static analysis

fuzzing/symbolic exec imprecision

symbolic execution had some nice properties:

- could reliably enumerate possible paths
- could figure out inputs
- could prove paths are impossible

but had huge practical problems:

- not enough time/space to explore all those paths
- too complicated to actually solve equations to find inputs

greybox fuzzing: one practical compromise

- replaced equation solving with (educated) guessing
- tried to explore enough paths

complete versus sound

complete:

if way to reach assertion failure, analysis finds it

sound:

if analysis finds way to reach assertion failure, it's fails the assertion

symbolic execution, greybox fuzzing: always sound

because they actually run the program

symbolic execution: complete **if all paths are solved**

but that isn't practical for a large program

other program analysis designs

other design points than symbolic execution:

tracking *all the varaible values*

alternative: *just track properties of interest*

compute precisely *what paths through code are possible*

alternative: *use some approximation*

model for use-after-free

model for use-after-free, pointer is:

- allocated
- freed
- (other states?)

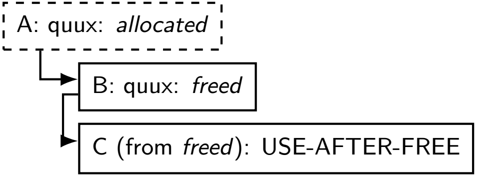
just track this logical state for each pointer

ignore everything else

assume all if statements/loop conditions can be true or false

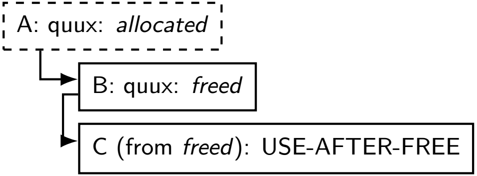
checking use-after-free (1)

```
void someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    ... /* omitted code that doesn't use quux */
    free(quux);
    // B
    ... /* omitted code that doesn't use quux */
    // C
    *quux = bar;
    ...
}
```



checking use-after-free (1)

```
void someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    ... /* omitted code that doesn't use quux */
    free(quux);
    // B
    ... /* omitted code that doesn't use quux */
    // C
    *quux = bar;
    ...
}
```



checking use-after-free (1)

```
void someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    ... /* omitted code that doesn't use quux */
    free(quux);
    // B
    ... /* omitted code that doesn't use quux */
    // C
    *quux = bar;
    ...
}
```

A: quux: *allocated*

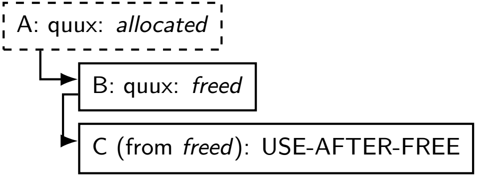
B: quux: *freed*

C (from *freed*): USE-AFTER-FREE

analysis can give warning — almost certainly bad

checking use-after-free (1)

```
void someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    ... /* omitted code that doesn't use quux */
    free(quux);
    // B
    ... /* omitted code that doesn't use quux */
    // C
    *quux = bar;
    ...
}
```



analysis can give warning — almost certainly bad

exercise: how could this be a false positive?

result from clang's scan-build

Summary > **Report bf2b5d**

Bug Summary

File:	example1.c
Warning:	line 8, column 11 Use of memory after it is freed

Report Bug

Annotated Source Code

Press '?' to see keyboard shortcuts

Show analyzer invocation

☐ Show only relevant lines

```
1   extern void SomethingUnknown();
2
3   int *someFunction(int foo, int bar) {
4       int *quux = malloc(sizeof(int));
                  1  Memory is allocated →
5       SomethingUnknown();
6       free(quux);
          2  ← Memory is released →
7       SomethingUnknown();
8       *quux = bar;
              3  ← Use of memory after it is freed
9       SomethingUnknown();
10  }
```

