Vulnerability Detection - Static Analysis

Holistic Software Security

Aravind Machiry

Credits: Prof. Mayur Naik, UPenn

What is it?

- Finding vulnerabilities in a given piece of software:
  - Software could be:
    - Binaries or
    - Source code or
    - Both.

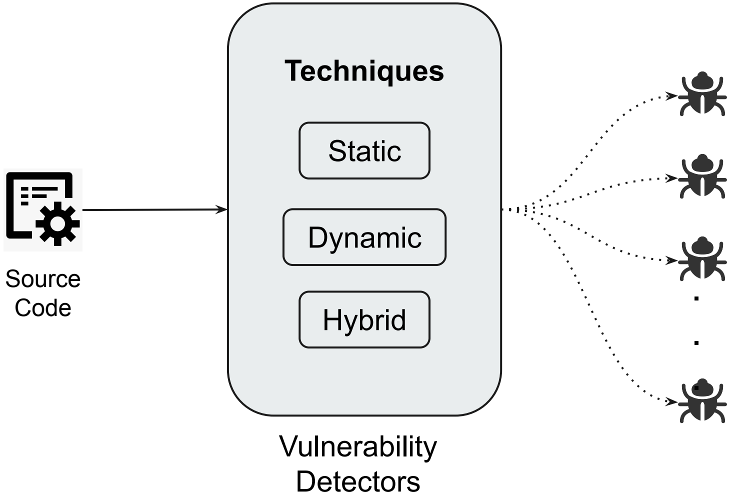
What is it?

- Finding vulnerabilities in a given piece of software:
    - Software could be:
        - Binaries or
        - Source code or
        - Both.

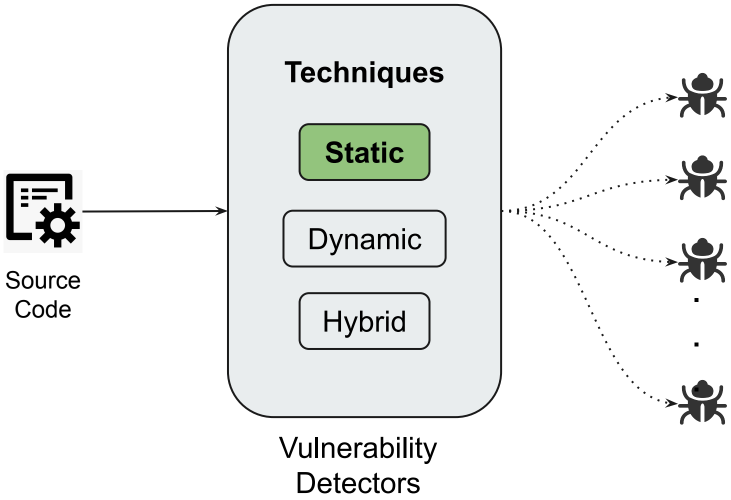
Our focus

Overview

Overview


Techniques
Static
Dynamic
Hybrid
Source
Code
Vulnerability
Detectors

Overview


Source
Code
Techniques
Static
Dynamic
Hybrid
Vulnerability
Detectors

Static

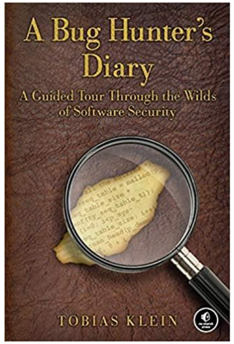
- Static w.r.t to the software being analysed:
  - We **do not run** (or dynamically execute) the program.
- Example:
  - `grep -r "sscanf[^)]*,[^)]*%s"`
  - To find: CWE-120 - Buffer Copy without Checking Size of Input ('Classic Buffer Overflow').

Static

- `grep -r "sscanf[^)]*,[^)]*%s"`

```
static char cs;
...
int ret = sscanf(buf, "%s", &cs);
if (ret != 1) {
   accdet_error("..");
   return -1;
}
...
```

CVE-2016-8472: In MediaTek Kernel Driver



Most successful technique

Static

- `grep -r "sscanf[^)]*,[^)]*%s" -> CVE-2016-8472`
    - Along with **2,300** other matches which **are not vulnerabilities (False positives)**.

```
...
char *ptr = “CMD 12”;
char buf[64]
...
sscanf(ptr, "%s", buf);
```

Maximum size could be 6 (less than 64 -> size of buf)

It becomes worse on complex codebases!

	CppCheck	flawfinder	RATS
Qualcomm	18	4,365	693
Samsung	22	8,173	2,244
Huawei	34	18,132	2,301
MediaTek	168	14,230	3,730
Total	**242**	**44,990**	**8,968**

Static

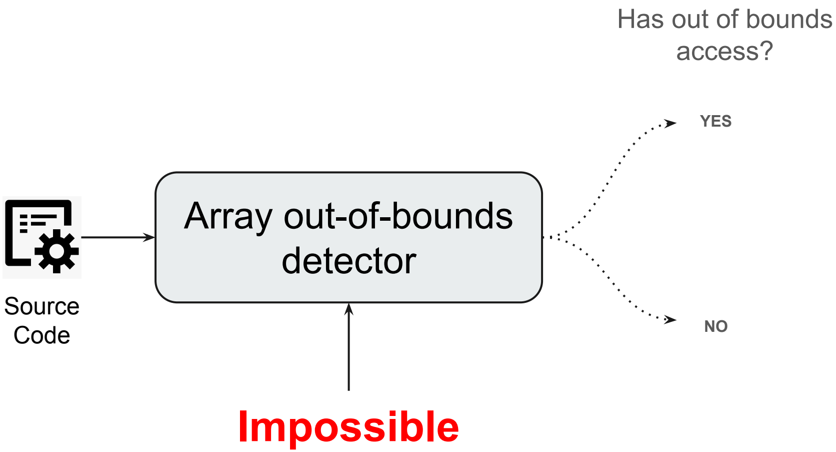
- How does a human find vulnerabilities?

```
void overflow() {
  char *out;
  int in = get_int();        1073741824
  if (in <= 0) { return; }             0
  out = malloc(in*sizeof(char*));
  for (i = 0; i < in; i++)
    out[i] = get_string();
}
```

Static

- How does a human find vulnerabilities?
    - Understands the program and tries to find if any vulnerable conditions are possible.
    - We need a way to analyze the given program or software:
        - Program Analysis -> **Static Program Analysis** or **Static Analysis**

But, computing program properties is undecidable!



But, computing program properties is undecidable!

```
void foo() {
  int a[2];
  M(X);
  a[3] = 0;
}
```

- Halting Problem: Impossible to say whether a program terminates.
- Proof by contradiction:
  - Yes -> Execution reaches a[3] i.e., program M(X) terminates.
  - No -> Execution does not reach a[3] i.e., program M(X) does not terminate.
- Contradiction: We can say if a program terminates.

Static analysis design choices for vulnerability detection

Impossible

True Result	Sound	Complete	Neither sound nor complete	Sound and Complete
Bug	Bug	May or May not be a bug.	May or May not be a bug.	Bug
Not a bug	May or May not be a bug.	Not a bug.	May or May not be a bug.	Not a bug
	false positives No false negatives	No false positives false negatives	false positives false negatives	No false positives No false negatives

Precision and Recall

		Analysis Outcome	
		Accept	Reject
Program's Ground Truth	Good	True Negative	False Positive
	Bad	False Negative	True Positive

$$precision = \frac{\text{\# True Positives}}{\text{\# Rejected}} \qquad recall = \frac{\text{\# True Positives}}{\text{\# Bad}}$$

Static analysis design choices for vulnerability detection

	Recall=1	Precision=1		
True Result	**Sound**	**Complete**	**Neither sound nor complete**	**Sound and Complete**
Bug	Bug	May or May not be a bug.	May or May not be a bug.	Bug
Not a bug	May or May not be a bug.	Not a bug.	May or May not be a bug.	Not a bug
	false positives No false negatives	No false positives false negatives	false positives false negatives	No false positives No false negatives

Sound Static Analysis

- Used to be the popular choice. Why?
  - Guarantees that all bugs will be found.
    - Over Approximation.
    - Caveat: False positives.
  - If a sound static analysis says, there are no bugs*, then we can be sure that the program does not have bugs.

* of specific type.

Sound Static Analysis

```
void foo(unsigned i) {
  int a[2];
 if (i < 2) a[i] = 0; //p3
 else a[i] = 1; //p4
}

int main() {
  unsigned i, j;
  scanf("%u %u", &i, &j);
  if (i < 2) foo(i); //p1
  foo(j); //p2
  return 0;
}
```

Consider the following out-of-bounds detectors with the following warnings at corresponding lines:

- SA1: P1, P2, P3, P4
- SA2: P3 and P4
- SA3: P4
- SA4: P4 only when called from P2

Are these analyses sound?

Sound Static Analysis

```
void foo(unsigned i) {
  int a[2];
 if (i < 2) a[i] = 0; //p3
 else a[i] = 1; //p4
}

int main() {
  unsigned i, j;
  scanf(“%u %u”, &i, &j);
  if (i < 2) foo(i); //p1
  foo(j); //p2
  return 0;
}
```

Consider the following out-of-bounds detectors with the following warnings at corresponding lines:

- SA1: P1, P2, P3, P4
- SA2: P3 and P4
- SA3: P4
- SA4: P4 only when called from P2

Are these analyses sound?

What about precision?

