Static Taint Analysis

What we have learned so far about static analysis techniques

Recap

Static analysis ≠ perfect precision

Need to choose between:

- **Safety** (conservatively warn of all software vulnerabilities)
- **Certainty** (guarantee properties for compiler optimization)

Recap

Static analysis ≠ perfect precision

Need to choose between:

- **Safety** (conservatively warn of all software vulnerabilities)
- **Certainty** (guarantee properties for compiler optimization)

Reconcile two views

- **Over-approximation**: conservative, safe
- **Under-approximation**: precise

Over-approximation

May Analysis: What Might Be True

- Describes properties that **may hold** on **some paths**

Over-approximates possible runtime behavior

Key characteristics:

- Powerset Lattice order: $\subseteq$ (least element = $\emptyset$)
- Flow Equation: use **union (U)**

Examples:

- **Live Variables Analysis**

Under-approximation

Must Analysis: What Must Be True

- Describes properties that **hold on all paths**

Under-approximates – only keeps what's guaranteed

Key characteristics:

- Powerset Lattice order: ⊇ (greatest element = universal set)
- Flow Equation: use **intersection (∩)**

Examples:

- **Available Expressions**

Lattice is the same, but order is the opposite of the may analysis.
Greatest element gives the most information

Comparison

Feature	May Analysis	Must Analysis
Interpretation	May be true	Must be true
Approximation type	Over-approximation	Under-approximation
Powerset lattice order	$\subseteq$ (least = $\emptyset$)	$\supseteq$ (greatest = universal)
Combine operator	$\cup$ (union)	$\cap$ (intersection)
Typical goal	Vulnerability detection	Optimization

↳ Typically used for compiler optimization, because you want certainty

Comparison

Adopt May analysis when:

- We care about **possible behaviors** (e.g., potential vulnerabilities)

Adopt Must analysis when:

- We need **guaranteed facts** (e.g., safe optimizations)

The main difference lies in:

- **Lattice structure** ($\subseteq$ vs $\supseteq$)
- **Combining constraints** ($\cup$ vs $\cap$)

Question:
What structure would you choose for static taint analysis?

May analyses: Taint analyses as vulnerability detection, not optimization

Static Taint Analysis	
Interpretation	??
Approximation	??
Lattice order	??
Combine operator	??
Goal	??

Static Taint Analysis	
Interpretation	May
Approximation	Over-approximation
Lattice order	$untaint \sqsubseteq taint$
Combine operator	$\sqcup$ Least upper bound
Goal	Taint detection

From Dynamic to Static taint tracking

- Dynamic taint analysis operates at runtime: It tracks taint information as the program executes.
  - It ensures precise detection of vulnerabilities but can incur performance overhead.
  - Incomplete Coverage: Only the executed paths are analyzed.
    - If a piece of code or a dangerous path is never executed during runtime, it is never checked.
  - This leads to false negatives (i.e., vulnerabilities may go undetected).

DA was like a digital twin of execution that analyzed flow of execution and put taint stickers on variables and was ready to abort execution when tainted data reach sinks.

2 drawbacks: runtime overhead, dta only analyzes current path. Cannot establish that on all possible paths something holds

From Dynamic to Static taint tracking

- Static taint analysis ensures a broad coverage, works before execution
  - No runtime costs
  - Other issues … later

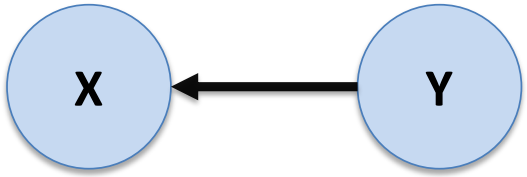
(Static) Intuition: typing flow of assignments

What is interesting behind statically? Think of usage of types, that statically prove properties of variables. Think of assignment: X=Y. There is a flow of info from Y to X.

X := Y

The flow from Y to X is legale whenever the (type of) value of variable Y (τ_Y must be compatible with the (type of) value of variablle x (τ_X)

From PoV of static type system, if you need to check if X=Y is well typed, type associated to Y must be compatible with type associated to X. If type of Y is a subset of the types X can take, that is good.



FLOW

$$\frac{\tau_Y < \tau_X}{X := Y \text{ is ok}}$$

TYPING RULE

(Static) Intuition: typing flow of assignments

The degree of taint lattice

We use this information with taint propagation. We use a very simple lattice for this analysis.

(Static) Intuition: typing flow of assignments

taint

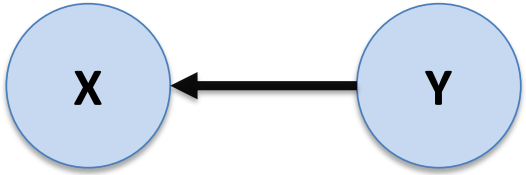
untaint

The degree of Taint lattice

X := Y

Same intuitive idea: taint degree must be compatible

The flow from Y to X is legale whenever the the degree of taint τ_Y of variable Y is less than the degree of taint τ_X of variablle x.



FLOW

$$\frac{\tau_Y < \tau_X}{X := Y \text{ is ok}}$$

TYPING RULE

untaint ≤ taint. Compatible.