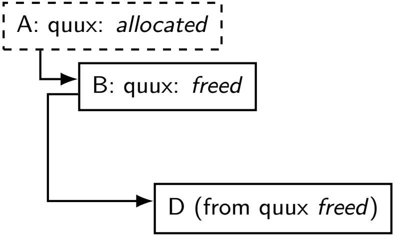
checking use-after-free (2)

```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ... /* omitted code that doesn't use quux */
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ... /* omitted code that doesn't use quux */
    // D
}
```

A: quux: *allocated*

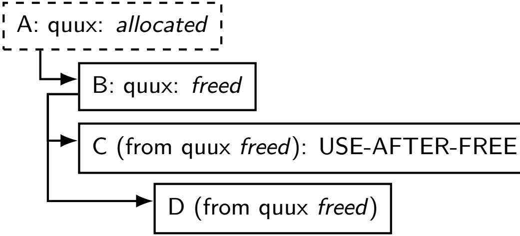
checking use-after-free (2)

```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ... /* omitted code that doesn't use quux */
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ... /* omitted code that doesn't use quux */
    // D
}
```



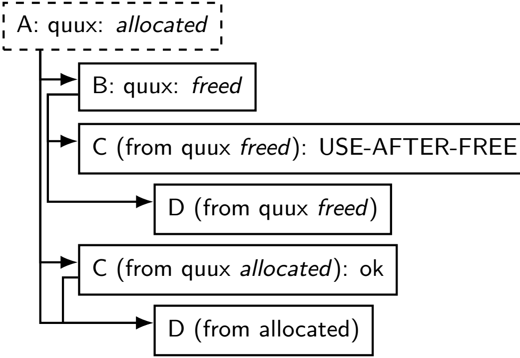
checking use-after-free (2)

```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ... /* omitted code that doesn't use quux */
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ... /* omitted code that doesn't use quux */
    // D
}
```



checking use-after-free (2)

```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ... /* omitted code that doesn't use quux */
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ... /* omitted code that doesn't use quux */
    // D
}
```



one idea: guess that Complex(foo) can be probably be true

option 1: say "something wrong maybe"?

result from clang's scan-build

```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));

            1  Memory is allocated →

    if (Complex(foo)) {

        2  ← Assuming the condition is true →

      3  ← Taking true branch →

        free(quux);

        4  ← Memory is released →

    }
    SomethingUnknown();
    if (Complex(bar)) {

        5  ← Assuming the condition is true →

      6  ← Taking true branch →

        *quux = bar;

          7  ← Use of memory after it is freed

    }
    SomethingUnknown();
}
```

exercise: holes in the model?

```
void example(int a) {
    int *p;
    int *q;
    q = malloc(...);
    p = malloc(...);
    // (A)
    if (a > 0) {
        // (A1)
        p = q;
    }
    // (B)
    free(p);
    // (C)
    ...
}
```

exercise: what should state of pointer q be at C?

A. allocated B. freed

C. allocated if+only if reached via path with A1

D. freed if+only if reached via path with A1

E. something else?

clang-analyzer output

```
void example(int a) {
    int *p;
    int *q;
    q = malloc(4);

        1 Memory is allocated →

    p = malloc(4);
    // (A)
    if (a > 0) {

        2 ← Assuming 'a' is > 0 →

      3 ← Taking true branch →

        // (A1)
        p = q;
    }
    // (B)
    free(p);

      4 ← Memory is released →

    // (C)
    *q = 1;

       5 ← Use of memory after it is freed

}
```

analysis building blocks

needed to track that p and q could point to same thing

common prerequisite for all sorts of program analysis

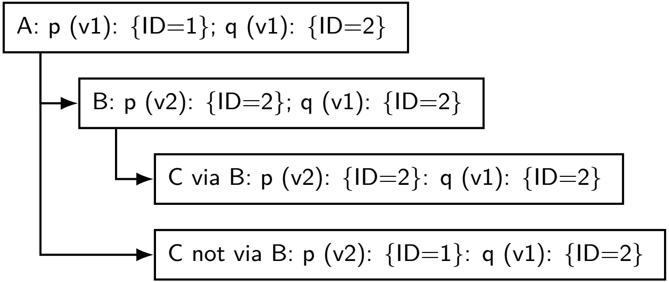
overly simple algorithm for points-to analysis

