Software Security 05

Sven Schäge

Overview

- Recap: Our Starting Point
- Data Flow Analysis: Conceptual Idea
- Gaining Intuition via Examples
  - Computing a Reaching Definitions Analysis
  - Computing a Very Busy Expressions Analysis
- Identifying the Overall Pattern
- Applications to Security
- Limits and Advanced Topics

Learning Outcomes:

By the end of this lecture, you will:

- Be able to define useful facts about a program that can automatically be computed via data flow analysis. These facts can be interpreted in a security-context.
- Be able to apply data flow analysis to automatically compute useful facts about program code.
- Be able to design your own data flow analysis and introduce incompleteness to make it run automatically.
- Be able to reflect on the limits and advantages of data flow analysis.
- Be able to identify suitable application scenarios for data flow analysis.
- Understand the necessity for advanced techniques like pointer analysis, points-to analysis and alias analysis in complex programs.

Recap: Starting Point

- Rice‘s Theorem:
  - Rice’s Theorem essentially states that functional properties of programs are undecidable and thus cannot be computed automatically with perfect completeness and soundness.

  => Need approximations or concentrate on non-functional properties.
- Conservative Approximations
  - Does either only introduce non-perfect soundness or non-perfect completenes.
- CFG
  - Comprises all possible program executions but also impossible ones.
  - If absence of error is established for CFG, it holds for concrete program as well.
  - Approximation introduced at decision points. (CFG takes if and else branch while execution only takes one branch.)

How to Circumvent Rice’s Theorem

- Establish useful properties (facts) of the approximate program in a way that allows for a **fixed-point computation** that can be solved in finite amount of time.
  - At the beginning each step of the approximated program is initialized to some default **set**.
  - The computations now proceed step-wisely, slowly changing the state of the approximated program.
- Idea for finite termination
  - Sets either grow or shrink **monotonously.** This is guaranteed if the properties are organized as a (mathematical) lattice, a partial order where unions and intersections of all elements exist.
    - (CodeIsDead, CodeNotDead, No Information Yet/Uninitialized, Don’t Know - Can be Both)
  - The number of all possible configurations of all the sets is finite.
    (We only really need that the lattice has finite height.)
- The established properties are interpreted in a security context. This can be used to show that certain vulnerabilities are not present in the program.
- The established properties hold for the original program as well.

Example of a Non-functional, Security-Relevant Fact: Program has No Divisions by Zero

- Easy (trivial)
  - x/const
  - If const=0, bad
  - Otherwise, good
- Non-trivial
  - x/y
  - Need to know that in no execution of program, y will be set to 0!
  - For all inputs and thus for all program paths!

Can efficiently be addressed via Dataflow Analysis!
Strategic Goal: Know all values that y may take on when this program point is reached!

- Even Harder
  - x/Equations.LeftHandSide[i] (divisor is loaded by dereferencing a pointer to a memory)
  - Need to know that in no execution of program, Equations.LeftHandSide[i] (referenced to by a pointer) will be set to 0!
  - For all inputs and thus for all program paths!

Involves pointers and dynamic allocation, requires additionally pointer analysis

- Divisions by zero can be security-relevant:
  - In C++, division by zero on integers is undefined behavior. One of the rules about the C++ language is that the compiler may assume that undefined behavior will never occur, and thus is allowed to do anything if it does.
  - Often an exception handler will be invoked to handle the division by zero. In general, attackers know that exception handlers are not as well-tested as regular code flows.

Data Flow Analysis: Conceptual Ideas

Data Flow Analysis: Conceptual Ideas

- Result is a set of **facts at each program point**.
- Can analyse which program parts exchange **data** with each other and presents the resulting dependencies. (Enables the establishment of so-called data flow graph.)
- Intuitively, shows at each program point how certain variables may influence later (or earlier) variables based on the CFG.
- Two orthogonal dimensions in which data flow analysis can vary: direction and meet-operation.
  - Forward Analysis (like reaching definition analysis):
    What program parts/variables at position Y could be influenced by variable X?
  - Backward Analysis (like live variable analysis):
    What variables at point Y will be read afterwards (before their next write update)?
    - Used for deadcode elimination -> remove statements that assign variables which are not read
  - Must-Property (every path taken must have this property)
  - May-Property (at least one path may have this property)
- Can be used, for example, to reason about
  - critical **variables** being influenced by input variables -> taint analysis
  - or critical **variables** being reflected in output -> avoid leakage of critical data
- Can help to inform dynamic testing

Data Flow Analysis: Classical and Security-Related Goals

- Classical Dataflow Analysis
    - **Reaching Definitions Analysis -> The reaching definitions for a given program point are those assignments that may have defined the current values of variables**
        - **Find uninitialized variable uses**
    - **Available Expression Analysis -> Expression is available if current value has been computed earlier**
        - **Avoid recomputing expressions**
    - Very Busy Expressions Analysis -> Find expressions that will definitely be evaluated again before its value changes.
        - Reduce Code Size
    - Live Variables Analysis -> A variable is live at a program point if there exists an execution where its value is read later in the execution without it being written to in the meantime
        - Allocate Registers Efficiently
- Modern, Security-Related Dataflow Analysis
    - Proper Variable Initialization
    - Division-by-Zero (constant)
        - Execute reaching definitions analysis
        - Consider all program points that have division expression x/y
        - If at each of these points, there are only reaching definitions for y that assign constants !=0 to y => no problem
        - Absence of these specific division by (constant) zero holds for all possible program paths since we abstracted the program flow! Soundness!
        - However, more complicated division by variable zero is not excluded.
    - Sign Analysis
    - Taint Analysis
        - Check Information Flow (sensitive data leak, code injection,…)
    - Interval Analysis
        - Check Memory Safety (integer overflows, buffer overflows,…)
    - Concurrency Analysis
        - Concurrency Safety Properties (data races, deadlocks,…)
    - Side-Channel Analysis
        - E.g.: Division by Secret-Dependent Value