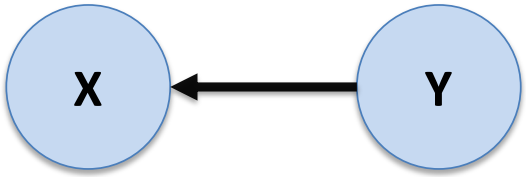
(Static) Intuition: typing flow of assignments

taint

untaint

The degree of
Taint lattice

$\tau_y = untaint$



FLOW

$$\frac{\tau_Y < \tau_X}{X := Y \ is \ ok}$$

TYPING RULE

(Static) Intuition: typing flow of assignments

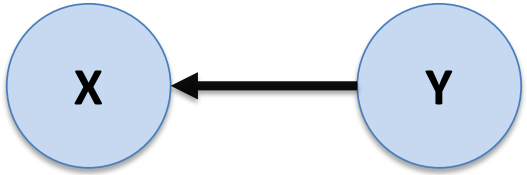
taint

untaint

The degree of
Taint lattice

$\tau_y = taint$

$\tau_x = untaint$



FLOW

$$\frac{\tau_y < \tau_x = false}{X := Y \text{ is not ok}}$$

The ingredients of the analysis

The analysis by examples

```
string name = source()
string x = name;
sink(x)
```

requires only untainted values

```
sink (untainted string s) = printfun(untainted  string s) { … // trusted sink};

Source() = tainted  string getsFromNetwork(….); //untrusted source
```

Ingredient 1: the analysis

- **May analysis**:
- Lattice order is $\subseteq$
- Join function is **union ($\cup$)**
- Taint facts: set of **tainted** variables -- we track which variables are **tainted**

Over approximation: we don't consider value of variables, just taint status.

Ingredient 2: the flow equation

We define for each program point

- **GEN(n)**: New taint facts **generated** at statement n
- **KILL(n)**: Taint facts **removed** or overwritten at statement n
- **IN(n)**: Taint facts **before** statement n
- **OUT(n)**: Taint facts **after** statement n
- **OUT(n) = (IN(n) – KILL(n))** ∪ **GEN(n)**

1. string name = source()
2. string x = name;
3. sink(x)

GEN(1) = {name}
KILL(1) = ∅
IN(1) = ∅

OUT(1) = GEN(1) ∪ (IN(1) – KILL(1)) = { name }

1. string name = source()
2. string x = name;
3. sink(x)

IN(1) = ∅
GEN(1) = {name}
KILL(1) = ∅

OUT(1) = GEN(1) ∪ (IN(1) – KILL(1)) = { name }

take all the possible predecessors

IN(2) = OUT(1)
if name ∈ IN(1) then x ∈ GEN(2)
KILL(2) = ∅

OUT(2) = GEN(2) ∪ (IN(2) – KILL(2)) = { name, x }

we have to know the flow of execution: to say that x is tainted, you should know if name is tainted.

OUT(1)

1. string name = source()
2. string x = name;
3. sink(x)

IN(1) = ∅
GEN(1) = {name}
KILL(1) = ∅

OUT(1) = GEN(1) ∪ (IN(1) – KILL(1)) = { name }

IN(2) = OUT(1)
if name ∈ IN(1) then x ∈ GEN(2)
KILL(2) = ∅

OUT(2) = GEN(2) ∪ (IN(2) – KILL(2)) = { name, x }

IN(3) = OUT(2) = { name, x }
GEN(3) = ∅
KILL(2) = ∅

Warning!!! Since x ∈ IN(3) a tainted value reaches sink!

Key Issue: Flow Sensitivity in Static Taint Analysis

The analysis depend on the flow of execution (order of execution). Y_r depends on r because gen depends on the order of execution. Y_r is context.

Flow-dependent (flow-sensitive) taint analysis considers the **order of statements** in a program.

The analysis tracks **how taint facts change over time** as execution flows through the control-flow graph.

GEN and KILL in Flow Sensitivity

GEN(s): What is generated (newly tainted) by basic block s

GEN(s): **adds taint** to variables.

- x = source() then GEN = { x }
- x = E and $\exists y \in PVar(E)\ and\ y \in IN(s)$ then $y \in GEN(s)$

→ Now you see the flow dependency: there is a flow of taint variables.

- because x becomes tainted at this point

KILL(s): What is killed (make untainted) or overwritten by block s

KILL(s) removes taint from variables, either by sanitizing or overwriting.

- x = 0 then KILL(s) = { x }
- x = E and $\forall y \in PVar(E)\ y \notin IN(s)$ then KILL(s) = {x}

→ every program variable

- the taint degree that was on x is now gone.

→ x equal to a constant

The flow equation

- At each program point, the analysis computes

$$OUT(s) = \big(IN(s) - KILL(s)\big) \cup GEN(s)$$

  - *IN(s)*: taint info **before** s
  - *OUT(s)*: taint info **after** s

- The flow equation precisely captures **flow sensitivity**:
- If a variable was tainted earlier (*IN(s)*), but gets overwritten or sanitized (*KILL(s)*), it will no longer be tainted in *OUT(s)*.
- If a taint is introduced (*GEN(s)*), it appears in *OUT(s)* even if it was not previously present.

8

Why this matters?

Without GEN/KILL, we would have to assume taints never disappear. This may lead to:

- **Overtainting** (false positives)
- Poor precision

Using GEN/KILL sets enables:

- Accurate **taint propagation**
- **Precise flow-sensitive reasoning** about taint evolution

1: x = source(); // **GEN = { x }**
2: x = 0; // **KILL = { x }**
3: sink(x);

$x \in OUT(1)$

$x \notin OUT(2)$

sink(x) is sanitized!!

What about the monotone framework?

The Monotone Framework

- Analysis Lattice: $L = untainted \leq tainted$
- Analysis State: $[i] \in PVar \rightarrow L$ *
- The Join operation: $JOIN(i) = \bigcup [j]\ j \in pred(i)$

↑ least upper bound of all incoming info

* Take a program point: it is a function from program variable to the lattice of ranked value.

