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 ($\cup$)**

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**

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

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?

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).

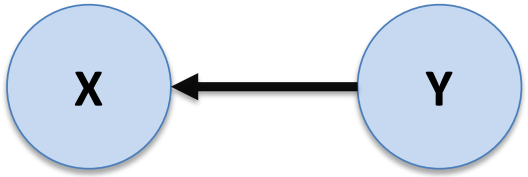
From Dynamic to Static taint tracking

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

(Static) Intuition: typing flow of assignments

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)

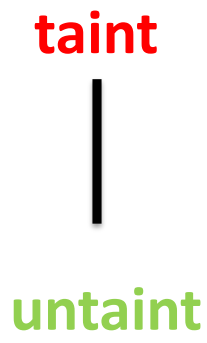


FLOW

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

TYPING RULE

(Static) Intuition: typing flow of assignments



The degree of taint lattice

(Static) Intuition: typing flow of assignments

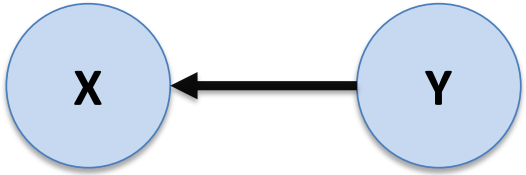
taint

untaint

The degree of Taint lattice

X := Y

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

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

TYPING RULE

(Static) Intuition: typing flow of assignments

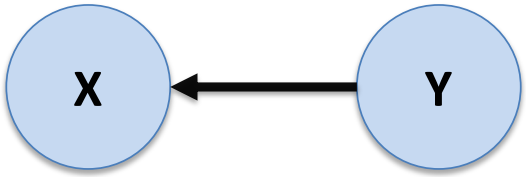
taint

|

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

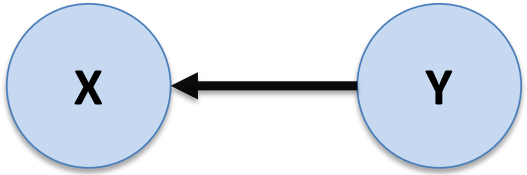
|

untaint

The degree of Taint lattice

$$\tau_y = \boldsymbol{taint}$$

$$\tau_x = \boldsymbol{untaint}$$



FLOW

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

The ingredients of the analysis

The analysis by examples

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

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

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

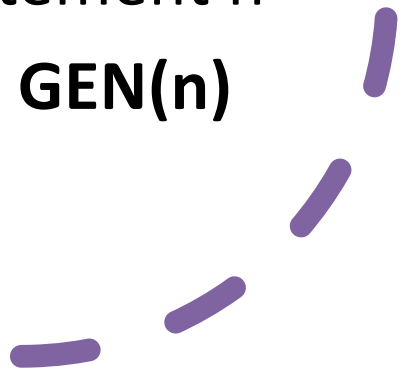
Ingredient 1: the analysis

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

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 }

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

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

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 ∈ OUT(1) then x ∈ GEN(2)
KILL(2) = ∅

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

IN(3) = OUT2) = { name, x }
GEN(3) = ∅
KILL(2) = ∅

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

Key Issue: Flow Sensitivity in Static Taint Analysis

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 $y \in GEN(s)$

- 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}

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

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.

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

The Monotone Framework

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

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 $\widehat{+}$ $\boldsymbol{v}$ = ***tainted***
$\boldsymbol{v}$ $\widehat{+}$ ***tainted*** = ***tainted***

Example

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

$$[1] = [name = \boldsymbol{tainted}]$$
$$[2] = JOIN(2)[x = eval(JOIN(2), name)] = [1][\, x = \ eval([1], name)]$$
$$= [name = \boldsymbol{tainted}][eval(x = \ [name = \boldsymbol{tainted}](name)]$$
$$= [name = \boldsymbol{tainted}][x \ = \boldsymbol{tainted}] = [name = \boldsymbol{tainted}, x = \boldsymbol{tainted}]$$

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)
```

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

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

```
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)
```

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

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

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

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

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

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

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

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

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

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

```
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!

