The Penultimate Challenge:
Bug report construction in the Clang Static Analyzer

Kristóf Umann
dkszelethus@gmail.com

Eötvös Loránd University, Budapest
Ericsson Hungary

2019. oct. 22.
Task failed successfully.
Clear, precise bug reports are important

- One of the main selling points of Clang back in the day
- Not only wording, it requires a good infrastructure
- Tools without it are miserable to use
Agenda

- Path-sensitive analysis in the Clang Static Analyzer
- Current state of bug report construction
- Difficulties, current state of research, future work
Path-sensitive analysis in the Clang Static Analyzer
The Clang Static Analyzer

It employs a variety of techniques to analyze C, C++, ObjectiveC, ObjectiveC++ code:

- AST matching
- CFG based analyses
- Symbolic execution
Exploring paths of execution

- Traverse the control flow graph (CFG) of a function
- On branches, explore a path on which the condition is true, and one on which its false
- How does this work interprocedurally?
Exploring paths of execution

- Traverse the control flow graph (CFG) of a function
- On branches, explore a path on which the condition is true, and one on which its false
- How does this work interprocedurally? Inlining!
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5;

main:
01 int flag;
02 bool coin();
03
04 void foo() {
05    flag = coin();
06 }
07
08 int main() {
09    int *x = 0;
10    flag = 1;
11    foo();
12    if (flag)
13        x = new int;
14    foo();
15    if (flag)
16        *x = 5;
17 }
18

foo:
[B2 (ENTRY)]
[B1] flag = coin();
[B0 (EXIT)]

[B5 (ENTRY)]
[B4]
int *x = 0;
flag = 1;
foo();
if (flag)
[B3]
    x = new int;
[B2] foo();
    if (flag)
        x = new int;
    foo();
    if (flag)
        *x = 5;
[B1] *x = 5
[B0 (EXIT)]
```c
int flag;
bool coin();

void foo() {
    flag = coin();
}

int main() {
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        x = new int;
    foo();
    if (flag)
        *x = 5;
}
```
01 int flag;
02 bool coin();
03
04 void foo() {
05   flag = coin();
06 }
07
08 int main() {
09   int *x = 0;
10   flag = 1;
11   foo();
12   if (flag)
13     x = new int;
14   foo();
15   if (flag)
16     *x = 5;
17 }
18
main:

foo:

01 int flag;
02 bool coin();
03
04 void foo() {
05   flag = coin();
06   foo();
07   if (flag)
08     *x = 5
09 }
10
11 int main() {
12   int *x = 0;
13   flag = 1;
14   foo();
15   if (flag)
16     x = new int;
17   foo();
18   if (flag)
19     *x = 5;
20 }
21
22 int flag;
23 bool coin();
24
25 void foo() {
26   flag = coin();
27   foo();
28   if (flag)
29     *x = 5
30 }
31
32 int main() {
33   int *x = 0;
34   flag = 1;
35   foo();
36   if (flag)
37     x = new int;
38   foo();
39   if (flag)
40     *x = 5;
41 }
```c
int flag;
bool coin();
void foo() {
    flag = coin();
}

int main() {
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        x = new int;
    foo();
    if (flag)
        *x = 5;
}
```
main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;
foo();
if (flag)

[B2]  
foo();
x = new int;
flag = coin();

[B1]

[B0 (EXIT)]
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5;

01  int flag;
02  bool coin();
03
04  void foo() {
05    flag = coin();
06  }
07
08  int main() {
09    int *x = 0;
10    flag = 1;
11    foo();
12    if (flag)
13        x = new int;
14    foo();
15    if (flag)
16        *x = 5;
17    }
18
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5;

int flag;
bool coin();

void foo() {
    flag = coin();
}

int main() {
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        x = new int;
    foo();
    if (flag)
        *x = 5;
}
main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

foo:
[B2 (ENTRY)]

[B1]
flag = coin();

[B0 (EXIT)]

main:

01 int flag;
02 bool coin();
03 void foo() {
04   flag = coin();
05 }
06 }
07
08 int main() {
09   int *x = 0;
10   flag = 1;
11   foo();
12   if (flag)
13     x = new int;
14   foo();
15   if (flag)
16     *x = 5;
17 }
18
int *x = 0;
flag = 1;
foo();
if (flag)
  x = new int;
foo();
if (flag)
  *x = 5;
}

int flag;
bool coin();

void foo() {
  flag = coin();
}

int main() {
  int *x = 0;
  flag = 1;
  foo();
  if (flag)
  x = new int;
  foo();
  if (flag)
  *x = 5;
}
main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;
foo();
if (flag)

[B2] foo();
if (flag)

[B1] flag = coin();

[B0 (EXIT)]
```c
int *x = 0;
flag = 1;
foo();
if (flag)
  x = new int;
foo();
if (flag)
  *x = 5;
```

```c
int main() {
  int *x = 0;
  flag = 1;
  foo();
  if (flag)
    x = new int;
  foo();
  if (flag)
    *x = 5;
}
```
```c
01 int flag;
02 bool coin();
03
04 void foo() {
05    flag = coin();
06 }
07
08 int main() {
09    int *x = 0;
10    flag = 1;
11    foo();
12    if (flag)
13       x = new int;
14    foo();
15    if (flag)
16       *x = 5;
17 }
18 }
```
int *x = 0;
flag = 1;
foo();
if (flag)
{
    x = new int;
    foo();
    if (flag)
    {
        *x = 5;
    }
}

flag = coin();
foo();
if (flag)
flag == 1;
x == nullptr;

foo:

main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1]
flag = coin();

[B0 (EXIT)]

* x = 5

[B0 (EXIT)]
flag == 1;
x == nullptr;

main:
[B5 (ENTRY)]
[B4]
int *x = 0;
flag = 1;
foo();
if (flag)
[B3]
x = new int;

foo:
[B2 (ENTRY)]
[B1]
flag = coin();

[B2 (EXIT)]
[B0 (EXIT)]

[B0 (EXIT)]

[B1]
* x = 5
flag == 1;
x == nullptr;

foo:

main:

[B5 (ENTRY)]

[B4] int *x = 0;
flag = 1;
foo();
if (flag)

[B3] x = new int;

[B2] foo();
if (flag)

[B1] *x = 5

[B0 (EXIT)]
flag ∈ (−∞, ∞);
x == nullptr;

foo:

main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1]
flag = coin();

[B0 (EXIT)]

[B0 (EXIT)]

* x = 5
`int *x = 0;`  
`flag = 1;`  
`foo();`  
`if (flag) flag ∈ (−∞, ∞);  
x == nullptr;`  

foo:

`[B2 (ENTRY)]`  
`flag = coin();`  
`[B1]`  
`[B0 (EXIT)]`

main:

`[B5 (ENTRY)]`  
`[B4] int *x = 0;`  
`flag = 1;`  
`foo();`  
`if (flag)`  
`[B3] x = new int;`  
`foo();`  
`if (flag)`  
`[B2]`  
`[B0 (EXIT)]`  
`*x = 5`  
`[B1]`  
`[B0 (EXIT)]`
```c
int *x = 0;
flag = 1;
foo();
if (flag)
    flag ∈ (−∞, ∞);
x == nullptr;
    x = new int;
foo();
if (flag)
    *x = 5;

main:
    flag = coin();

foo:
    if (flag)
        *x = 5;
```

![Flowchart diagram of the code flow]
flag ∈ (−∞, ∞);
x == nullptr;

foo:

[B2 (ENTRY)]

[B1]
flag = coin();

[B0 (EXIT)]
flag == 0;
x == nullptr;

foo:

main:

B5 (ENTRY)

B4
int *x = 0;
flag = 1;
foo();
if (flag)

B3
x = new int;

B2
foo();
if (flag)

B2 (ENTRY)

B1
flag = coin();

B1
* x = 5

B0 (EXIT)
flag == 0;
x == nullptr;

```c
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    flag == 0;
x == nullptr;
```
flag == 0;
x == nullptr;

foo:

[B2 (ENTRY)]
flag = coin();
[B1]  
[B0 (EXIT)]

main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1]
*x = 5

[B0 (EXIT)]
flag ∈ (−∞, ∞);
x == nullptr;

main:

[B5 (ENTRY)]
[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1]
flag = coin();

[B0 (EXIT)]

[B2 (ENTRY)]

[B0 (EXIT)]

[B1]
*x = 5
flag ∈ (−∞, ∞);
x == nullptr;

foo:
[B2 (ENTRY)]
[B1]
flag = coin();
[B0 (EXIT)]

main:
[B5 (ENTRY)]
[B4]
int *x = 0;
flag = 1;
foo();
if (flag)
[B3]
x = new int;
[B2]
foo();
if (flag)
[B1]
*x = 5
[B0 (EXIT)]
```c
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    flag ∈ (−∞, ∞);
x == nullptr;

main:
flag = coin();
foo:
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5
```
flag ∈ (−∞, ∞);
x == nullptr;

foo:

[B2 (ENTRY)]

[B1]
flag = coin();

[B0 (EXIT)]

main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1]
*x = 5

[B0 (EXIT)]
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5

flag ∈ (−∞, 0) ∪ (0, ∞);
x == nullptr;
dereference of x!

foo:

main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)
    *x = 5

[B3]
x = new int;

[B2]
foo();
if (flag)
    *x = 5

[B1]
flag = coin();
The ExplodedGraph

- Contains everything the analyzer learned during symbolic execution
- All explored paths of execution
- Every symbolic value in every program state
main:

{B5 (ENTRY)}

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B2]
int *x = 0;
flag = 1;
foo();
if (flag)

[B1]
flag = coin();

[B0 (EXIT)]

foo:

{B2 (ENTRY)}

[B1]
flag = coin();

[B0 (EXIT)]

[B0 (EXIT)]

[B1]
x = new int;

[B3]

[B2]
x = new int;

[B0 (EXIT)]

[B1]
*x = 5

[dereference of x!]

[B0 (EXIT)]

[B1]
flag = 0
x = nullptr

[B0 (EXIT)]

[dereference of x!]

[B0 (EXIT)]

flag = 0
x = nullptr

[dereference of x!]

flag = 0
x = nullptr
int *x = 0; flag = 1; foo(); if (flag)

x = new int;

foo(); if (flag)

*x = 5

main:

flag = coin();

foo:

int **x = 0; flag = 1; foo(); if (flag)

x = nullptr
flag ∈ (−∞, ∞)

x = nullptr
flag ∈ (−∞, 0) ∪ (0, ∞)

x = (heap allocated object)
dereference of x!

x = (heap allocated object)
*x = 5

flag = 0

x = (heap allocated object)
dereference of x!

flag = 0

x = nullptr
```c
main:
[B5 (ENTRY)]
[B4] int *x = 0;
flag = 1;
foo();
if (flag)
[B3] x = new int;
[B2] foo();
if (flag)
[B1] *x = 5
[B0 (EXIT)]
flag = coin();
[B2 (ENTRY)]
[B1] flag = 1
x = nullptr
[B0 (EXIT)]

flag = 1
x = nullptr

(x after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr
flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
dereference of x!

flag = 0
x = (heap allocated object)
flag = 0
x = nullptr
(x after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr
flag ∈ (−∞, 0) ∪ (0, ∞)
```
`int *x = 0; flag = 1; foo(); if (flag) x = new int; foo(); if (flag) *x = 5`
main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1]
x = 5

[B0 (EXIT)]

foo:

[B2 (ENTRY)]

[B1]
flag = coin();

[B0 (EXIT)]

(Bafter the call to foo

flag ∈ (−∞,∞)
x = nullptr

flag = 1
x = nullptr

(Bafter the call to foo

flag ∈ (−∞,∞)
x = nullptr

flag ∈ (−∞,0) ∪ (0,∞)
x = nullptr

dereference of x!

flag = 0
x = nullptr

(Bafter the call to foo

flag ∈ (−∞,∞)
x = nullptr

flag ∈ (−∞,0) ∪ (0,∞)
x = nullptr

dereference of x!
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5

main:
flag = coin();

foo:
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
(x after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

(x after the call to foo)
flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!

(x after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr
dereference of x!

flag = 0
x = nullptr
main:
[B5 (ENTRY)]

foo:
[B2 (ENTRY)]
intro *x = 0;
flag = 1;
foo();
if (flag)

[B4]
main:
flag = coin();
[B2 (ENTRY)]

[B0 (EXIT)]

[B1]
flag = coin();
[B0 (EXIT)]

[B3] x = new int;
[B2 (ENTRY)]
[B0 (EXIT)]

[B1]
*x = 5
[B0 (EXIT)]

[B3] x = new int;

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
dereference of x!

flag = 0
x = nullptr
dereference of x!

flag = 0
x = nullptr

10/38
int *x = 0; flag = 1; foo(); if (flag)

int *x = 0; flag = 1; foo(); if (flag)

x = new int;

foo(); if (flag)

*x = 5

main:

flag = coin();

foo:

[flag = 1
x = nullptr
(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr]

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!

flag = 0
x = nullptr

flag = 0
x = nullptr

x = (heap allocated object)
dereference of x!
int *x = 0; flag = 1; foo(); if (flag)

x = new int; x = new int;

foo(); if (flag)

*x = 5

main:
[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1]
*x = 5

[B0 (EXIT)]