The Monotone Framework

Assignment at program point i.

$$\frac{i.a = E}{[i] = JOIN(i)[a = eval(JOIN(i), E)]}$$

eval$(\sigma, a) = \sigma(a)$
$eval(\sigma, CstInt) =$ ***untainted***
$eval(\sigma, input) =$ ***tainted***
$eval(\sigma, E1\ op\ E2) = eval(\sigma, E1)\ \widehat{op}\ eval(\sigma, E2)$

Example: $\widehat{+}$ is the abstract additions over the abstract values given by
untainted $\widehat{+}$ ***untainted*** = ***untainted***
tainted $\widehat{+}$ v = ***tainted***
v $\widehat{+}$ ***tainted*** = ***tainted***

Example

1: string name = source()

2: string x = name;

3: sink(x)

$[1] = [name = \boldsymbol{tainted}]$ Mapping between variable name and value

$[2] = JOIN(2)[x = eval(JOIN(2), name)] = [1][\, x = \ eval([1], name)]$

$= [name = \boldsymbol{tainted}][eval(x = \ [name = \boldsymbol{tainted}](name)]$

$= [name = \boldsymbol{tainted}][x \ = \boldsymbol{tainted}] = [name = \boldsymbol{tainted}, x = \boldsymbol{tainted}]$

Conditionals

```
1: string name = source();      // taint source
2: string x;
3: if (...)                      // conditional
4:    x = name;                  // taint flows here
5: else x = "ciao";              // constant, untainted
6: sink(x);                      // potential leak
```

GOAL: Determine if tainted data may reach sink(x), using GEN/KILL sets

```
1: string name = source()
2: string x;
3: If ()
4:     x = name
5:    else  x = “ciao”;
6: sink(x)
```

$GEN(1) = \{name\}$

$KILL(1) = \emptyset$

$OUT(1) = GEN(1) \cup (IN(1) - KILL(1)) = \{name\}$

```
1: string name = source()
2: string x;
3: If ()
4:     x = name
5:    else  x = “ciao”;
6: sink(x)
```

$GEN(1) = \{name\}$

$KILL(1) = \emptyset$

$OUT(1) = GEN(1) \cup (IN(1) - KILL(1)) = \{name\}$

$GEN(2) = \emptyset$

$KILL(1) = \emptyset$

$OUT(2) = IN(2) = \{name\}$

assume that guard cannot modify tainted values, no assignment

```
1: string name = source()
2: string x;
3: If ()
4:     x = name
5:    else  x = “ciao”;
6: sink(x)
```

$$GEN(1) = \{name\}$$
$$KILL(1) = \emptyset$$
$$OUT(1) = GEN(1) \cup (IN(1) - KILL(1)) = \{name\}$$

$$GEN(2) = \emptyset$$
$$KILL(1) = \emptyset$$
$$OUT(2) = IN(2) = \{name\}$$

$$GEN(3) = \emptyset$$
$$KILL(3) = \emptyset$$
$$OUT(3) = IN(3) = \{name\}$$

```
1: string name = source()
2: string x;
3: If ()
4:     x = name
5:    else  x = “ciao”;
6: sink(x)
```

$GEN(1) = \{name\}$

$KILL(1) = \emptyset$

$OUT(1) = GEN(1) \cup (IN(1) - KILL(1)) = \{name\}$

$GEN(2) = \emptyset$

$KILL(1) = \emptyset$

$OUT(2) = IN(2) = \{name\}$

$GEN(3) = \emptyset$

$KILL(3) = \emptyset$

$OUT(3) = IN(3) = \{name\}$

$IN(4) = \{name\}$

$GEN(4) = \{x\}$

$KILL(4) = \emptyset$

$OUT(4) = \{x\} \cup IN(4) = \{x, name\}$

X is being tainted

```
1: string name = source()
2: string x;
3: If ()
4:     x = name
5:    else  x = “ciao”;
6: sink(x)
```

$$GEN(1) = \{name\}$$
$$KILL(1) = \emptyset$$
$$OUT(1) = GEN(1) \cup (IN(1) - KILL(1)) = \{name\}$$

$$GEN(2) = \emptyset$$
$$KILL(1) = \emptyset$$
$$OUT(2) = IN(2) = \{name\}$$

$$GEN(3) = \emptyset$$
$$KILL(3) = \emptyset$$
$$OUT(3) = IN(3) = \{name\}$$

$$IN(4) = \{name\}$$
$$GEN(4) = \{x\}$$
$$KILL(4) = \emptyset$$
$$OUT(4) = \{x\} \cup IN(4) = \{x, name\}$$

$$IN(5) = \{name\}$$
$$GEN(5) = \emptyset$$
$$KILL(5) = \{x\}$$
$$OUT(5) = IN(5) - \{x\} = \{name\}$$

GEN(1) = {name}

OUT(1) = (IN(1) ∪ {name}) − {name}

GEN(2)

GEN(4) = {x}

OUT(4) = (IN(4)) ∪ {x} = {x, name}

OUT(5) = IN(5)

```
1: string name = source()
2: string x;
3: If ()
4:     x = name
5:    else  x = "ciao";
6: sink(x)
```

$GEN(1) = \{name\}$

$KILL(1) = \emptyset$

$OUT(1) = GEN(1) \cup (IN(1) - KILL(1)) = \{name\}$

$GEN(2) = \emptyset$

$KILL(1) = \emptyset$

$OUT(2) = IN(2) = \{name\}$

$GEN(3) = \emptyset$

$KILL(3) = \emptyset$

$OUT(3) = IN(3) = \{name\}$

$IN(4) = \{name\}$

$GEN(4) = \{x\}$

$KILL(4) = \emptyset$

$OUT(4) = \{x\} \cup IN(4) = \{x, name\}$

$IN(5) = \{name\}$

$GEN(5) = \emptyset$

$KILL(5) = \{x\}$

$OUT(5) = IN(5) - \{x\} = \{name\}$

$IN(6) = OUT(4) \cup OUT(5) = \{name\ x\}$

Warning!!! Since x ∈ IN(6) a tainted value reaches sink!