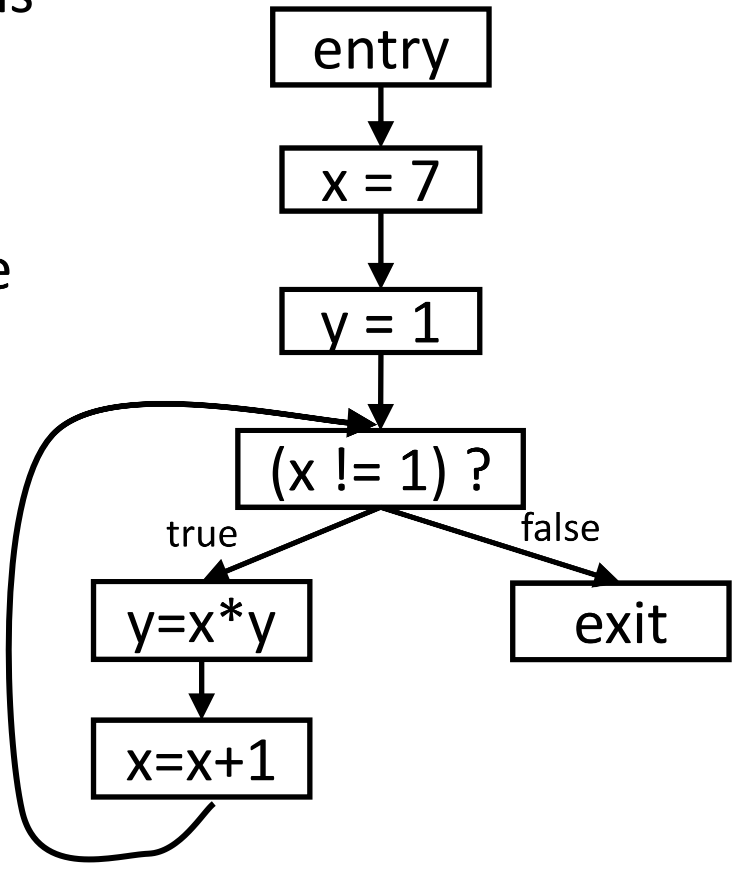
Data Flow Analysis Overview

- Method (intuitively):
  - Generalize/Approximate Program
  - Specify for each node of the CFG the set of facts (e.g. all assignments to all variables) fulfilled before the node and the set of facts fulfilled after the node. These facts are usually related to variables and more generally the state of the program.
  - Setup data-flow equations for each node of the control-flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a **fixpoint**.
- A basic algorithm for solving data-flow equations is an **iterative** algorithm:
  - for i ← 1 to N
    - initialize node i
  - while (sets are still changing)
    - for i ← 1 to N
    - recompute sets at node i
- Important conditions for termination:
  - Changes to sets' facts are monotonic! – May only grow or shrink. (The set of all possible sets forms a lattice.)
  - The set of all possible facts is finite!
- In simple form, does not deal with pointers but only basic types.

Data Flow Analysis –
A Detailed Example
(based on compact presentation of Mayur Naik)

Abstracting Control-Flow Conditions

- Abstracts away control-flow conditions with non-deterministic choice (*)
- Non-deterministic choice => assumes condition can evaluate to true or false
- Considers all paths possible in actual runs, and maybe paths that are never possible.



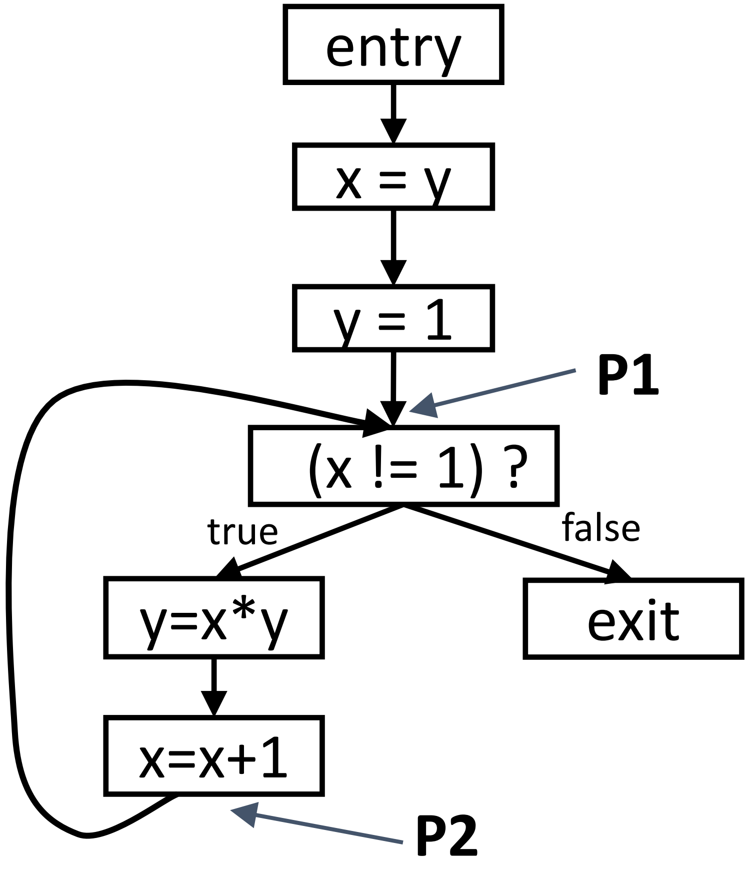
Property 1: Reaching Definitions Analysis