for each pointer/reference track which objects it can refer to

if multiple paths: take union of all possible

simple points-to analysis

```
void example(int a) {
    int *p;
    int *q;
    q = malloc(...); // ID=1
    p = malloc(...); // ID=2
    // (A)
    if (a > 0) {
        p = q;
        // (B)
    }
    // (C)
    ...
}
```



simple points-to analysis

```
void example(int a) {
    int *p;
    int *q;
    q = malloc(...); // ID=1
    p = malloc(...); // ID=2
    // (A)
    if (a > 0) {
        p = q;
        // (B)
    }
    // (C)
    ...
}
```

A: *p (v1)*: {ID=1}; *q (v1)*: {ID=2}

B: *p (v2)*: {ID=2}; *q (v1)*: {ID=2}

C via B: p (v2): {ID=2}: q (v1): {ID=2}

C not via B: p (v2): {ID=1}: q (v1): {ID=2}

likely first step: mark different versions of p, q
and track them as separate variables
this way: can avoid storing set of values for q for every block of code
(instead just point to q (v1) set)

simple points-to analysis

```
void example(int a) {
    int *p;
    int *q;
    q = malloc(...); // ID=1
    p = malloc(...); // ID=2
    // (A)
    if (a > 0) {
        p = q;
        // (B)
    }
    // (C)
    ...
}
```

A: p (v1): {ID=1}; q (v1): {ID=2}

B: p (v2): {ID=2}; q (v1): {ID=2}

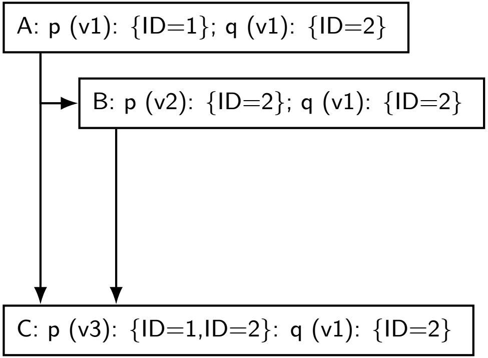
C via B: p (v2): {ID=2}: q (v1): {ID=2}

C not via B: p (v2): {ID=1}: q (v1): {ID=2}

one idea: keep track of each path separately
(but limit to how much one can do this)

simple points-to analysis

```
void example(int a) {
    int *p;
    int *q;
    q = malloc(...); // ID=1
    p = malloc(...); // ID=2
    // (A)
    if (a > 0) {
        p = q;
        // (B)
    }
    // (C)
    ...
}
```



alternate idea: avoid path explosion by merging possible sets

complicating points-to analysis

would like to analyze program function-at-a-time, but...

- functions can change values shared by other functions

what about computed array indices?

what about pointers to pointers?

...

high false-positive solution:

- when incomplete info: assume value points to anything of right type

high false-negative solution:

- when incomplete info: assume value points to nothing

checking use-after-free (3)

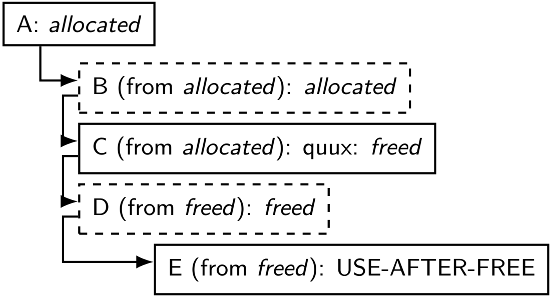
```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (anotherFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
    ...
}
```