Sink will be the union of the two

SUMMARY

	IN	GEN	KILL	OUT
1	Ø	{ name }	Ø	{ name }
2	{ name }	Ø	Ø	{ name }
3	{ name }	Ø	Ø	{ name }
4	{ name }	{ x }	Ø	{ name, x }
5	{ name }	Ø	{ x }	{ name }
6	{ name, x }	Ø	Ø	{ name, x }

SUMMARY

	IN	GEN	KILL	OUT
1	Ø	{ name }	Ø	{ name }
2	{ name }	Ø	Ø	{ name }
3	{ name }	Ø	Ø	{ name }
4	{ name }	{ x }	Ø	{ name, x }
5	{ name }	Ø	{ x }	{ name }
6	{ name, x }	Ø	Ø	{ name, x }

Analysis Result

Variable x **may be tainted** at the sink (line 6)

Static analysis **reports a possible taint flow**

Dropping Conditional

```
1: string name = source();
2: string x;
3: if (...)
4:    x = name;
5: else x = "ciao";
6: sink(x);
```

```
1: string name = source();
2: string x1;
3: string x2;
4: x1 = name;
5: x2 = "ciao";
6: sink(x2);
```

Imagine compiler is smart and removes gen/kill by using the two variables

GOAL: Determine if tainted data may reach sink(), using GEN/KILL sets

SUMMARY

	IN	GEN	KILL	OUT
1	Ø	{ name }	Ø	{ name }
2	{ name }	Ø	Ø	{ name }
3	{ name }	Ø	Ø	{ name }
4	{ name }	{ x1 }	Ø	{ name, x1 }
5	{ name, x1 }	Ø	{ x2}	{ name, x1}
6	{ name, x1}	Ø	Ø	{ name, x1 }

Analysis Result

Variable x2 **is not tainted** at the sink (line 6)

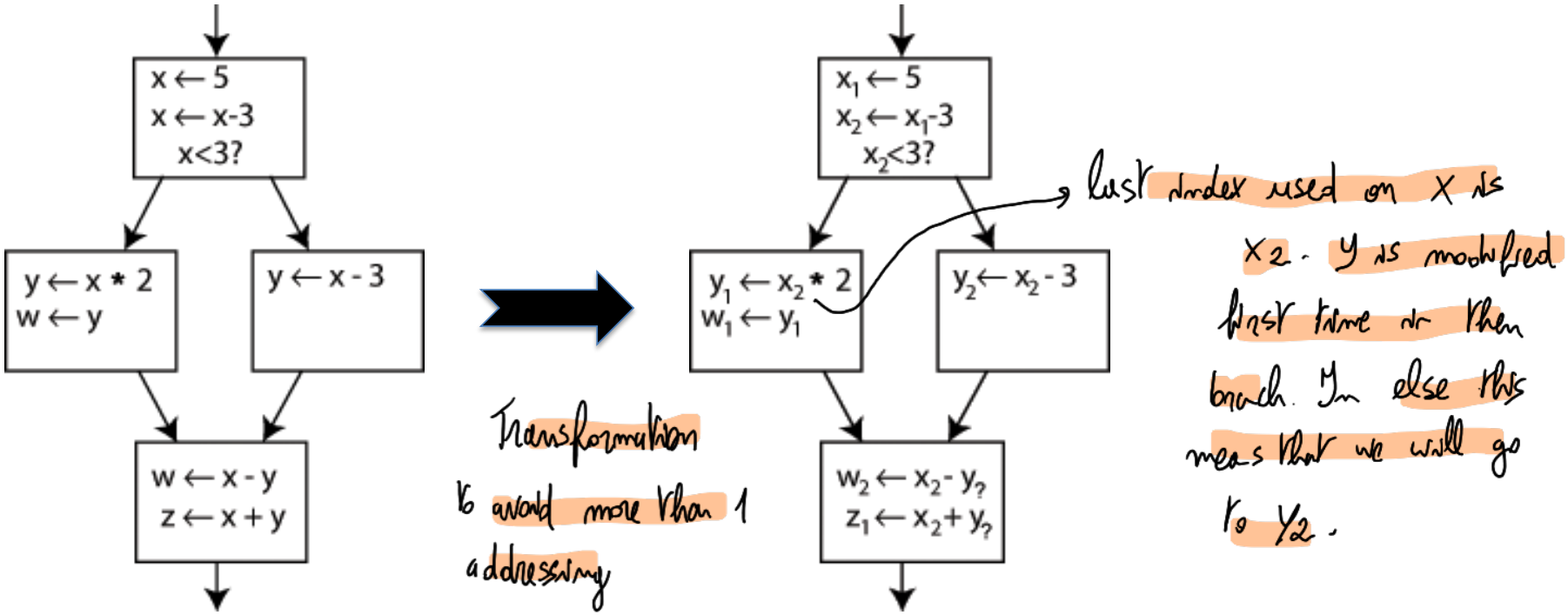
Static analysis **does not report a possible taint flow**

In back end of compiler we have SSA: possibility to assign variables only once. Lot of theory behind, but you can always transform a program with multiple assignment in one in which you have this property.

