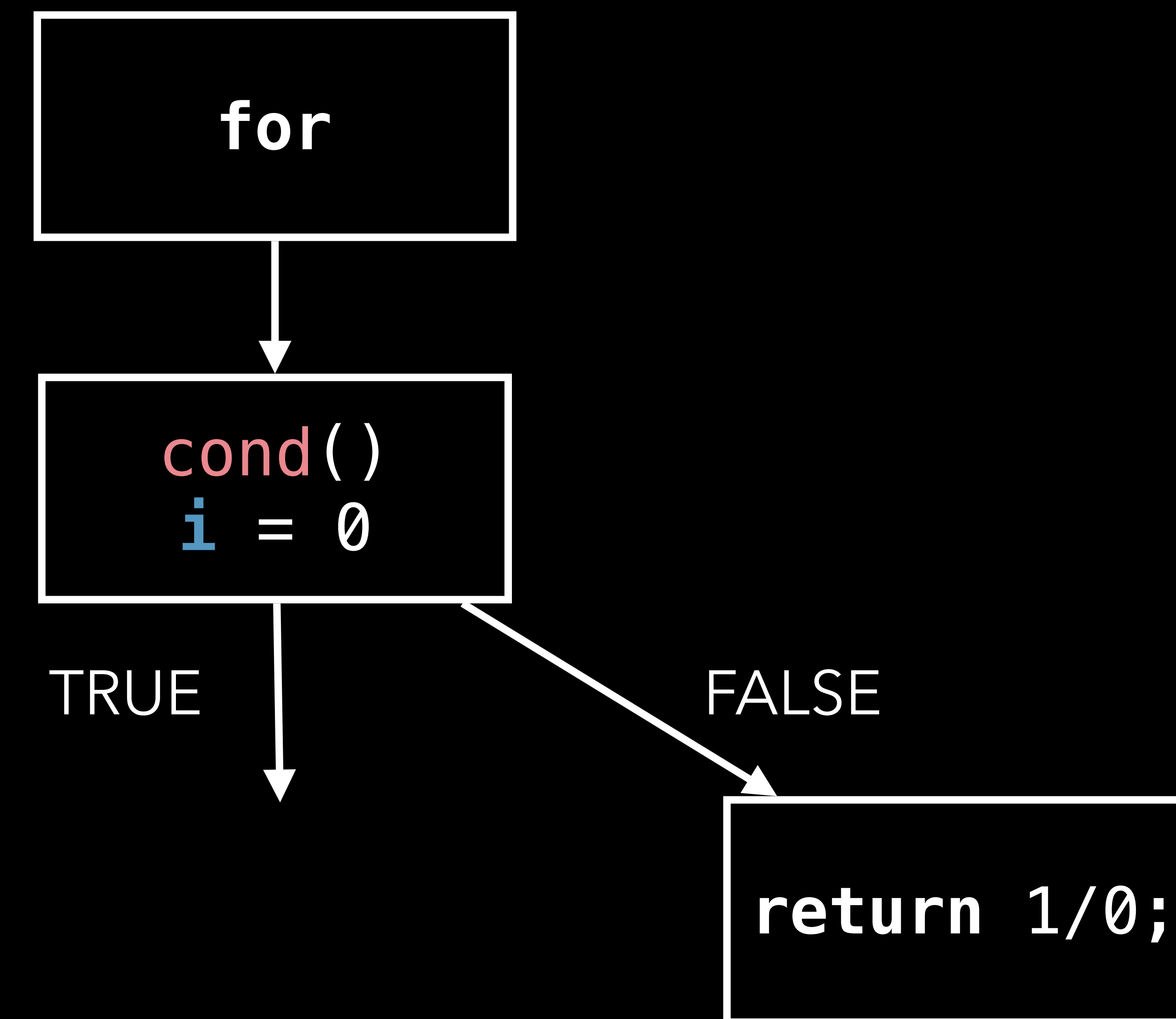
- Deduplication
    - If same report is found multiple times, return shortest path
- Budget per source location
    - Paths that visit a location more than 3 times get dropped
- Budget per number of inlinings
- …
- In many **unfortunate** cases, shortest path not found at all