A: *allocated*

→ B (from *allocated*): *allocated*

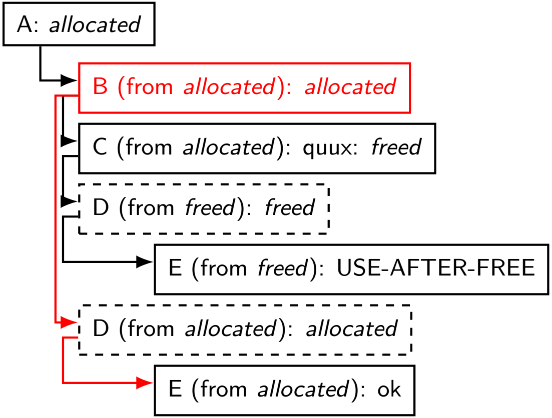
checking use-after-free (3)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (anotherFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
    ...
}
```



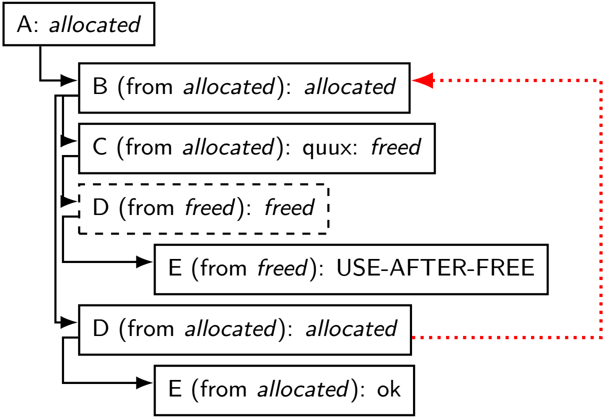
checking use-after-free (3)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (anotherFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
    ...
}
```



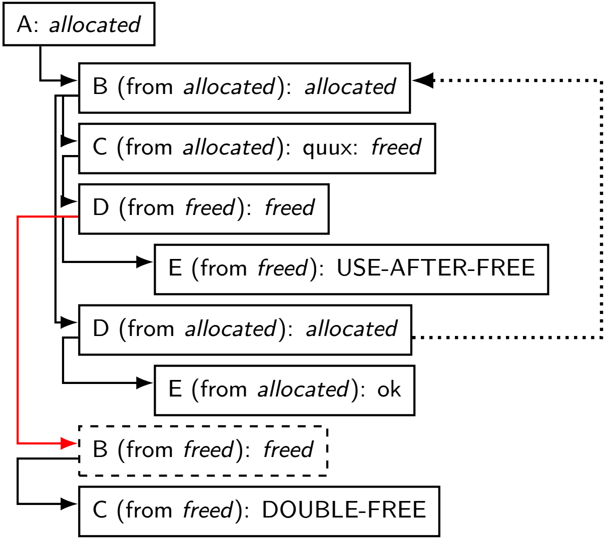
checking use-after-free (3)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (anotherFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
    ...
}
```



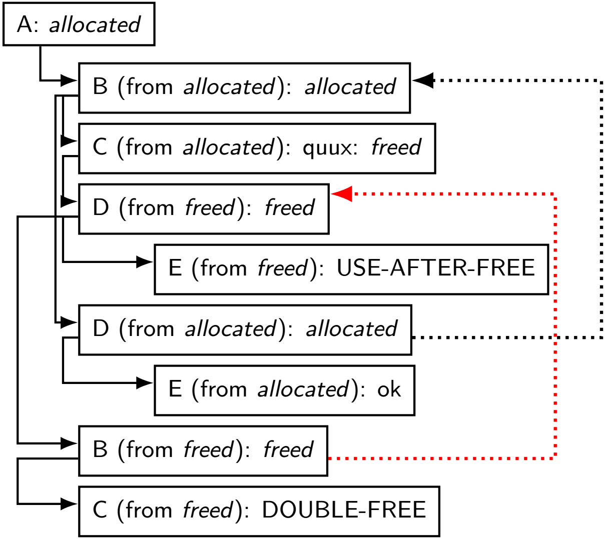
checking use-after-free (3)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (anotherFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
    ...
}
```



checking use-after-free (3)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (anotherFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
    ...
}
```



result from clang's scan-build