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!

flag = 0
x = nullptr

(flag = 0)

x = nullptr

flag = 0
x = nullptr
dereference of x!

flag = 0
x = nullptr
```c
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
    x = new int;
    foo();
    if (flag)
        *x = 5;
flag = coin();
```
int *x = 0;
flag = 1; foo();
if (flag)
int *x = new int;
foo();
if (flag)
*x = 5

main:
[B5 (ENTRY)]

foo:
[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;
foo();
if (flag)

[B2]
*x = 5

[B0 (EXIT)]

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (-∞, ∞)
x = nullptr

flag ∈ (-∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!

flag ∈ (-∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*x = undefined

flag = 0
x = nullptr
int *x = 0; flag = 1; foo(); if (flag)
    x = new int;
foo(); if (flag)
    *x = 5

main:
    flag = coin();

[88 (ENTRY)]

[86 (EXIT)]

foo:
    flag = 1
    x = nullptr
    flag = 1
    x = nullptr
    (after the call to foo)
    flag ∈ (−∞, ∞)
    x = nullptr

    flag = 0
    x = (heap allocated object)
    *x = undefined

    flag = 0
    x = nullptr
    (after the call to foo)
    flag ∈ (−∞, 0) ∪ (0, ∞)
    x = nullptr
    dereference of x!

    flag = 0
    x = nullptr
int *x = 0; flag = 1; foo();
if (flag)
x = new int;
foo(); if (flag)
*x = 5

main:
[B0 (EXIT)]

foo:
[B1] flag = coin();
[B0 (EXIT)]

[B0 (EXIT)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1] *x = 5
[B0 (EXIT)]

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞,∞)
x = nullptr

flag ∈ (−∞,0) ∪ (0,∞)
x = nullptr
dereference of x!
flag = 0
x = nullptr

flag ∈ (−∞,0) ∪ (0,∞)
x = (heap allocated object)
*x = undefined

flag = 0
x = nullptr

10/38
int *x = 0; flag = 1; foo();
if (flag)
  x = new int;
    foo();
    if (flag)
      *x = 5
main:
  flag = coin();
  flag = coin();
  foo:
    flag = 1
    x = nullptr
  (after the call to foo)
  flag ∈ (−∞, ∞)
  x = nullptr
  flag = coin();
  flag = coin();
  if (flag)
    x = new int;
    foo();
    if (flag)
      *x = 5
  (after the call to foo)
  flag ∈ (−∞, 0) ∪ (0, ∞)
  x = nullptr
  dereference of x!
  flag = 0
  x = nullptr
  (after the call to foo)
  flag ∈ (−∞, 0) ∪ (0, ∞)
  x = nullptr
  dereference of x!
  flag = 0
  x = nullptr
main:
[B5 (ENTRY)]

foo:
[B4]
int *x = 0;
flag = 1;
foo();
if (flag)
[B2]
x = new int;
foo();
if (flag)
[B1]*x = 5

flag = coin();

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

x = (heap allocated object)
*x = undefined

(dereference of x!)
flag = 0
x = nullptr

(flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

(flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*x = undefined

(after the call to foo)
flag ∈ (−∞, ∞)
x = (heap allocated object)
*x = undefined

(dereference of x!)
flag = 0
x = nullptr

x = (heap allocated object)
```c
int *x = 0; flag = 1; foo();
if (flag)
    x = new int;
foo(); if (flag)
    *x = 5

main:
    flag = coin();
    foo();
    if (flag)
        x = new int;
    foo(); if (flag)
        *x = 5

foo:
    flag = coin();
    foo(); if (flag)
        *x = 5
```

Diagram:

```
main:
    flag = coin();
    foo(); if (flag)
        x = new int;
    foo(); if (flag)
        *x = 5
```

```
foo:
    flag = coin();
    foo(); if (flag)
        *x = 5
```

```
flag = 1
    x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
    x = nullptr
```

```
flag = (−∞, 0) ∪ (0, ∞)
    x = nullptr

(dereference of x)
```

```
flag ∈ (−∞, 0) ∪ (0, ∞)
    x = (heap allocated object)
    *x = undefined

(after the call to foo)
flag ∈ (−∞, ∞)
    x = (heap allocated object)
    *x = undefined
```

```
flag = 0
    x = nullptr

(dereference of x)
```

```
flag ∈ (−∞, 0) ∪ (0, ∞)
    x = nullptr
```

```
flag ∈ (−∞, ∞)
    x = nullptr
```

```
flag ∈ (−∞, ∞)
    x = (heap allocated object)
    *x = undefined
```

```
flag = 0
    x = nullptr

(dereference of x)
```

```
flag ∈ (−∞, ∞)
    x = nullptr
```

```
flag ∈ (−∞, ∞)
    x = (heap allocated object)
    *x = undefined
```

10/38
int *x = 0; flag = 1; foo();
if (flag)
   x = new int;
foo();
if (flag)
   *x = 5

main:
[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)
   [B3] x = new int;

[B2] foo();
if (flag)
   [B1] *x = 5

[B0 (EXIT)]

flag = coin();
[B2 (ENTRY)]

[B1]
flag = coin();

[B0 (EXIT)]

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
*x = undefined

(dereference of x!)
flag ∈ (−∞, ∞)
x = nullptr

flag = 0
x = nullptr

(dereference of x!)
flag ∈ (−∞, ∞)
x = nullptr

flag = 0
x = nullptr

10/38
int *x = 0; flag = 1; foo(); if (flag)  
x = new int; foo(); if (flag)  
*x = 5

flag = coin();

main:

foo:

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
*x = undefined

x = (heap allocated object)
derreference of x!
*x = 5

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

flag ∈ (−∞, ∞)
x = nullptr
main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1]
x = nullptr

[B0 (EXIT)]

foo:

[B2 (ENTRY)]

[B1]
flag = coin();

[B0 (EXIT)]

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞,∞)
x = nullptr

flag ∈ (−∞,0) ∪ (0,∞)
x = nullptr
dereference of x!
*x = 5

flag ∈ (−∞,0) ∪ (0,∞)
x = (heap allocated object)
*x = undefined

flag ∈ (−∞,0) ∪ (0,∞)
x = (heap allocated object)
*x = undefined

flag ∈ (−∞,0) ∪ (0,∞)
x = (heap allocated object)
*dereference of x!
*x = 5

flag ∈ (−∞,0) ∪ (0,∞)
x = nullptr

flag ∈ (−∞,0) ∪ (0,∞)
x = nullptr

flag ∈ (−∞,0) ∪ (0,∞)
x = nullptr
int *x = 0;
flag = 1; foo();
if (flag)
    x = new int;
foo();
if (flag)
    foo();
if (flag)
    *x = 5

flag = coin();

foo:

B2 (ENTRY)
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
B1
x = new int;
B2 (EXIT)

B1
x = 5
B0 (EXIT)

B5 (ENTRY)
main:

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!
*x = undefined

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
dereference of x!
*x = undefined

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*x = undefined

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*deref = 5

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*deref = 5

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*x = undefined
```c
int *x = 0; flag = 1; foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5

main:

flag = coin();

foo:

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
  x = nullptr

x = (heap allocated object)
dereference of x!
  *x = 5

flag = 0
x = (heap allocated object)
  *x = undefined
```
main:
[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1]
*x = 5

[B0 (EXIT)]

flag = coin();

foo:
[B2 (ENTRY)]

[B1]
flag = coin();

[B0 (EXIT)]

flag = 1
x = nullptr

(flag after the call to foo)
flag ∈ (-∞, ∞)
x = nullptr

flag ∈ (-∞, 0) ∪ (0, ∞)
x = heap allocated object
*x = undefined

(flag after the call to foo)
flag ∈ (-∞, 0)
x = heap allocated object
*x = undefined

(flag after the call to foo)
flag ∈ (-∞, 0)
x = heap allocated object
*x = undefined

flag = 0
x = (heap allocated object)
*dereference of x!
*x = 0

flag ∈ (-∞, 0) ∪ (0, ∞)
x = heap allocated object
*x = undefined

flag ∈ (-∞, 0) ∪ (0, ∞)
x = heap allocated object
*x = undefined
int *x = 0;flag = 1;foo();if (flag)
tax = new int;
foo();if (flag)
foo();if (flag)
*x = 5

flag = coin();

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*x = undefined

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
dereference of x!
*x = 0

flag ∈ (−∞, ∞)
x = (heap allocated object)
*x = undefined

flag = 0
x = nullptr

flag = 0
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*x = undefined

flag ∈ (−∞, ∞)
x = (heap allocated object)
*x = undefined

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
dereference of x!
*x = 0
```c
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5

main:
flag = coin();
foo();
if (flag)
    x = new int;
    *x = 5
    flag = 1;
x = nullptr
flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

x = (heap allocated object)
dereference of x!
*x = 0

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*x = undefined

(flag = 0)

x = (heap allocated object)
*x = undefined
```

```c
flag = 0
x = nullptr
```

```c
flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*x = undefined
```

```c
flag = 0
x = nullptr
```

```c
flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*x = undefined
```
```c
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5
```

```
main:
[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1]
x = 5

[B0 (EXIT)]
```
```c
int *x = 0;
flag = 1;
foo();
if (flag)
x = new int;
foo();
if (flag)
*x = 5

main:

flag = coin();
flag = coin();

foo:

int *x = 0;
flag = 1;
foo();
if (flag)
x = new int;
flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
*x = undefined

x = (heap allocated object)
dereference of x!
*x = 5

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*x = 5

flag ∈ (−∞, 0) ∪ (0, ∞)
*x = undefined

flag = 0
x = nullptr

flag = 0
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr
```

main:
  [B5 (ENTRY)]
  [B4]
  int *x = 0;
  flag = 1;
  foo();
  if (flag)
    x = new int;
    foo();
    if (flag)
      *x = 5

foo:
  [B2 (ENTRY)]
  flag = coin();
  [B1]
  [B0 (EXIT)]
  [B2]
  foo();
  if (flag)
    [B3]
    x = new int;
    [B1]
    *x = 5

  [B0 (EXIT)]

flag ∈ (−∞,0) ∪ (0,∞)
  x = (heap allocated object)
  *x = undefined

flag = 1
  x = nullptr

(flag after the call to foo)
  flag ∈ (−∞,∞)
  x = nullptr

flag ∈ (−∞,0) ∪ (0,∞)
  x = (heap allocated object)
  *x = undefined

flag = 0
  x = nullptr

(flag after the call to foo)
  flag ∈ (−∞,∞)
  x = nullptr

(flag after the call to foo)
  flag ∈ (−∞,∞)
  x = nullptr

flag = 0
  x = nullptr

(flag after the call to foo)
  flag ∈ (−∞,∞)
  x = nullptr

(flag after the call to foo)
  flag ∈ (−∞,∞)
  x = nullptr

(dereference of x!)
  *x = 5

flag = 0
  x = (heap allocated object)
  *x = undefined

flag ∈ (−∞,0) ∪ (0,∞)
  x = (heap allocated object)
int *x = 0; flag = 1; foo(); if (flag)
    x = new int;

foo:
    flag = coin();
    if (flag)
        *x = 5

main:
    flag = coin();
    foo();
    if (flag)
        *x = 5
    flag = 1
    x = nullptr

(after the call to foo)
    flag ∈ (−∞, ∞)
    x = nullptr

flag = 0
    x = nullptr

(after the call to foo)
    flag ∈ (−∞, ∞)
    x = nullptr

flag = 0
    x = nullptr

(flag ∈ (−∞, 0) ∪ (0, ∞)
    x = (heap allocated object)
    *x = undefined

(dereference of x!)

flag ∈ (−∞, 0) ∪ (0, ∞)
    x = (heap allocated object)
    *x = undefined

flag = 0
    x = (heap allocated object)
    *x = undefined
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    x = nullptr;
(after the call to foo)
    flag ∈ (−∞, ∞)
    x = nullptr
flag = 0
    x = nullptr
(after the call to foo)
    flag ∈ (−∞, ∞)
    x = nullptr
flag = 0
foo();
if (flag)
    x = (heap allocated object)
    *x = 5
(after the call to foo)
    flag ∈ (−∞, 0) ∪ (0, ∞)
    x = (heap allocated object)
    *x = undefined
(after the call to foo)
    flag ∈ (−∞, 0) ∪ (0, ∞)
    x = (heap allocated object)
    *x = undefined
flag ∈ (−∞, 0) ∪ (0, ∞)
    x = (heap allocated object)
    dereference of x!
    *x = 5
flag = 0
    x = (heap allocated object)
    *x = undefined
flag = 0
    x = (heap allocated object)
    *x = undefined
```c
main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;

[B2]
foo();
if (flag)

[B1]
x = 5

[B0 (EXIT)]

flag = 0
x = nullptr

flag = 1
x = nullptr

(flag after the call to foo)

flag ∈ (−∞, ∞)

x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*x = undefined

(flag after the call to foo)

flag ∈ (−∞, ∞)
x = nullptr
dereference of x!

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
dereference of x!

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
*dereference of x!

* = 5
```

```c
flag = 0
x = nullptr
dereference of x!
```
```c
int *x = 0; flag = 1; foo(); if (flag)
    x = new int;
foo(); if (flag)
    foo(); if (flag)
    *x = 5
main:
    flag = coin();
    [B2 (ENTRY)]
    [B1]
    flag = coin();
    [B0 (EXIT)]
    foo();
    [B2 (ENTRY)]
    [B3]
    x = new int;
    [B1]
    *x = 5
    [B0 (EXIT)]
    flag = 1
    x = nullptr
    (after the call to foo)
    flag ∈ (−∞, ∞)
    x = nullptr
    flag = 0
    x = nullptr
    (after the call to foo)
    flag ∈ (−∞, ∞)
    x = nullptr
    flag ∈ (−∞, 0) ∪ (0, ∞)
    x = (heap allocated object)
    *x = undefined
    dereference of x!
    flag ∈ (−∞, 0) ∪ (0, ∞)
    x = (heap allocated object)
    *x = undefined
    flag = 0
    x = (heap allocated object)
    *x = undefined
    (after the call to foo)
    flag ∈ (−∞, ∞)
    x = nullptr
    flag ∈ (−∞, 0) ∪ (0, ∞)
    x = (heap allocated object)
    *x = undefined
    dereference of x!
```
```c
int *x = 0;
flag = 1; foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5

main:
flag = coin();

foo:
int *x = 0;
flag = 1; foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5

(dereference of x)
```
There is always more to talk about...