Solution: Coverage-Based Iteration order

- Record the number of times the analyzer visits each location
- Use a priority queue:
    - Prefers source locations analyzer has visited fewer times so far
- Finds bugs on first iteration when possible

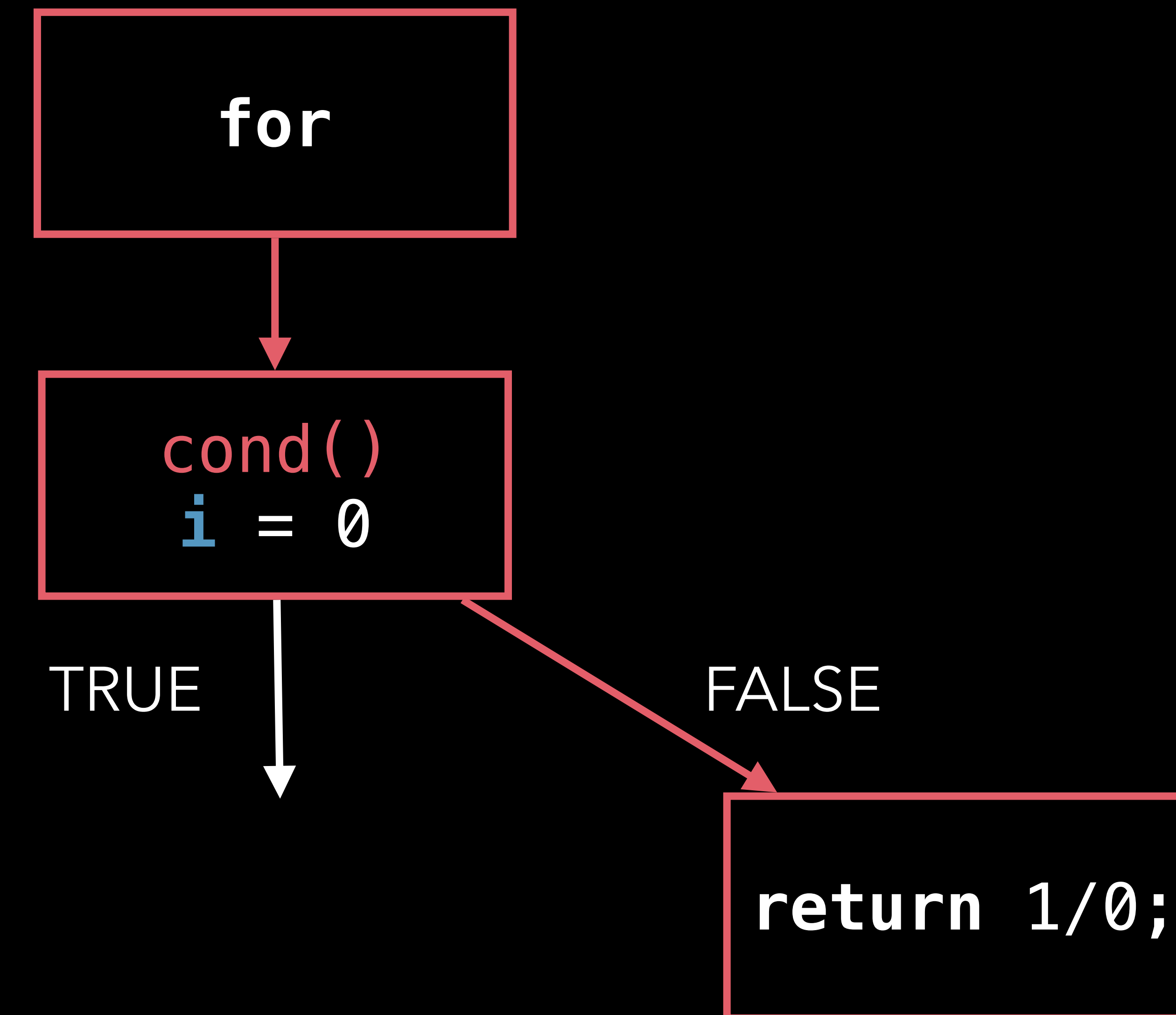
Coverage-Based Iteration Order

```
int main() {
  for (int i = 0; i < 2; ++i) {
    if (cond())
      continue;
    return 1/0; // 💥 crash
  }
}
```