Determine, the reaching definitions!

Reaching definitions:

- Determine for each program point
    - which assignments have been made and not overwritten, when execution reaches that point along some path
    - those assignments that may have defined the current values of variables.
- For each point: “All variable definitions/assignments, that reach until that point”

- In this example we have four assignments: x=y, y=1, y=x*y, and x=x+1
- As an example, x=y reaches P1 but not P2 because x gets overwritten via x=x+1

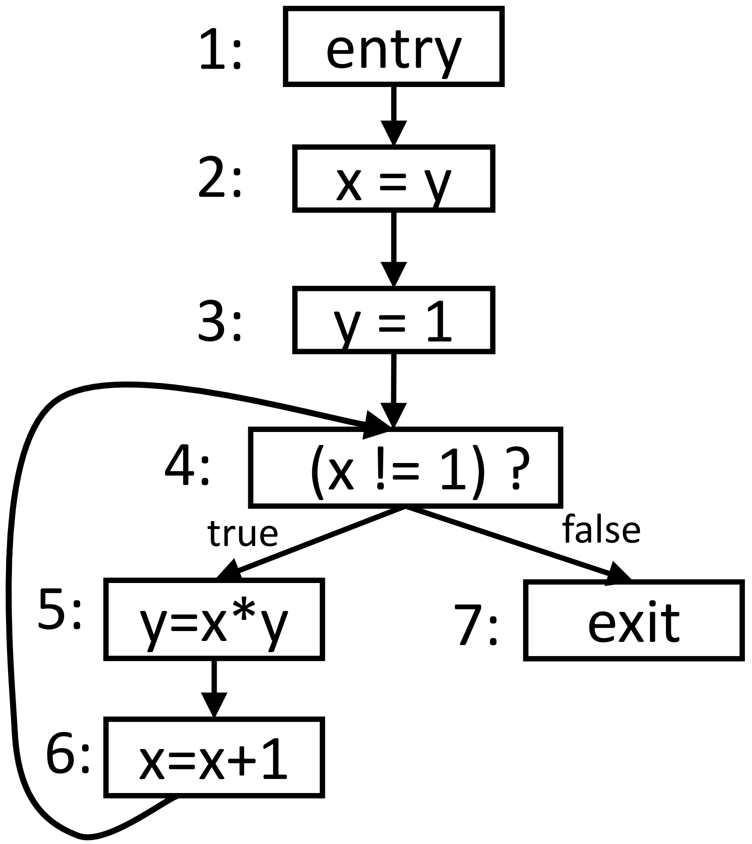


Result of Dataflow Analysis (Intuitively)

- Set of facts at each program point, where each fact is a pair of the form:
  <defined variable name, defining node label>
- Examples: <x,2> , <y,5>
- This helps to distinguish distinct assignments to the same variable!

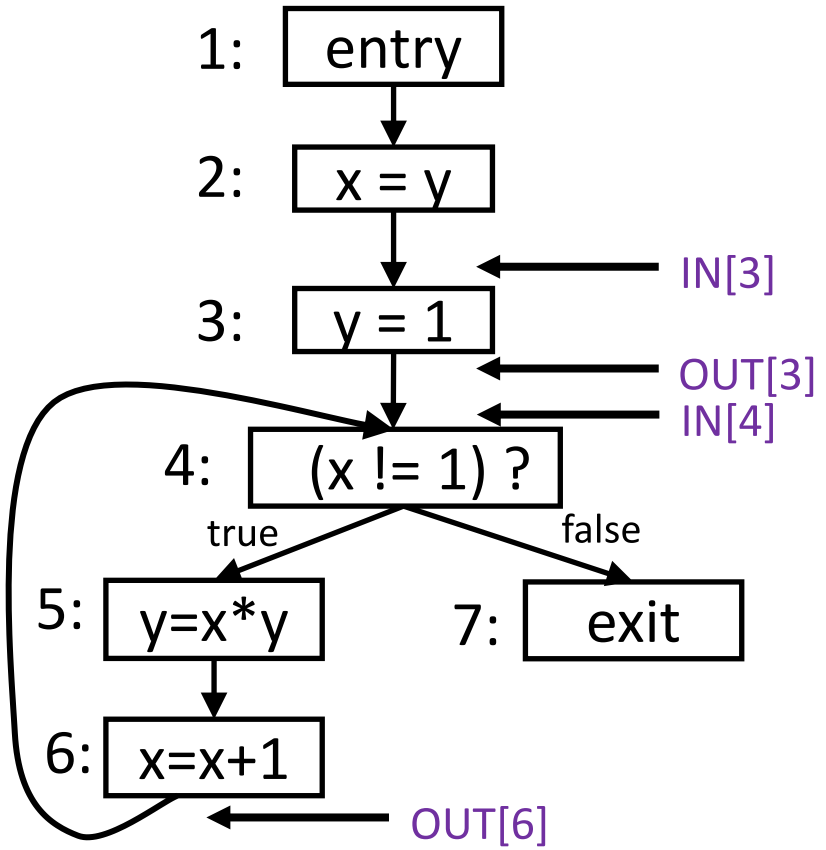
- In other data flow analyses, other interesting facts maybe specified
  - Sets of assignments
  - Sets of variables
  - Sets of expressions