```
int *quux = malloc(sizeof(int));
    1  Memory is allocated →
SomethingUnknown();
// A
do {
    5  ← Loop condition is false.  Exiting loop →
    // B
    SomethingUnknown();
    if (anotherFunction()) {
        2  ← Assuming the condition is true →
        3  ← Taking true branch →
        free(quux);
        4  ← Memory is released →
        // C
    }
    SomethingUnknown();
    // D
} while (complexFunction());
SomethingUnknown();
// E
*quux++;
    6  ← Use of memory after it is freed
```

```
int *quux = malloc(sizeof(int));
    1  Memory is allocated →
SomethingUnknown();
// A
do {
    5  ← Loop condition is true. Execution continues on line 12 →
    // B
    SomethingUnknown();
    if (anotherFunction()) {
        2  ← Assuming the condition is true →
        3  ← Taking true branch →
        6  ← Assuming the condition is true →
        7  ← Taking true branch →
        free(quux);
        4  ← Memory is released →
        8  ← Attempt to free released memory
        // C
    }
    SomethingUnknown();
    // D
} while (complexFunction());
```

checking for array bounds

can *try* to apply same technique to array bounds

but much more complicated/more likely to have false positives/negatives

for each array or pointer track:
- minimum number of elements before/after what it points to

for each integer track:
- minimum bound
- maximum bound

similar analysis looking at paths?

checking array bounds (1)

```c
int array[100];
void someFunction(int foo) {
    // A
    if (foo > 100) {
        return;
    }
    // B
    array[foo] += 1;
}
```

A: foo: $[-\mathrm{inf}, +\mathrm{inf}]$; array: indices [0, 99]

B: foo: $[-\mathrm{inf}, +100]$; array: indices [0, 99]

checking array bounds (1)

```
int array[100];
void someFunction(int foo) {
    // A
    if (foo > 100) {
        return;
    }
    // B
    array[foo] += 1;
}
```

A: foo: $[-\inf, +\inf]$; array: indices [0, 99]

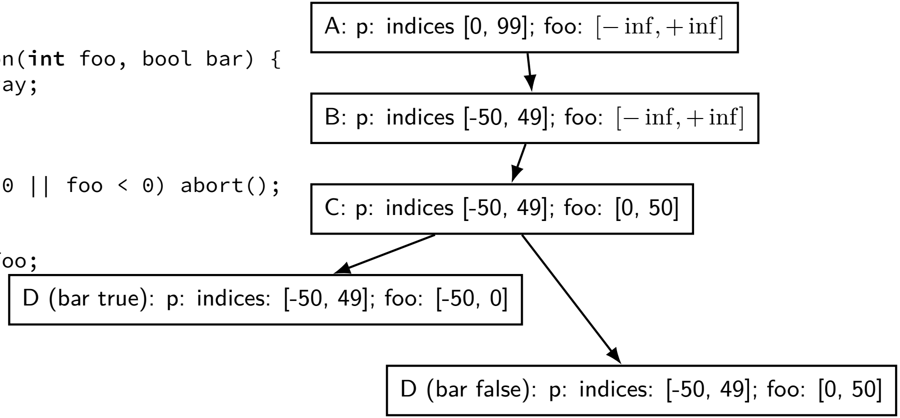
B: foo: $[-\inf, +100]$; array: indices [0, 99]

give warning about `foo == 100`? probably bug!

give warning about `foo < 0`? maybe??

checking array bounds (2)

```
int array[100];
void someFunction(int foo, bool bar) {
    int *p = array;
    // A
    p += 50;
    // B
    if (foo >= 50 || foo < 0) abort();
    // C
    if (bar) {
        foo = -foo;
    }
    // D
    p[foo] = 1;
}
```



checking array bounds (2)

