Definitions of Regions of Interest for EyeWorks Eye Tracking Analysis for Study “Effects of Precise and Imprecise Value-Set Analysis (VSA) Information on Manual Code Analysis”

Michelle Leger, David Hannasch

February 16, 2021

Contents

1 Questions of Interest

Here are the questions we want to answer:

1. Do people tend to start at the top or the bottom of the code, and does that vary by expertise? To answer this question, we have an ROI for the print statement and an ROI at the top of the code (assumptions and/or allocation statements).
2. Do people focus on the SENSITIVE parts and ignore the PUBLIC parts, or do they pay equal attention to both as they are working through these problems? To answer this question, we have an ROI for lines that deal with SENSITIVE parts of the code (tracing flow through the function) and another for lines that deal with PUBLIC parts of the code. If there are problems where those can't be teased apart, we leave those out of the item analysis.
3. Does the allocation of attention to parts of the code change when different types of models are available?

Any code that could be associated with any one of multiple ROIs (e.g., flow statements between conditional source and conditional destination selections) or is not associated obviously with a given ROI is just "other code". It is not assigned to an ROI.

2 Regions of Interest

We attempt to have multiple lines associated with each ROI. Exceptions are the print line, which is at the end of the code and will not bleed into an ROI below it, and Sensitive or Public regions that are difficult to separate. In these cases, we leave these regions in if

1. there are "other roi" regions adjacent to one side or the other, as those code regions will not be tracked separately
2. there are other regions of the same type immediately adjecent to one side or the other, although they may be split by an outer ROI wrapping only part of what is technically one ROI
3. there is only one one-line ROI

There are two patterns of initialization in the code:

1. allocations are performed in a chunk and then initializations are performed in a chunk, or
2. allocations and initializations are interleaved.

In the first case, we use the ROIs allocations and memory and variable initializations (see always-easy-callsite-code).

In the second case, we use sensitive, public, and memory initialization ROIs (ignoring allocations), depending on whether we are initializing immediately afterwards (and writing to memory or variables afterwards) with SENSITIVE, PUBLIC, or some other variable (see always-easy-flow-code). These three regions are not as clear-cut as the allocations, memory, and variable regions, and it's not clear whether that is acceptable. Effectively we lump allocations in with the thing happening around them.

2.1 Code ROIs

Start

Start: initial code; comments on entry to the code that state assumptions about initial state (here, only about values of conditionals), initial two code statements that would be observed first in reading the code from top to bottom (because the number of lines of comments changes)

Memory initialization

Memory initialization: chunks of writes to memory (statements that begin with an asterisk *) and including at least nested writes (multiple asterisks)

Conditional source

Conditional source: an **if** writing to a variable or **if**/**else** where the conditional statements write different values to the same variable

Conditional destination

Conditional destination: an **if**/**else** where the conditional statements write the same value to different variables

Sensitive

Sensitive: regions of code where the information being accessed is known to be SENSITIVE or the code instructions are directly related to only SENSITIVE information.

Public

Public: regions of code where the information being accessed is known to be PUBLIC or the code instructions are directly related to only PUBLIC information.

Print

Print: the final line of code in every example

2.2 Memory ROIs

Memory ROIs from both types of models are easily distinguishable. The ROIs are broken apart by a line break.

Variables

```
Variables, usually receiving the
    results of an alloc() call.
    These are named as they appear
     in the code.
```

Memory

```
Memory cells, whether they contain
     values or pointers.  These
    cells are named Mem<i> for
    increasing values of i.
```

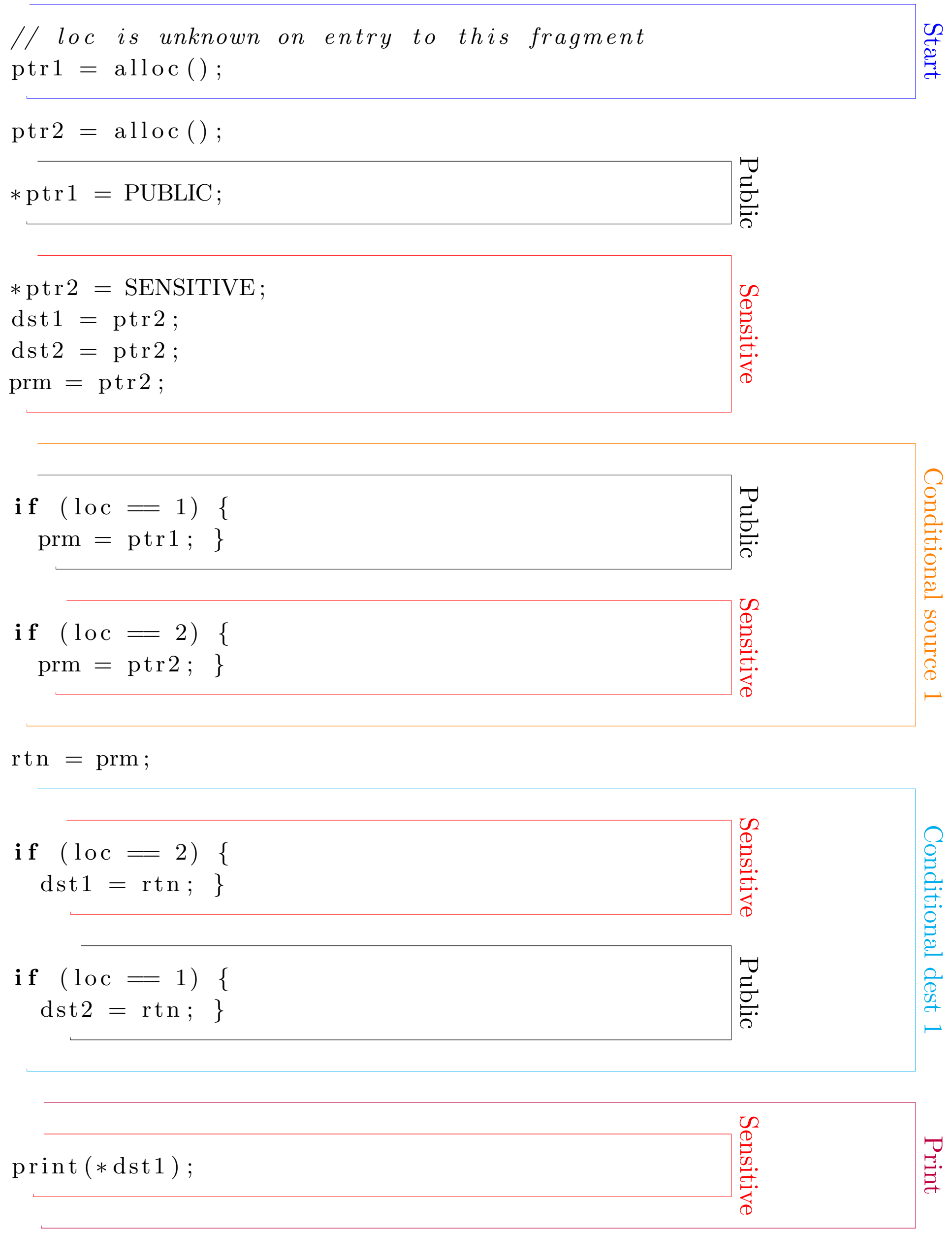
At the moment, we do not make machine-readable diagrams because it does not appear EyeWorks provides a way to enter them; Laura has to draw all the boxes by hand to get EyeWorks to track them.

We want regions to be at least two lines in height when possible, because the eye-tracking is not sensitive enough for one-line regions. Thus we may even use definitions of regions that are less clear-cut, because there's no point in using rigorous definitions that will only yield garbage data when we feed them into the eye-tracker.

3 ROIs for each code snippet

3.1 callsite-sensitivity

3.1.1 always-easy-callsite-code



```
// loc is unknown on entry to this fragment
ptr1 = alloc();

ptr2 = alloc();

*ptr1 = PUBLIC;

*ptr2 = SENSITIVE;
dst1 = ptr2;
dst2 = ptr2;
prm = ptr2;

if (loc == 1) {
  prm = ptr1; }

if (loc == 2) {
  prm = ptr2; }

rtn = prm;

if (loc == 2) {
  dst1 = rtn; }

if (loc == 1) {
  dst2 = rtn; }

print(*dst1);
```

This is inspired by an identity function call made from 2 sites where the pre-conditionals and post-conditionals are in a different order. If called from site 1, the function is called with PUBLIC data that is never printed. If called from site 2, the function ALWAYS prints SENSITIVE data.

***** MODEL A *****

Variables

```
ptr1: ->Mem1
ptr2: ->Mem2
dst1: ->Mem1, ->Mem2
dst2: ->Mem1, ->Mem2
prm: ->Mem1, ->Mem2
loc: ?
rtn: ->Mem1, ->Mem2
```

Memory

```
Mem1: PUBLIC
Mem2: SENSITIVE
```

***** MODEL B *****

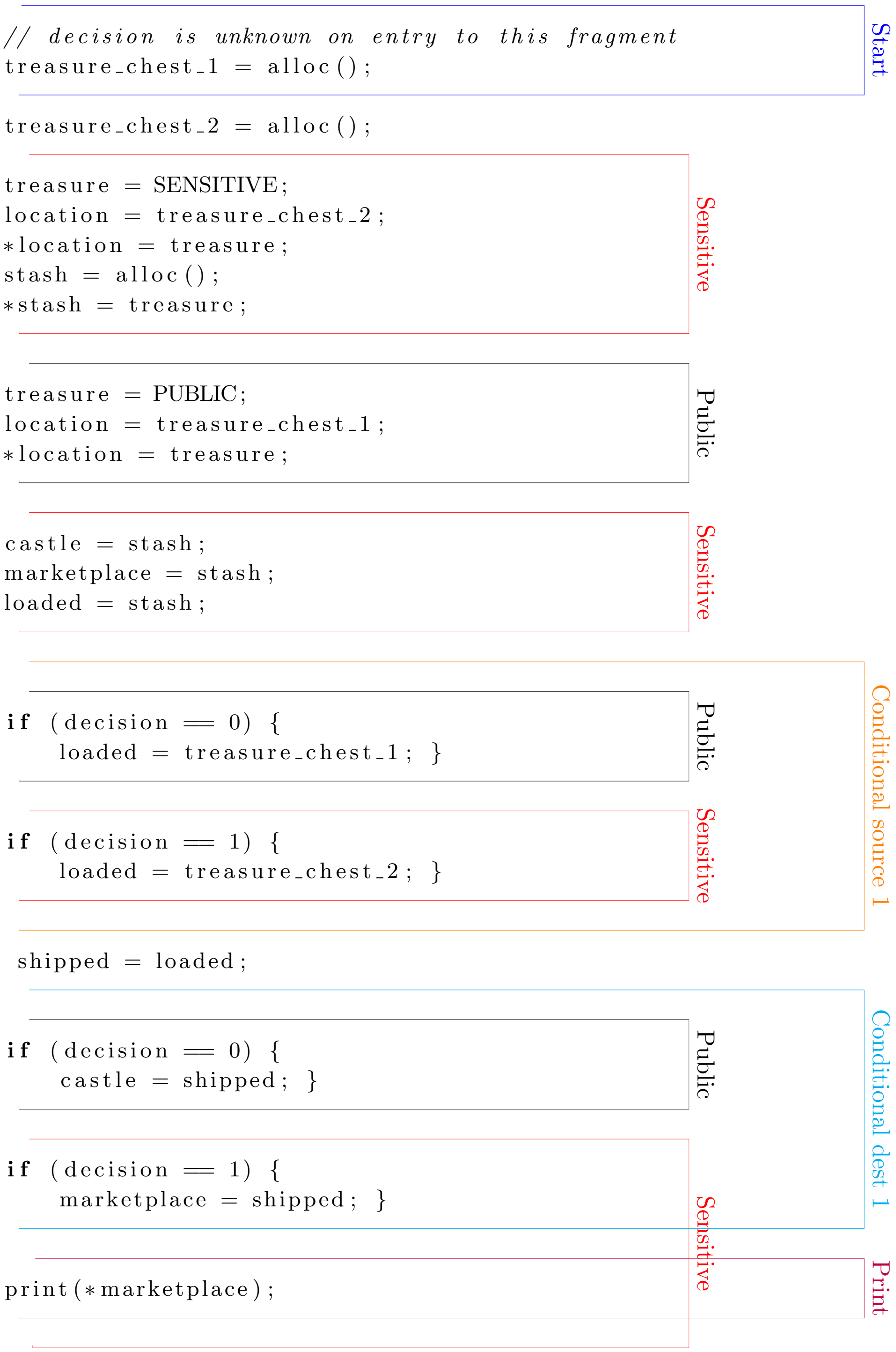
Variables

```
ptr1: ->Mem1
ptr2: ->Mem2
dst1: ->Mem2
dst2: ->Mem1, ->Mem2
prm: ->Mem1, ->Mem2
loc: ?
rtn: ->Mem1, ->Mem2
```