major design space $\longrightarrow$



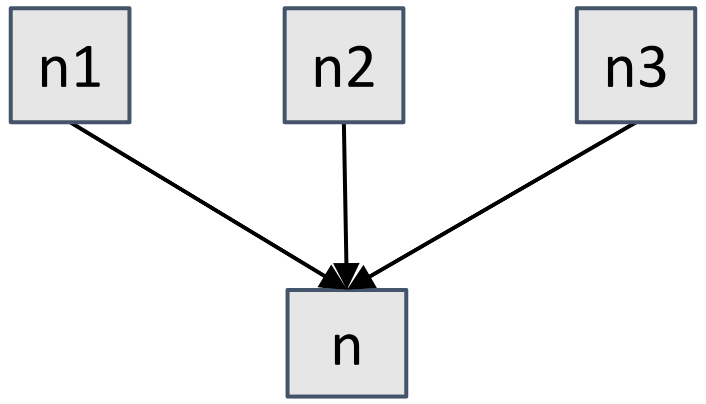
Result of Dataflow Analysis (Formally)

- Assign to each node label n
- 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 the two operations until IN[n] and OUT[n] stop changing for all n
    - "Fixed point"



Reaching Definitions Analysis: Operation #1

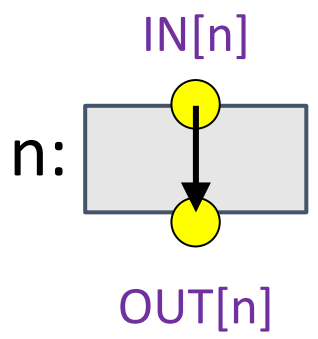
$$\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']$$



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

Reaching Definitions Analysis: Operation #2

OUT[n] = (IN[n] - KILL[n]) ∪ GEN[n]



boolean expression evaluated

n: b ?

GEN[n] = ∅ KILL[n] = ∅

assignment

n: x = a

GEN[n] = { <x, n> }

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

Overall Algorithm: Chaotic Iteration

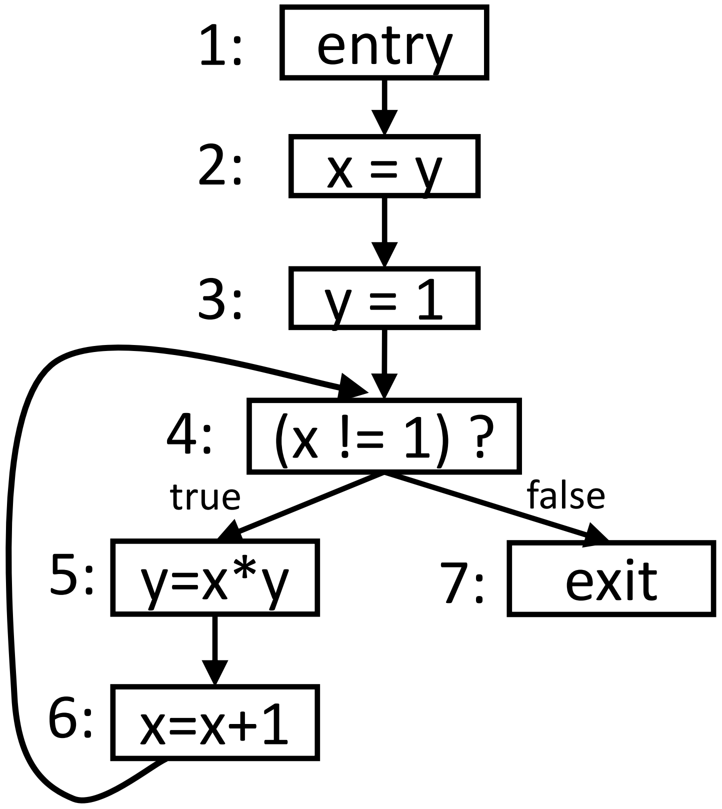
```
for (each node n):
    IN[n] = OUT[n] = ∅
OUT[entry] = { <v, ?> : v is a program variable }
repeat:
    for (each node n):
        $IN[n] = \bigcup_{n' \in predecessors(n)} OUT[n']$
        OUT[n] = (IN[n] - KILL[n]) ∪ GEN[n]
until IN[n] and OUT[n] stop changing for all n
```