Designing a Sound Static Analysis

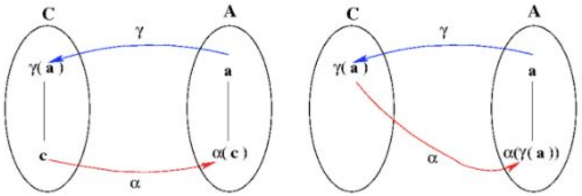
- Guaranteed Termination: Should finish in reasonable time.
- **Over Approximate** program behavior.

Abstract Interpretation

- Interpret the program over abstract states.
- Abstract semantics:
  - How to interpret operations over abstract values.
- Guaranteed Termination (Kleene fixed-point theorem):
  - Galois Connection.
  - Monotonic Transfer functions:
    - The state computed at a program point should never decrease.

Abstract Interpretation

- Galois Connection:
    - Abstraction function (α) -> Maps a set of concrete values to abstract value.
    - Concretization function (γ) -> Maps an abstract value to set of concrete values.
    - **1. $\alpha(c) \leq a \Longleftrightarrow c \leq \gamma(a)$**
    - **2. $\alpha(\gamma(a)) \leq a$**



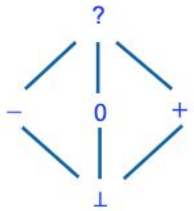
Relationship 1: abstracting followed by concretizing

Relationship 2: concretizing followed by abstracting

Sign Abstract Domain

To handle properties related to integers.

Abstract Values: {-, 0, +, ⊥, ?}



$$\alpha(S) = \begin{cases} 0 & \text{if all elements of S are 0} \\ + & \text{if all elements of S are positive} \\ - & \text{if all elements of S are negative} \\ ? & \text{otherwise} \end{cases} \qquad \gamma(S) = \begin{cases} \{0\} & \text{if } S = 0 \\ \{\text{pos int}\} & \text{if } S = + \\ \{\text{neg int}\} & \text{if } S = - \\ \{0 \text{ pos neg}\} & \text{if } S = ? \end{cases}$$

ADD	-	0	+	?
-	-	-	?	?
0	-	0	+	?
+	?	+	+	?
?	?	?	?	?

MULT	-	0	+	?
-	+	0	-	?
0	0	0	0	0
+	-	0	+	?
?	?	0	?	?

Divide by Zero Detector

- We do not care about absolute values of integers.
- We **just need to know if a number can be 0 or not**.
- Sign abstract domain provides a decent choice.
- Possible values for numbers: {-, 0, +, ⊥, ?}

```
void main() {
   ...
   if (x > 0) {
      ...1/x.. // x:+
   }
   ...2/x.. // x: ?
}
```

numRequests: ?

```
int averageResponseTime(int totalTime, int numRequests) {
  return totalTime / numRequests;
}
```

CVE-2019-14498

A divide-by-zero error exists in VLC media player that can be exploited by a crafted audio file

Data flow analysis

- Most vulnerabilities need reasoning of the flow of data through the program.
    - E.g., user input used as an index into an array => User data flows into index of an array.

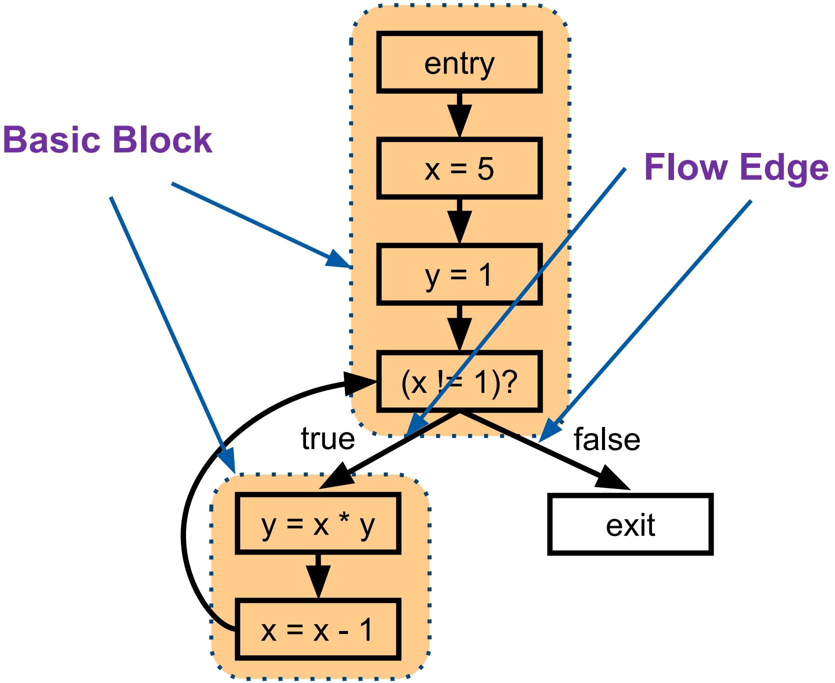
- Reasoning about flow of data in programs.

- Different kinds of data: constants, expressions, taint, etc.

Data flow concepts

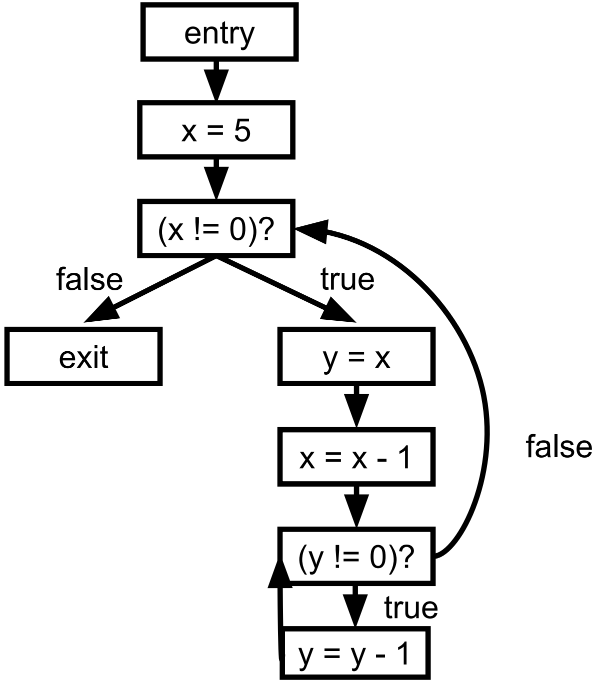
- Control flow graph (CFG):
    - Represents possible control flows with in the function.
    - Graph of basic blocks.
    - **Basic block:** Sequence of instructions always executed in the order.
    - **Edges** -> Flow of control.

Control flow graph (CFG)




```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```

Control flow graph (CFG)