- Branches in the ExplodedGraph may happen far more often
There is always more to talk about...

- Branches in the ExplodedGraph may happen far more often
- Representation of values, regions: symbols
There is always more to talk about...

- Branches in the ExplodedGraph may happen far more often
- Representation of values, regions: symbols
- ExplodedGraphs are usually very-very large, and contain tremendous amount of information
Bug report construction
Processing of the ExplodedGraph

- Bugs are represented with error nodes
- The graph may contain several of them
Processing of the ExplodedGraph

- Bugs are represented with error nodes
- The graph may contain several of them
- The goal is to explain the path to these nodes
Processing of the ExplodedGraph

- Bugs are represented with error nodes
- The graph may contain several of them
- The goal is to explain the path to these nodes
- For each node, construct the shortest path from the root to the error node
- This is called a bug path
flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
* x = undefined

(after the call to foo)
flag ∈ (−∞, ∞)
x = (heap allocated object)
* x = undefined

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
dereference of x!
* x = 5

flag = 0
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

flag = 0
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = (heap allocated object)
dereference of x!
* x = 5
01 int flag;
02 bool coin();
03
04 void foo() {
05    flag = coin();
06 }
07
08 int main() {
09    int *x = 0;
10    flag = 1;
11    foo();
12    if (flag)
13        x = new int;
14    foo();
15
16    if (flag)
17        *x = 5;
18 }
The ideal bug report

The goal is to generate a bug report from the bug path that is
- complete: contains every information necessary to understand how the bug occurred
- minimal: contains no unnecessary information
Techniques used by the analyzer

- 2 techniques:
Techniques used by the analyzer

- 2 techniques:
  - BugReporterVisitors
  - Interestingness propagation
Techniques used by the analyzer

- 2 techniques:
  - BugReporterVisitors
  - Interestingness propagation
- Visit the nodes of the bugpath from the error node to the root
flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag = 0
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!
flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!
flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

dereference of x!

(flag = 0
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

(flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

dereference of x!

<diagnostic msg 4>
flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag = 0
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!

<diagnostic msg 3>
(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag = 0
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
derereference of x!

<diagnostic msg 2>
flag = 1
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag = 0
x = nullptr

(after the call to foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!

<diagnostic msg 1>
BugReporterVisitors

- An arbitrary number of visitors can be registered
An arbitrary number of visitors can be registered
Their `visitNode()` is called on each node visit
BugReporterVisitors

- An arbitrary number of visitors can be registered
- Their `visitNode()` is called on each node visit
- Visitors may create diagnostic messages about the node they are currently visiting
BugReporterVisitors

- An arbitrary number of visitors can be registered
- Their `visitNode()` is called on each node visit
- Visitors may create diagnostic messages about the node they are currently visiting
- Despite the misleading name, they are more like callbacks than visitors
Visitors

- **ConditionBRVisitor**: Describes conditions of if branches, loops, conditional operators etc.
Visitors

- **ConditionBRVisitor**: Describes conditions of if branches, loops, conditional operators etc.
- **FindLastStoreBRVisitor**: Finds the last store to a given variable
Visitors

- ConditionBRVisitor: Describes conditions of if branches, loops, conditional operators etc.
- FindLastStoreBRVisitor: Finds the last store to a given variable
- TrackControlDependencyCondBRVisitor
TrackControlDependencyCondBRVisitor

- Most recent addition, available in Clang 10.0.0
- GSoC’19 project mentored by Artem Dergachev, Gábor Horváth and Ádám Balogh
- https://szelethus.github.io/gsoc2019/
TrackControlDependencyCondBRVisitor

- Most recent addition, available in Clang 10.0.0
- GSoC’19 project mentored by Artem Dergachev, Gábor Horváth and Ádám Balogh
- [https://szelethus.github.io/gsoc2019/](https://szelethus.github.io/gsoc2019/)
- Calculates control dependencies to points of interest
TrackControlDependencyCondBRVisitor

- Most recent addition, available in Clang 10.0.0
- GSoC’19 project mentored by Artem Dergachev, Gábor Horváth and Ádám Balogh
- [https://szelethus.github.io/gsoc2019/](https://szelethus.github.io/gsoc2019/)
- Calculates control dependencies to points of interest
- Tells the analyzer to explain the conditions of control dependency blocks
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
    foo();
    if (flag)
        *x = 5;

int flag;
bool coin();
void foo()
{
    flag = coin();
}
int main()
{
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        x = new int;
        foo();
        if (flag)
            *x = 5;
}
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5;

main:
int flag;
bool coin();
void foo() {
    flag = coin();
}

int main() {
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        x = new int;
    foo();
    if (flag)
        *x = 5;
}

foo:
flag = coin();
x = new int;
foo();
if (flag)
x = new int;
foo();
if (flag)
    *x = 5;
```c
int flag;
bool coin();

void foo() {
    flag = coin();
}

int main() {
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        x = new int;
    foo();
    if (flag)
        *x = 5;
}
```
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5;

main:

void foo() {
    flag = coin();
    foo();
    if (flag)
        *x = 5;
}

int main() {
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        x = new int;
    foo();
    if (flag)
        *x = 5;
}
main:

[B5 (ENTRY)]

[B4]
int *x = 0;
flag = 1;
foo();
if (flag)

[B3]
x = new int;
foo();
if (flag)

[B2]
foo();
if (flag)