Static Single Assignment (SSA)

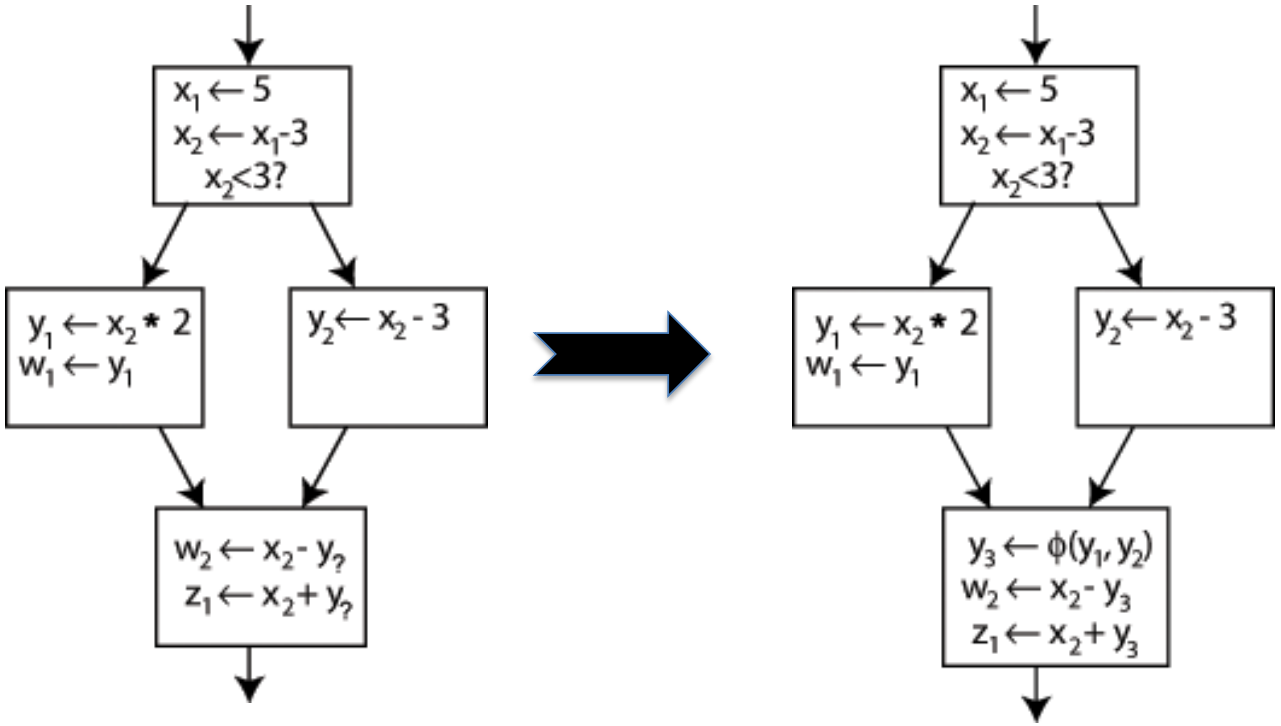
- SSA is a way of structuring the intermediate representation (IR) of programs so that **every variable is assigned exactly once** and and **every variable is defined before it is used**
- Intuition: Existing variables in the original IR are split into versions, new variables typically indicated by the original name with a subscript, so that every definition gets its own version.
- This is formally equivalent to continuation-passing style (CPS) translation

SSA Example



y in the bottom block could be referring to either y_1 or y_2,

SSA Example



Φ (Phi) function generates a new definition of *y* called y_3 by "choosing" either y_1 or y_2, depending on the control flow.

y_3 is the new variable we have, returned by ϕ that follows out either y_1 or y_2 depending value of condition

(1) Compiler can insert code to compute ϕ at runtime

SSA: discussion

- Given an arbitrary control-flow graph, it can be difficult to tell where to insert Φ functions, and for which variables.
  - this general question has an efficient solution that can be computed using a concept called *dominance frontiers*
- A compiler can implement a Φ function by inserting "move" operations at the end of every predecessor block.
  - The compiler might insert a move from y_1 to y_3 at the end of the left block and a move from y_2 to y_3 at the end of the right block.

SSA

- The LLVM Compiler Infrastructure uses SSA form
- The GNU Compiler Collection makes extensive use of SSA.
- Oracle's HotSpot Java Virtual Machine uses an SSA-based intermediate language in its JIT compiler.
- Microsoft Visual C++ compiler (2015 Update) uses SSA

Strategy several compilers use. So we will use SSA in our programs

Without SSA, if you see `x = ...` multiple times, you don't even know *where* the ambiguity starts. With SSA, you know: "Ah, `x3 = φ(x1, x2)`—this is the *exact point* where taintedness could come in, depending on the control flow." It gives you a **clean structure** for your analysis to walk through.

Also, some static taint analyzers use **path-sensitive** analysis, where they try to track under which conditions a value would be tainted or not. So even though they can't know exactly which branch will execute, they might keep track like:

- "x3 is tainted *if* condition is true."

And if later on that `condition` is used again, or can be reasoned about, the analyzer might sharpen its guesses.

So no, SSA doesn't magically solve the taint ambiguity—but it turns the problem from "ugh, where did this taint come from?" into "okay, here's the exact place where control flow merges and taint status splits." That's a massive help for keeping the analysis **precise but manageable**.

Multiple conditionals

```
int printfun(untainted  string) { … } ;
tainted   string getsFromNetwork(….);

void f (int x) {
    string y;
    If (x) y = ”ciao”
    else  y = getsFromNetwork()
    if (x)  printfun(y);
}
```

y becomes untainted

y would become tainted

If we combine results, y is tainted and untainted.

y = untainted

Multiple conditionals

```
int printfun(untainted  string) { … } ;
tainted   string getsFromNetwork(….);

void f (int x) {
    α string y;
    If (x) y = ”ciao”
    else  y = getsFromNetwork()
    if (x)  printfun(y);
}
```

y = tainted

Multiple conditionals

```
int printfun(untainted  string) { … } ;
tainted   string getsFromNetwork(….);

void f (int x) {
    α string y;
    If (x) y = ”ciao”
    else  y = getsFromNetwork()
    if (x)  printfun(y);
}
```

y = untainted

y = tainted

We would say that y is tainted because we take the union

We use an exception, but we only print if x is true, and if x is true, y is untainted.