Memory

```
Mem1: PUBLIC
Mem2: SENSITIVE
```

3.1.2 always-medium-callsite-code



```
// decision is unknown on entry to this fragment
treasure_chest_1 = alloc();

treasure_chest_2 = alloc();

treasure = SENSITIVE;
location = treasure_chest_2;
*location = treasure;
stash = alloc();
*stash = treasure;

treasure = PUBLIC;
location = treasure_chest_1;
*location = treasure;

castle = stash;
marketplace = stash;
loaded = stash;

if (decision == 0) {
    loaded = treasure_chest_1; }

if (decision == 1) {
    loaded = treasure_chest_2; }

 shipped = loaded;

if (decision == 0) {
    castle = shipped; }

if (decision == 1) {
    marketplace = shipped; }

print(*marketplace);
```

***** MODEL A *****

```
Variables
treasure_chest_1: ->Mem1
treasure_chest_2: ->Mem2
treasure: PUBLIC, SENSITIVE
location: ->Mem1, ->Mem2
stash: ->Mem3
castle: ->Mem1, ->Mem2, ->Mem3
marketplace: ->Mem1, ->Mem2, ->Mem3
loaded: ->Mem1, ->Mem2 , ->Mem3
decision: ?
shipped: ->Mem1, ->Mem2, ->Mem3
```

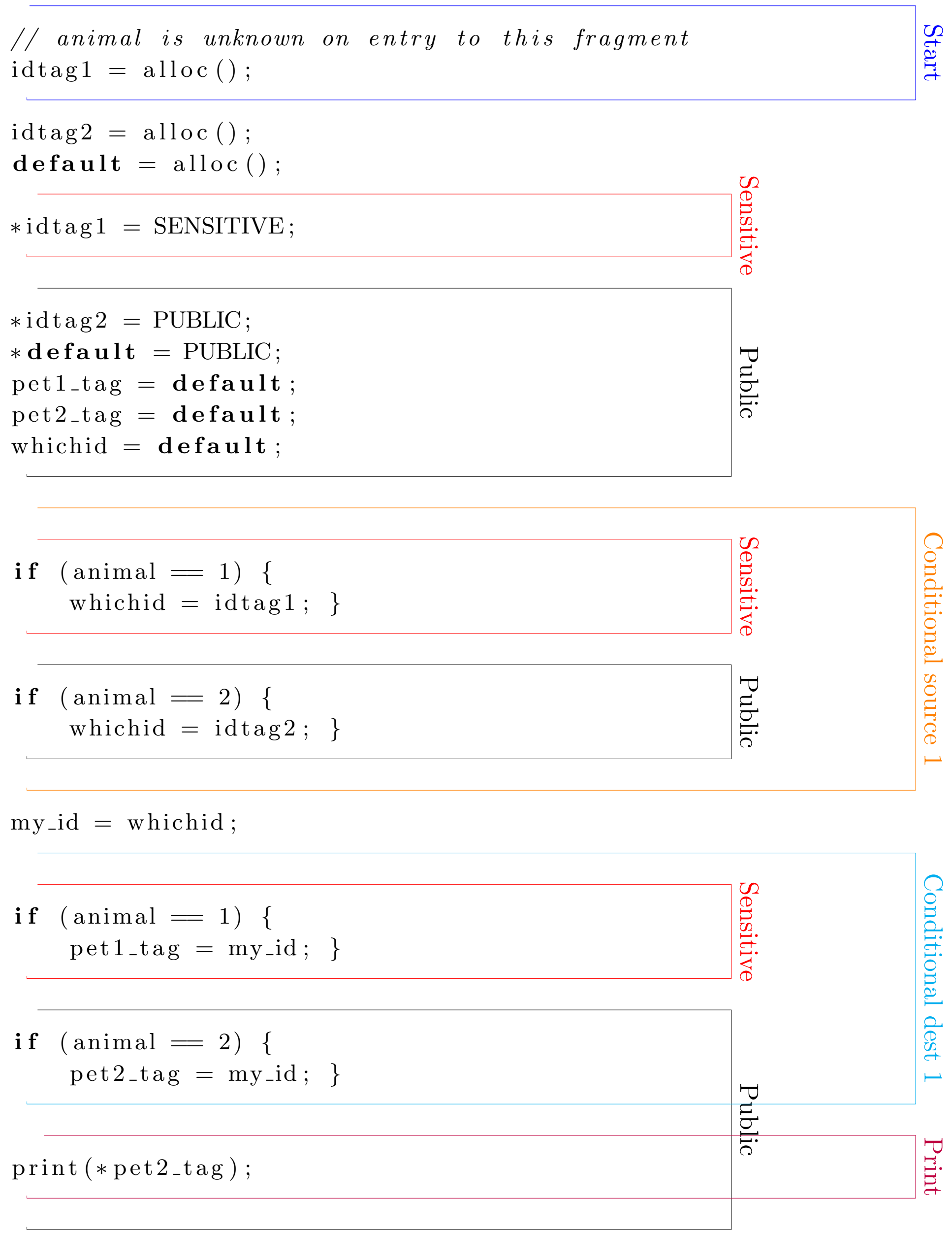
```
Memory
Mem1: PUBLIC, SENSITIVE
Mem2: PUBLIC, SENSITIVE
Mem3: PUBLIC, SENSITIVE
```

***** MODEL B *****

```
Variables
treasure_chest_1: ->Mem1
treasure_chest_2: ->Mem2
treasure: PUBLIC, SENSITIVE
location: ->Mem1, ->Mem2
castle: ->Mem1, ->Mem3
marketplace: ->Mem2, ->Mem3
loaded: ->Mem1, ->Mem2, ->Mem3
decision: ?
shipped: ->Mem1, ->Mem2, ->Mem3
```

```
Memory
Mem1: PUBLIC
Mem2: SENSITIVE
Mem3: SENSITIVE
```

3.1.3 never-easy-callsite-code



```
// animal is unknown on entry to this fragment
idtag1 = alloc();

idtag2 = alloc();
default = alloc();

*idtag1 = SENSITIVE;

*idtag2 = PUBLIC;
*default = PUBLIC;
pet1_tag = default;
pet2_tag = default;
whichid = default;

if (animal == 1) {
    whichid = idtag1; }

if (animal == 2) {
    whichid = idtag2; }

my_id = whichid;

if (animal == 1) {
    pet1_tag = my_id; }

if (animal == 2) {
    pet2_tag = my_id; }

print(*pet2_tag);
```

This is inspired by an identity function call made from 2 sites. If called from site 1, the function is called with SENSITIVE data that is never printed. If called from site 2, the function is called with PUBLIC data that is printed. This example NEVER prints SENSITIVE data.

***** MODEL A *****

```
Variables
idtag1: ->Mem1
idtag2: ->Mem2
default: ->Mem3
pet1_tag: ->Mem1, ->Mem2, ->Mem3
pet2_tag: ->Mem1, ->Mem2, ->Mem3
whichid: ->Mem1, ->Mem2, ->Mem3
animal: ?
my_id: ->Mem1, ->Mem2, ->Mem3
```

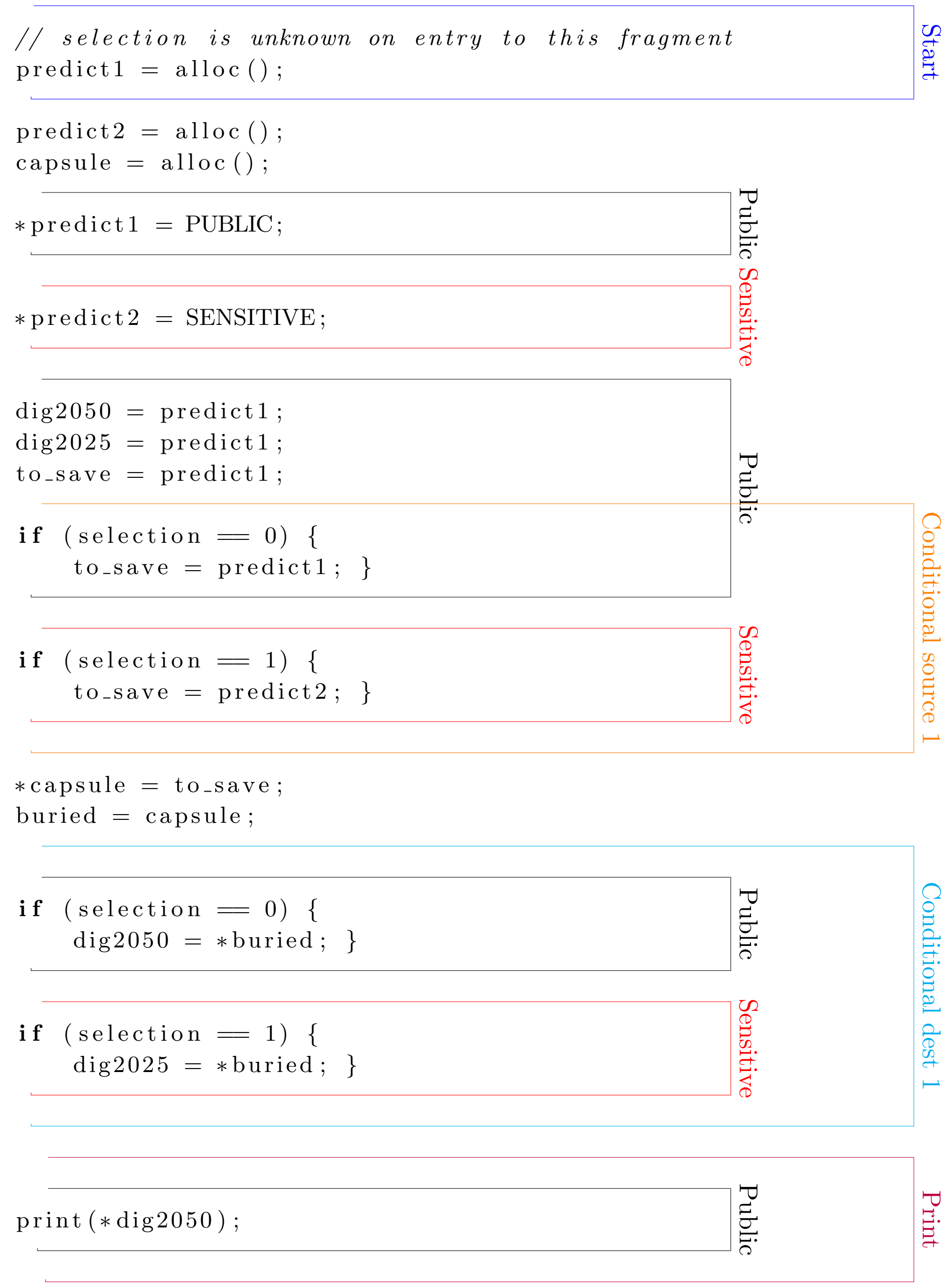
```
Memory
Mem1: SENSITIVE
Mem2: PUBLIC
Mem3: PUBLIC
```