[B1]
* x = 5

[B0 (EXIT)]

main:
01 int flag;
02 bool coin();
03
04 void foo() {
05 flag = coin();
06 }
07
08 int main() {
09 int *x = 0;
10 flag = 1;
11 foo();
12 if (flag)
13 x = new int;
14 foo();
15
16 if (flag)
17 * x = 5;
18 }
```c
int flag;
bool coin();

void foo() {
  flag = coin();
}

int main() {
  int *x = 0;
  flag = 1;
  foo();
  if (flag)
    x = new int;
  foo();
  if (flag)
    x = new int;
  foo();
  if (flag)
    *x = 5;
}
```
Interestingness propagation

- During analysis, some symbolic regions or values may have been marked as "interesting".
Interestingness propagation

- During analysis, some symbolic regions or values may have been marked as "interesting".
- During bug report construction, propagate interestingness to entities that interact with an interesting entity.
Interestingness propagation

- During analysis, some symbolic regions or values may have been marked as "interesting".
- During bug report construction, propagate interestingness to entities that interact with an interesting entity.
- Nodes in the bug path that do not describe an interesting entity are pruned.
Combining Visitors and Interestingness

Expression value tracking!
Combining Visitors and Interestingness

Expression value tracking!

- Mark the expression as interesting
Combining Visitors and Interestingness

Expression value tracking!

- Mark the expression as interesting
- Register visitors to describe events related to it
  - FindLastStoreBRVisitor
  - TrackControlDependencyCondBRVisitor
  - ReturnVisitor
  - UndefOrNotNullArgVisitor
  - etc...
Combining Visitors and Interestingness

Expression value tracking!

- Mark the expression as interesting
- Register visitors to describe events related to it
  - FindLastStoreBRVisitor
  - TrackControlDependencyCondBRVisitor
  - ReturnVisitor
  - UndefOrNullArgVisitor
  - etc...
- TrackControlDependencyCondBRVisitor does that as well
01 int flag;
02 bool coin();
03
04 void foo() {
05  flag = coin();
06 }
07
08 int main() {
09  int *x = 0;
10  flag = 1;
11  foo();
12  if (flag)
13     x = new int;
14  foo();
15  if (flag)
16     *x = 5;
17 }

flag = 1
x = nullptr

(after the call to foo)
flag \in \mathbb{R}

x = nullptr

flag = 0
x = nullptr

(after the call to foo)
flag \in \mathbb{R}

x = nullptr

dereference of x!
(calling foo)

(entering foo)
flag = 1
x = nullptr

(flag ∈ (−∞, ∞)
x = nullptr)

(returning from foo)
flag ∈ (−∞, ∞)
x = nullptr

dereference of x!
(calling foo)

(entering foo)
flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

flag ∈ (−∞, ∞)
x = nullptr

(returning from foo)
flag ∈ (−∞, ∞)
x = nullptr

dereference of x!
flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!
Stage 1: Visitor notes
The checker tracks \( x \)'s value

\[
\text{flag } \in (-\infty, 0) \cup (0, \infty) \\
x = \text{nullptr} \\
\text{dereference of } x!
\]

Tracked variables: \( \{ x \} \)
ConditionBRVisitor: Assuming 'flag' is not equal to 0

\[ \text{flag} \in (-\infty, 0) \cup (0, \infty) \]
\[ x = \text{nullptr} \]

Tracked variables: \{\text{x}\}

note: Assuming flag is true
TrackControlDependencyCond-BRVisitor tracks flag

flag ∈ (−∞,0)∪(0,∞)  
  x = nullptr

note: Assuming flag is true

TrackControlDependencyCond-BRVisitor tracks flag

flag ∈ (−∞,0)∪(0,∞)  
  x = nullptr

Tracked variables: \{x, flag\}
(returning from foo)
flag ∈ (−∞, ∞)
x = nullptr

Tracked variables: \{x, flag\}

note: Assuming flag is true
FindLastStoreBRVisitor: Value assigned to 'flag'

flag ∈ (−∞, ∞)

x = nullptr

Tracked variables: \{x, flag\}

note: flag is assigned a value

note: Assuming flag is true
(entering foo)
flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

Tracked variables: {x, flag}

note: flag is assigned a value
note: Assuming flag is true
ConditionBRVisitor: Assuming 'flag' is 0

flag = 0
x = nullptr

Tracked variables: \{x, flag\}

note: Assuming flag is false

note: flag is assigned a value

note: Assuming flag is true

dereference of x!

T racked variables: \{x, flag\}
flag = 1
x = nullptr
(calling foo)

(flag ∈ (−∞, ∞)
 x = nullptr)

(flag ∈ (−∞, ∞)
 x = nullptr)

(returning from foo)

(flag ∈ (−∞, ∞)
 x = nullptr)

(flag = 0
x = nullptr)

(calling foo)

(flag ∈ (−∞, 0) ∪ (0, ∞)
 x = nullptr)

(flag ∈ (−∞, ∞)
 x = nullptr)

(flag ∈ (−∞, 0) ∪ (0, ∞)
 x = nullptr)

(note: Assuming flag is false)

(note: flag is assigned a value)

(note: Assuming flag is true)

Tracked variables: \{x, flag\}
flag ∈ (−∞, ∞)
x = nullptr

Tracked variables: \{x, \text{flag}\}

**FindLastStoreBRVisitor is already satisfied, no notes**

**note:** Assuming \text{flag} is false

**note:** \text{flag} is assigned a value

**note:** Assuming \text{flag} is true

```plaintext
T racked variables: \{x, \text{flag}\}
```
(entering foo)
flag = 1
x = nullptr

note: Assuming flag is false

note: flag is assigned a value

note: Assuming flag is true

Tracked variables: \{x, flag\}
Tracked variables: \{x, \text{flag}\}

- flag = 1
  - x = nullptr

- (calling foo)

- (entering foo)
  - flag \in (-\infty, \infty)
    - x = nullptr

- (returning from foo)
  - flag \in (-\infty, \infty)
    - x = nullptr

- flag = 0
  - x = nullptr

- (calling foo)

- (entering foo)
  - flag \in (-\infty, 0) \cup (0, \infty)
    - x = nullptr

- (returning from foo)
  - flag \in (-\infty, 0) \cup (0, \infty)
    - x = nullptr

- (calling foo)

- (entering foo)
  - flag \in (-\infty, 0) \cup (0, \infty)
    - x = nullptr

- (returning from foo)
  - flag \in (-\infty, 0) \cup (0, \infty)
    - x = nullptr
dereference of x!

- flag \in (-\infty, 0) \cup (0, \infty)
  - x = nullptr
dereference of x!