Multiple conditionals

```
int printfun(untainted  string) { … } ;
tainted   string getsFromNetwork(….);

void f (int x) {
    α string y;
    If (x) y = ”ciao”
    else  y = getsFromNetwork()
    if (x)  printfun(y);
}
```

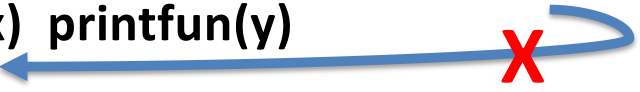
y = untainted

y = tainted

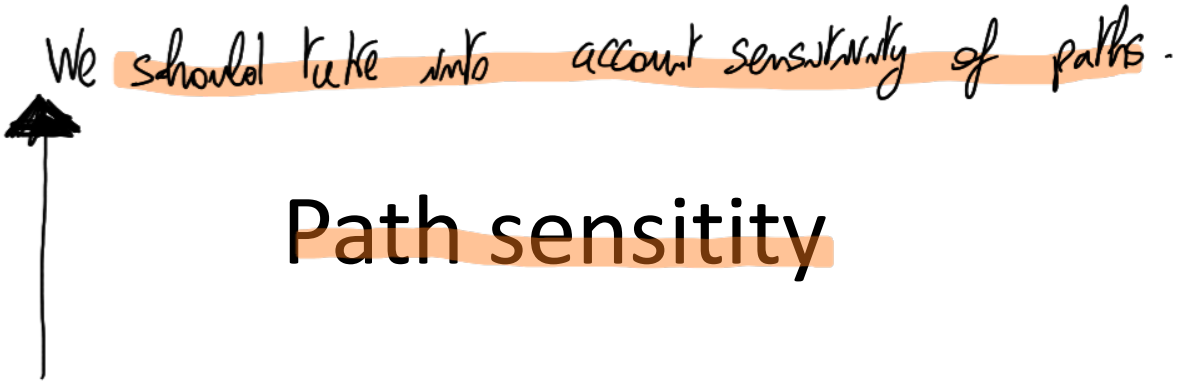
No solution

Multiple conditionals

```
int printfun(untainted  string) { … };
tainted   string getsFromNetwork(….);

void f (int x) {
    α string y;
    If (x) y = ”ciao”
    else  y = getsFromNetwork()
    if (x)  printfun(y)
}
```



No solution
False Alarm: because of the conditions on the guard



Path sensitity

- The problem is that the constraints we generates do not correspond to **feasible paths** (i.e. **feasible executions**)

- **Solution**: *We develop an analysis which considers the feasibility of paths when generating constraints*

EXERCIZE

A simple while program

```
string input = source();
char[] buffer = new char[32];
int i = 0;

 while (i < input.length() && i < 32) {
    buffer[i] = input.charAt(i);
     i = i + 1;
 }

 string userInput = new String(buffer);
grantAccess(userInput);
}
```

Not considering path sensitivity.