***** MODEL B *****

```
Variables
idtag1: ->Mem1
idtag2: ->Mem2
default: ->Mem3
pet1_tag: ->Mem1, ->Mem3
pet2_tag: ->Mem2, ->Mem3
whichid: ->Mem1, ->Mem2, ->Mem3
animal: ?
my_id: ->Mem1, ->Mem2
```

```
Memory
Mem1: SENSITIVE
Mem2: PUBLIC
Mem3: PUBLIC
```

3.1.4 never-medium-callsite-code

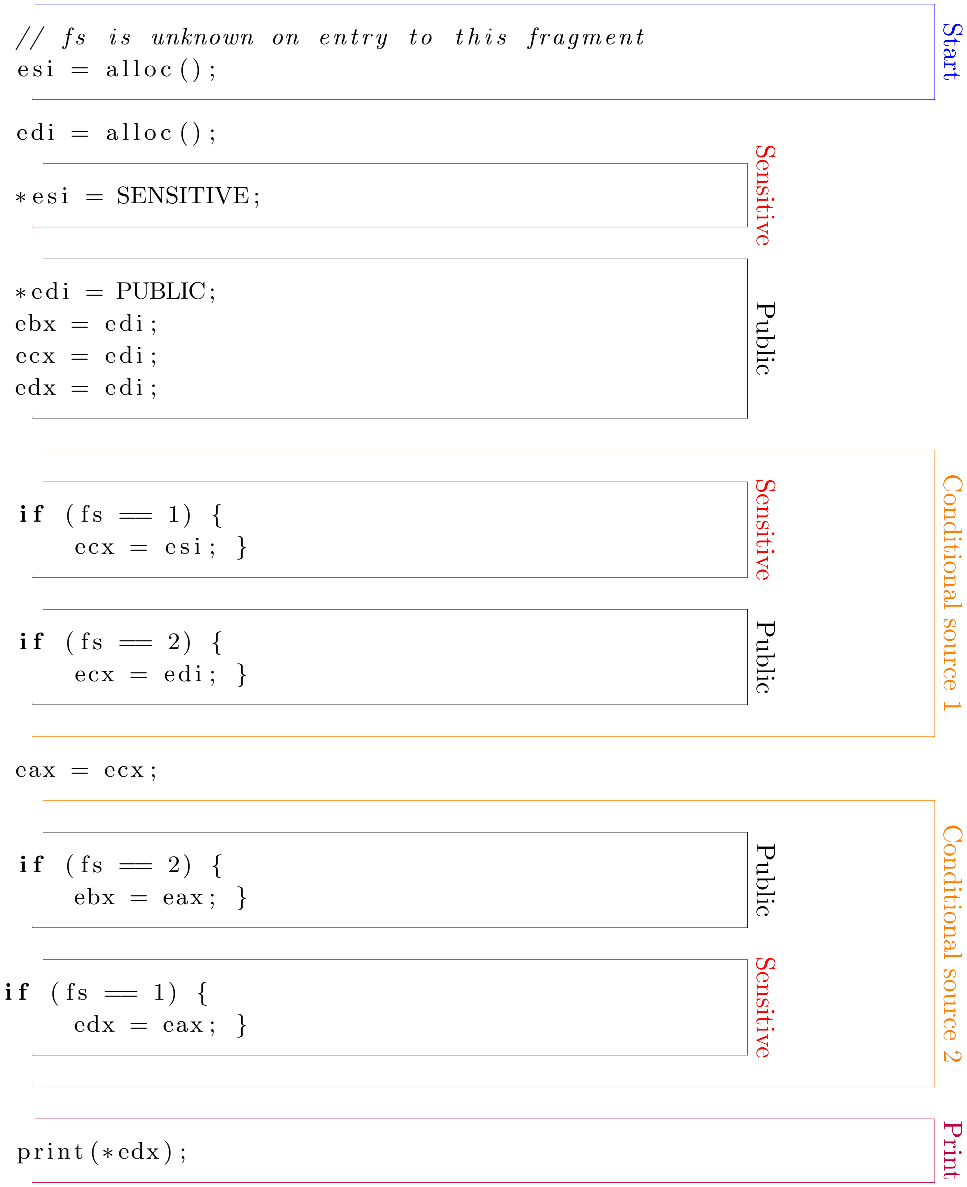


```
// selection is unknown on entry to this fragment
predict1 = alloc();

predict2 = alloc();
capsule = alloc();

*predict1 = PUBLIC;

*predict2 = SENSITIVE;

dig2050 = predict1;
dig2025 = predict1;
to_save = predict1;

if (selection == 0) {
    to_save = predict1; }

if (selection == 1) {
    to_save = predict2; }

*capsule = to_save;
buried = capsule;

if (selection == 0) {
    dig2050 = *buried; }

if (selection == 1) {
    dig2025 = *buried; }

print(*dig2050);
```

WARNING to users of these ROIs. This may be a problematic case that you want to exclude from the itemset analysis – one liners exist in ROIs. –

This is inspired by the identity function call made from 2 sites where the return code is pass by reference instead of pass by value. If called from site 0, the function is called with PUBLIC data that is printed. If called from site 1, the function is called with SENSITIVE data that is never printed. This NEVER prints SENSITIVE data.

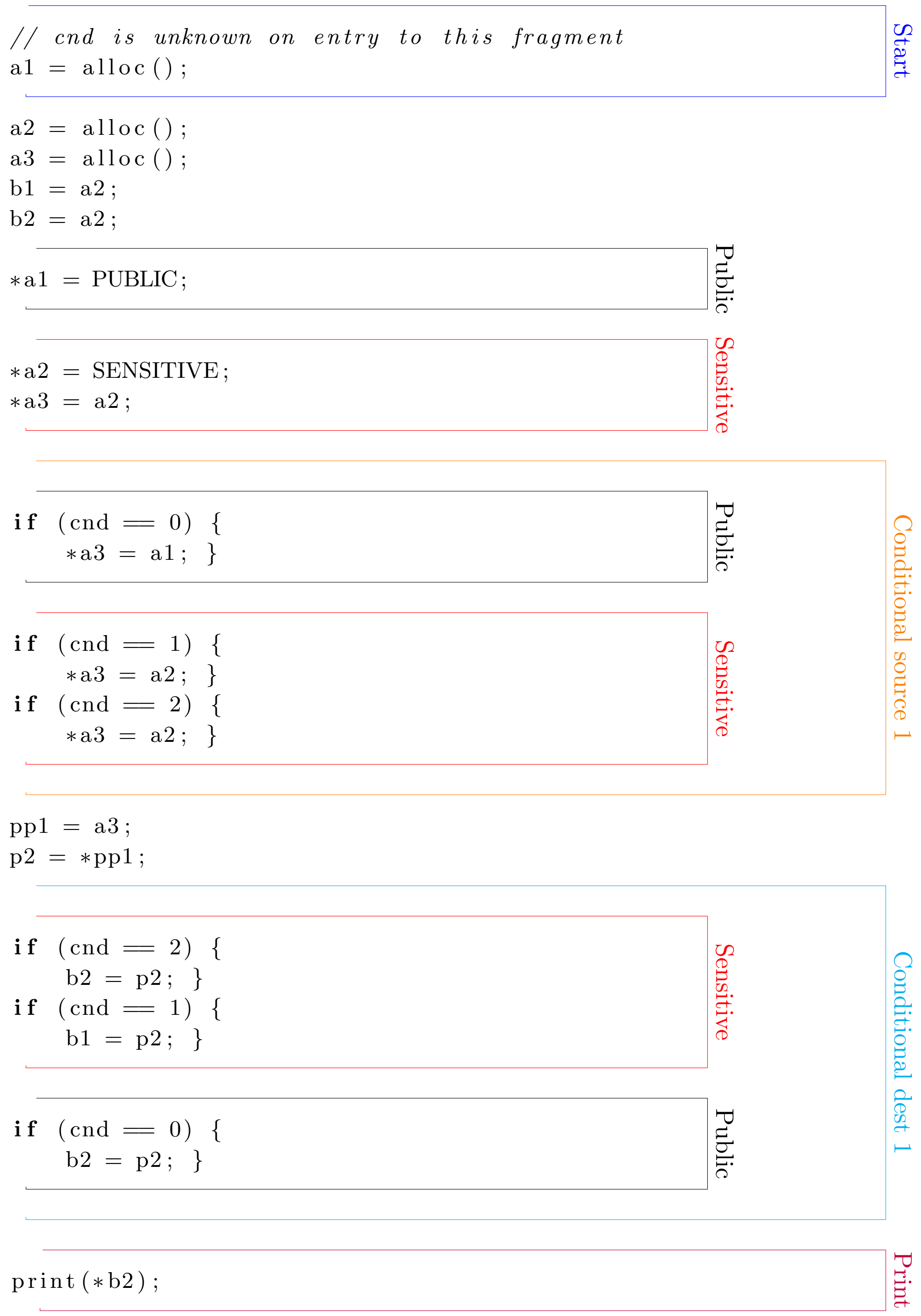
3.1.5 sometimes-easy-callsite-code



```
// fs is unknown on entry to this fragment
esi = alloc();

edi = alloc();

*esi = SENSITIVE;

*edi = PUBLIC;
ebx = edi;
ecx = edi;
edx = edi;

if (fs == 1) {
    ecx = esi; }

if (fs == 2) {
    ecx = edi; }

eax = ecx;

if (fs == 2) {
    ebx = eax; }

if (fs == 1) {
    edx = eax; }

print(*edx);
```

This is inspired by an identity function call made from 2 call sites. If called from site 1, the function is called with SENSITIVE data. From the other site, it is called with PUBLIC data. From call site 2, the return value is placed in ebx. From the other site, it is placed in edx and is then printed. From call site 1, this would definitely leak sensitive data. From call site 2, this would not leak any data (and would not even print PUBLIC data). However, this problem has an indeterminate answer because we do not know if call site 1 is reachable (if fs can be 1). Thus this example SOMETIMES prints SENSTIVE data, depending on fs.