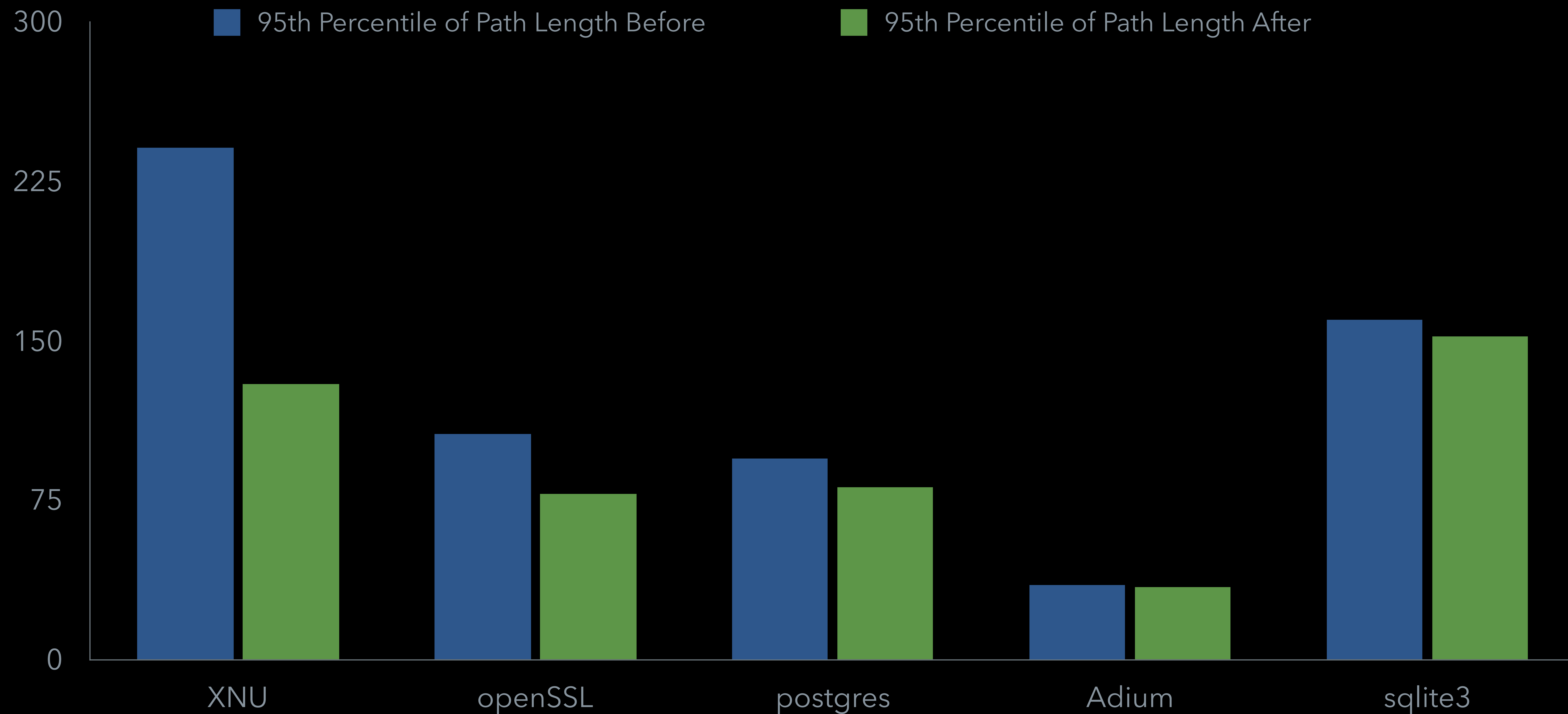


Coverage-Based Iteration Order

```
int main() {
  for (int i = 0; i < 2; ++i) {
    if (cond())
      continue;
    return 1/0; // 💥 crash
  }
}
```