- T racked variables: \{x, \text{flag}\}

- note: Assuming flag is false

- note: flag is assigned a value

- note: Assuming flag is true
FindLastStoreBRVisitor: 'x' initialized to null pointer value

Tracked variables: \{x, flag\}
Stage 2: Non-visitor notes

- `flag = 1
  x = nullptr
  (calling foo)
  (entering foo)
  flag ∈ (−∞, 0) ∪ (0, ∞)
  x = nullptr
  (returning from foo)
  flag ∈ (−∞, 0) ∪ (0, ∞)
  x = nullptr
- `flag = 0
  x = nullptr
  (calling foo)
  (entering foo)
  flag ∈ (−∞, 0) ∪ (0, ∞)
  x = nullptr
  (returning from foo)
  flag ∈ (−∞, 0) ∪ (0, ∞)
  x = nullptr
- `note: x initialized to nullptr`
- `note: Assuming flag is false`
- `note: flag is assigned a value`
- `note: Assuming flag is true`
- `note: Dereference of x!`
flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!

The warning message is supplied by the checker

note: x initialized to nullptr

note: Assuming flag is false

flag is assigned a value

note: flag is assigned a value

note: Assuming flag is true

warning: Nullptr dereference

dereference of x!
flag ∈ (−∞, 0) ∪ (0, ∞)

x = nullptr

note: x initialized to nullptr

dereference of x!

warning: Nullptr dereference

note: Assuming flag is false

note: flag is assigned a value

note: Assuming flag is true

warning: Nullptr dereference
Returning from 'foo'

\[(\text{returning from } \text{foo}) \quad \text{flag } \in (\neg \infty, \infty) \quad x = \text{nullptr}\]

- flag = 1
  - \text{returning from } \text{foo}
  - \text{flag } \in (\neg \infty, \infty)
  - x = \text{nullptr}

- flag = 0
  - x = \text{nullptr}

- note: x initialized to nullptr

- note: Assuming flag is false

- note: flag is assigned a value

- note: Returning from foo

- warning: Nullptr dereference

- note: Assuming flag is true
flag $\in (-\infty, \infty)$

\[ x = \text{nullptr} \]

warning: Nullptr dereference

note: x initialized to nullptr

note: flag is assigned a value

note: Returning from foo

note: Assuming flag is false

note: Assuming flag is true
Entering call from 'main'

(entering foo)
flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

warning: Nullptr dereference
Calling 'foo'

flag = 1
x = nullptr
(calling foo)

flag = 1
x = nullptr
(entering foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr
dereference of x!

warning: Nullptr dereference

note: x initialized to nullptr

note: Assuming flag is true
note: Returned from foo
note: flag is assigned a value
note: Entered function from main

Calling 'foo'

(flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr)

(flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr)

(flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr)

(expecting_flag)

note: Assuming flag is false
note: Calling foo
note: Entered function from main
note: flag is assigned a value
note: Returning from foo
note: Assuming flag is true
flag = 0
x = nullptr

note: x initialized to nullptr

note: Assuming flag is false

note: Calling foo

note: Entered function from main

note: flag is assigned a value

note: Returning from foo

warning: Nullptr dereference
(returning from foo)
flag ∈ (−∞, ∞)
x = nullptr

Note: x initialized to nullptr

Note: Returning from foo
Note: Assuming flag is false
Note: Calling foo
Note: Entered function from main
Note: flag is assigned a value
Note: Returning from foo
Note: Assuming flag is true
Warning: Nullptr dereference
flag ∈ (−∞, ∞)

x = nullptr

warning: Nullptr dereference

note: x initialized to nullptr

note: Returning from foo

note: Assuming flag is false

note: Calling foo

note: Entered function from main

note: flag is assigned a value

note: Returning from foo

note: Assuming flag is true

warning: Nullptr dereference
Entering call from 'main'

(entering foo)
flag = 1
x = nullptr

(flag = 1
x = nullptr)
(calling foo)

(flag = 1
x = nullptr)
(entering foo)

(flag ∈ (−∞,∞)
x = nullptr)
(calling foo)

(flag ∈ (−∞,∞)
x = nullptr)
(entering foo)

(flag ∈ (−∞,0) \cup (0,∞)
x = nullptr)
(calling foo)

(flag ∈ (−∞,0) \cup (0,∞)
x = nullptr)
(entering foo)

(flag ∈ (−∞,0) \cup (0,∞)
x = nullptr)
dereference of x!

warning: Nullptr dereference

note: x initialized to nullptr
note: Entered function from main
note: Returning from foo
note: Assuming flag is false
note: Calling foo
note: Entered function from main
note: flag is assigned a value
note: Returning from foo
note: Assuming flag is true

flag = 0
x = nullptr
Calling 'foo'

(calling foo)

- flag = 1
  - x = nullptr

Note: x initialized to nullptr

- flag = 0
  - x = nullptr

Note: flag is assigned a value

- flag = 0
  - x = nullptr

Note: Assuming flag is false

- flag = 1
  - x = nullptr

Note: flag is assigned a value

- flag = 1
  - x = nullptr

Note: flag is assigned a value

Note: Returning from foo

Warning: Nullptr dereference

Note: Assuming flag is true
flag = 1
x = nullptr

note: x initialized to nullptr

note: Calling foo

note: Enter function from main

note: Calling foo

note: Returning from foo

note: Assuming flag is false

note: Calling foo

note: Enter function from main

note: flag is assigned a value

note: Returning from foo

note: Assuming flag is true

warning: Nullptr dereference
Stage 3: Pruning

flag = 1
x = nullptr
(calling foo)

entering foo
flag = 1
x = nullptr

(returning from foo)
flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

(dereference of x!)

warning: Nullptr dereference
note: x initialized to nullptr
note: Calling foo
note: Entered function from main

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

(returning from foo)
flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

(dereference of x!)

note: Returning from foo
note: Assuming flag is false
note: Calling foo
note: Entered function from main

flag is assigned a value
note: Returning from foo
note: Assuming flag is true
warning: Nullptr dereference

Stage 3: Pruning
flag = 1
x = nullptr

(calling foo)

(entering foo)
flag = 1
x = nullptr

flag ∈ (−∞, ∞)
x = nullptr

(returning from foo)
flag ∈ (−∞, ∞)
x = nullptr

flag = 0
x = nullptr

(calling foo)

(entering foo)
flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

flag ∈ (−∞, ∞)
x = nullptr

(returning from foo)
flag ∈ (−∞, ∞)
x = nullptr

flag ∈ (−∞, 0) ∪ (0, ∞)
x = nullptr

warning: Nullptr dereference

dereference of x!

note: x initialized to nullptr
note: Calling foo

note: Entered function from main

note: Returning from foo
note: Assuming flag is false
note: Calling foo

note: Entered function from main

note: flag is assigned a value
note: Returning from foo
note: Assuming flag is true
flag = 1
x = nullptr