3.1.6 sometimes-medium-callsite-code



```
// cnd is unknown on entry to this fragment
a1 = alloc();

a2 = alloc();
a3 = alloc();
b1 = a2;
b2 = a2;

*a1 = PUBLIC;

*a2 = SENSITIVE;
*a3 = a2;

if (cnd == 0) {
    *a3 = a1; }

if (cnd == 1) {
    *a3 = a2; }
if (cnd == 2) {
    *a3 = a2; }

pp1 = a3;
p2 = *pp1;

if (cnd == 2) {
    b2 = p2; }
if (cnd == 1) {
    b1 = p2; }

if (cnd == 0) {
    b2 = p2; }

print(*b2);
```

** Need more information about the value of cnd.

This is inspired by the identity function call made from 3 sites where the calling code is pass by reference instead of pass by value. If called from site 0, the function is called with PUBLIC data that is printed. If called from site 1, the function is called with SENSITIVE data that is not printed. If called from call site 2, the function is called with SENSITIVE data that is printed. However, this problem has an indeterminate answer because we do not know if call site 2 is even possible (if cnd can be anything but 0 and 1). Thus this example SOMETIMES prints SENSITIVE data, depending on cnd.

***** MODEL A *****

```
Variables
a1: ->Mem1
a2: ->Mem2
a3: ->Mem3
b1: ->Mem1, ->Mem2
b2: ->Mem1, ->Mem2
cond: ?
pp1: ->Mem3
p2: ->Mem1, ->Mem2
```

```
Memory
Mem1: PUBLIC
Mem2: SENSITIVE
Mem3: ->Mem1, ->Mem2
```

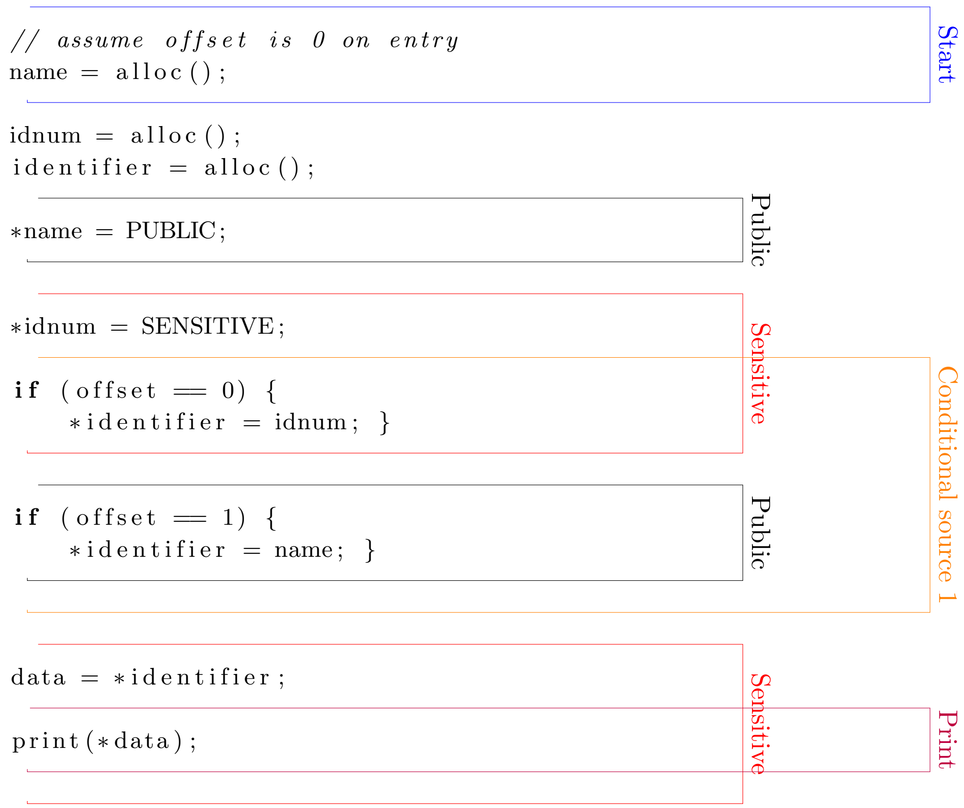
***** MODEL B *****

```
Variables
a1: ->Mem1
a2: ->Mem2
a3: ->Mem3
b1: ->Mem2
b2: ->Mem1, ->Mem2
cond: ?
pp1: ->Mem3
p2: ->Mem1, ->Mem2
```

```
Memory
Mem1: PUBLIC
Mem2: SENSITIVE
Mem3: ->Mem1, ->Mem2
```

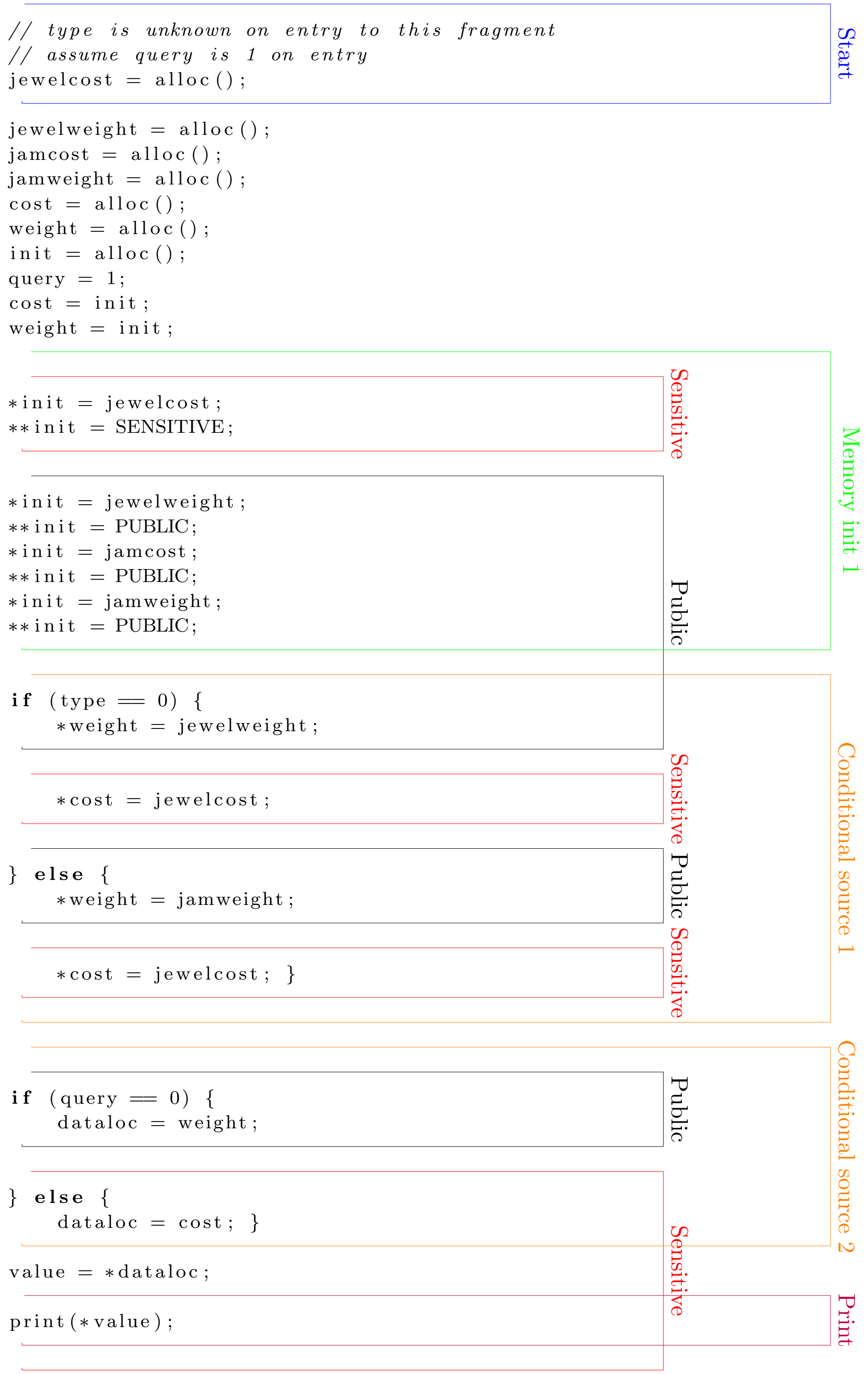
3.2 field-sensitivity

3.2.1 always-easy-field-code



```
// assume offset is 0 on entry
name = alloc();

idnum = alloc();
identifier = alloc();

*name = PUBLIC;

*idnum = SENSITIVE;

if (offset == 0) {
    *identifier = idnum; }

if (offset == 1) {
    *identifier = name; }

data = *identifier;

print(*data);
```

3.2.2 always-medium-field-code

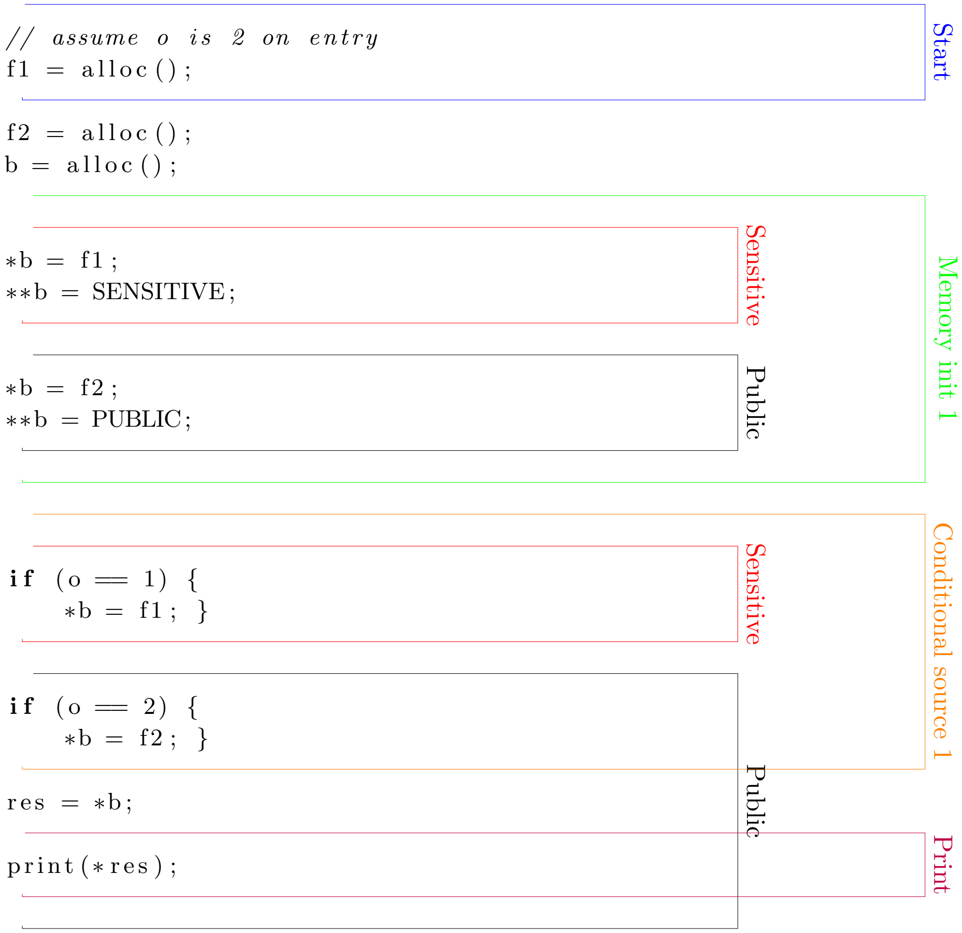


```
// type is unknown on entry to this fragment
// assume query is 1 on entry
jewelcost = alloc();

jewelweight = alloc();
jamcost = alloc();
jamweight = alloc();
cost = alloc();
weight = alloc();
init = alloc();
query = 1;
cost = init;
weight = init;

*init = jewelcost;
**init = SENSITIVE;

*init = jewelweight;
**init = PUBLIC;
*init = jamcost;
**init = PUBLIC;
*init = jamweight;
**init = PUBLIC;

if (type == 0) {
    *weight = jewelweight;

    *cost = jewelcost;

} else {
    *weight = jamweight;

    *cost = jewelcost; }

if (query == 0) {
    dataloc = weight;

} else {
    dataloc = cost; }

value = *dataloc;

print(*value);
```