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(2);
```

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

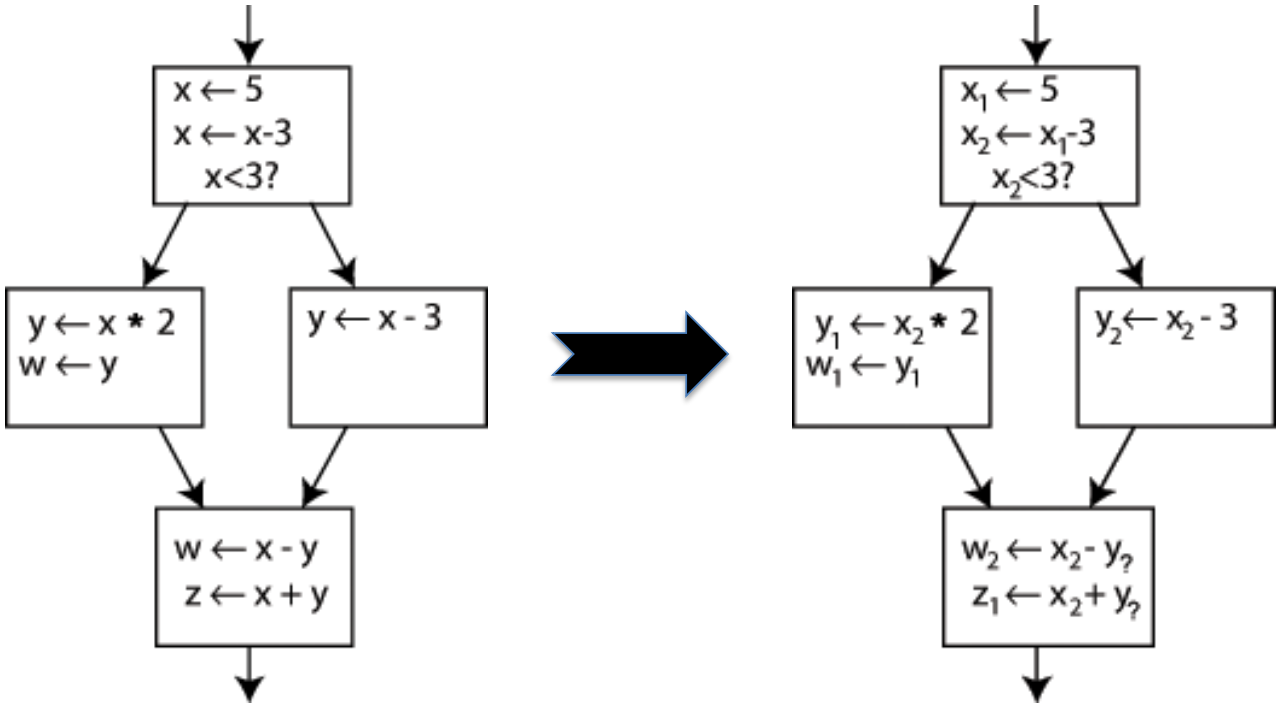
Variable x2 **is not tainted** at the sink (line 6)

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

Static Single Assignment (SSA)

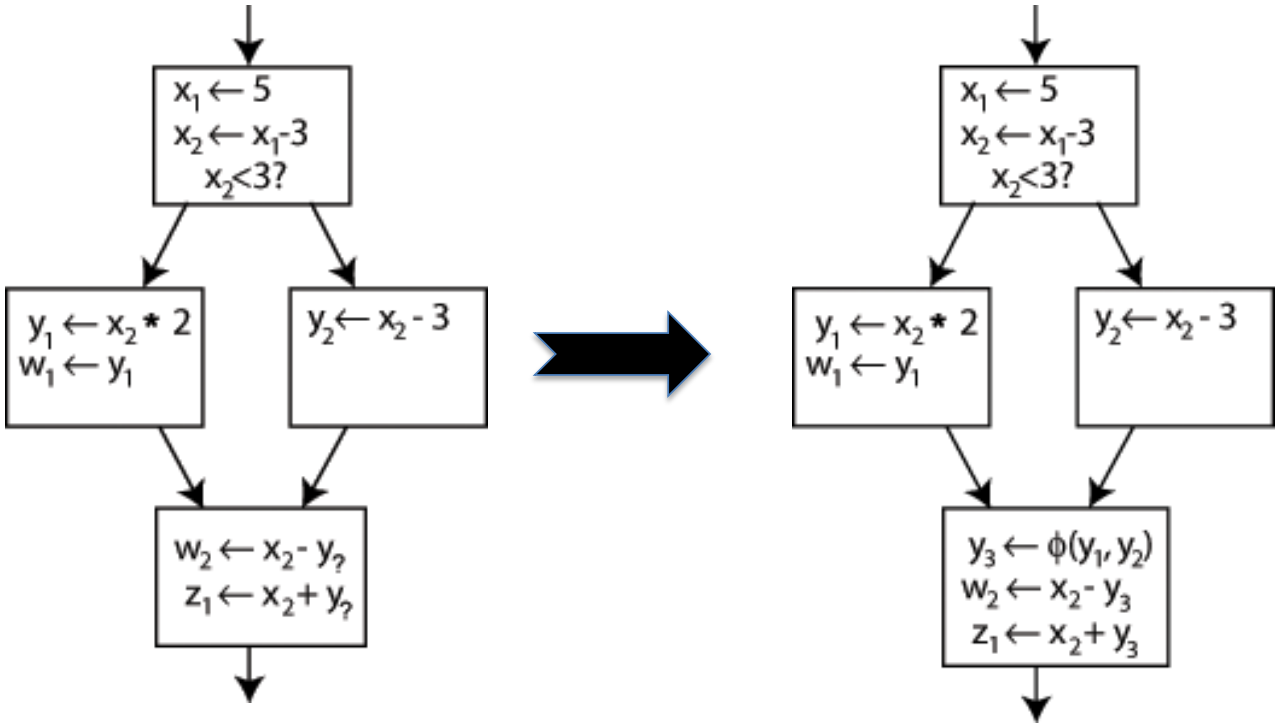
- SSA is a way of structuring the intermediate representation (IR) of programs so that **every variable is assigned exactly once** and and **every variable is defined before it is used**
- Intuition: Existing variables in the original IR are split into versions, new variables typically indicated by the 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.

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

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 = 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

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);
}
```