(calling foo)

(entering foo)
flag ∈ (−∞, 0] ∪ (0, ∞)
x = nullptr

(returning from foo)
flag ∈ (−∞, 0] ∪ (0, ∞)
x = nullptr

flag = 0
x = nullptr

(calling foo)

(entering foo)
flag ∈ (−∞, 0] ∪ (0, ∞)
x = nullptr

(flag is assigned a value)

(returning from foo)
flag ∈ (−∞, 0] ∪ (0, ∞)
x = nullptr

(dereference of x!)
warning: Nullptr dereference

note: x initialized to nullptr
note: Assuming flag is false
note: Calling foo
note: Entered function from main
note: flag is assigned a value
note: Returning from foo
note: Assuming flag is true
warning: Nullptr dereference
```c
int flag;
bool coin();
void foo() {
    flag = coin();
    // Entered call from 'main'

    // Value assigned to 'flag', which participates in a condition later
}

int main() {
    int *x = 0;
    // 'x' initialized to a null pointer value

    flag = true;
    foo();
    // Calling 'foo'

    if (flag) {
        x = new int;
    }
    // Returning from 'foo'
    foo();
    // Returning from 'foo'

    if (flag) {
        // Assuming 'flag' is not equal to 0
        *x = 5;
        // Dereference of null pointer (loaded from variable 'x'
    }
}
```
Present problems, research
Very hard to solve problems

- Relevant information isn’t found in the bug path
Very hard to solve problems

- Relevant information isn’t found in the bug path
- Ad absurdum, not even in the ExplodedGraph
```c
int flag;

bool coin();

void foo() {
  flag = coin();
  // Value assigned to 'flag', which participates in a condition later
}

int main() {
  int *x = 0;
  // 'x' initialized to a null pointer value
  flag = true;
  foo();
  if (flag) {
    x = new int;
    // Assuming 'flag' is 0
  }
  foo();
  // Calling 'foo'
  if (flag) {
    x = 5;
    // Assuming 'flag' is not equal to 0
  }
  // Dereference of null pointer (loaded from variable 'x')
```
```c
int flag;
bool coin();

void foo() {
    flag = coin();
}

int main() {
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        x = new int;
    foo();
    if (flag)
        *x = 5;
}
```
int main() {
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        x = new int;
    foo();
    if (flag)
        *x = 5;
}
int main() {
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        printf("Nothing to see here!");
    foo();
    if (flag)
        *x = 5;
}
Reaching definitions analysis

- An algorithm to find a set of last stores (definitions) to a variable
- Regard all definitions to a variable as a point of interest
- https://reviews.llvm.org/D64991
# main:

```c
int flag;
bool coin();

void foo() {
  flag = coin();
}

int main() {
  int *x = 0;
  flag = 1;
  foo();
  if (flag)
    x = new int;
  foo();
  if (flag)
    *x = 5;
}
```

- `main` function:
  - declares `int flag` and `bool coin()`
  - defines `void foo()` function
  - initializes `int *x` and `flag` to 1
  - calls `foo()`
  - checks `flag` and initializes `x` with a new integer
  - calls `foo()` again
  - checks `flag` and assigns 5 to `*x`

- `foo` function:
  - initializes `flag` with `coin()`
  - calls `foo()`
  - initializes `x` with a new integer
  - calls `foo()` again
  - checks `flag` and assigns 5 to `*x`
```c
int *x = 0;
flag = 1;
foo();
if (flag)
   x = new int;
foo();
if (flag)
   *x = 5;

int main() {
   int flag;
   bool coin();

   void foo() {
      flag = coin();
   }

   int main() {
      int *x = 0;
      flag = 1;
      foo();
      if (flag)
         x = new int;
      foo();
      if (flag)
         *x = 5;
   }
```
```c
int flag;
bool coin();

void foo() {
    flag = coin();
}

int main() {
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        x = new int;
    foo();
    if (flag)
        *x = 5;
}
```
```c
int *x = 0;
flag = 1;
foo();
if (flag)
  x = new int;
foo();
if (flag)
  *x = 5;
```

```c
int flag;
bool coin();

void foo() {
  flag = coin();
}

int main() {
  int *x = 0;
  flag = 1;
  foo();
  if (flag)
    x = new int;
  foo();
  if (flag)
    *x = 5;
}
```
```c
01 int flag;
02 bool coin();
03 
04 void foo() {
05   flag = coin();
06 }
07 
08 int main() {
09   int *x = 0;
10   flag = 1;
11   foo();
12   if (flag)
13       x = new int;
14   foo();
15   if (flag)
16       *x = 5;
17   }
18 }
```
int *x = 0;
flag = 1;
foo();
if (flag)
    x = new int;
foo();
if (flag)
    *x = 5;

int main() {
    int flag;
    bool coin();
    void foo() {
        flag = coin();
        foo();
        if (flag)
            *x = 5;
    }
    int *x = 0;
    flag = 1;
    foo();
    if (flag)
        x = new int;
    foo();
    if (flag)
        *x = 5;
}
Problems with reaching definitions

- Originally concieved for instructions
- Incredibly complex to implement for C, C++, etc...
- Doesn’t argue about aliasing
- Only works in a given CFG...
Problems with reaching definitions

- Originally conceived for instructions
- Incredibly complex to implement for C, C++, etc...
- Doesn’t argue about aliasing
- Only works in a given CFG...
- Using visitors, it’s possible to make this algorithm semi-interprocedural
Conclusion
Clear and precise bug reports are important.
Conclusion

- Clear and precise bug reports are important
- The analyzer uses callbacks (or visitors) and interestingness propagation to construct path-sensitive bug reports
Conclusion

- Clear and precise bug reports are important
- The analyzer uses callbacks (or visitors) and interestingness propagation to construct path-sensitive bug reports
- Problems that require arguing outside the bugpath, especially the ExplodedGraph are insanely difficult
Conclusion

- Clear and precise bug reports are important
- The analyzer users callbacks (or visitors) and interestingness propagation to construct path-sensitive bug reports
- Problems that require arguing outside the bugpath, especially the ExplodedGraph are insanely difficult
- The analyzer gets better by the minute
```c
int flag;
bool coin();
void foo() {
    flag = coin();
}

void bar() {
    int *x = 0;
    flag = true;
    foo();
    if (!flag) {
        x = new int;
    }
    foo();
    if (!flag) {
        *x = 1;
    }
}
```
```c
int flag;

bool coin();

void foo() {
    flag = coin();
}

int main() {
    int *x = 0;
    flag = true;
    foo();
    if (flag) {
        x = new int;
    }
    foo();
    if (flag) {
        x = 5;
    }
}
Questions?