```
x = 5;
while (x != 0) {
  y = x;
  x = x - 1;
  while (y != 0) {
    y = y - 1
  }
}
```

Classic Dataflow Analyses -> Primarily used in compiler optimization

Reaching Definitions Analysis

- Find uninitialized variable uses

Very Busy Expressions Analysis

- Reduce code size

Available Expressions Analysis

- Avoid recomputing expressions

Live Variables Analysis

- Allocate registers efficiently

Security related Dataflow Analyses

Interval Analysis

- Check memory safety
  (integer overflows, buffer overruns, …)

Type-State Analysis

- Temporal safety properties
  (APIs of protocols, libraries, …)

Taint Analysis

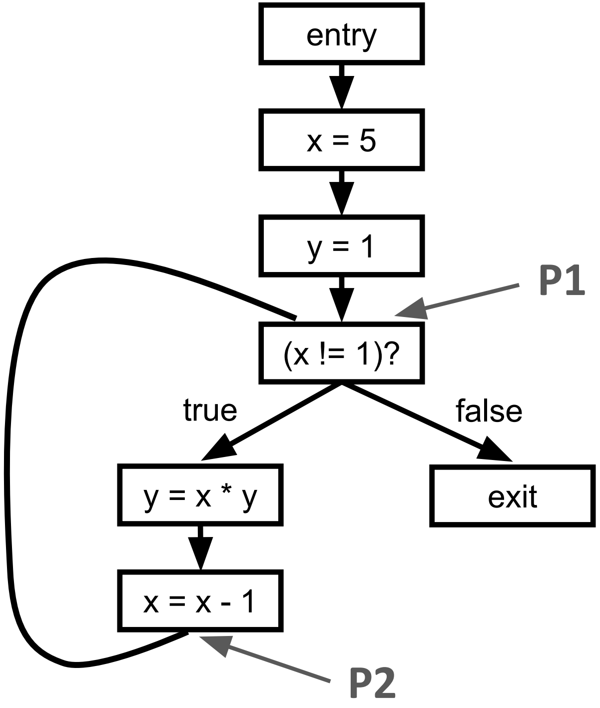
- Check information flow
  (Sensitive data leak, code injection, …)

Concurrency Analysis

- Concurrency safety properties
  (dataraces, deadlocks, …)

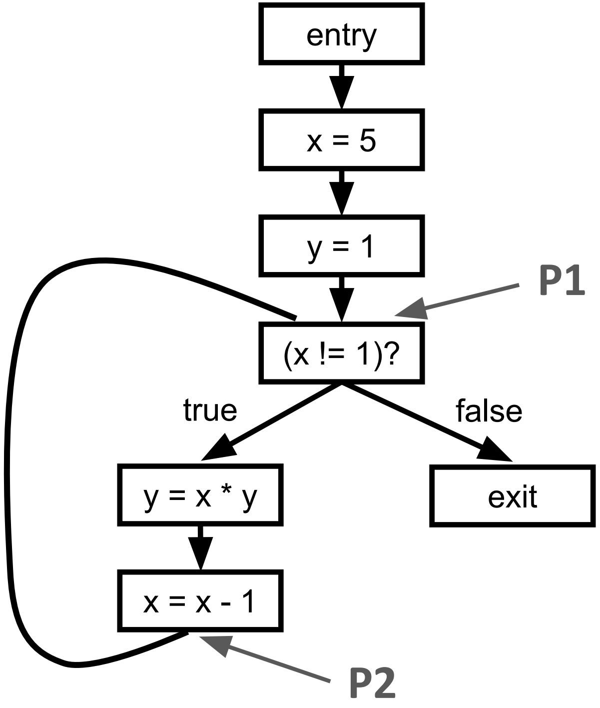
Reaching Definition Analysis

Determine, for each program point, which assignments (definitions) have been made and not overwritten, when execution reaches that point along some path.



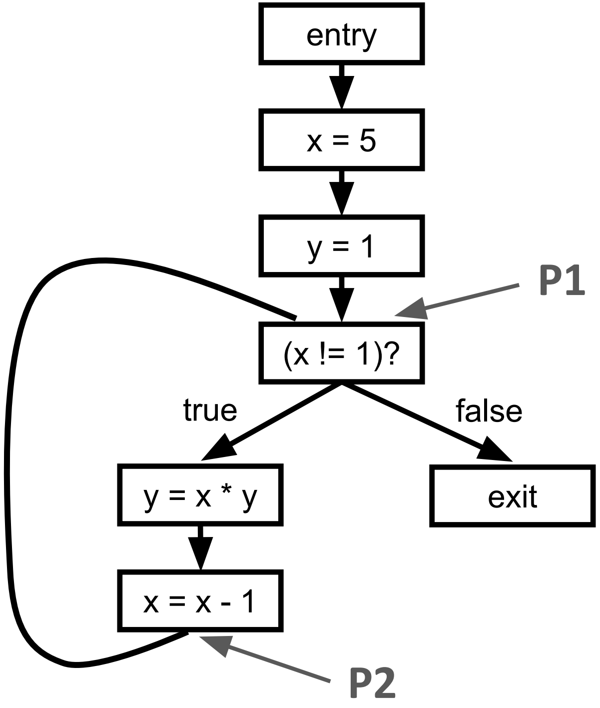
Reaching Definition Analysis

1. The assignment y = 1 reaches P1
1. The assignment y = 1 reaches P2
1. The assignment y = x * y reaches P1



Reaching Definition Analysis

1. The assignment y = 1 reaches P1
1. The assignment y = 1 reaches P2
1. The assignment y = x * y reaches P1



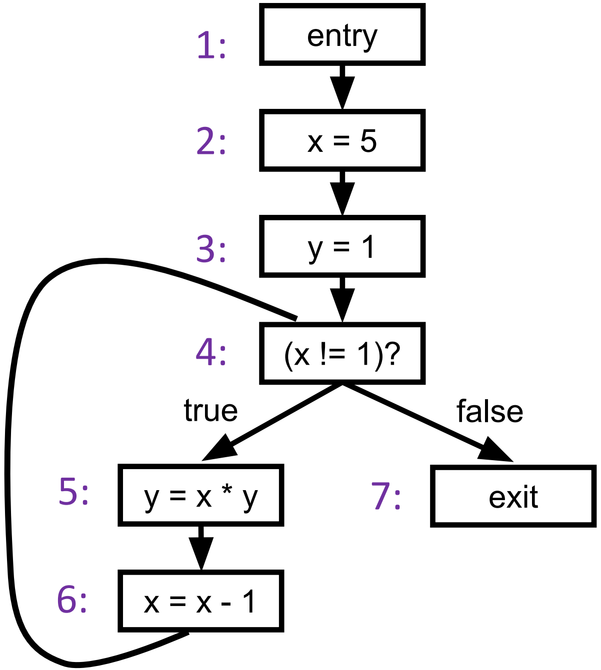
Reaching Definition Analysis

- Result: Set of definitions at each program point

- A definition is a pair of the form:

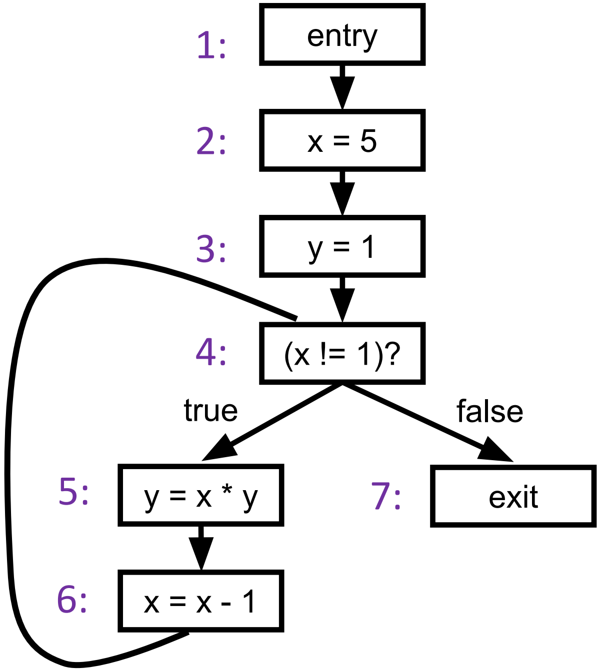
  <defined variable name, defining node label>

- Examples: <x,2> , <y,5>



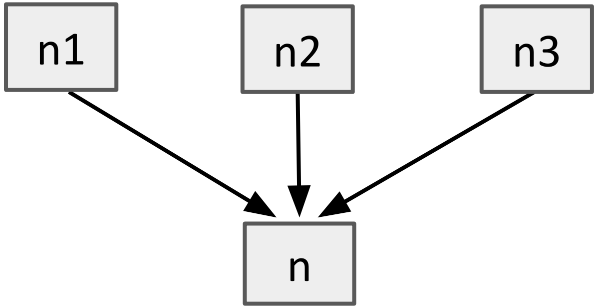
Reaching Definition Analysis

- Give a distinct label n to each node
- IN[n] = set of facts at entry of node n
- OUT[n] = set of facts at exit of node n
- Dataflow analysis computes IN[n] and OUT[n] for each node
- Repeat two operations until IN[n] and OUT[n] stop changing
    - Called “saturated” or “fixed point”



Reaching Definition Analysis: Computing IN

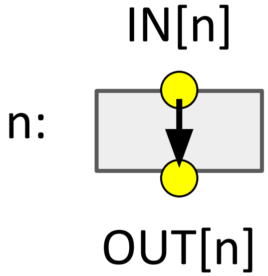
$$IN[n] = \bigcup_{n' \in \text{predecessors}(n)} OUT[n']$$



$$IN[n] = OUT[n1] \cup OUT[n2] \cup OUT[n3]$$

Reaching Definition Analysis: Computing OUT

OUT[n] = (IN[n] - KILL[n]) ∪ GEN[n]



n: `b ?` GEN[n] = ∅ KILL[n] = ∅

n: `x = a` GEN[n] = { <x, n> }

KILL[n] = { <x, m> : m != n }

Overall algorithm: Chaotic Iteration

for (each node n):

IN[n] = OUT[n] = $\varnothing$

OUT[entry] = { <v, ?> : v is a program variable }

repeat:

for (each node n):

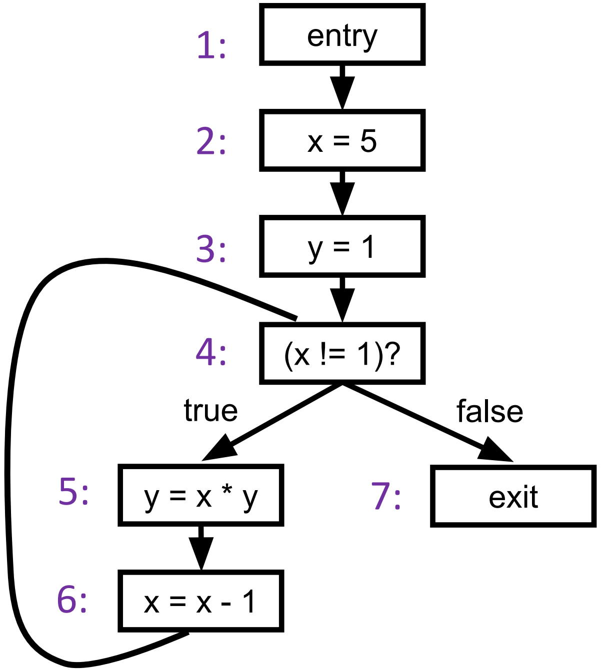
$$\mathrm{IN}[n] = \bigcup_{n' \in \mathrm{predecessors}(n)} \mathrm{OUT}[n']$$

OUT[n] = (IN[n] - KILL[n]) ∪ GEN[n]

until IN[n] and OUT[n] stop changing for all n

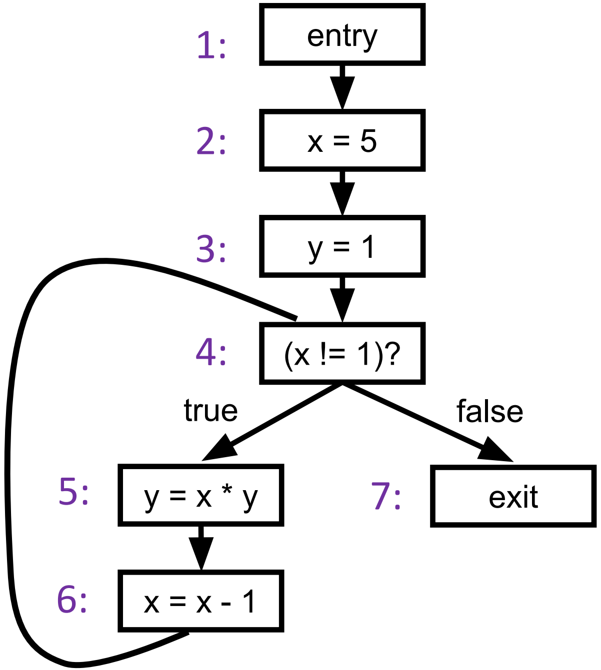
Reaching Definition Analysis: Example