$\}$ initialization

Reaching Definitions Analysis Example

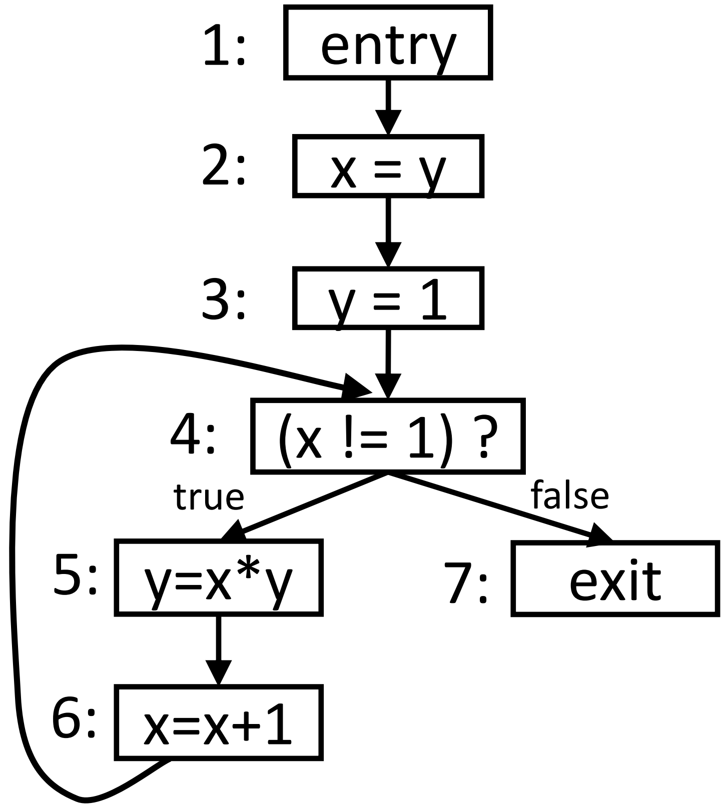
all variables, each of them uninitialised