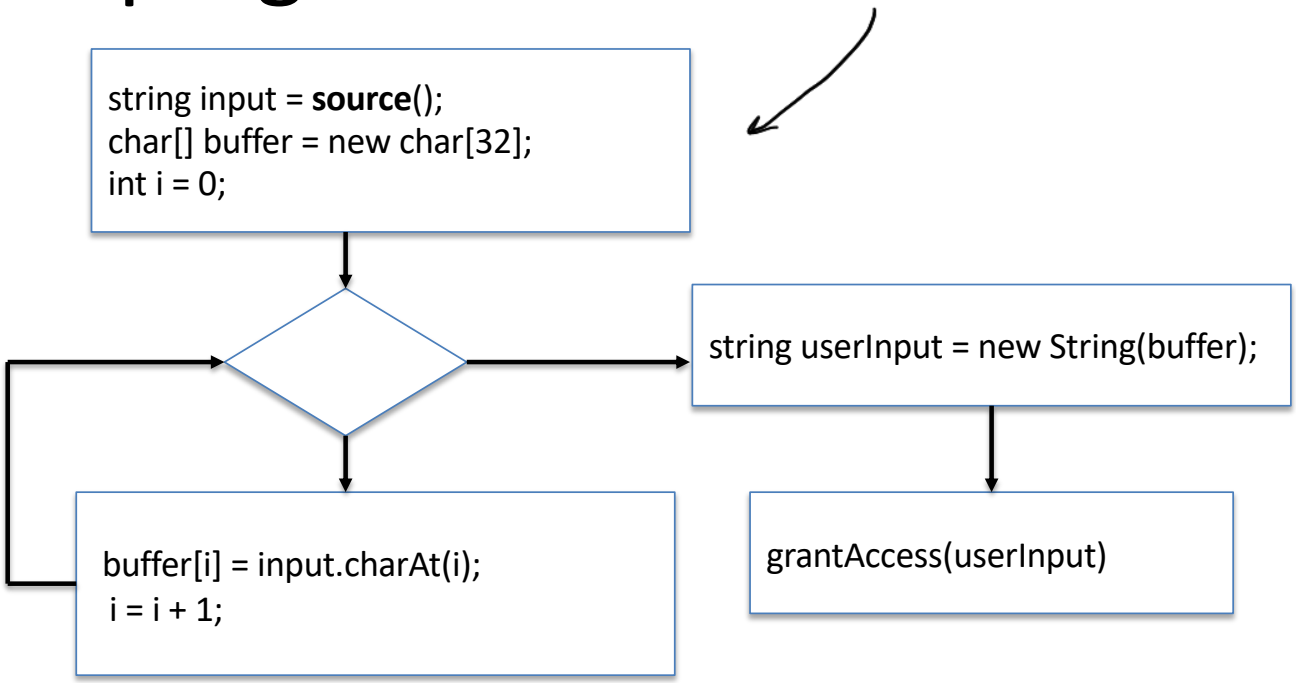
transform buffer into a string

↳ Something like a password

GOAL: Determine if tainted data may reach grantAccess

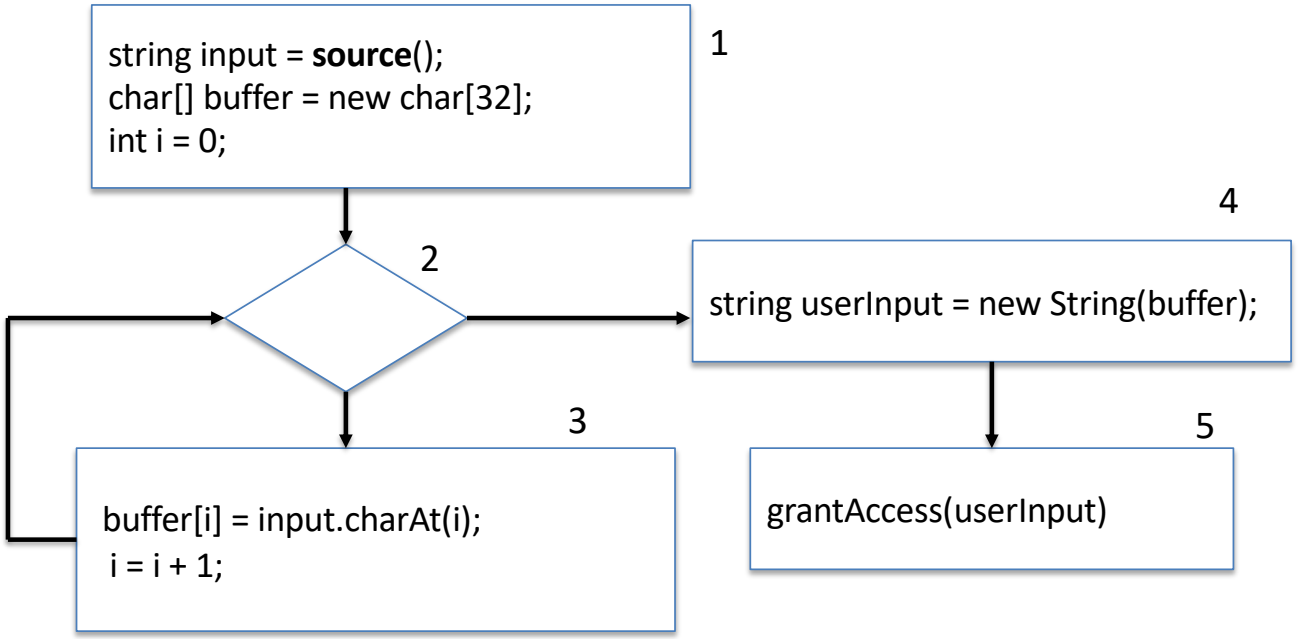
A simple while program and its CFG

```
string input = source();
char[] buffer = new char[32];
int i = 0;

 while (i < input.length() && i < 32) {
    buffer[i] = input.charAt(i);
     i = i + 1;
 }

 string userInput = new String(buffer);
grantAccess(userInput);
}
```