n	IN[n]	OUT[n]
1	--	{<x,?>,<y,?>}
2	∅	∅
3	∅	∅
4	∅	∅
5	∅	∅
6	∅	∅
7	∅	--



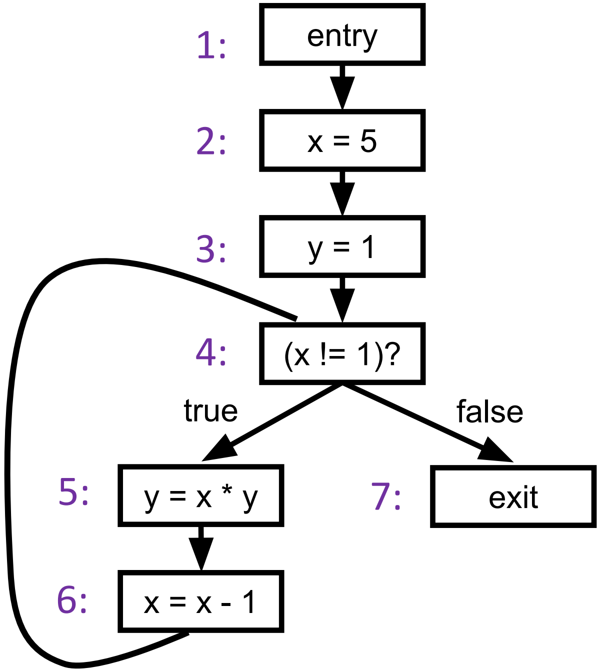
Reaching Definition Analysis: Example

n	IN[n]	OUT[n]
1	--	{<x,?>,<y,?>}
2	{<x,?>,<y,?>}	{<x,2>,<y,?>}
3	{<x,2>,<y,?>}	{<x,2>,<y,3>}
4	∅	∅
5	∅	∅
6	∅	∅
7	∅	--



Reaching Definition Analysis: Example

n	IN[n]	OUT[n]
1	--	{<x,?>,<y,?>}
2	{<x,?>,<y,?>}	{<x,2>,<y,?>}
3	{<x,2>,<y,?>}	{<x,2>,<y,3>}
4	{<x,2>,<y,3>,<y,5>,<x,6>}	{<x,2>,<y,3>,<y,5>,<x,6>}
5	{<x,2>,<y,3>,<y,5>,<x,6>}	{<x,2>,<y,5>,<x,6>}
6	{<x,2>,<y,5>,<x,6>}	{<y,5>,<x,6>}
7	{<x,2>,<y,3>,<y,5>,<x,6>}	--



Reaching Definition Analysis: Abstract Domain

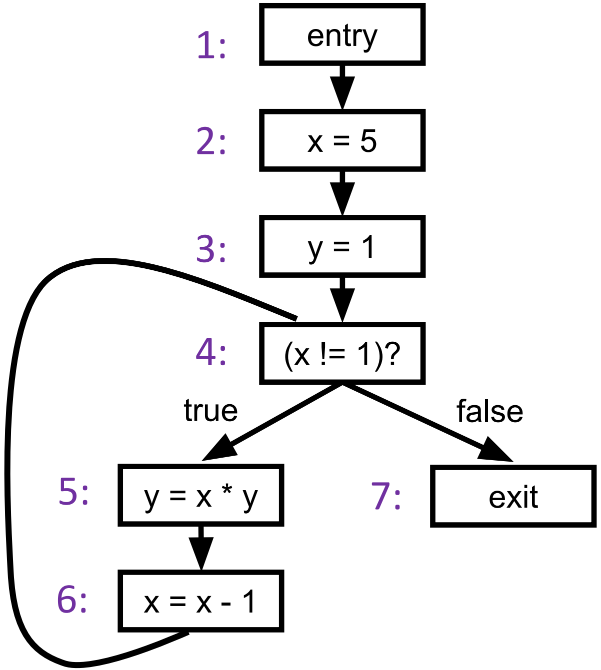
- Any combination of the definitions <x, 2>, <y,3>, <y,5>, <y,6> may reach a particular program point

- So, each combination of definitions is an abstract value

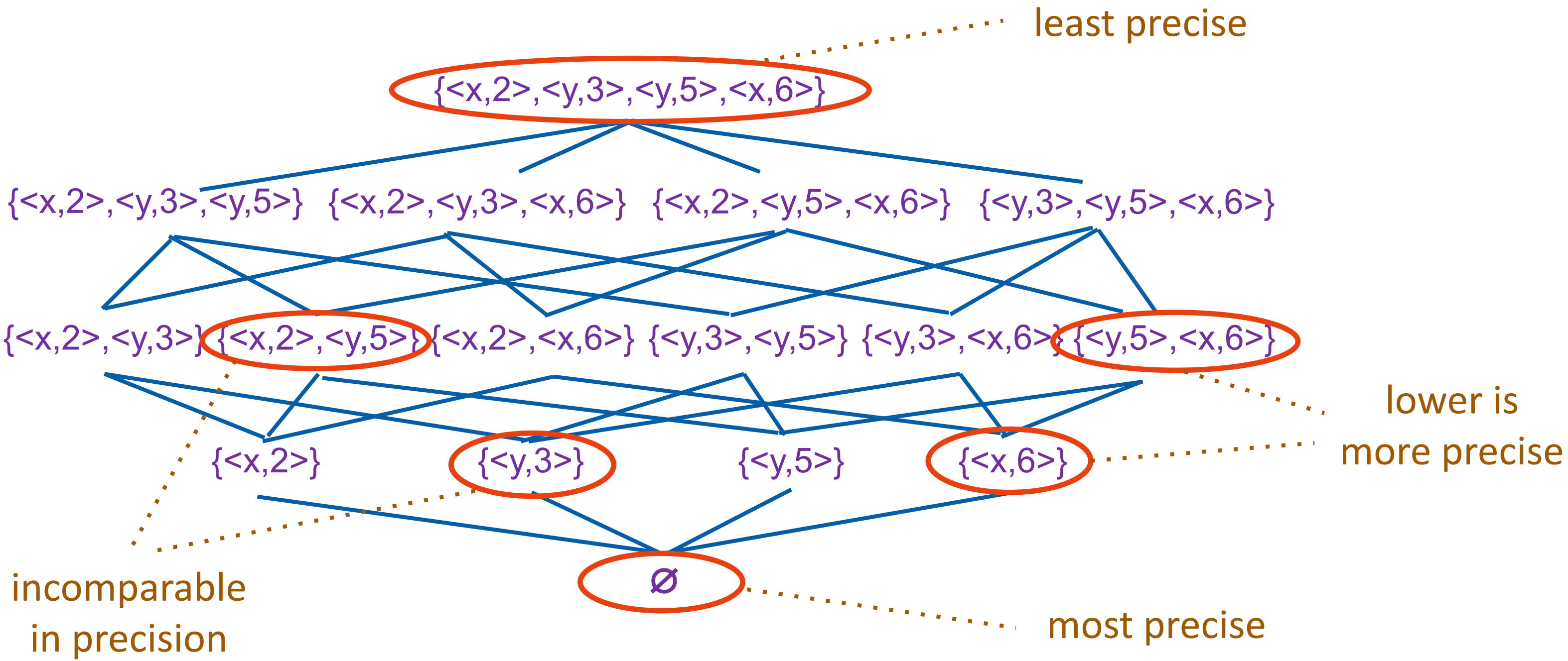
- Abstract domain is:

  set inclusion

  $$\langle\, 2^{\{<x,2>,\ <y,3>,\ <y,5>,\ <y,6>\}},\ \subseteq \,\rangle$$



Reaching Definition Analysis: Abstract Domain



Galois Connection and Termination

Abstract and Concrete domain form Galois connection.

The Chaotic Iteration algorithm always terminates!

- The two operations of reaching definitions analysis are "monotonic"

  => IN and OUT sets never shrink, only grow

- Largest they can be is set of all definitions in program, which is finite

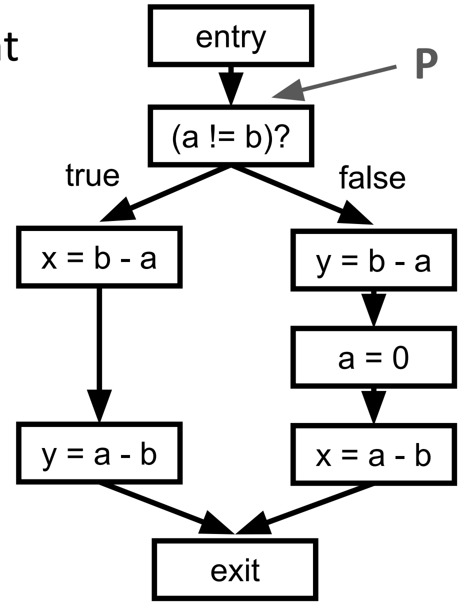
  => IN and OUT cannot grow forever

=> IN and OUT will stop changing after some iteration

Very Busy Expression Analysis

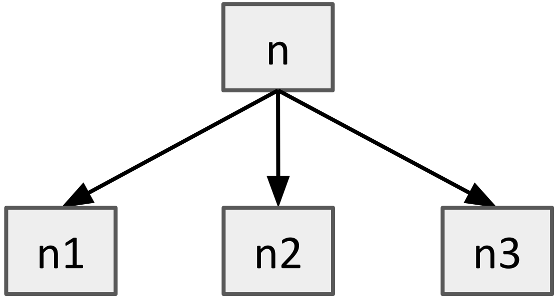
Goal: Determine very busy expressions at each program point

An expression is very busy if, no matter what path is taken, the expression is used before any of the variables occurring in it are redefined



Very Busy Expression Analysis: Computing OUT

$$OUT[n] = \bigcap_{n' \in successors(n)} IN[n']$$

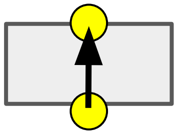


$$OUT[n] = IN[n1] \cap IN[n2] \cap IN[n3]$$

Very Busy Expression Analysis: Computing IN

$$IN[n] = (OUT[n] - KILL[n]) \cup GEN[n]$$

IN[n]

n: 

OUT[n]

n: `b ?`

$GEN[n] = \varnothing$ $KILL[n] = \varnothing$

n: `x = a`

$GEN[n] = \{ a \}$

$KILL[n] = \{ \text{expression } e : e \text{ contains } x \}$

Overall algorithm: Chaotic Iteration (again!)

for (each node n):

  IN[n] = OUT[n] = set of all expressions in program

IN[exit] = $\emptyset$

repeat:

  for (each node n):

$$OUT[n] = \bigcap_{n' \in successors(n)} IN[n']$$

$$IN[n] = (OUT[n] - KILL[n]) \cup GEN[n]$$

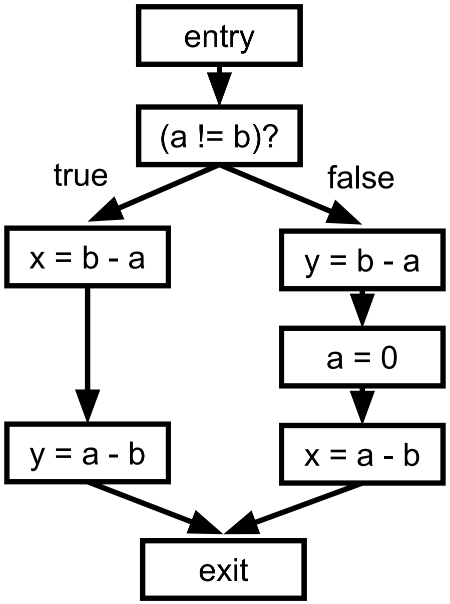
until IN[n] and OUT[n] stop changing for all n

Very Busy Expression Analysis: Abstract Domain

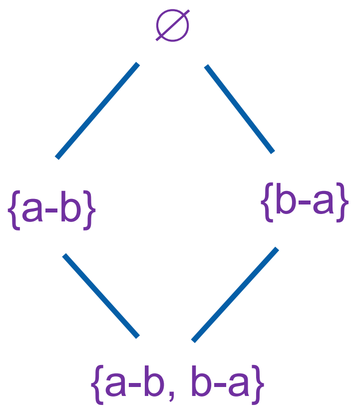
- Expressions a-b, b-a may independently be "very busy" at a particular program point
- So, each combination of these expressions is an abstract value