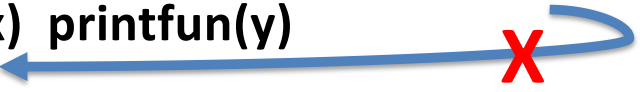
y = untainted

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*

TEST

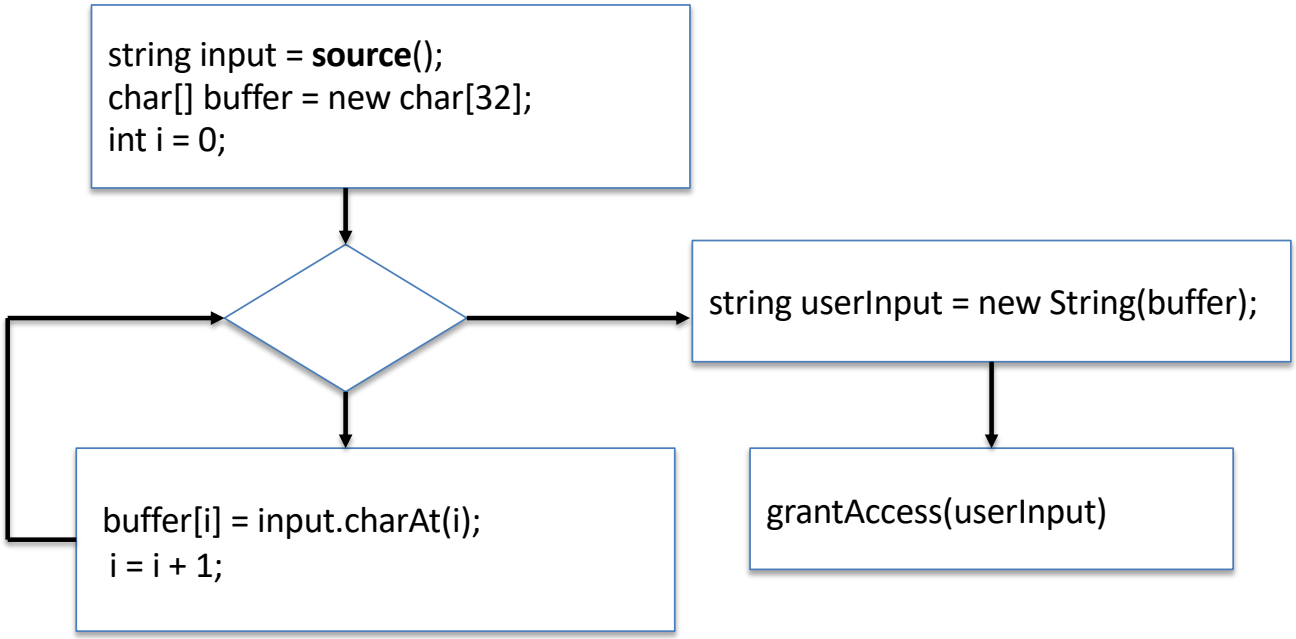
A simple while program

```
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);
}
```

GOAL: Determine if tainted data may reach grantAccess

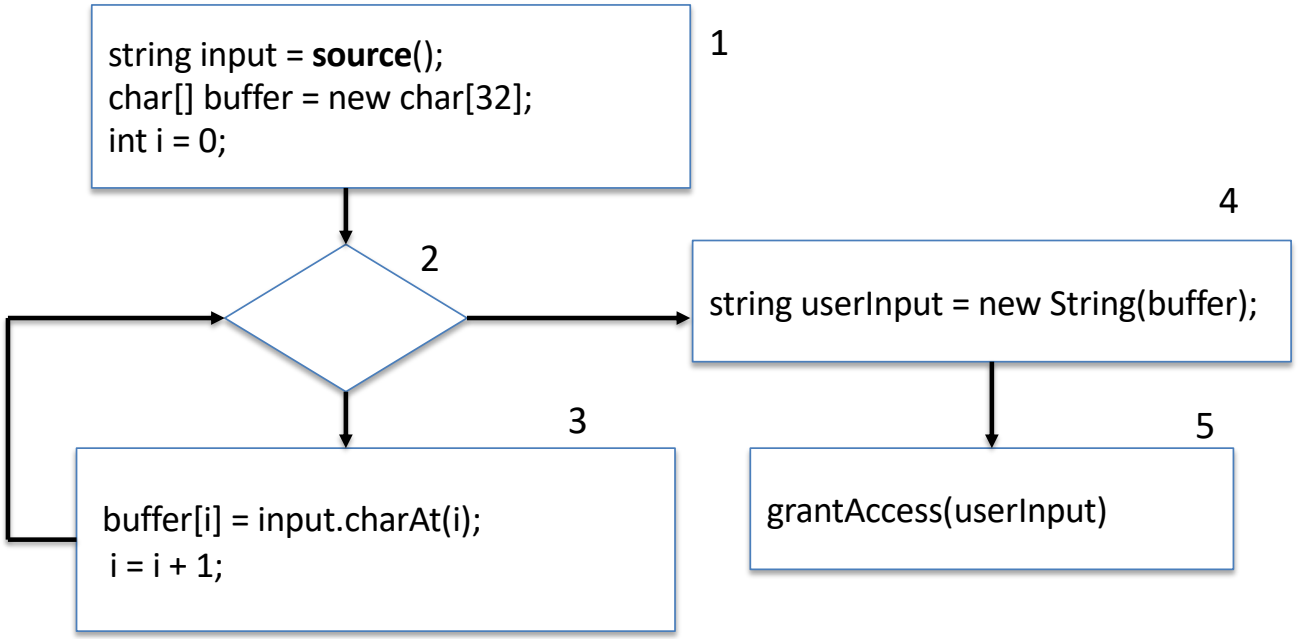
A simple while program and its CFG

```
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);
}
```