```
int array[100];
void someFunction(int foo, bool bar) {
    int *p = array;
    // A
    p += 50;
    // B
    if (foo >= 50 || foo < 0) abort();
    // C
    if (bar) {
        foo = -foo;
    }
    // D
    p[foo] = 1;
}
```

A: p: indices [0, 99]; foo: $[-\mathrm{inf}, +\mathrm{inf}]$

B: p: indices [-50, 49]; foo: $[-\mathrm{inf}, +\mathrm{inf}]$

C: p: indices [-50, 49]; foo: [0, 50]

D (bar true): p: indices: [-50, 49]; foo: [-50, 0]

D (bar false): p: indices: [-50, 49]; foo: [0, 50]

warn about possible out-of-bounds?

common bug patterns

effectively detecting things like "arrays are in bounds"
or "values aren't used after being freed"
is not very reliable for large programs

(but analysis tools true and are getting better)

but static analysis tools shine for *common bug patterns*

patterns clang's analyzer knows

```
struct foo *p = malloc(sizeof(struct foo*)); // meant struct foo?
long *p = malloc(16 * sizeof(int)); // meant sizeof(long)?
```

```
strncat(foo, bar, sizeof(foo));
```

```
int *global;
int *foo() {
    int x;
    int *p = &x;
    ...
    global = p; // putting pointer to stack in global
    return p;    // returning pointer to stack
}
```

more suspect patterns

SpotBugs: Java static analysis tool

```
// pattern: connecting to database with empty password:
connection = DriverManager.getConnection(
    "jdbc:hsqldb:hsql://db.example.com/xdb" /* database ID */,
    "sa" /* username */, "" /* password */);

// pattern: Sql.hasResult()'s second argument isn't a constant
Sql.hasResult(c, "SELECT␣1␣FROM␣myTable␣WHERE␣code='"+code+"'");

// pattern: new FileReader's argument comes from request
HttpRequest request = ...;
String path = request.getParameter("path");
BufferedReader r = new BufferedReader(
    new FileReader("data/" + path));
```

static analysis

need to avoid exploring way too many paths

- clang-analyzer: only a procedure at a time
- other analyzers: some way of pruning paths

need to avoid false positives

- probably can't always assume every if can be true/false
- one idea: apply symbolic-execution like techniques to prune
- clang-analyzer: limited by being procedure-at-a-time

taint tracking?

preview: information flow

really common pattern we want to find:
data from somewhere gets to dangerous place

- pointer to stack escapes function
- input makes it to SQL query, file name

we'll talk about it specially next

information flow

so far: static analysis concerned with control flow

often, we're really worried about how *data* moves

many applications:

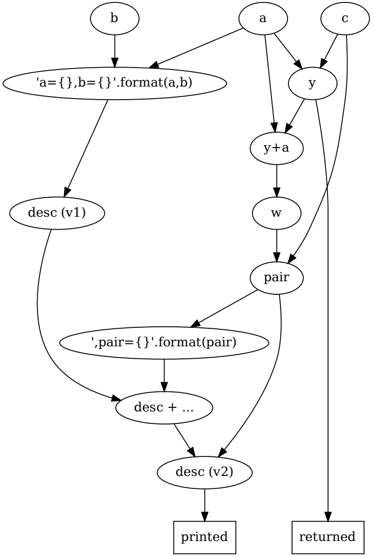
- does an array index depend on user input?
- does an SQL query depend on user input?
- does data sent over network depend on phone number?

...

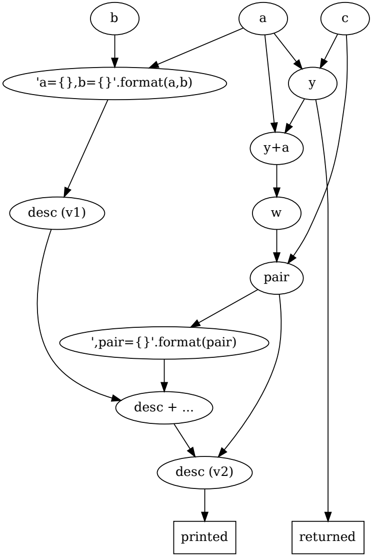
can do this *statically* (potential dependencies)
or *dynamically* (actual dependencies as program runs)

information flow graph (1a)