- Abstract domain is:

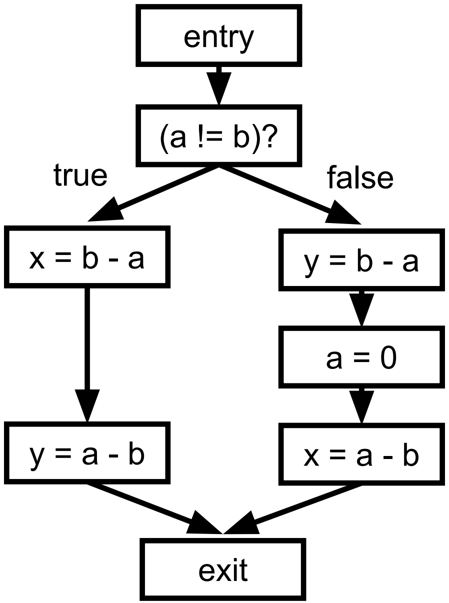
reverse set inclusion

$$\langle\, 2^{\{\,b\text{-}a,\ a\text{-}b\,\}},\ \supseteq \,\rangle$$



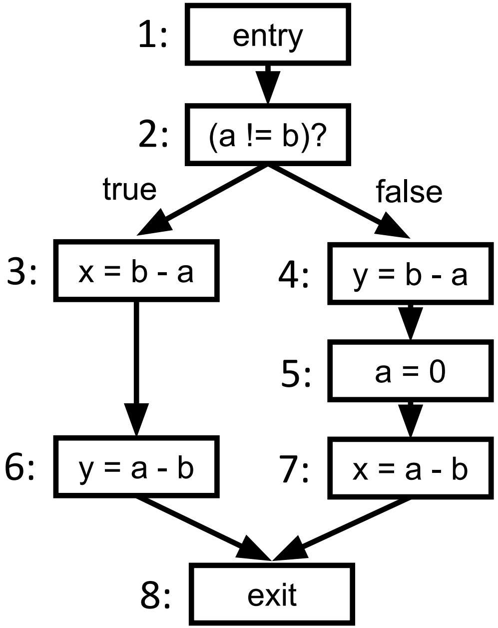
Very Busy Expression Analysis: Abstract Domain





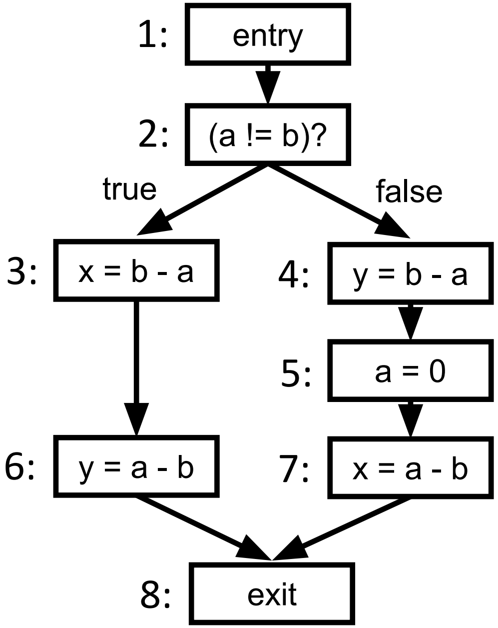
Very Busy Expression Analysis: Example

n	IN[n]	OUT[n]
1	--	{ b-a, a-b }
2	{ b-a, a-b }	{ b-a, a-b }
3	{ b-a, a-b }	{ b-a, a-b }
4	{ b-a, a-b }	{ b-a, a-b }
5	{ b-a, a-b }	{ b-a, a-b }
6	{ b-a, a-b }	{ b-a, a-b }
7	{ b-a, a-b }	{ b-a, a-b }
8	∅	--



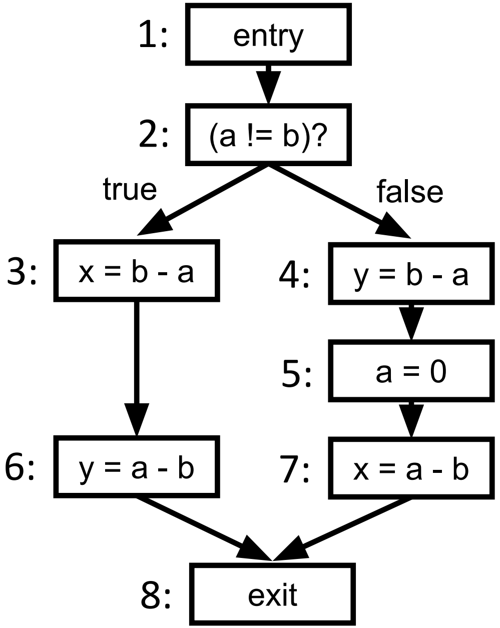
Very Busy Expression Analysis: Example

n	IN[n]	OUT[n]
1	--	{ b-a, a-b }
2	{ b-a, a-b }	{ b-a, a-b }
3	{ b-a, a-b }	{ b-a, a-b }
4	{ b-a, a-b }	{ b-a, a-b }
5	{ b-a, a-b }	{ b-a, a-b }
6	{ a-b }	∅
7	{ a-b }	∅
8	∅	--



Very Busy Expression Analysis: Example

n	IN[n]	OUT[n]
1	--	{ b-a }
2	{ b-a }	{ b-a }
3	{ b-a, a-b }	{ a-b }
4	{ b-a }	∅
5	∅	{ a-b }
6	{ a-b }	∅
7	{ a-b }	∅
8	∅	--



Overall Pattern of Dataflow Analysis

$$\color{blue}{\boxed{\quad\quad}}\color{black}{[n] = \left(\color{red}{\boxed{\quad\quad}}\color{black}{[n] - \mathrm{KILL}[n]\right) \cup \mathrm{GEN}[n]}}$$

$$\color{red}{\boxed{\quad\quad}}\color{black}{[n] = \color{cyan}{\boxed{\quad\quad}}\;\color{blue}{\boxed{\quad\quad}}\color{black}{[n']}}$$

$$n' \in \boxed{\quad\quad}(n)$$

$\color{blue}{\boxed{\quad\quad}}$ $\color{red}{\boxed{\quad\quad}}$ = IN or OUT

$\color{cyan}{\boxed{\quad\quad}}$ = ∪ (may) or ∩ (must)

$\boxed{\quad\quad}$ = predecessors or successors

Reaching Definition Analysis

$$\boxed{\text{OUT}}[n] = (\boxed{\text{IN}}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\boxed{\text{IN}}[n] = \underset{n' \in \boxed{\text{preds}}(n)}{\boxed{\bigcup}} \boxed{\text{OUT}}[n']$$

$\boxed{}$ $\boxed{}$ = IN or OUT

$\boxed{}$ = ∪ (may) or ∩ (must)

$\boxed{}$ = predecessors or successors

Very Busy Expression Analysis

$$\boxed{\text{IN}}\,[n] = \left(\boxed{\text{OUT}}\,[n] - \text{KILL}[n]\right) \cup \text{GEN}[n]$$

$$\boxed{\text{OUT}}\,[n] = \underset{n' \in \boxed{\text{succs}}(n)}{\boxed{\;\cap\;}}\ \boxed{\text{IN}}\,[n']$$

$\boxed{\quad}$ $\boxed{\quad}$ = IN or OUT

$\boxed{\quad}$ = ∪ (may) or ∩ (must)

$\boxed{\quad}$ = predecessors or successors

Type of analysis

Forward: Predecessors

Backward: Successors

May: Join (e.g., union)

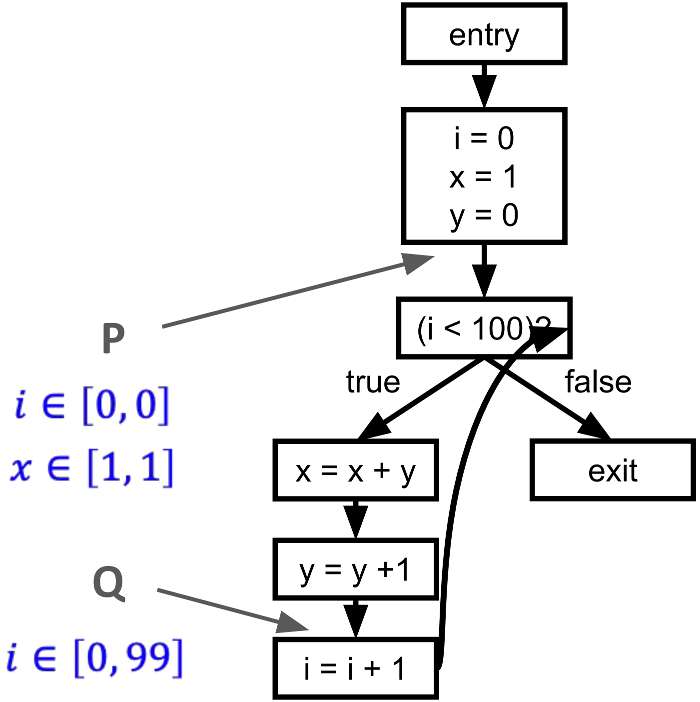
Must: Meet (e.g., intersection)

	May	Must
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

Interval Analysis

Goal: Determine, for each integer variable at each program point, a lower bound and an upper bound on its possible values at that point.

Improving Integer Security for Systems with KINT [OSDI 12]



Uses of Interval Analysis: Integer overflow detection

$in \in [1, +\infty]$

$\cancel{in \in [-\infty, +\infty]}$