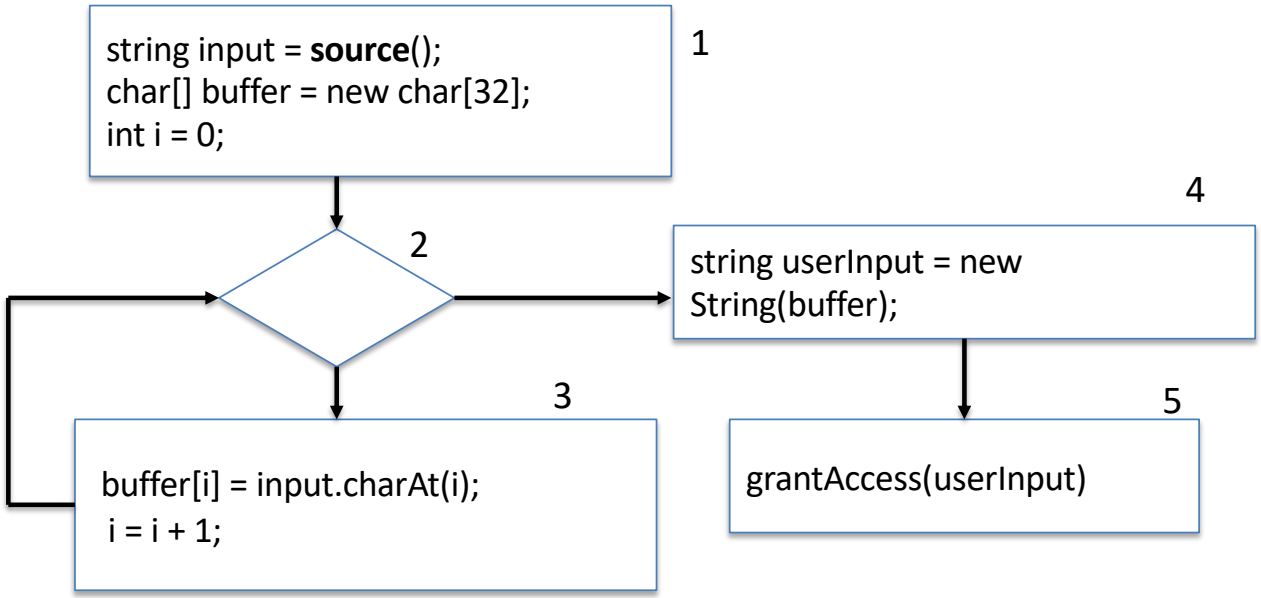


GOAL: Determine if tainted data may reach grantAccess

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) = ??

	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}

- Abstraction step: if I have a buffer and each of the entries of buffer is tainted, then buffer is tainted. This is safe because now buffer as a whole is given as input. So we say "because all the elements of buffer are tainted then buffer is tainted".

The Constraints

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

The Constraints

IN	OUT
IN(1) = ∅	OUT(1) = (IN(1) – KILL(1)) ∪ $\mathrm{GEN}(1)$
IN(2) = OUT(1) ∪ OUT(3)	OUT(2) = (IN(2) – KILL(2)) ∪ $\mathrm{GEN}(2)$
IN(3) = OUT(2)	OUT(3) = (IN(3) – KILL(3)) ∪ $\mathrm{GEN}(3)$
IN(4) = OUT(2)	OUT(4) = (IN(4) – KILL(4)) ∪ $\mathrm{GEN}(4)$
IN(5) = OUT(4)	OUT(5) = (IN(5) – KILL(5)) ∪ $\mathrm{GEN}(5)$

The Constraints

IN	OUT
IN(1) = Ø	OUT(1) = (Ø – Ø)) ∪ {input}
IN(2) = OUT(1) ∪ OUT(3)	OUT(2) = (IN(2) – KILL(2)) ∪ GEN(2)
IN(3) = OUT(2)	OUT(3) = (IN(3) – KILL(3)) ∪ GEN(3)
IN(4) = OUT(2)	OUT(4) = (IN(4) – KILL(4)) ∪ GEN(4)
IN(5) = OUT(4)	OUT(5) = (IN(5) – KILL(5)) ∪ GEN(5)

The Constraints

IN	OUT
IN(1) = Ø	OUT(1) = (Ø – Ø)) U {input}
IN(2) = {input} U OUT(3) = {input}	OUT(2) = {input}
IN(3) = {input}	OUT(3) = (IN(3) – KILL(3)) U GEN(3)
IN(4) = OUT(2)	OUT(4) = (IN(4) – KILL(4)) U GEN(4)
IN(5) = OUT(4)	OUT(5) = (IN(5) – KILL(5)) U GEN(5)

The Solution

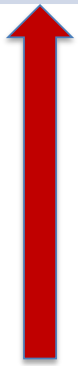
IN	OUT
IN(1) = Ø	OUT(1) = {input}
IN(2) = {input, buffer}	OUT(2) = {input, buffer}
IN(3) = {input, buffer}	OUT(3) = {input, buffer}
IN(4) {input, buffer}	OUT(4) = {input, buffer, userInput}
IN(5) = {input, buffer, userInput}	OUT(5) = {input, buffer, userInput}

The Solution

IN	OUT
IN(1) = Ø	OUT(1) = {input}
IN(2) = {input, buffer}	OUT(2) = {input, buffer}
IN(3) = {input, buffer}	OUT(3) = {input, buffer}
IN(4) {input, buffer}	OUT(4) = {input, buffer, userInput}
IN(5) = {input, buffer, userInput}	OUT(5) = {input, buffer, userInput}

The Constraints

IN	OUT
IN(1) = Ø	OUT(1) = {
IN(2) = OUT(1) ∪ OUT(3)	OUT(2) = (IN(2) – KILL(2)) ∪ GEN(2)
IN(3) = OUT(2)	OUT(3) = (IN(3) – KILL(3)) ∪ GEN(3)
IN(4) = OUT(2)	OUT(4) = (IN(4) – KILL(4)) ∪ GEN(4)
IN(5) = OUT(4)	OUT(5) = (IN(5) – KILL(5)) ∪ GEN(5)



WARNING

The algorithm

```
Out(s) := Gen(s) for all basic block s
W := {all basic block}
repeat {
  take s from W
  In(s) := U s' ∈ pred(s) Out(s')
  temp := Gen(s) U (In(s) - Kill(s))
  if (temp != Out(s)) {
    Out(s) := temp
    W := W U succ(s)}
} until W = empty
```

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*

Path sensitivity: a running example

```
1: string name = source();    // taint source
2: string x;
3: if (...) {
4:    x = name;
5: } else {
6:    x = "ciao";
7: }
8: sink(x);                   // potential taint sink
```

We analyze each path through the program separately:

Path P1 (True branch):

1 → 2 → 3 → 4 → 7 → 8

First flow of execution

Path sensitivity

```
1: string name = source();    // taint source
2: string x;
3: if (...) {
4:    x = name;
5: } else {
6:    x = "ciao";
7: }
8: sink(x);                   // potential taint sink
```

We analyze each path through the program separately:

Path P2 (False branch):