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



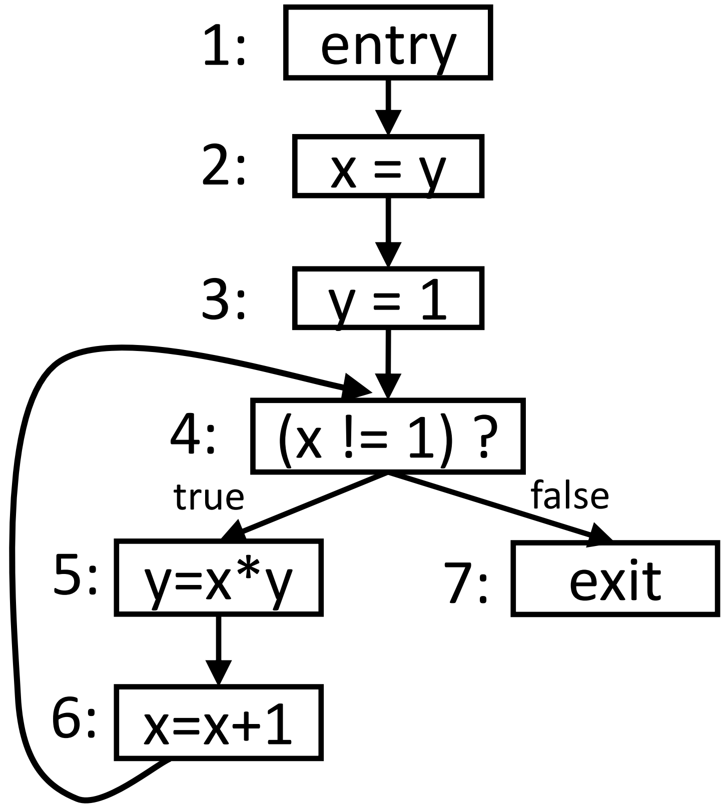
Reaching Definitions Analysis Example

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	∅	--



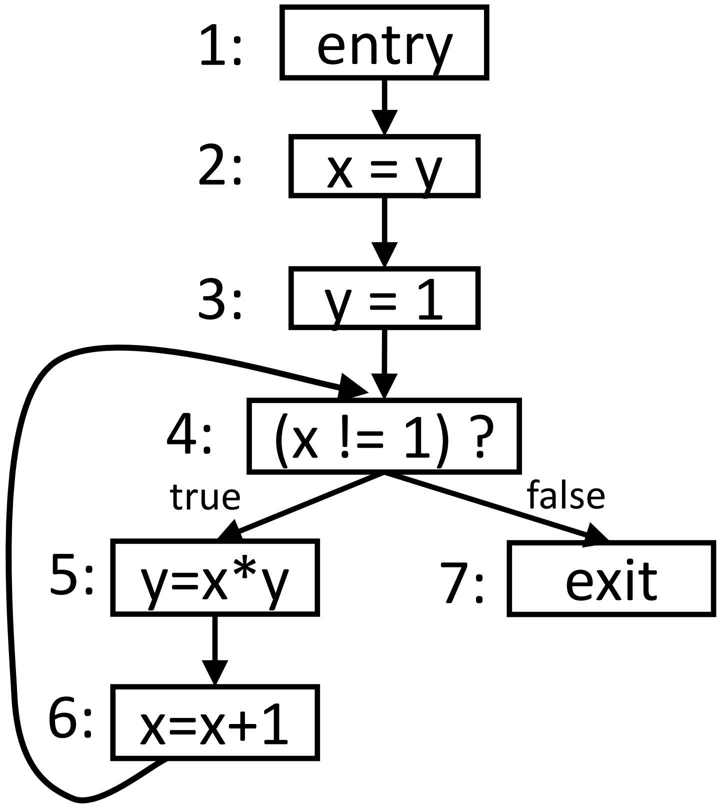
Reaching Definitions Analysis

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 Definitions Analysis

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>}	--



Does It Always Terminate?

Chaotic Iteration algorithm always terminates

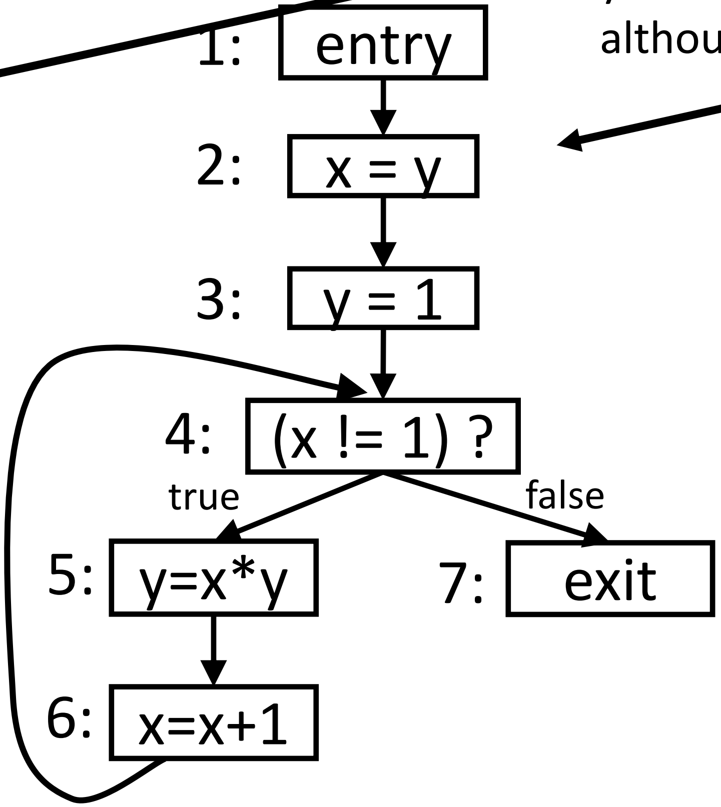
- The two operations of reaching definitions analysis are monotonic
  => IN and OUT sets never shrink, only grow
- Largest they can be is set of all definitions in program, which is finite
  => IN and OUT cannot grow forever

=> IN and OUT will stop changing after some iteration

Reaching Definitions Analysis

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>}	--

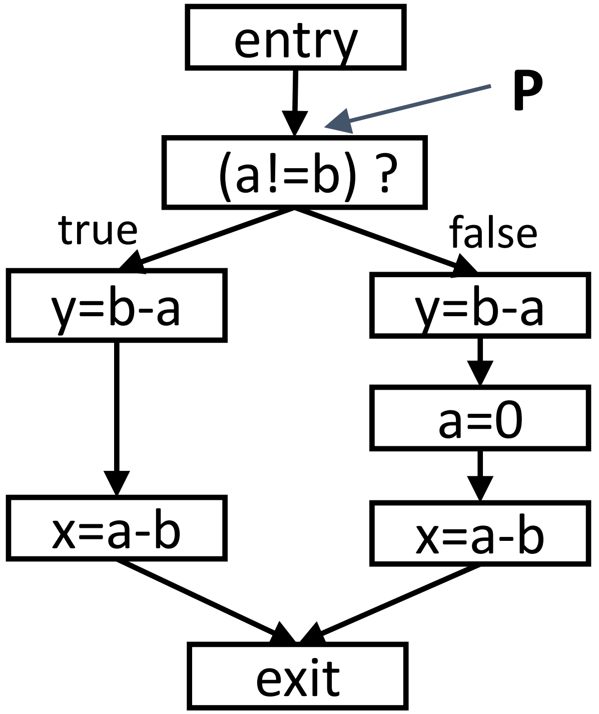
security critical observation:
y used in assigment although uninitialised



Very Busy Expressions Analysis

Goal: Determine very busy expressions at the exit from the 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.



b-a is very busy here since

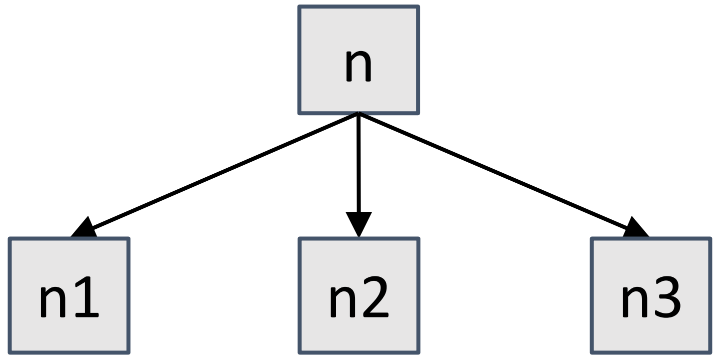
- it is used on all paths and
- none of its variables are redefined on any path (between expression b-a and program point P)

a-b is not very busy at P since a is redefined!