```
void overflow() {
  char *out;
  int in = get_int();
  if (in <= 0) { return; }
  out = malloc(in*sizeof(char*));
  for (i = 0; i < in; i++)
    out[i] = get_string();
}
```

1073741824

0

CVE-2019-3855

In LibSSH, an attacker can exploit to execute code on the client system when a user connects to the server

CVE-2019-8099

In Adobe Acrobat, an attacker can use to steal information

Uses of Interval Analysis: Out of bounds access

index $\in [0, 3]$

~~index $\in [-\infty, +\infty]$~~

```
int main () {
  char *items[] = {"boat", "car", "truck", "train"};
  int index = get_int();
  if (index < 0 || index > 3) { return; }
  printf("You selected %s\n", items[index]);
}
```

Uses of Interval Analysis: Divide by zero detection

`numRequests` $\in [-\infty, +\infty]$

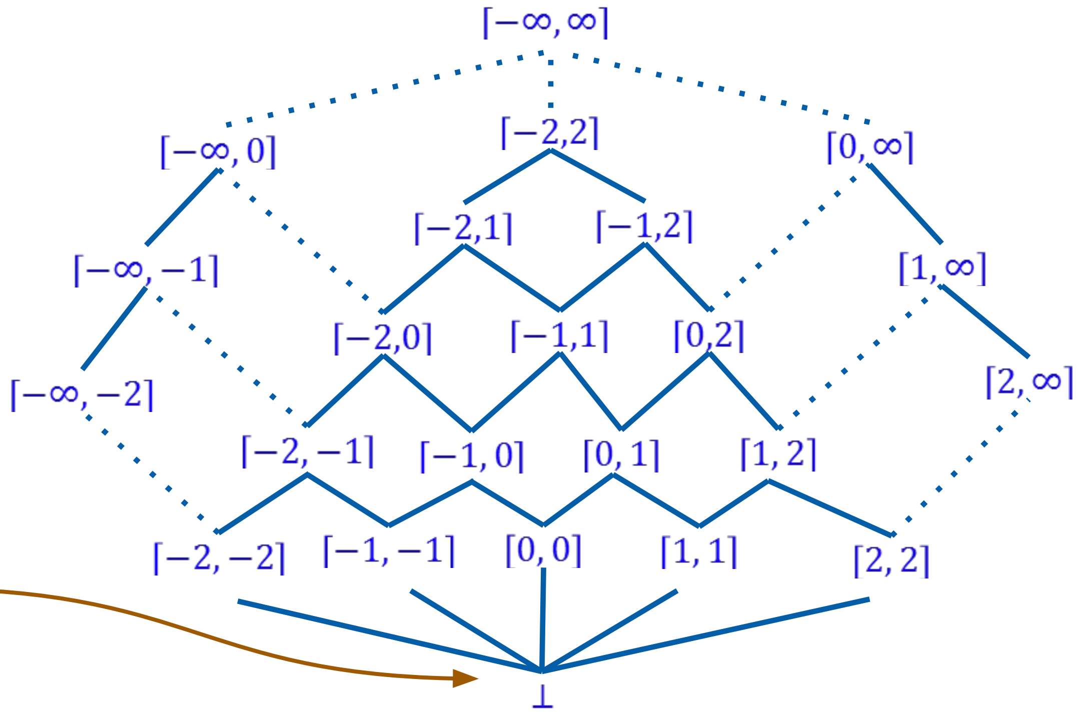
```
int averageResponseTime(int totalTime, int numRequests) {
  return totalTime / numRequests;
}
```

CVE-2019-14498

A divide-by-zero error exists in VLC media player that can be exploited by a crafted audio file

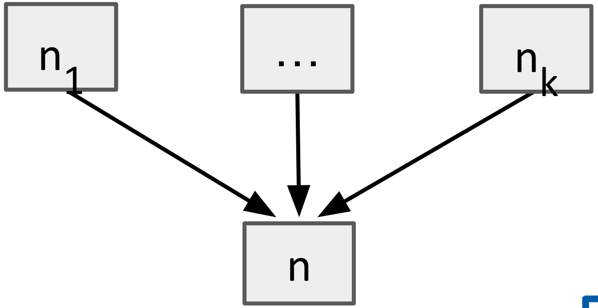
Interval Analysis: Abstract Domain

- Intervals ordered by inclusion
- The lattice has infinite height!



Bottom element interpreted as "not an integer"

Interval Analysis: Computing IN



$$\mathrm{IN}[n] = \bigsqcup_{n' \in \text{predecessors}(n)} \mathrm{OUT}[n']$$

for each variable **x**:

$$\mathrm{IN}[n](\mathbf{x}) = [\mathbf{MIN}(\mathbf{l_1}, \ldots, \mathbf{l_k}), \mathbf{MAX}(\mathbf{h_1}, \ldots, \mathbf{h_k})]$$

where:

$$\mathrm{OUT}[n_1](\mathbf{x}) = [\mathbf{l_1}, \mathbf{h_1}]$$

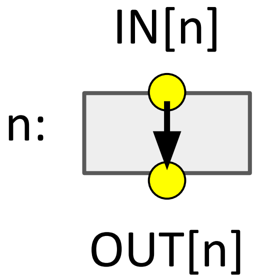
$$\ldots$$

$$\mathrm{OUT}[n_k](\mathbf{x}) = [\mathbf{l_k}, \mathbf{h_k}]$$

Interval Analysis: Computing OUT, differs with operations

n: x = y

OUT[n](x) = IN[n](y)



n: x = y - z

OUT[n](x) = [y1 - z2, y2 - z1]

where:

IN[n](y) = [y1, y2]

IN[n](z) = [z1, z2]

OUT[n](w) = IN[n](w) for each variable w other than x

Interval Analysis: Computing OUT, differs with operations

n: $\boxed{x = y + z}$

$$OUT[n](x) = [\ \boxed{y1 + z1}\ ,\ \boxed{y2 + z2}\ ]$$

where:

$$IN[n](y) = [y1, y2]$$

$$IN[n](z) = [z1, z2]$$

OUT[n](w) = IN[n](w) for each variable w other than x

Interval Analysis: Computing OUT, differs with operations

n: $\boxed{x = y * z}$

$$OUT[n](x) = \left[MIN(\boxed{x1y1, x1y2, x2y1, x2y2}),\ MAX(\boxed{x1y1, x1y2, x2y1, x2y2})\right]$$

where:

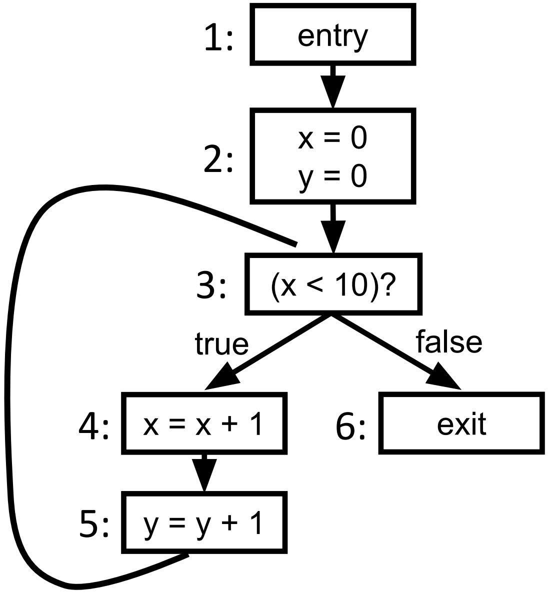
IN[n](y) = [y1, y2]

IN[n](z) = [z1, z2]

OUT[n](w) = IN[n](w) for each variable w other than x

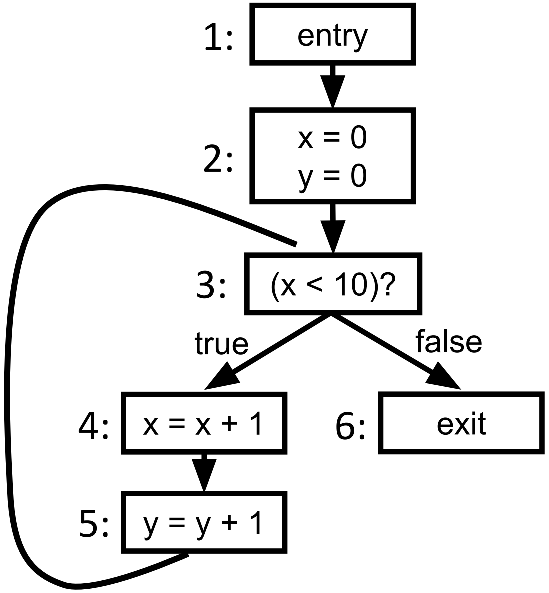
Interval Analysis: Example

n	Iter # 0
1	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$
2	$x \in \{\perp\}$ $y \in \{\perp\}$
3	$x \in \{\perp\}$ $y \in \{\perp\}$
4	$x \in \{\perp\}$ $y \in \{\perp\}$
5	$x \in \{\perp\}$ $y \in \{\perp\}$



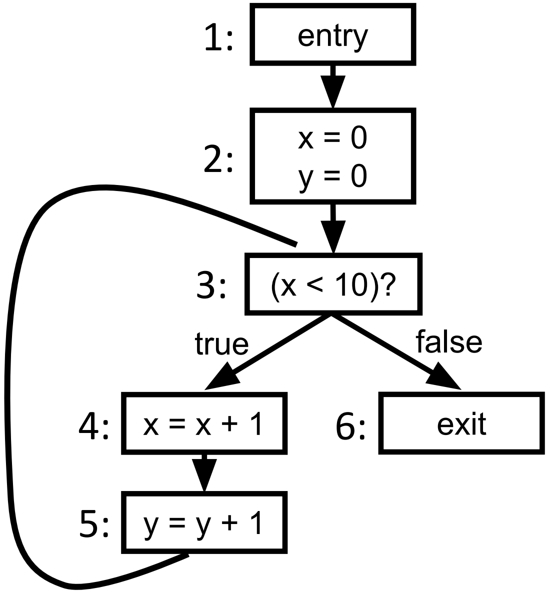
Interval Analysis: Example

n	Iter # 0	Iter # 1
1	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$
2	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [0,0]$ $y \in [0,0]$
3	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [0,0]$ $y \in [0,0]$
4	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [1,1]$ $y \in [0,0]$
5	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [1,1]$ $y \in [1,1]$



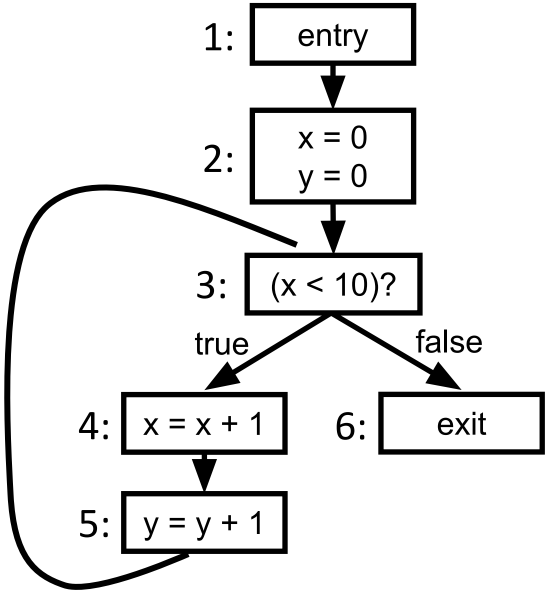
Interval Analysis: Example

n	Iter # 0	Iter # 1	Iter # 2	Iter # 3	Iter # k
1	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$
2	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$
3	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,1]$ $y \in [0,1]$	$x \in [0,2]$ $y \in [0,2]$	$x \in [0,k-1]$ $y \in [0,k-1]$
4	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [1,1]$ $y \in [0,0]$	$x \in [1,2]$ $y \in [0,1]$	$x \in [1,3]$ $y \in [0,2]$	$x \in [1,k]$ $y \in [0,k-1]$
5	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [1,1]$ $y \in [1,1]$	$x \in [1,2]$ $y \in [1,2]$	$x \in [1,3]$ $y \in [1,3]$	$x \in [1,k]$ $y \in [1,k]$



Interval Analysis: Example

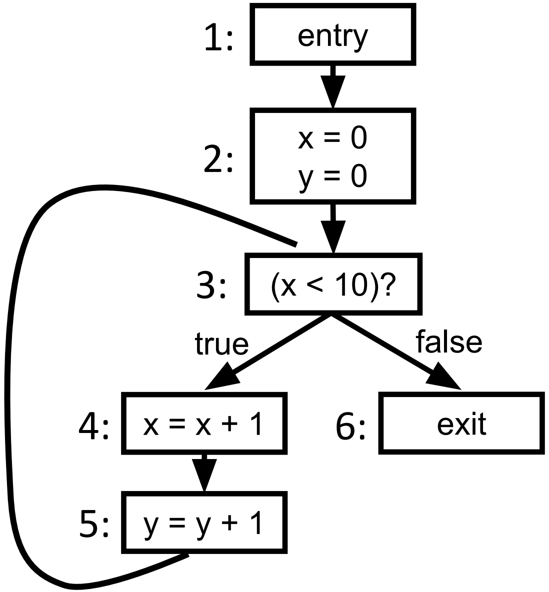
n	Iter # 0	Iter # 1	Iter # 2	Iter # 3	Iter # k
1	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$
2	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$
3	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,1]$ $y \in [0,1]$	$x \in [0,2]$ $y \in [0,2]$	$x \in [0,k-1]$ $y \in [0,k-1]$
4	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [1,1]$ $y \in [0,0]$	$x \in [1,2]$ $y \in [0,1]$	$x \in [1,3]$ $y \in [0,2]$	$x \in [1,k]$ $y \in [0,k-1]$
5	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [1,1]$ $y \in [1,1]$	$x \in [1,2]$ $y \in [1,2]$	$x \in [1,3]$ $y \in [1,3]$	$x \in [1,k]$ $y \in [1,k]$



Interval Analysis: Example

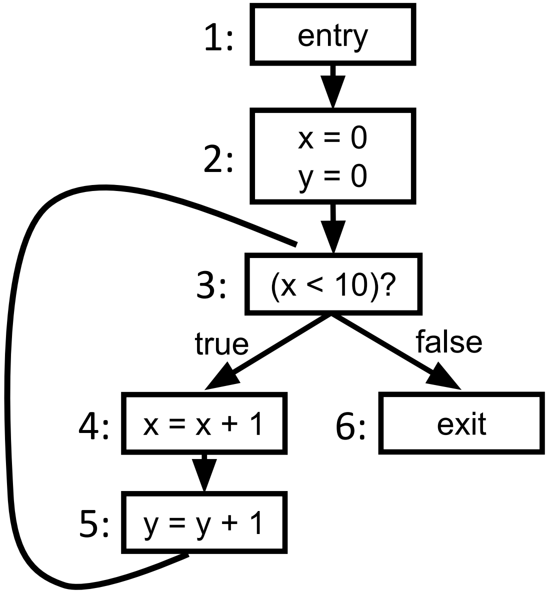
n	Iter # 0	Iter # 1	Iter # 2	Iter # 3	Iter # k
1	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$
2	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$
3	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,1]$ $y \in [0,1]$	$x \in [0,2]$ $y \in [0,2]$	$x \in [0,9]$ $y \in [0,9]$
4	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [1,1]$ $y \in [0,0]$	$x \in [1,2]$ $y \in [0,1]$	$x \in [1,3]$ $y \in [0,2]$	$x \in [1,10]$ $y \in [0,9]$
5	$x \in \{\bot\}$ $y \in \{\bot\}$	$x \in [1,1]$ $y \in [1,1]$	$x \in [1,2]$ $y \in [1,2]$	$x \in [1,3]$ $y \in [1,3]$	$x \in [1,10]$ $y \in [1,10]$

more precise analysis



Interval Analysis: Example

n	Iter # 0	Iter # 1	Iter # 2	Iter # 3	Iter # k	Iter # ∞
1	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$
2	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$
3	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,1]$ $y \in [0,1]$	$x \in [0,2]$ $y \in [0,2]$	$x \in [0,k-1]$ $y \in [0,k-1]$	$x \in [0,\infty]$ $y \in [0,\infty]$
4	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [1,1]$ $y \in [0,0]$	$x \in [1,2]$ $y \in [0,1]$	$x \in [1,3]$ $y \in [0,2]$	$x \in [1,k-1]$ $y \in [0,k-1]$	$x \in [1,\infty]$ $y \in [0,\infty]$
5	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [1,1]$ $y \in [1,1]$	$x \in [1,2]$ $y \in [1,2]$	$x \in [1,3]$ $y \in [1,3]$	$x \in [1,k-1]$ $y \in [1,k-1]$	$x \in [1,\infty]$ $y \in [1,\infty]$



Interval Analysis: Example

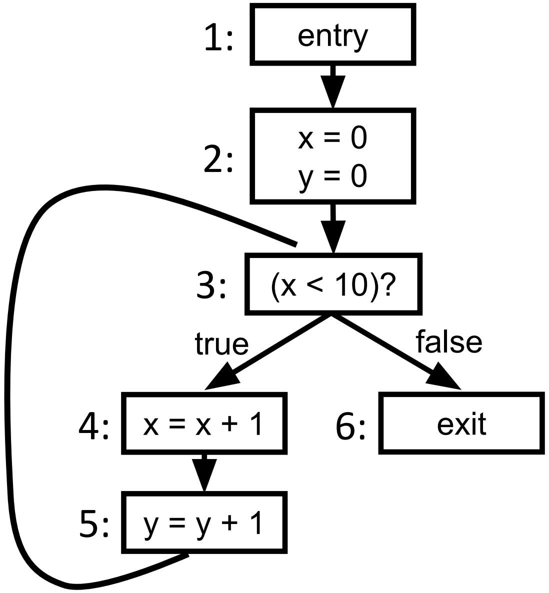
- In infinite-height lattice, the fixed-point computation does not terminate!

- Solution: Widening

  Infinite ascending chain:

  $y \rightarrow \perp$, [1,1], [1,2], [1,3], …, [1,k-1], …, [1,∞]

  Finite ascending chain: $y \rightarrow \perp$, [1,1], [1,2], [1,∞]



Static analysis for vulnerability detection