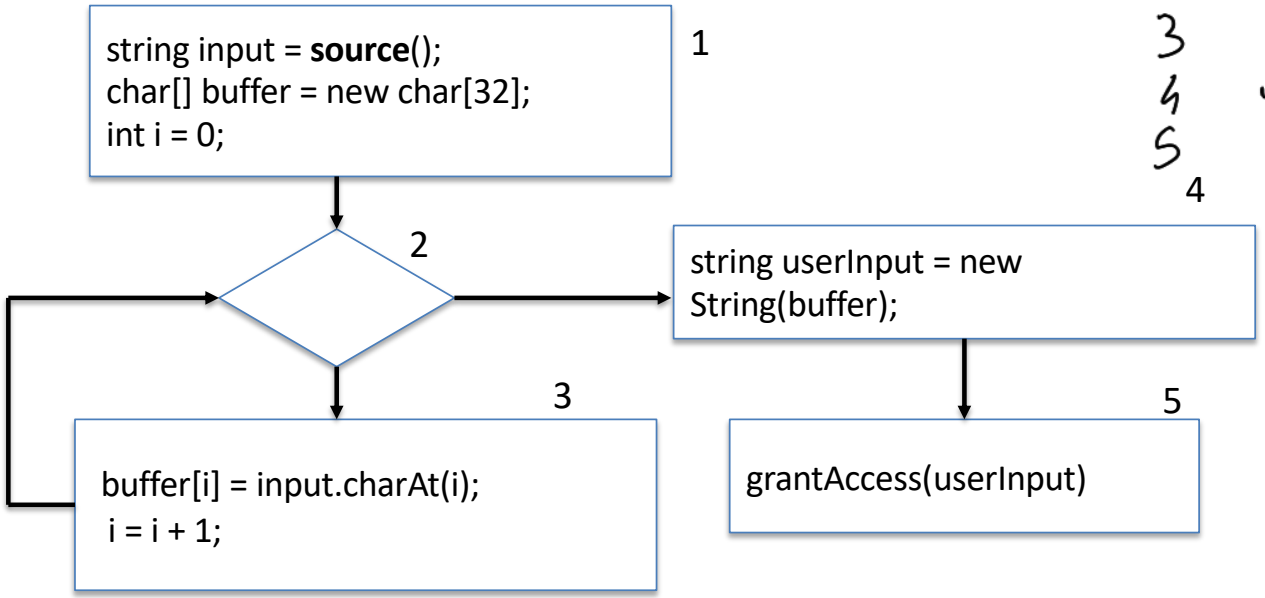


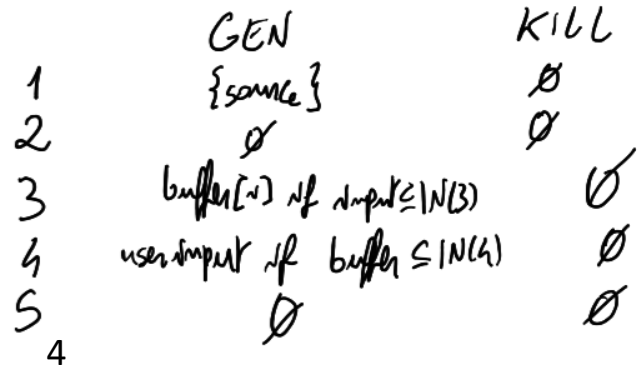
GOAL: Determine if tainted data may reach grantAccess

The CFG



GOAL: Determine if tainted data may reach grantAccess at block 5





BASIC BLOCK	GEN SET	KILL SET
1	{input}	Ø
2	Ø	Ø
3	$buffer[i] \in GEN(3)$	Ø
4	{userInput, buffer} $\subseteq GEN(4)$	Ø
5	Ø	Ø

The Constraints

IN	OUT
IN(1) = ??	OUT(1) =??
IN(2) = ??	OUT(2) = ??
IN(3) = ??	OUT(3) = ??
IN(4) = ??	OUT(4) = ??
IN(5) = ??	OUT(5) = ??

The Constraints

IN	OUT
IN(1) = Ø	OUT(1) = { input }
IN(2) = ??	OUT(2) = ??
IN(3) = ??	OUT(3) = ??
IN(4) = ??	OUT(4) = ??
IN(5) = ??	OUT(5) = ??

	GEN	KILL
1	{input}	∅
2	∅	∅
3	buffer[n] if input ⊆ IN(3)	∅
4	user input if buffer ⊆ IN(4)	∅
5	∅	∅

$$OUT(S) = (IN(S) - KILL(S)) \cup GEN(S)$$

IN(1)	∅
IN(2)	OUT(1) ∪ OUT(3)
IN(3)	OUT(2)
IN(4)	OUT(2)
IN(5)	OUT(4)

OUT(1)	{input}
OUT(2)	IN(2)
OUT(3)	IN(3) ∪ {buffer} if input ⊆ IN(3)
OUT(4)	IN(4) ∪ {user input} if buffer ⊆ IN(4)
OUT(5)	IN(5)

IT. 0

	IN(S)	OUT(S)
1	∅	∅
2	∅	∅
3	∅	∅
4	∅	∅
5	∅	∅

IT. 1

	IN(S)	OUT(S)
1	∅	{input}
2	∅	∅
3	∅	∅
4	∅	∅
5	∅	∅

IT. 2

	IN(S)	OUT(S)
1	∅	{input}
2	{input}	∅
3	∅	∅
4	∅	∅
5	∅	∅

IT. 3

	IN(S)	OUT(S)
1	∅	{input}
2	{input}	{input}
3	∅	∅
4	∅	∅
5	∅	∅

It. 4

	IN(s)	OUT(s)
1	$\emptyset$	{input}
2	{input}	{input}
3	{input}	$\emptyset$
4	{input}	$\emptyset$
5	$\emptyset$	$\emptyset$

It. 5

	IN(s)	OUT(s)
1	$\emptyset$	{input}
2	{input}	{input}
3	{input}	{input, buffer}
4	{input}	{input}
5	$\emptyset$	$\emptyset$

It. 6

	IN(s)	OUT(s)
1	$\emptyset$	{input}
2	{input, buffer}	{input}
3	{input}	{input, buffer}
4	{input}	{input}
5	{input}	$\emptyset$

It. 7

	IN(s)	OUT(s)
1	$\emptyset$	{input}
2	{input, buffer}	{input, buffer}
3	{input}	{input, buffer}
4	{input}	{input}
5	{input}	{input}

It. 8

	IN(s)	OUT(s)
1	$\emptyset$	{input}
2	{input, buffer}	{input, buffer}
3	{input, buffer}	{input, buffer}
4	{input, buffer}	{input}
5	{input}	{input}

It. 9

	IN(s)	OUT(s)
1	$\emptyset$	{input}
2	{input, buffer}	{input, buffer}
3	{input, buffer}	{input, buffer}
4	{input, buffer}	{input, buffer, usedinput}
5	{input}	{input}

It. 9

	IN(s)	OUT(s)
1	$\emptyset$	{input}
2	{input, buffer}	{input, buffer}
3	{input, buffer}	{input, buffer}
4	{input, buffer}	{input, buffer, usedinput}
5	{input, buffer, usedinput}	{input}

It. 10

	IN(s)	OUT(s)
1	$\emptyset$	{input}
2	{input, buffer}	{input, buffer}
3	{input, buffer}	{input, buffer}
4	{input, buffer}	{input, buffer, usedinput}
5	{input, buffer, usedinput}	{input, buffer, usedinput}

IN(1)	$\emptyset$
IN(2)	OUT(1) ∪ OUT(3)
IN(3)	OUT(2)
IN(4)	OUT(2)
IN(5)	OUT(4)

OUT(1)	{input}
OUT(2)	IN(2)
OUT(3)	IN(3) ∪ {buffer} if input ⊆ IN(3)
OUT(4)	IN(4) ∪ {user input} if buffer ⊆ IN(4)
OUT(5)	IN(5)