1 → 2 → 3 → 5 → 6 → 7 → 8

Second possible flow of execution

Path sensitivity

```
1: string name = source();    // taint source
2: string x;
3: if (...) {
4:    x = name;
5: } else {
6:    x = "ciao";
7: }
8: sink(x);                   // potential taint sink
```

We analyze each path through the program separately:

Path P1 (True branch):

1 → 2 → 3 → 4 → 7 → 8

Path P2 (False branch):

1 → 2 → 3 → 5 → 6 → 7 → 8

We maintain separate **IN/OUT sets** per block **per path**.

```
1: string name = source();    // taint source
2: string x;
3: if (...) {
4:    x = name;
5: } else {
6:    x = "ciao";
7: }
8: sink(x);                   // potential taint sink
```

GEN/KILL SETS

Line	Statement	GEN	KILL
1	name = source();	{ name}	Ø
2	string x;	Ø	Ø
4	x = name;	{ x } if name tainted	Ø
6	x = "ciao";	Ø	{ x }
8	sink(x);	Ø	Ø

Step by step analysis

```
1: string name = source();    // taint source
2: string x;
3: if (...) {
4:    x = name;
5: } else {
6:    x = "ciao";
7: }
8: sink(x);                   // potential taint sink
```

Lines 1–2 (both paths)

Line 1:

$$IN(1) = \emptyset$$

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

Line 2:

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

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

Step by step analysis

```
1: string name = source();    // taint source
2: string x;
3: if (...) {
4:    x = name;
5: } else {
6:    x = "ciao";
7: }
8: sink(x);                   // potential taint sink
```

True branch (path P1)

we do not generate new paths (not C assumption)

IN(4) = OUT(3) = OUT(2) = { name }

Since name ∈ IN(4), then GEN(4) = { x }

KILL(4) = Ø

OUT(4) = { name, x }

IN(P1:8) = OUT(4) = { name, x }

Since x ∈ IN(P1:8) !!!warning tainted value reaches sink

Path P1: x is tainted at the sink

Step by step analysis

```
1: string name = source();    // taint source
2: string x;
3: if (...) {
4:    x = name;
5: } else {
6:    x = "ciao";
7: }
8: sink(x);                   // potential taint sink
```

False branch (path P2)

IN(6)= OUT(3) = OUT(2) = { name }
Since "ciao" is a constant, variable x is overwritten with untainted value
GEN(6) = Ø, KILL(6) = { x }
OUT(6) = IN(6) – KILL(6) = { name) (no taint value on variable x)

IN(P2:8) = OUT(6) = { name }
Since x ∉ IN(P2:8)] then untainted value reaches sink

Path P2: x is untainted at the sink

SUMMARY of the function we have analyzed

```
1: string name = source();    // taint source
2: string x;
3: if (...) {
4:    x = name;
5: } else {
6:    x = "ciao";
7: }
8: sink(x);                   // potential taint sink
```

Path	vulnerability at sink?	Explanation
Path P1	Yes	Tainted name copied to x
Path P2	No	x overwritten with untainted value

Evaluation

- Extending the analysis with **path sensitivity**, allows us to **distinguish** that **sink(x)** is only unsafe on some paths.
- The (standard) conservative (safe) analyzer would still **report a potential taint violation** at **sink(x)**.
- But a **path-sensitive analyzer** could:
  - Mark the violation with path conditions (e.g., *only if true branch is taken*)
  - Use this precision to **reduce false positives**.

Analysis is definitely more precise than what we did before. But what is the strategy you have to define to perform this analysis? You need to keep track of feasibility and conditions for reaching a certain path. We need a precise strategy

Path sensitivity

A **path-sensitive analysis** tracks data flow facts depending on the path taken

- Path often represented by which branches of conditionals taken

A **path-sensitive analysis** can reason more accurately about correlated conditionals

The issue: how can we make a path sensitive analysis?

- a dataflow analysis where we track facts for each possible path

Handling Path Sensitivity

Approach: The analysis exploits **path-sensitive predicates** in order to associate taint facts with **symbolic conditions** representing the execution paths.

Incorporate symbolic conditions that represent the path you are referring to the taint facts that we are gathering

Symbolic Condition: Path predicates

- Use **path predicates** to:
  - **Distinguish** between different execution paths
  - Track **conditional taint facts** of variables
  - Evaluate whether **taint values** may reach the **sink** under specific conditions

Path condition: conditions to be satisfied in order to reach that point.
Question now is: is sink reached by tainted value? Yes/No depending on the conditions for path traversal: so we will say "sink is reached by tainted value if..."

Conditional taint facts

- We associate each taint fact with a **path predicate** φ (a symbolic condition).
- A path predicate φ represents **under what condition** a taint fact holds.
- We perform a **forward symbolic propagation** of of pairs

$$(\varphi, x)$$

meaning "variable x is tainted under condition φ "

Taint facts: set of tainted variables. To take into account paths we define φ, path predicate representing conditions under which a certain fact about taintness of a variable holds.