```
void main() {
  uint n, j, k, m;
  char buf[16];
  scanf("%u", &n);
 L1: j = sizeof(buf) + 2;
 L2: k = foo(j) + 4;
 L3: m = n;
 L4: m = m*2;
}
```

Integer overflow detector says possible overflows at:

- L1, L2, L3, L4

But, the values at lines L1 and L2 are constants so **most likely the overflow is not possible or programmer expected it.**

However, the overflow is quite possible at lines L3 and L4 as the data is controlled by the user.

Taint analysis

- Identifying the flow of user (tainted) data in the program.
    - **Taint sources:** Sources of tainted data.
        - E.g., scanf, fread, fwrite, etc.
    - **Taint propagation**: How each instruction propagates the taint from its operands to results.
        - E.g., a = b + c ==> Taint(a) = Taint(b) || Taint(c).

Vulnerability detection using taint analysis

- **Integer overflow:** Use of tainted data as an operand in arithmetic operation.
- **Out of bounds access:** Use of tainted data as index into an array.
- **Possible infinite loop detector:** Use of tainted data as the loop bound.
- …

Sensitive Sinks: Instructions or program points where tainted data should not be used.

Vulnerability detection using taint analysis

Track the flow of tainted data through the program and identify if any tainted data is used at sensitive sinks.

Taint analysis: Example