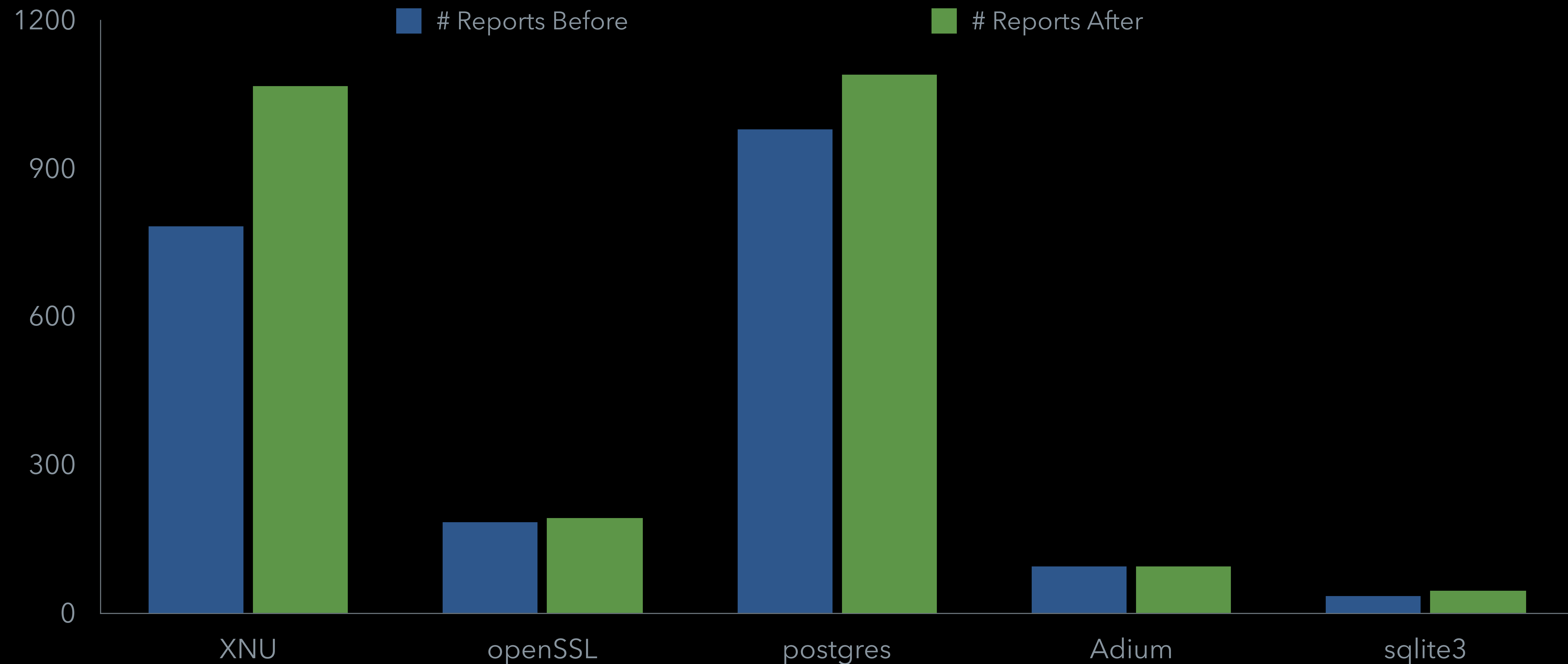


Results: 95th Percentile of Path Length



Results: Total Bug Reports

16% Increase in Number of Reports Found



Agenda

Introduction to Clang Static Analyzer

Using coverage-based iteration order

Improved C++ constructor and destructor support

Incomplete C++ Support Caused False Positives

- Analyzer lost information on object construction
- Analyzer lost track of objects before they were destroyed
- Temporaries are hard!