=> b-a can be executed before branch, saving code

Very Busy Expressions Analysis: Operation #1

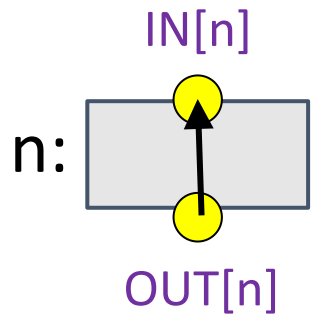
$$\mathrm{OUT}[n] = \bigcap_{n' \in \mathrm{successors}(n)} \mathrm{IN}[n']$$



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

Very Busy Expressions Analysis: Operation #2

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



n: b ?

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

n: x = a

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

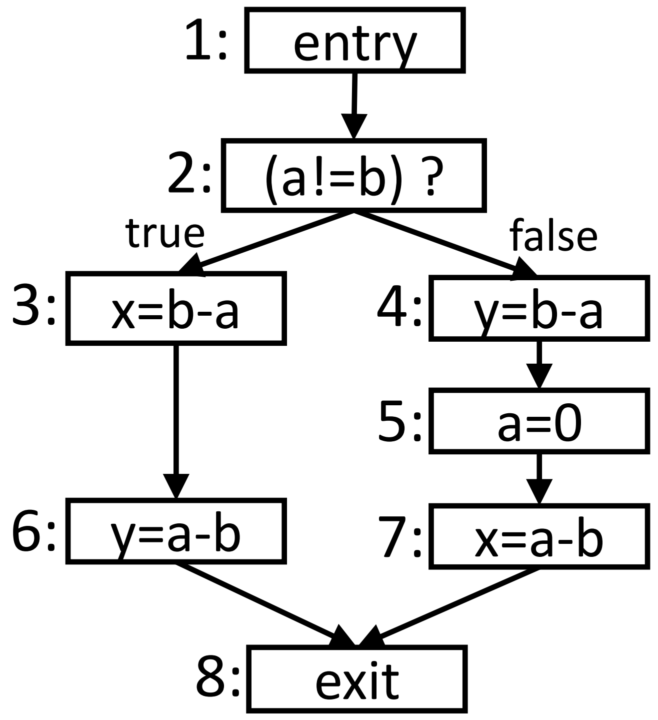
$KILL[n] = \{ \text{expr } e : e \text{ contains } x \}$

Overall Algorithm: Chaotic Iteration

```
for (each node n)
    IN[n] = OUT[n] = set of all exprs in program
IN[exit] = Ø
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
```