Our running example

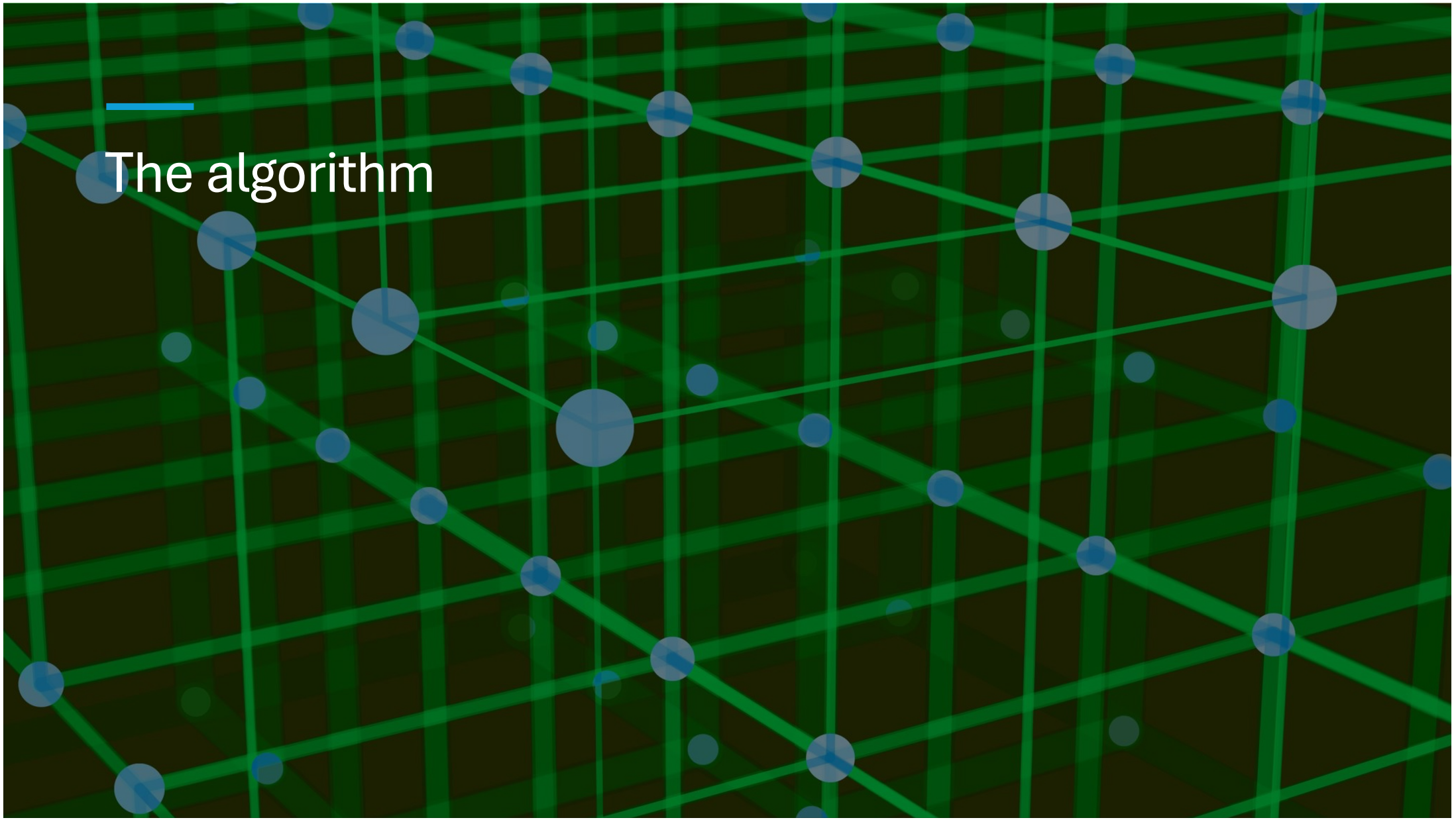
```
1: string name = source();    // taint source
2: string x;
3: if (...) {
4:    x = name;
5: } else {
6:    x = "ciao";
7: }
8: sink(x);                   // potential taint sink
```

Path Conditions:

φ_1 condition for then branch

φ_2 condition for else branch

We know in this case $\varphi_2 = !\varphi_1$.


The algorithm

Design ideas

- The algorithm keeps **separate taint facts for each path** by attaching them to **path predicates**
- This allows us to express properties of the form:
  - Variable x is tainted if φ holds
  - Variable x is untainted if φ holds
- At a sink, the algorithm checks:
  - **Does any taint fact of the form (φ, x) hold?**
- If yes then a taint value may reach sink under that path condition

We look out if taint values reach sinks under certain paths

The (ideal) data structures

TaintFacts: a set of pairs (φ, x), where:
φ is a Boolean expression (path condition)
x is a program variable — set of program variables

It is a group of different analyses for each path.

PathCond
symbolic conditions accumulated during analysys traversal

- I have 2 data structs: 1 collecting Taint facts, a set of pairs (path condition, program variable)
  X is tainted if φ is true.
- Path conditions: conditions to be true to follow a path (that taints x)
  conditions that identify paths to which TaintFacts refer to

Some auxiliary routines

ROUTINE: GEN/KILL TOGETHER

Takes known facts, statement and path condition φ

gen_kill(s, φ, facts) = <**gen(s, φ, facts)**, **kill(s, φ, facts)** >:

Computes which taint facts are added or removed at node s, under condition φ

[φ is given because it is needed to enrich facts]

↳ two modes of CFG

edgeCondition(n1, n2):

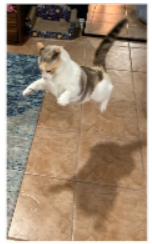
Returns the condition required to take the conditional branch (e.g., φ_1, φ_2)

Restrict(facts, φ): Takes a set of taint facts and a path condition, and remove from the set, all the taint facts that do not satisfy φ.

Keeps only facts that are consistent with the current path predicate

Used to establish if taints we have are compatible with our current node reachable with path conditions φ.

GO, GO, GO!



```
X = source()
if X > 0 then
    x = 5
else
    x = x + 1
if x > 0 then
    y = 0
else
    y = x

sink(y)
```

$(true, \{x\})$ PATH 1,2(,3,4)

$(x>0 \wedge true, \emptyset) = (x>0, \emptyset)$ PATH 1

$(x \leq 0, \{x\})$ PATH 2

TWO PATHS

$(x>0 \wedge x>0, \emptyset)$ PATH 1(,3)

$(x>0 \wedge x \leq 0, \{x\})$ — False → unfeasible path

$(x \leq 0 \wedge x>0, \emptyset)$ — false = unfeasible path

$(x \leq 0 \wedge x \leq 0, \{x, y\})$ PATH 2(,4)

$T = \{ \text{PATH1}(x>0, \emptyset), \text{PATH2}(x \leq 0, \{x, y\}) \}$ RESTRICT$(T, x>0) = \{(x>0, \emptyset)\}$

T contains two taint facts associated to paths.