WARNING to users of these ROIs. This may be a problematic case that you want to exclude from the itemset analysis – one liners exist in ROIs. –

Sometimes we want to keep track of two different conditional-memory-initialization regions. Two if/else conditional statements should only be in the same conditional-memory-initialization block if they write to the same set of locations in memory. However, in our actual analysis, we do not differentiate between conditional source 1 and conditional source 2 regions.

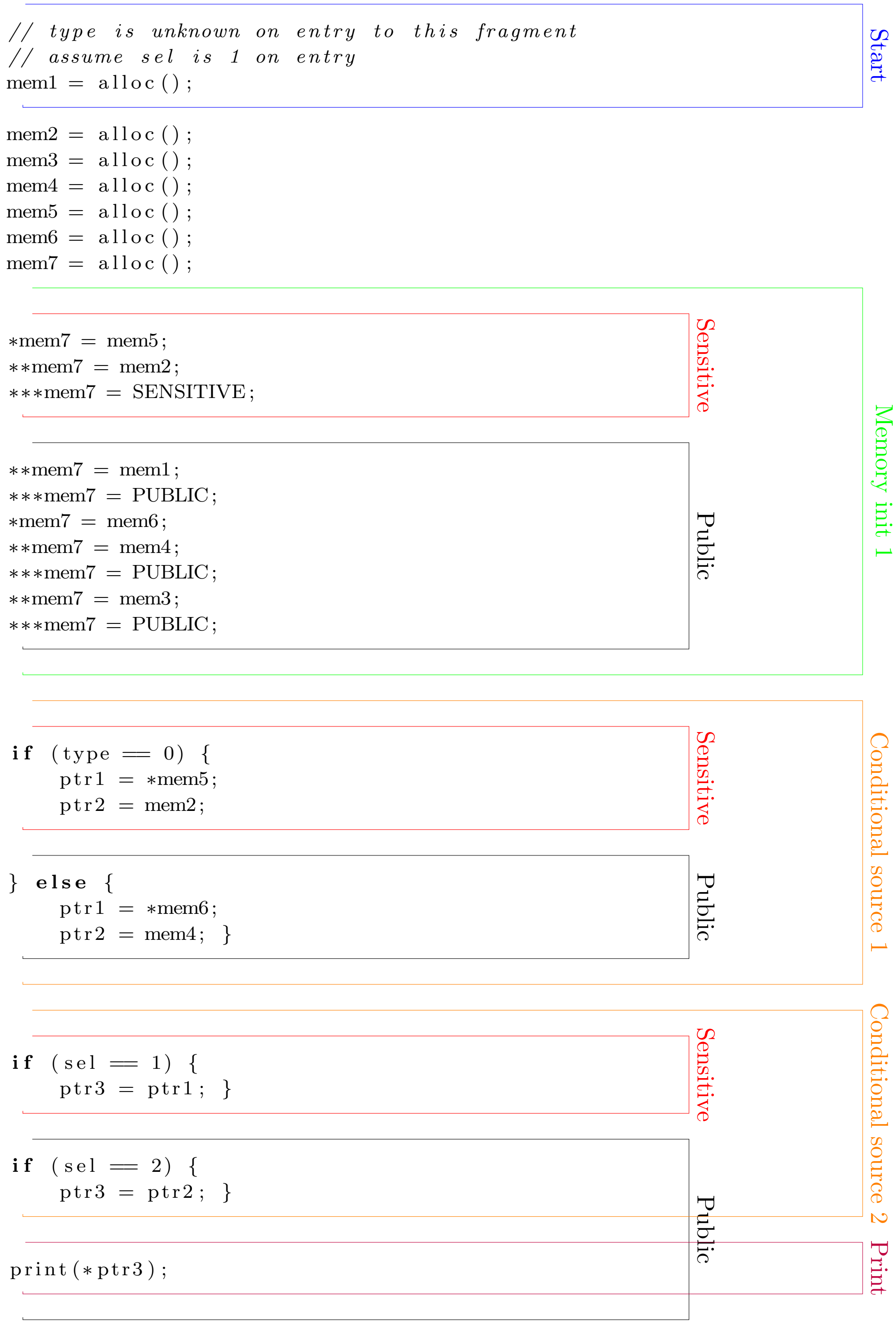
3.2.3 never-easy-field-code



```
// assume o is 2 on entry
f1 = alloc();

f2 = alloc();
b = alloc();

*b = f1;
**b = SENSITIVE;

*b = f2;
**b = PUBLIC;

if (o == 1) {
    *b = f1; }

if (o == 2) {
    *b = f2; }

res = *b;

print(*res);
```

This example is inspired by a base object that has two fields, the first of which contains SENSITIVE data and the second of which contains PUBLIC data. We use the base object to set the fields initially and to access them. We force flow to model accessing and printing the second field, which NEVER prints senstive data.

3.2.4 never-medium-field-code



```
// type is unknown on entry to this fragment
// assume sel is 1 on entry
mem1 = alloc();

mem2 = alloc();
mem3 = alloc();
mem4 = alloc();
mem5 = alloc();
mem6 = alloc();
mem7 = alloc();

*mem7 = mem5;
**mem7 = mem2;
***mem7 = SENSITIVE;

**mem7 = mem1;
***mem7 = PUBLIC;
*mem7 = mem6;
**mem7 = mem4;
***mem7 = PUBLIC;
**mem7 = mem3;
***mem7 = PUBLIC;

if (type == 0) {
    ptr1 = *mem5;
    ptr2 = mem2;

} else {
    ptr1 = *mem6;
    ptr2 = mem4; }

if (sel == 1) {
    ptr3 = ptr1; }

if (sel == 2) {
    ptr3 = ptr2; }

print(*ptr3);
```

This example is inspired by a base object that has two fields, each of which has two fields. We use the base object and intermediate objects to access all leaf fields, setting the first leaf field of the first object to PUBLIC, the second leaf field of the first object to SENSITIVE, and the leaf fields of the first object to PUBLIC. We select one of the two intermediate objects based on the unknown variable. We then the force the flow to model selecting the first field (here, ptr1) of the selected intermediate object. We then print that field. Since both first fields are PUBLIC, this program NEVER prints SENSITIVE data.

3.2.5 sometimes-easy-field-code

```
// offset is unknown on entry to this fragment
// assume evidence is 0 on entry
person = alloc();
```

Start

```
partner = alloc();
suspect = alloc();
```

```
if (evidence == 0) {
    *suspect = person;
} else {
    *suspect = partner; }
towrite = *suspect;
```

Conditional source 1

```
if (offset == 0) {
    *towrite = SENSITIVE;
```

Sensitive

```
} else {
    *towrite = PUBLIC; }
```

Public

Conditional source 2

```
print(*person);
```

Print

This is inspired by a base object that has two fields. We force the flow to model selecting the first field, write some SENSITIVE or PUBLIC data to that field depending on an unknown conditional, and then print the first field. Without knowing the value of the conditional, we cannot reason about this program. It SOMETIMES prints SENSITIVE.

The first conditional assignment cannot be definitively labeled as either Sensitive or Public.

3.2.6 sometimes-medium-field-code

```
// type is unknown on entry to this fragment
field_a1 = alloc();
```

Start

```
field_a2 = alloc();
field_b1 = alloc();
field_b2 = alloc();
object_a = alloc();
object_b = alloc();
base = alloc();
```

```
*base = object_a;
**base = field_a1;
***base = PUBLIC;
**base = field_a2;
***base = PUBLIC;
*base = object_b;
**base = field_b1;
***base = PUBLIC;
```

Public

```
**base = field_b2;
***base = SENSITIVE;
```

Sensitive

Memory init 1

```
if (type == 0) {
    object = object_a;
```

Public

```
} else {
    object = object_b; }
```

Sensitive

Conditional source 1

```
loc = *object;
```

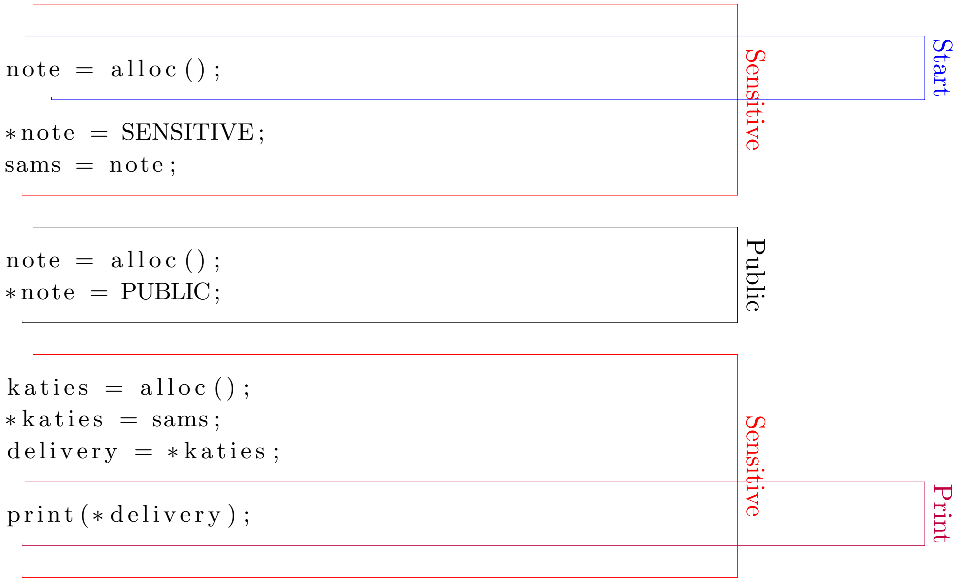
```
print(*loc)
```

Print

This problem is inspired by a base object that has two fields, each of which has two fields. We use the base object and intermediate objects to access all leaf fields, setting the leaf fields of the first object to PUBLIC, the first leaf field of the second object to PUBLIC, and the second leaf field of the second object to SENSITIVE. We model accessing the second field of whichever intermediate field object is selected, depending on an unknown conditional, and then print that leaf field. Without knowing the value of the conditional, we cannot reason about this program, and it MAY OR MAY NOT BE SAFE.