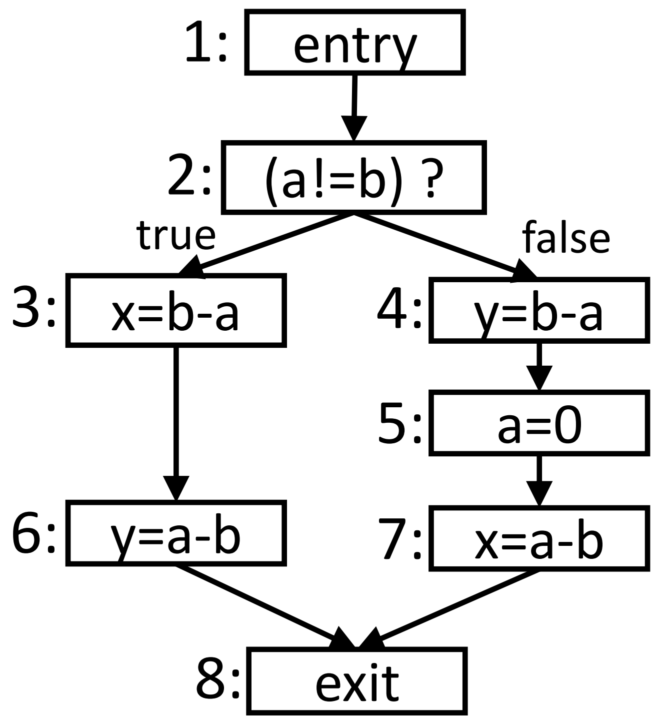
Very Busy Expressions Analysis Example

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	∅	--



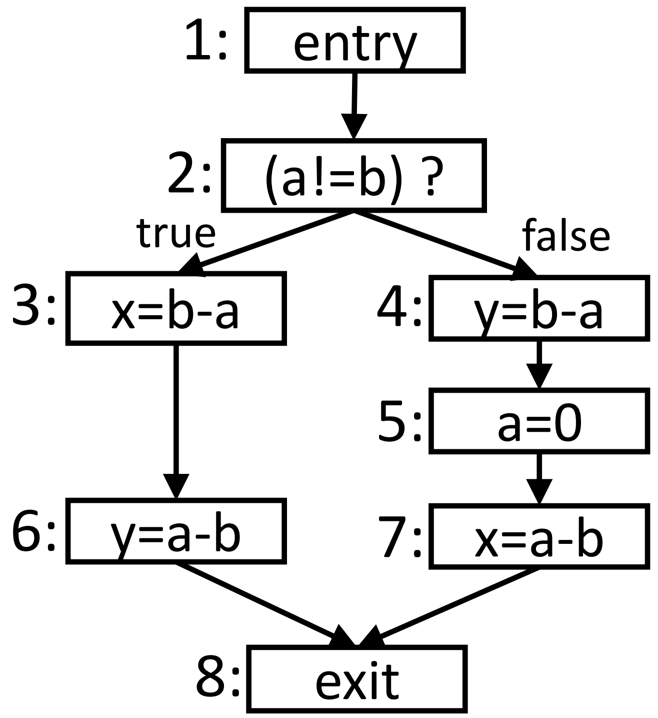
Very Busy Expressions Analysis Example

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 Expressions Analysis

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	∅	--



Identifying the Overall Pattern

Overall Pattern of Dataflow Analysis

🟦[n] = (🟥[n] - KILL[n]) ∪ GEN[n]

🟥[n] = ▢ 🟦[n']

n' ∈ ⬛(n)

🟦
🟥 = IN or OUT

▢ = ∪ (may) or ∩ (must)

⬛ = predecessors or successors

Reaching Definitions Analysis

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

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

▭
▭ = IN or OUT

▭ = ∪ (may) or ∩ (must)

▭ = predecessors or successors

Very Busy Expression Analysis

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

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

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

$\boxed{}$ = ∪ (may, any-path analysis) or ∩ (must, all-path analysis)

$\boxed{}$ = predecessors or successors

Forward, Backward, May, Must

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

- A forward analysis is one that for each program point computes information about the past behavior. Examples of this are sign analysis and available expressions analysis. They can be characterized by the right-hand sides of constraints only depending on predecessors of the CFG node. Thus, the analysis essentially starts at the entry node and propagates information forward in the CFG.
- A backward analysis is one that for each program point computes information about the future behavior. Examples of this are live variables analysis and very busy expressions analysis. They can be characterized by the right-hand sides of constraints only depending on successors of the CFG node.
- A may analysis is one that describes information that may possibly be true and, thus, computes an over-approximation. Examples of this are live variables analysis and reaching definitions analysis where the union is used to combine information.
- A must analysis is one that describes information that must definitely be true and, thus, computes an under-approximation. Examples of this are available expressions analysis and very busy expressions analysis. Information is combined via intersection.

Application to Security

Design Lattice for Basic Data Types

- Specify at each program point which tainted values will reach that point
- Sign Analysis (includes Divide-by-Zero Analysis)
  - Lattice: (0, +, -, ⊥, ⊤)
  - ⊤ denotes any of the choices (0, +, -) or none:
    - Least precise information
    - Don’t know
  - ⊥ denotes none of the choices
    - Most precise information
    - Empty set of integer values
    - For expressions that are not numbers but, say pointers
    - Or have no value in any execution because they are unreachable from the program entry.
  - Forward, May-Analysis

Taint Analysis

- Example of a security-focused form of data flow analysis
- A) Track how untrusted input flows through the program.
- B) Track how information flows through the program and if it is leaked to public observers.
- Taint analysis associates a "taint" tag or marker with data that comes from untrusted sources. This tag is then propagated through the program, allowing analysts to trace the tainted data.
- Common use cases for taint analysis include identifying and preventing security vulnerabilities such as Cross-Site Scripting (XSS), SQL injection, command injection, and other attacks that involve the manipulation of user inputs
- Taint Analysis A): Inputs to Security Critical Program Points
    - Lattice: (Tainted, NotTainted, $\bot$, $\top$)
    - Forward, May-Analysis
- Taint Analysis B): Secrets to Output Program Points
    - Lattice: (Tainted, NotTainted, $\bot$, $\top$)
    - Backward, May-Analysis

Limits and Advanced Topics

Pointers? -> Pointer Analysis

- Simple Data Flow Analysis assumes that the mapping of variable names map to memory units is injective.
- Thus it cannot directly deal with pointers where memory units may be referred to by several variable names. This is called aliasing.
- To deal with this additional complexity, one can additionally apply pointer analysis, that collects sets of pointers that could possibly map to the same memory unit.
    - In points to analysis we keep track of all the memory units a pointer can refer to.
    - In Alias analysis we keep track of the memory units that might be pointed to by several pointers.
- As for the CFG abstraction, this can typically produces sets of pointers that might in real program executions never occur.

Summary

- Data flow analysis is a powerful tool for static analysis.
- It allows defining interesting facts about program code that can be established automatically (under certain circumstances).
- The facts need to form a finite space organized as a lattice while the data-flow analysis will ensures monotonic behavior.
- It circumvents Rice’s theorem by introducing incompleteness and focusing on non-functional properties.
- It can be extended to work with pointers, besides constants, (primitive) variables, and expressions via Pointer Analysis.
- The facts that can be established with data flow analysis can be highly security-relevant and exclude the presence of many error types.
- Test your skills:
    - References to other examples, and pointer analysis in Canvas.
    - Practical challenges in the homework.