Constructor Call = Initialization Bookkeeping + Method Call

Initialization Bookkeeping In C Is Easy

```
typedef struct {...} Point;
Point makePoint();

Point P = makePoint();
```

```
DeclStmt
`-VarDecl 'P' 'Point'
  `-CallExpr 'makePoint' 'Point'
```

1. `CallExpr`
Call '`makePoint()`' to evaluate contents of the structure

2. `DeclStmt`
Put these contents into '`P`'

Initialization Bookkeeping In C++ Is More Complicated

```
struct Point {
  ...
  Point();
};

Point P;
```

```
DeclStmt
`-VarDecl 'P' 'Point'
  `-CXXConstructExpr 'Point()'
```

1. `CXXConstructExpr`
   Call constructor like a method on the object `P`

↓

2. `DeclStmt`
   Learn about the existence of variable `P`

Initialization Bookkeeping In C++ Is More Complicated

```
struct Point {
  ...
  Point();
};

Point P;
```

```
DeclStmt
`-VarDecl 'P' 'Point'
  `-CXXConstructExpr 'Point()'
```

2. **DeclStmt**
Learn about the existence of variable **P**

1. **CXXConstructExpr**
Call constructor like a method on the object **P**

Initialization Bookkeeping In C++ Is More Complicated

```
struct Point {
  ...
  Point();
};

Point P;
```

```
DeclStmt
`-VarDecl 'P' 'Point'
  `-CXXConstructExpr 'Point()'
```

1. **DeclStmt**
   Learn about the existence of variable **P**

2. **CXXConstructExpr**
   Call constructor like a method on the object **P**

Initialization Bookkeeping In C++ Is More Complicated

- The constructor needs to know what object is being constructed
- `CXXConstructExpr` doesn't tell us everything in advance

Initialization Bookkeeping In C++ Takes Many Forms

Variables:

```
Point P(1, 2, 3);
Point P = Point(1, 2, 3);
Point P = Point(1); // cast from 1
Point P = 1; // implicit cast from 1
```

Constructor initializers:

```
struct Vector {
  Point P;
  Vector() : P(1, 2, 3) {}
};
struct Vector {
  Point P = Point(1, 2, 3);
};
```

Aggregates and brace initializers:

```
Point P{1, 2, 3};
PointPair PP{Point(1, 2),
             Point(3, 4)};
PointPairPair PPP{{{1, 2}, {3, 4}},
                  {{5, 6}, {7, 8}}};
std::vector<Point> V{{1, 2, 3}};
```

Heap allocation:

```
Point *P = new Point(1, 2, 3);
Point *P = new Point[N + 1];
```

Temporaries:

```
Point(1, 2, 3);
const Point &P = Point(1, 2, 3);
const int &x = Point(1, 2, 3).x;
// determine in run-time
const Point &P =
  lunarPhase() ? Point(1, 2, 3)
               : Point(3, 2, 1);
```

Return values:

```
Point getPoint() {
  return Point(1, 2, 3); // RVO
}
Point getPoint() {
  Point P(1, 2, 3); // NRVO
  return P;
}
```

Argument values:

```
draw(Point(1, 2, 3));
Point(1, 2, 3) - Point(4, 5, 6);
void draw(Point P = Point(1, 2, 3));
draw(); // construct P
```