```
def f(a, b, c):
    desc = 'a={},b={}'.format(a, b)
    if b > 10:
        y = a
    else:
        y = c
    w = y + a
    pair = (w, c)
    desc = desc + \
        ',pair={}'.format(pair)
    print(desc)
    return y
```



information flow graph (1b)

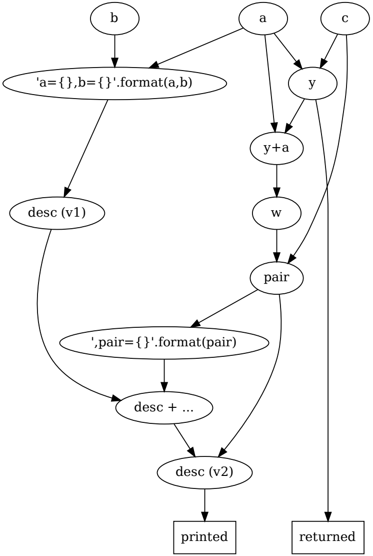


information flow graph (1b)

ex: does returned value depend on a, b, c?

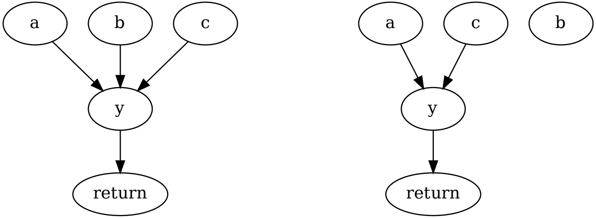
ex: does value of pair depend on a, b, c?

ex: does printed value depend on a, b, c?



information flow and control flow

```
def f(a, b, c):
    if b > 10:
        y = a
    else:
        y = c
    return y
```



Q: which is better ...

if we're trying to see if user input makes it to SQL query?

if we're trying to determine if private info goes out over network?

static info flow challenges (1)

```python
# Python example
def stash(a):
    global y
    y = a
x = [0,1,2,3]
stash(x)
x[2] = input()
print(y[2])
```

```c
// C example
int *y;
void stash(int *a) {
    y = a;
}
int main() {
    int x[3];
    stash(x);
    y[2] = GetInput();
    printf("%d\n",x[2]);
}
```

same points-to problem with static analysis

need to realize that x[2] and y[2] are the same!

even if assignment to/usage of y is more cleverly hidden

can fix this with dynamic approach: monitor running program

static info flow challenges (2)

```
def retrieve(flag):
    global the_default
    if flag:
        value = input()
    else:
        value = the_default
    value = process(value)
    if not flag:
        print("base␣on␣default:␣",value)
    return value
retrieve(True)
retrieve(False)
```

input can't make it to print here

...but need *path-sensitive* analysis to tell

can fix this w/ dynamic approach: monitor running program

static info flow challenges (3)

```
x = int(input())
if x == 0:
    print(0)
elif x == 1:
    print(1)
elif ...
```

does input make it to output?

should we try to detect this?

probably depends on intended use of analysis

harder to fix this issue

sources and sinks

needed choose *sources* (so far: function arguments)
and *sinks* (so far: print, return)

choice depends on application

SQL injection:

- sources: input from network
- sinks: SQL query functions

private info leak:

- sources: private data: phone number, message history, email, …
- sinks: network output

static analysis practicality

good at finding some kinds of bugs

array out-of-bounds probably not one — complicated tracking needed

excellent for "bug patterns" like:

```
struct Foo* foo;
...
foo = malloc(sizeof(struct Bar));
```

false positive rates are often 20+% or more

some tools assume lots of annotations

not limited to C-like languages

static analysis tools

Coverity, Fortify — commerical static analysis tools

Splint — unmaintained?

written by David Evans and his research group in the late 90s/early 00s

FindBugs (Java)

clang-analyzer — part of Clang compiler

Microsoft's Static Driver Verifier — required for Windows drivers:

mostly checks correct usage of Windows APIs