```
struct kernel_obj ko;

void increment(int *ptr) {
     *ptr +=1;
}

void entry_point(void *uptr){
     c_data->item = &ko;

     memcpy(&ko, uptr, sizeof(ko));

     increment(&(c_data->s));

     for (int i=0; i < ko.count; i++) {
         increment(&(ko.data[i]));
     }
     strcpy(..., c_data->buf);
     strcat(..., c_data->item);
     atoi(c_data->item);
}
```

Taint analysis: Taint propagation

```
struct kernel_obj ko;

void increment(int *ptr) {
    *ptr +=1;  // only when called from L1
}

void entry_point(void *uptr){
    c_data->item = &ko;

    memcpy(&ko, uptr, sizeof(ko));

    increment(&(c_data->s));

    for (int i=0; i < ko.count; i++) {
        L1: increment(&(ko.data[i]));
    }
    strcpy(..., c_data->buf);
    strcat(..., c_data->item);
    atoi(c_data->item);
}
```

Taint analysis: Vulnerabilities

```
struct kernel_obj ko;

void increment(int *ptr) {
     *ptr +=1; // only when called from L1
}

void entry_point(void *uptr){
     c_data->item = &ko;

     memcpy(&ko, uptr, sizeof(ko));

     increment(&(c_data->s));

     for (int i=0; i < ko.count; i++) {
         L1: increment(&(ko.data[i]));
     }
     strcpy(..., c_data->buf);
     strcat(..., c_data->item);
     atoi(c_data->item);
}
```

Integer overflow

Infinite Loop

Buffer overflow

Taint analysis: Challenges

```
struct kernel_obj ko;

void increment(int *ptr) {
     *ptr +=1;  // only when called from L1
}

void entry_point(void *uptr){
     c_data->item = &ko;

     memcpy(&ko, uptr, sizeof(ko));

     increment(&(c_data->s));

     for (int i=0; i < ko.count; i++) {
         L1: increment(&(ko.data[i]));
     }
     strcpy(..., c_data->buf);
     strcat(..., c_data->item);
     atoi(c_data->item);
}
```

- **Need alias analysis**: Can two pointers point to same object?

Taint analysis: Challenges

```
struct kernel_obj ko;

void increment(int *ptr) {
     *ptr +=1;  // only when called from L1
}

void entry_point(void *uptr){
     c_data->item = &ko;

     memcpy(&ko, uptr, sizeof(ko));

     increment(&(c_data->s));

     for (int i=0; i < ko.count; i++) {
         L1: increment(&(ko.data[i]));
     }
     strcpy(..., c_data->buf);
     strcat(..., c_data->item);
     atoi(c_data->item);
}
```

- Need alias analysis: Can two pointers point to same object?
- **Field sensitivity**: Should be able to distinguish between different fields of a same object.

Taint analysis: Challenges

```
struct kernel_obj ko;

void increment(int *ptr) {
     *ptr +=1;  // only when called from L1
}

void entry_point(void *uptr){
     c_data->item = &ko;

     memcpy(&ko, uptr, sizeof(ko));

     increment(&(c_data->s));

     for (int i=0; i < ko.count; i++) {
         L1: increment(&(ko.data[i]));
     }
     strcpy(..., c_data->buf);
     strcat(..., c_data->item);
     atoi(c_data->item);
}
```

- Need alias analysis: Can two pointers point to same object?
- Field sensitivity: Should be able to distinguish between different fields of a same object.
- **Context sensitivity**: Should be able to analyze function based on their calling context.

Analysis sensitivities

- **Flow-sensitive:** Analysis results depends on the program flow. Each program point has different results.
- **Path-sensitive:** Results depend on the control flow path. Each path in the CFG has different results.
- **Field-sensitive:** Results depend on the field of the structure or class. Each field of a structure or class has potentially different results.
- **Context-sensitive: Results depend on the context.** Results of a function differs with callers.
- **Object-sensitive**: ...

* **Sensitive** -> Analysis results depends on the * entity.

Precision Comparability of different sensitivities

- Is path-sensitive more precise than flow-sensitive or vice versa?
- Is flow-sensitive more precise than context-sensitive?
- Is field-sensitive more precise than field-insensitive?

Precision Comparability of different sensitivities

- Is path-sensitive more precise than flow-sensitive or vice versa?
- Is flow-sensitive more precise than context-sensitive?
- Is field-sensitive more precise than field-insensitive?

In general, we *cannot* compare precision of different sensitivities.

However, * sensitive analysis ***are definitely more precise*** than * insensitive analysis.

Vulnerability Detection: Expectations

"What Developers Want and Need from Program Analysis" [FSE 2016]

- Extremely less false positives (<10%).
- Can be unsound: need not find all the bugs, but should find most of the bugs.
- Need not be completely automated:
  - Developers are willing to provide input.
- Should be relatively fast, but it is okay if tool needs some prior processing time.

Vulnerability Detection Trend

Technique	2000	2001	2002	2003	2004	2005	2006	2008	2012	2014	2015	2017
Pattern based						[83]	[81]					
Smart Pattern Based									[82, 74]	[78, 79]	[75]	
Unsound, no pointer handling.	[51]			[42]	[43]							
Smart Unsound												
Interactive											[90]	[91]
Annotation Based		[72, 65]	[63, 67, 76]	[64]	[66]		[68]					
Sound, pointer handling				[38, 39]				[56]				

*Numbers are from references pdf.