We stick with one path and bring with us the conditions to follow that path and memorize tainted variables. We do this for all paths. That's it.

The (pseudo) code

Input:
Program **P** with control-flow graph CFG **G**

Output: Taint facts (with path conditions) at each program point

Initialize:
  For each node n in G:
    TaintIn[n] = Ø
    TaintOut[n] = Ø

Worklist = [(entry_node, true)] // pair of node and path predicate

↓ we start with entry node, and path condition is true because of course we follow always that path.

The code

```
while Worklist is not empty:
    (n,φ) = Worklist.pop()

// Step 1: Compute TaintIn[n] under condition φ

in_facts = ∪ [ Restrict(TaintOut[p], φ ∧ edgeCondition(p,n))
               for p in pred(n) ]

// Step 2: Apply GEN/KILL for this node under φ

gen_set, kill_set = gen_kill(n, φ, in_facts)
out_facts = (in_facts - kill_set) ∪ gen_set
```

We take a node

1 2

↳ We take info about incoming known facts, but we filter with current path conditions + conditions to get to n.

Note: Union of predecessors: merging of info we have with different paths

Look at example

The code

```
while Worklist is not empty:
        (n,φ) = Worklist.pop()

// Step 1: Compute TaintIn[n] under condition φ

in_facts = ∪ [ Restrict(TaintOut[p], φ ∧ edgeCondition(p,n))
                for p in pred(n) ]

// Step 2: Apply GEN/KILL for this node under φ

  gen_set, kill_set = gen_kill(n, φ, in_facts)
  out_facts = (in_facts - kill_set) ∪ gen_set
```

JUST DON'T THINK TOO MUCH ABOUT IT

The code

```
// Step 3: Update and propagate
if out_facts ≠ TaintOut[n]:
  TaintOut[n] = TaintOut[n] ∪ out_facts
  for s in succ(n):
    φ' = φ ∧ edgeCondition(n,s)
    Worklist.push((s, φ'))
Return TaintOut[n] for each node n
```

if out has been updated, we update

(put the ones not already there)

Take successors in the working list

and take conditions to get there

(3)

We consider a specific path

and for different paths we should add different elements in worklist?

Supported by fact that we do Union here!

You should look at one path at a time since you have one path condition

Our running example

```
1: string name = source();    // taint source
2: string x;
3: if (...) {
4:    x = name;
5: } else {
6:    x = "ciao";
7: }
8: sink(x);                   // potential taint sink
```

Path Conditions:

φ_1 condition for then branch

φ_2 condition for else branch

TaintIn[8] = {

(φ_1, {name, x}), // x is tainted in true branch

(φ_2 , {name}) // x is safe in false branch

}

An example

```
1:  string name = source();       // taint source
2:  string x;
3:  string log = "";

4:  if (name.length() > 0) {      // φ1
5:     x = name;
6:  } else {                      // φ2 = ¬φ1
7:     x = "default";
8:  }

9:  int i = 0;
10: while (i < x.length()) {      // φ3. (loop condition)
11:    log = log + x.charAt(i);   // potential use of tainted x
12:    i = i + 1;
13: }

14: sink(log);                    // sink
```

Step-by-Step Analysis with Path Predicates

Meta-notation

The notation

taint(x)

is used to indicate that variable x belongs to the set of tainted variables

The notation

!taint(x)

is used to indicate that variable x does npot belong top the set of tant variables (i.e. it has to be removed from that set)

Line 1: name = source();

Always executed (path predicate always true)

(true, taint(name))

```
1:  string name = source();       // taint source
2:  string x;
3:  string log = "";

4:  if (name.length() > 0) {      // φ1
5:     x = name;
6:  } else {                      // φ2 = ¬φ1
7:     x = "default";
8:  }

9:  int i = 0;
10: while (i < x.length()) {      // φ3. (loop condition)
11:    log = log + x.charAt(i);   // potential use of tainted x
12:    i = i + 1;
13: }

14: sink(log);                    // sink
```

Line 1: name = source();

Always executed (path predicate always true)

(true, taint(name))

Line 4–8: Conditional Assignment to x

True Branch (φ_1 = name.length() > 0)

Line 5: x = name

Taint from name propagates to x

(φ_1 ,taint(x))

False Branch ($\varphi_2 = \neg\varphi_1$)

Line 7: x = "default"

Overwrites x with untainted value **!taint(x)**

Removes variable x from taint facts

```
1:  string name = source();       // taint source
2:  string x;
3:  string log = "";

4:  if (name.length() > 0) {      // φ1
5:     x = name;
6:  } else {                      // φ2 = ¬φ1
7:     x = "default";
8:  }

9:  int i = 0;
10: while (i < x.length()) {      // φ3. (loop condition)
11:    log = log + x.charAt(i);   // potential use of tainted x
12:    i = i + 1;
13: }

14: sink(log);                    // sink
```

Line 1: name = source();
Always executed (path predicate always true)
(true, taint(name))

Line 4–8: Conditional Assignment to x
True Branch (φ_1 = name.length() > 0)
Line 5: x = name
Taint from name propagates to x
(φ_1 ,taint(x))
False Branch ($\varphi_2 = \neg\varphi_1$)
Line 7: x = "default"
Overwrites x with untainted value **!taint(x)**
Removes variable x from taint facts

Lines 9–13: Loop Copies x into log
We treat the loop as **symbolic propagation**. We reason symbolically
Assume:
- On every iteration, one character from x is appended to log
- We abstract the loop effect as: "log becomes tainted if x is tainted"