3.3 flow-sensitivity

3.3.1 always-easy-flow-code

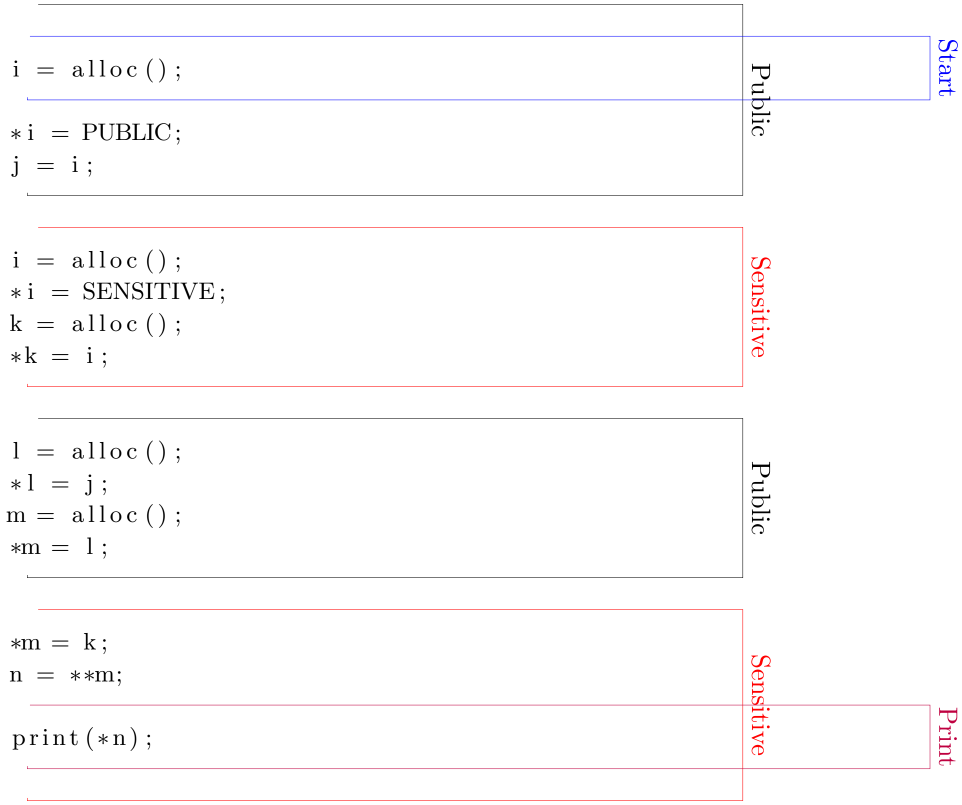


```
note = alloc();

*note = SENSITIVE;
sams = note;

note = alloc();
*note = PUBLIC;

katies = alloc();
*katies = sams;
delivery = *katies;

print(*delivery);
```

```
***** MODEL A *****
Variables
note: ->Mem1, ->Mem2
sams: ->Mem1, ->Mem2
katies: ->Mem3
delivery: ->Mem1, ->Mem2

Memory
Mem1: PUBLIC, SENSITIVE
Mem2: PUBLIC, SENSITIVE
Mem3: ->Mem1, ->Mem2
```

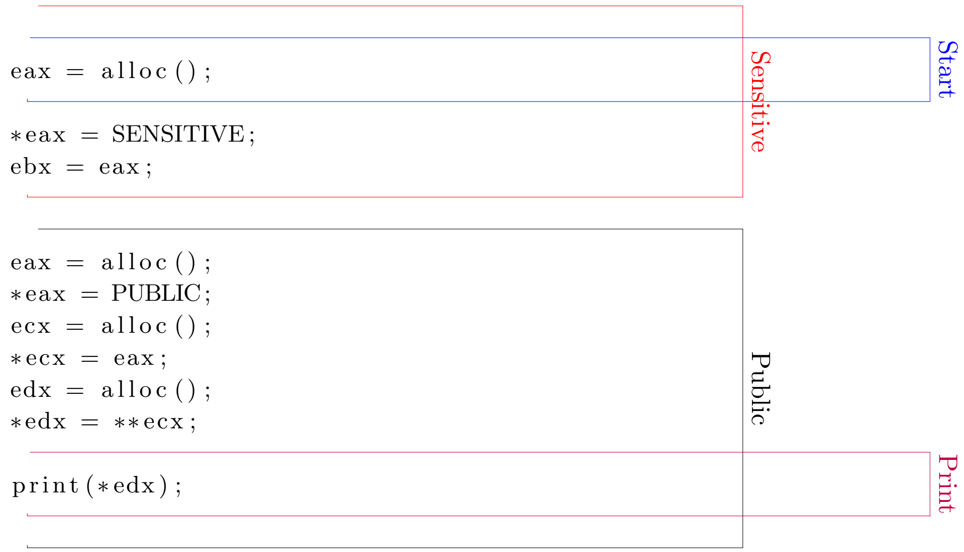
```
***** MODEL B *****
Variables
note: ->Mem1, ->Mem2
sams: ->Mem1
katies: ->Mem3
delivery: ->Mem1

Memory
Mem1: SENSITIVE
Mem2: PUBLIC
Mem3: ->Mem1
```

3.3.2 always-medium-flow-code



```
i = alloc();

*i = PUBLIC;
j = i;

i = alloc();
*i = SENSITIVE;
k = alloc();
*k = i;

l = alloc();
*l = j;
m = alloc();
*m = l;

*m = k;
n = **m;

print(*n);
```

3.3.3 never-easy-flow-code



```
eax = alloc();

*eax = SENSITIVE;
ebx = eax;

eax = alloc();
*eax = PUBLIC;
ecx = alloc();
*ecx = eax;
edx = alloc();
*edx = **ecx;

print(*edx);
```

3.3.4 never-medium-flow-code

```
dat_ptr = alloc();
```

Start

```
*dat_ptr = PUBLIC;
ptr_data1 = dat_ptr;
pptr_d1 = alloc();
*pptr_d1 = dat_ptr;
ppp_d1 = alloc();
*ppp_d1 = pptr_d1;
```

Public

```
dat_ptr = alloc();
*dat_ptr = SENSITIVE;
ptr_data2 = dat_ptr;
pptr_d2 = alloc();
*pptr_d2 = dat_ptr;
ppp_d2 = alloc();
*ppp_d2 = pptr_d2;
temp = ppp_d2;
```

Sensitive

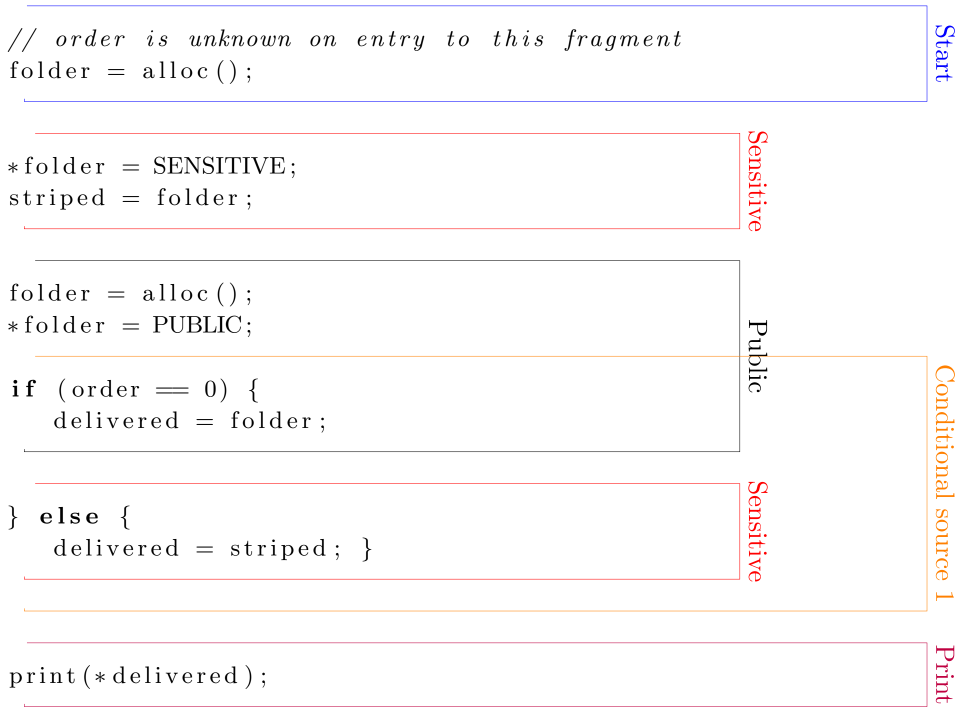
```
temp = ppp_d1;
answer = **temp;
```

```
print(*answer);
```

Print

Public

3.3.5 sometimes-easy-flow-code



```
// order is unknown on entry to this fragment
folder = alloc();

*folder = SENSITIVE;
striped = folder;

folder = alloc();
*folder = PUBLIC;

if (order == 0) {
   delivered = folder;

} else {
   delivered = striped; }

print(*delivered);
```

3.3.6 sometimes-medium-flow-code

```
// cond is unknown on entry to this fragment
r1 = alloc();
```