Captured values:

```
// copy to capture
Point P; [P]{ return P; }();
```

IT IS ONLY GETTING ~~WORSE~~ *better*

There is a common theme

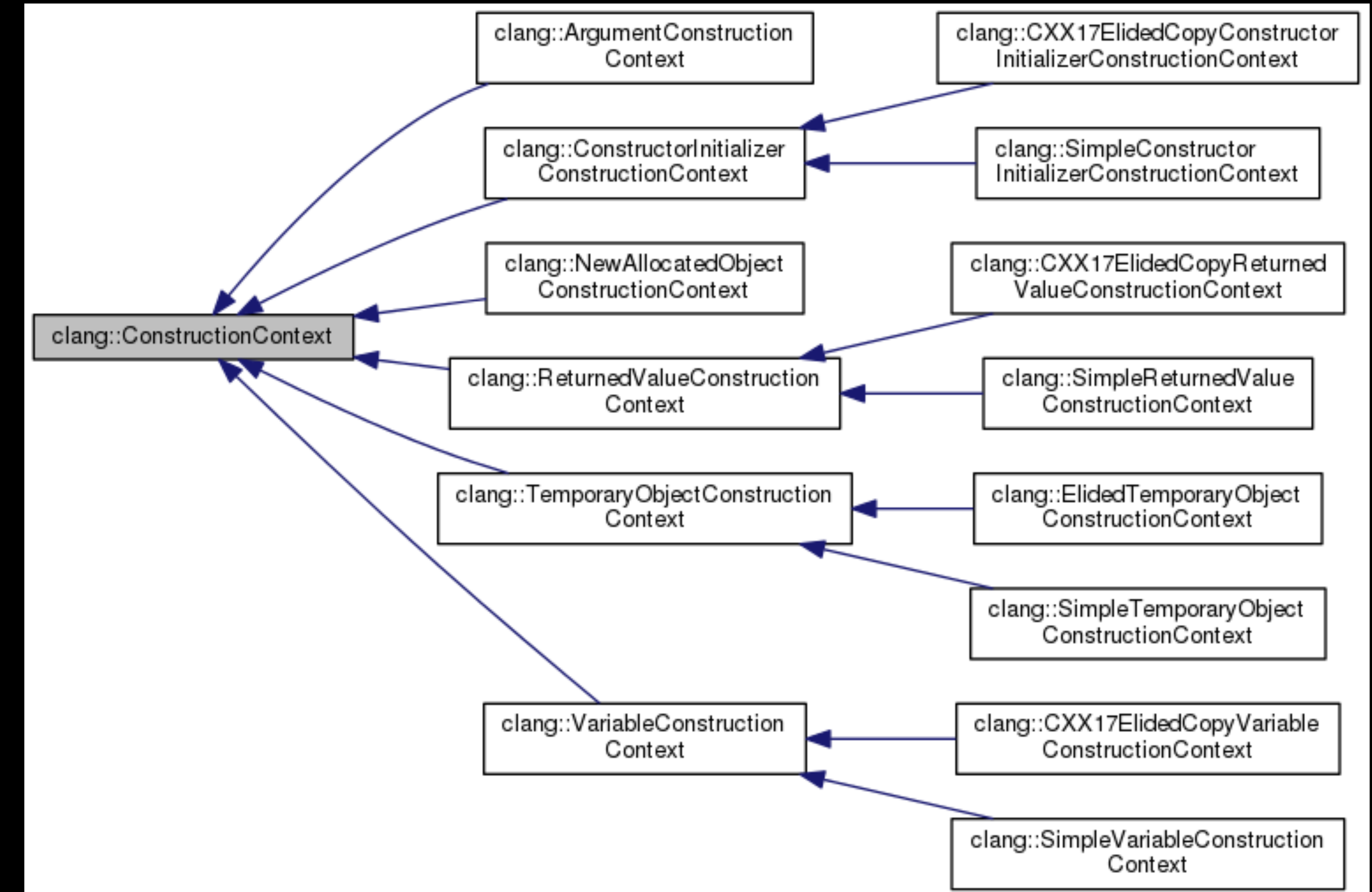
Need to track the constructed object’s address
until the analyzer processes the statement
that represents the object’s storage

Solution: Construction Context

- Augments CFG constructor call elements
- Describes the construction site:
    - What object is constructed?
    - Who is responsible for destroying it?
    - Is it a temporary that requires materialization?
    - Is the constructor elidable?