Start

```
r2 = r1;
*r1 = PUBLIC;
```

Public

```
r1 = alloc();
r3 = r1;
*r1 = SENSITIVE;
```

Sensitive

```
r1 = alloc();
```

```
 if (cond == 1) {
   *r1 = r3;
```

Sensitive

```
} else {
   *r1 = r2; }
```

Public

Conditional source 1

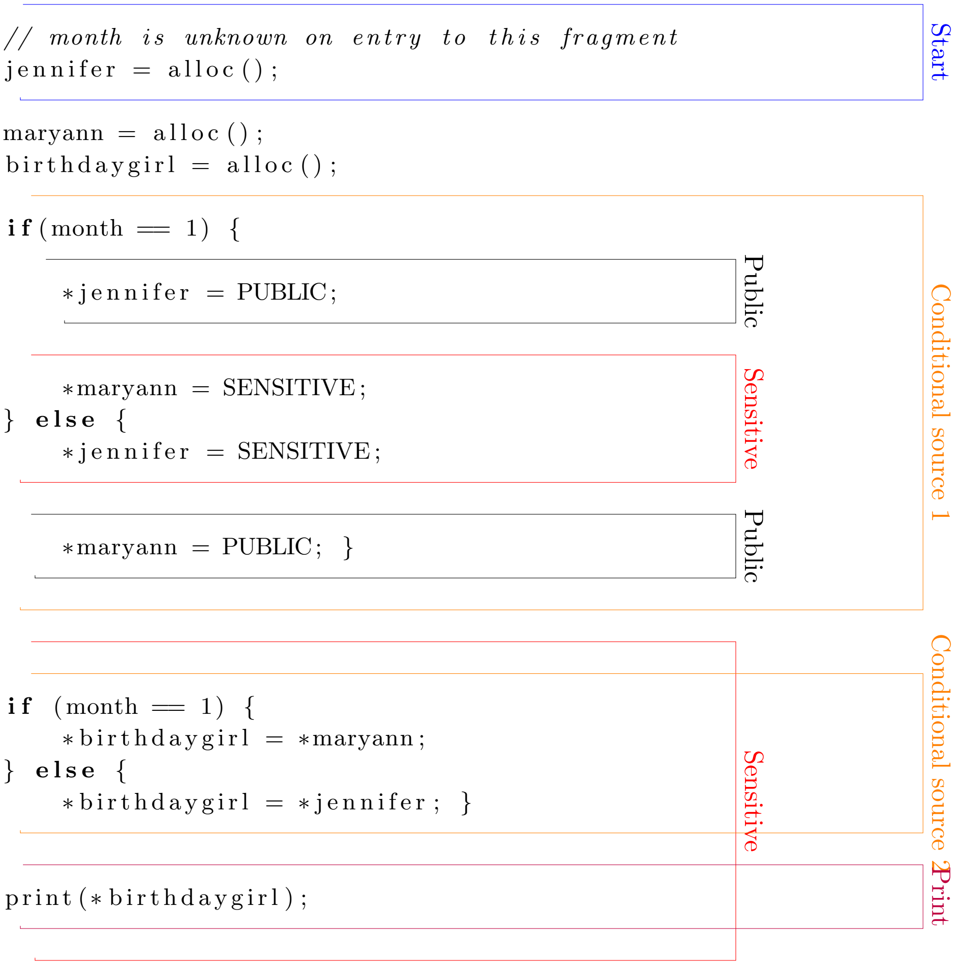
```
r4 = *r1;
```

```
print(*r4);
```

Print

3.4 path-sensitivity

3.4.1 always-easy-path-code



```
// month is unknown on entry to this fragment
jennifer = alloc();

maryann = alloc();
birthdaygirl = alloc();

if(month == 1) {

    *jennifer = PUBLIC;

    *maryann = SENSITIVE;
} else {
    *jennifer = SENSITIVE;

    *maryann = PUBLIC; }


if (month == 1) {
    *birthdaygirl = *maryann;
} else {
    *birthdaygirl = *jennifer; }

print(*birthdaygirl);
```

WARNING to users of these ROIs. This may be a problematic case that you want to exclude from the itemset analysis – one liners exist in ROIs. –

On all paths, the printed variable is only ever assigned a SENSITIVE value, thus this program ALWAYS prints SENSITIVE.

***** MODEL A *****

Variables

```
jennifer: ->Mem1
maryann: ->Mem2
birthdaygirl: ->Mem3
month: ?
```

Memory

```
Mem1: PUBLIC, SENSITIVE
Mem2: PUBLIC, SENSITIVE
Mem3: PUBLIC, SENSITIVE
```

***** MODEL B *****

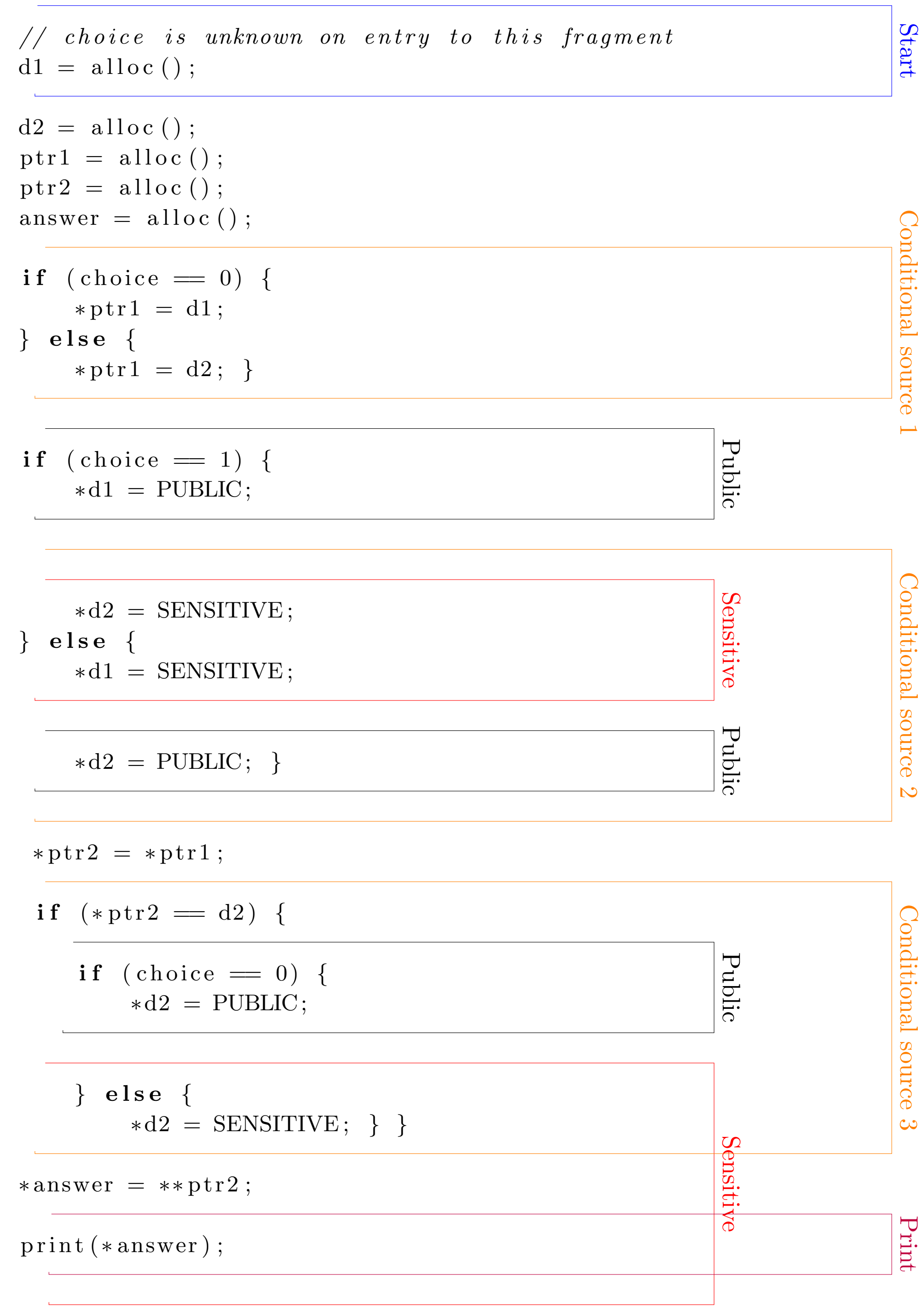
Variables

```
jennifer: ->Mem1
maryann: ->Mem2
birthdaygirl: ->Mem3
month: ?
```

Memory

```
Mem1: PUBLIC, SENSITIVE
Mem2: PUBLIC, SENSITIVE
Mem3: SENSITIVE
```

3.4.2 always-medium-path-code



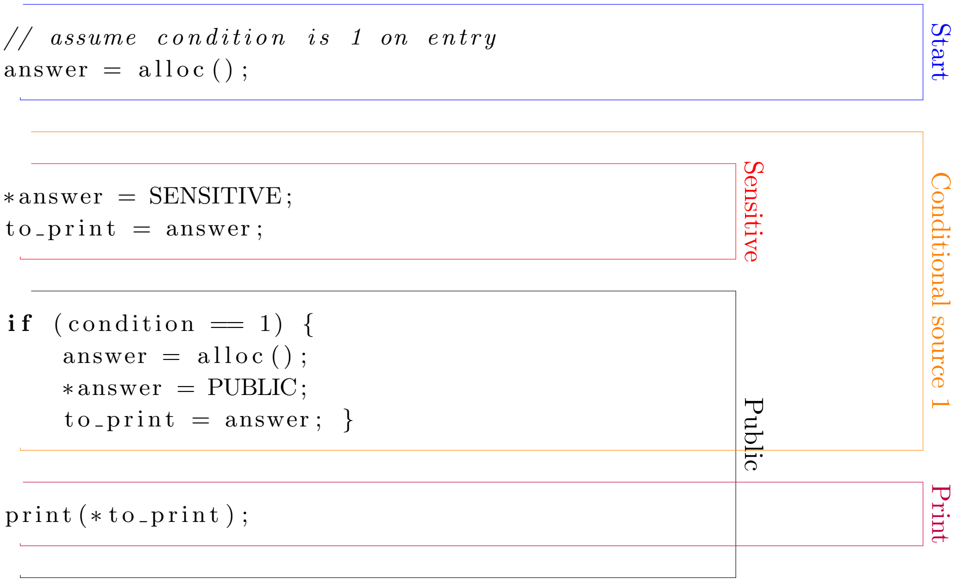
```
// choice is unknown on entry to this fragment
d1 = alloc();

d2 = alloc();
ptr1 = alloc();
ptr2 = alloc();
answer = alloc();

if (choice == 0) {
    *ptr1 = d1;
} else {
    *ptr1 = d2; }

if (choice == 1) {
    *d1 = PUBLIC;

    *d2 = SENSITIVE;
} else {
    *d1 = SENSITIVE;

    *d2 = PUBLIC; }

 *ptr2 = *ptr1;

 if (*ptr2 == d2) {

    if (choice == 0) {
        *d2 = PUBLIC;

    } else {
        *d2 = SENSITIVE; } }

*answer = **ptr2;

print(*answer);
```

On paths where choice is 0, ptr1 holds d1, d1 holds SENSITIVE, and the third if/else is not taken. On paths where choice is 1, ptr1 holds d2, d2 holds sensitive, and d2 is updated in the third if to SENSITIVE. On paths where choice is 2, ptr1 holds d2, d2 is set to PUBLIC and updated in the third if to SENSITIVE. Thus all paths result in SENSITIVE and the program ALWAYS prints SENSITIVE.

```
***** MODEL A *****
Variables
d1: ->Mem1
d2: ->Mem2
ptr1: ->Mem3
ptr2: ->Mem4
answer: ->Mem5
choice: ?

Memory
Mem1: PUBLIC, SENSITIVE
Mem2: PUBLIC, SENSITIVE
Mem3: ->Mem1, ->Mem2
Mem4: ->Mem1, ->Mem2
Mem5: PUBLIC, SENSITIVE
```

```
***** MODEL B *****
Variables
d1: ->Mem1
d2: ->Mem2
ptr1: ->Mem3
ptr2: ->Mem4
answer: ->Mem5
choice: ?

Memory
Mem1: PUBLIC, SENSITIVE
Mem2: PUBLIC, SENSITIVE
Mem3: ->Mem1, ->Mem2
Mem4: ->Mem1, ->Mem2
Mem5: SENSITIVE
```

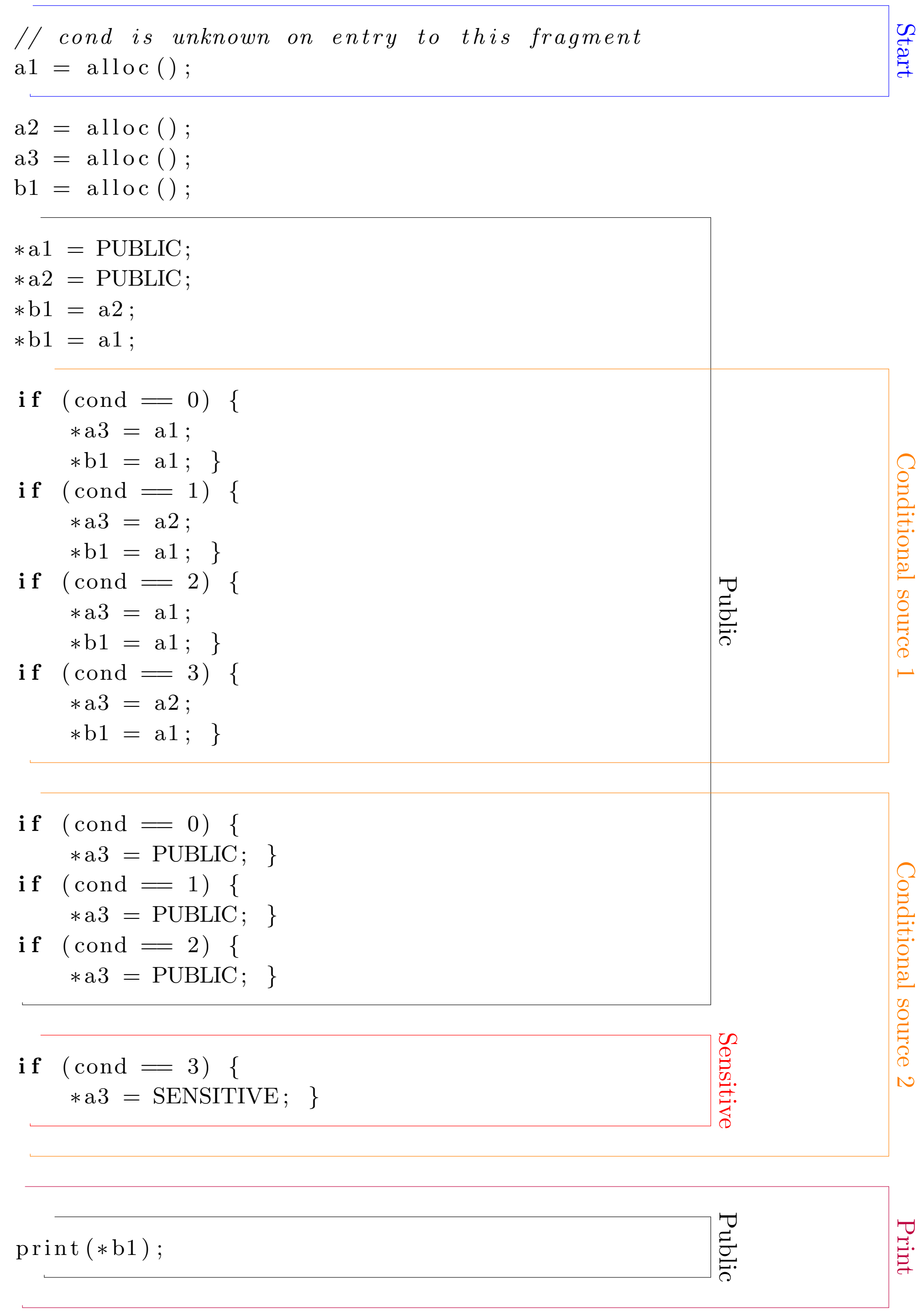
3.4.3 never-easy-path-code



```
// assume condition is 1 on entry
answer = alloc();

*answer = SENSITIVE;
to_print = answer;

if (condition == 1) {
    answer = alloc();
    *answer = PUBLIC;
    to_print = answer; }

print(*to_print);
```

The assignment to the printed variable only occurs on a safe path, whereon answer is set to public before to_print can be set. This program NEVER prints SENSITIVE.

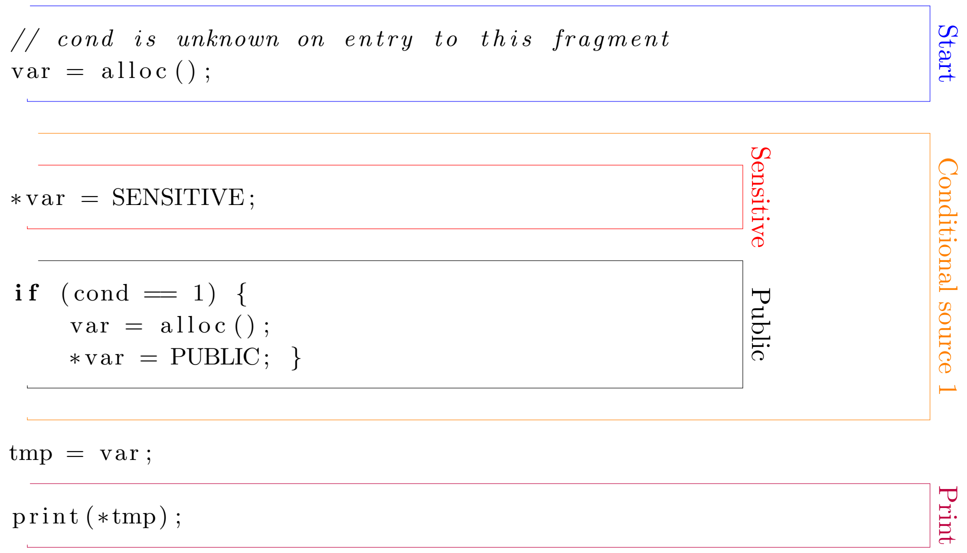
3.4.4 never-medium-path-code



```
// cond is unknown on entry to this fragment
a1 = alloc();

a2 = alloc();
a3 = alloc();
b1 = alloc();

*a1 = PUBLIC;
*a2 = PUBLIC;
*b1 = a2;
*b1 = a1;

if (cond == 0) {
    *a3 = a1;
    *b1 = a1; }
if (cond == 1) {
    *a3 = a2;
    *b1 = a1; }
if (cond == 2) {
    *a3 = a1;
    *b1 = a1; }
if (cond == 3) {
    *a3 = a2;
    *b1 = a1; }

if (cond == 0) {
    *a3 = PUBLIC; }
if (cond == 1) {
    *a3 = PUBLIC; }
if (cond == 2) {
    *a3 = PUBLIC; }

if (cond == 3) {
    *a3 = SENSITIVE; }

print(*b1);
```

This is inspired by reasoning about decision variables modulo 2 and specific values being set. Here, a1 is always set to a PUBLIC value and only a1 is printed.

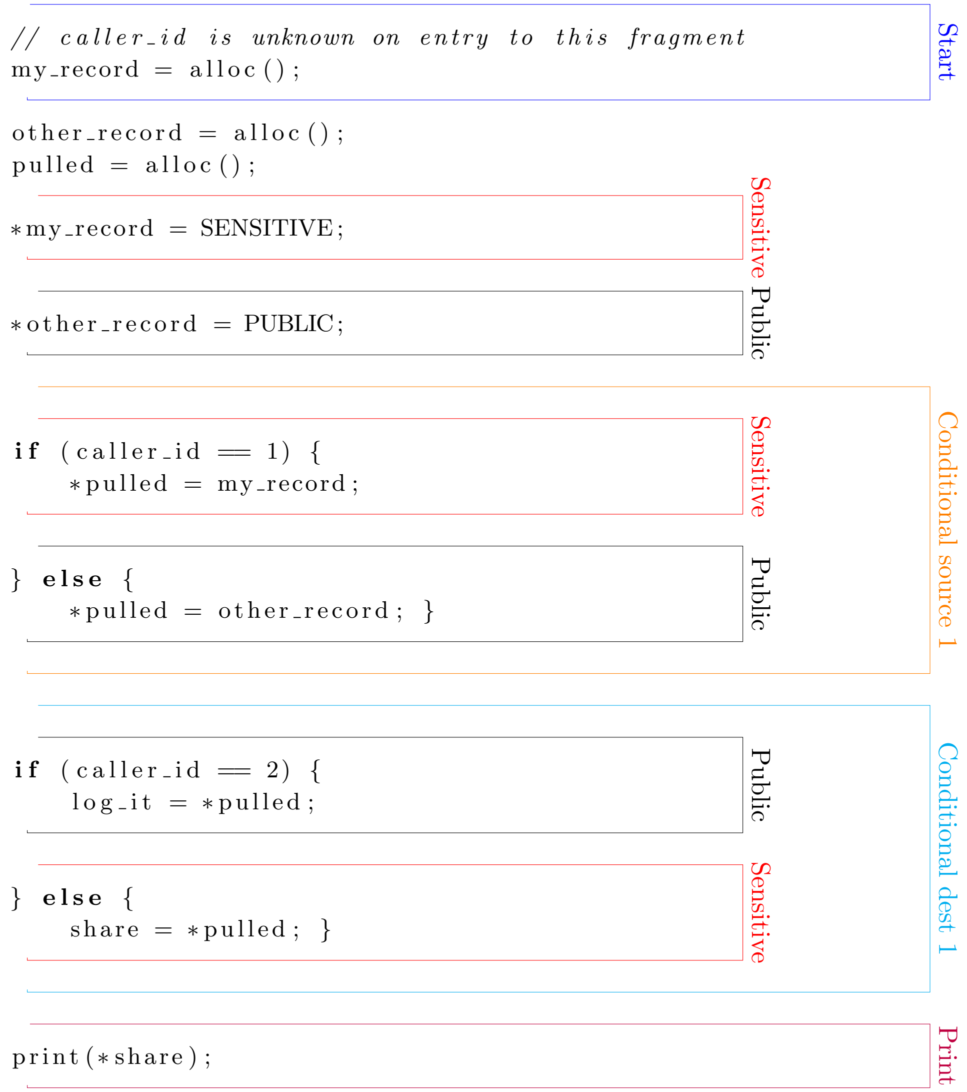
3.4.5 sometimes-easy-path-code



```
// cond is unknown on entry to this fragment
var = alloc();

*var = SENSITIVE;

if (cond == 1) {
    var = alloc();
    *var = PUBLIC; }

tmp = var;

print(*tmp);
```

The assignment to the printed variable occurs on both paths, so the program SOMETIMES prints SENSITIVE depending on cond.

3.4.6 sometimes-medium-path-code



```
// caller_id is unknown on entry to this fragment
my_record = alloc();

other_record = alloc();
pulled = alloc();

*my_record = SENSITIVE;

*other_record = PUBLIC;

if (caller_id == 1) {
    *pulled = my_record;

} else {
    *pulled = other_record; }

if (caller_id == 2) {
    log_it = *pulled;

} else {
    share = *pulled; }

print(*share);
```

WARNING to users of these ROIs. This may be a problematic case that you want to exclude from the itemset analysis – one liners exist in ROIs. –

This example is inspired by reasoning on paths where conditionals are related but different. If caller_id is 1, pulled points to the memory with SENSITIVE data. For all other values, it is called with PUBLIC data. If caller_id is 2, log_it holds a pointer to PUBLIC data. For all other values, share holds a pointer to the data. If caller_id is 1, share holds a pointer SENSITIVE data, but if anything else, share holds a a pointer to PUBLIC data. share is printed. Thus this problem SOMETIMES prints SENSITIVE data, depending on the value of caller_id.