Solution: Construction Context

- A construction syntax catalog
  - There are currently 15 classes
- Easy to identify and to support



Progress made…

Variables:

```
Point P(1, 2, 3);                        BEFORE
Point P = Point(1, 2, 3);
Point P = Point(1); // cast from 1        NOW
Point P = 1; // implicit cast from 1
```

Constructor initializers:

```
struct Vector {
  Point P;                               BEFORE
  Vector() : P(1, 2, 3) {}
};
struct Vector {
  Point P = Point(1, 2, 3);
};
```

Aggregates and brace initializers:

```
Point P{1, 2, 3};                        BEFORE
PointPair PP{Point(1, 2),
             Point(3, 4)};
PointPairPair PPP{{{1, 2}, {3, 4}},
                  {{5, 6}, {7, 8}}};
std::vector<Point> V{{1, 2, 3}};
```

Heap allocation:

```
Point *P = new Point(1, 2, 3);    NOW
Point *P = new Point[N + 1];
```

Temporaries:

```
Point(1, 2, 3);
const Point &P = Point(1, 2, 3);
const int &x = Point(1, 2, 3).x;
// determine in run-time                 NOW
const Point &P =
  lunarPhase() ? Point(1, 2, 3)
               : Point(3, 2, 1);
```

Return values:

```
Point getPoint() {
  return Point(1, 2, 3); // RVO          NOW
}
Point getPoint() {
  Point P(1, 2, 3); // NRVO
  return P;
}
```

Argument values:

```
draw(Point(1, 2, 3));                    NOW
Point(1, 2, 3) - Point(4, 5, 6);
void draw(Point P = Point(1, 2, 3));
draw(); // construct P
```

Captured values:

```
// copy to capture
Point P; [P]{ return P; }();
```

Progress made… but help wanted!

Variables:

BEFORE
```
Point P(1, 2, 3);
```
NOW
```
Point P = Point(1, 2, 3);
Point P = Point(1); // cast from 1
Point P = 1; // implicit cast from 1
```

Constructor initializers:

BEFORE
```
struct Vector {
  Point P;
  Vector() : P(1, 2, 3) {}
};
```
WANTED
```
struct Vector {
  Point P = Point(1, 2, 3);
};
```

Aggregates and brace initializers:

BEFORE
```
Point P{1, 2, 3};
```
WANTED
```
PointPair PP{Point(1, 2),
             Point(3, 4)};
PointPairPair PPP{{{1, 2}, {3, 4}},
                  {{5, 6}, {7, 8}}};
std::vector<Point> V{{1, 2, 3}};
```

Heap allocation:

NOW
```
Point *P = new Point(1, 2, 3);
```
WANTED
```
Point *P = new Point[N + 1];
```

Temporaries:

NOW
```
Point(1, 2, 3);
const Point &P = Point(1, 2, 3);
const int &x = Point(1, 2, 3).x;
// determine in run-time
const Point &P =
  lunarPhase() ? Point(1, 2, 3)
               : Point(3, 2, 1);
```

Return values:

NOW
```
Point getPoint() {
  return Point(1, 2, 3); // RVO
}
```
WANTED
```
Point getPoint() {
  Point P(1, 2, 3); // NRVO
  return P;
}
```

Argument values:

NOW
```
draw(Point(1, 2, 3));
Point(1, 2, 3) - Point(4, 5, 6);
```
WANTED
```
void draw(Point P = Point(1, 2, 3));
draw(); // construct P
```

Captured values:

WANTED
```
// copy to capture
Point P; [P]{ return P; }();
```

Achievements: False Positive Reduction on WebKit

Summary

- Improved exploration order
  - 16% more useful analyzer warnings generated
  - Resulting analyzer path are up to 3x shorter
- Improved understanding of C++ object construction and destruction
  - Fix most of the C++-specific false positives
- Available in LLVM-7.0.0
  - clang-analyzer.llvm.org

Questions?

Summary

- Improved exploration order
  - 16% more useful analyzer warnings generated
  - Resulting analyzer path are up to 3x shorter
- Improved understanding of C++ object construction and destruction
  - Fix most of the C++-specific false positives
- Available in LLVM-7.0.0
  - clang-analyzer.llvm.org