At each iteration we have 1 character. Assumption:
log becomes tainted as soon as we encounter 1 char tainted.

```
1:  string name = source();       // taint source
2:  string x;
3:  string log = "";

4:  if (name.length() > 0) {      // φ1
5:     x = name;
6:  } else {                      // φ2 = ¬φ1
7:     x = "default";
8:  }

9:  int i = 0;
10: while (i < x.length()) {      // φ3. (loop condition)
11:    log = log + x.charAt(i);   // potential use of tainted x
12:    i = i + 1;
13: }

14: sink(log);                    // sink
```

Line 1: name = source();

Always executed (path predicate always true)

(true, taint(name))

Line 4–8: Conditional Assignment to x

True Branch (φ_1 = name.length() > 0)

Line 5: x = name

Taint from name propagates to x

(φ_1 ,taint(x))

False Branch ($\varphi_2 = \neg\varphi_1$)

Line 7: x = "default"

Overwrites x with untainted value **!taint(x)**

Removes variable x from taint facts

Lines 9–13: Loop Copies x into log

From φ_1 branch:

$$(\varphi_1 \wedge \varphi_3 , taint(x)) \Longrightarrow (\varphi_1 \wedge \varphi_3 , taint(log))$$

From φ_2 branch:

$$(\varphi_2 \wedge \varphi_3, !taint(x)) \Longrightarrow !taint(log))$$

```
1:  string name = source();       // taint source
2:  string x;
3:  string log = "";

4:  if (name.length() > 0) {      // φ1
5:     x = name;
6:  } else {                      // φ2 = ¬φ1
7:     x = "default";
8:  }

9:  int i = 0;
10: while (i < x.length()) {      // φ3. (loop condition)
11:    log = log + x.charAt(i);   // potential use of tainted x
12:    i = i + 1;
13: }

14: sink(log);                    // sink
```

$$\varphi_2 = \neg \varphi_1$$

$$\varphi_3 = i < x.length()$$

$T = \{(true, \{name\})\}$

$T = \{(\varphi_1, \{x, name\}), (\varphi_2, \{name\})\}$

```
1:  string name = source();       // taint source
2:  string x;
3:  string log = "";

4:  if (name.length() > 0) {      // φ1
5:     x = name;
6:  } else {                      // φ2 = ¬φ1
7:     x = "default";
8:  }

9:  int i = 0;
10: while (i < x.length()) {      // φ3. (loop condition)
11:    log = log + x.charAt(i);   // potential use of tainted x
12:    i = i + 1;
13: }

14: sink(log);                    // sink
```

Line 1: name = source();

Always executed (path predicate always true)

(true, taint(name))

Line 4–8: Conditional Assignment to x

True Branch (φ_1 = name.length() > 0)

Line 5: x = name

Taint from name propagates to x

(φ_1 ,taint(x))

False Branch ($\varphi_2 = \neg\varphi_1$)

Line 7: x = "default"

Overwrites x with untainted value **!taint(x)**

Removes variable x from taint facts

Lines 9–13: Loop Copies x into log

From φ_1 branch:

$$(\varphi_1 \wedge \varphi_3, taint(x)) \Longrightarrow (\varphi_1 \wedge \varphi_3, taint(log))$$

From φ_2 branch:

$$(\varphi_2 \wedge \varphi_3, !taint(x)) \Longrightarrow (\varphi_2 \wedge \varphi_3, !taint(log))$$

Line 14: sink(log);

From φ_1: $(\varphi_1 \wedge \varphi_3, taint(log))$ **sink receive taint data**

From φ_2 : $(\varphi_2 \wedge \varphi_3, !taint(log))$ sink is safe

$$\varphi_2 = \neg\varphi_1$$

$$\varphi_3 = i < x.length()$$

You can also consider $\varphi_4 = \neg\varphi_3$,

so while not executed.

Line	Path Predicate	GEN	KILL
1	true	{ name }	∅
2–3	true	∅	∅
5	φ_1= (name.length() > 0)	{ x }	∅
7	$\varphi_2 = \neg\varphi_1$	∅	{ x }
10–13	φ_3(loop condition)	if x ∈ IN ⇒ { log }	∅
14	—	∅	∅

SUMMARY

Path Condition	Line 14	Explanation
$\boldsymbol{\varphi_1 \wedge \varphi_3}$	**Tainted**	*x* = *name*, loop propagates
$\boldsymbol{\neg\varphi_1 \wedge \varphi_3}$	**Untainted**	*x* = *"default"*, log remains clean

Remark

The **path-sensitive analysis** shows that *sink(log)* receives **tainted data only if**:

- name.length() > 0 (φ_1)
- and the loop executes (φ_3)

On other paths, the sink is safe

The analysis

- Avoids **false positives**
- Allows one to **generate warnings with path conditions**,

SUMMARY (AGAIN)

Path Condition	Line 14	Explanation
$\boldsymbol{\varphi_1 \wedge \varphi_3}$	**Tainted**	*x* = *name*, loop propagates
$\boldsymbol{\neg\varphi_1 \wedge \varphi_3}$	**Untainted**	*x* = *"default"*, log remains clean

Symbolic Taint Condition

A boolean φ_T that correspond to some conditions related to path traversal.

The analysis generates a **symbolic condition** under which taint reaches the sink:

If φ_T holds, we have taint reaching sink. You have to provide solution you need tainting solving constraint

$$Taint\ at\ sink \iff \varphi_T = \varphi_1 \wedge \varphi_3\ holds$$

$$\varphi_T = (name.length(\quad) > 0) \wedge (x.length(\quad) > 0)$$

What next?

Expoit Static taint analysis with **path predicates to determine the symbolic taint predicate**

Use a suitable solver to **determine if any path condition leading to taint(log) is satisfiable**

… but what is a suitable solver?

SMT Solver: theorem prover. SW which solves logical formulas. Their job is: give an input as a formula and solver tries to give a result.

SMT Solver

- **SMT (Satisfiability Modulo Theories)** solvers are software toolkits that takes **logical formulas** as input and determines whether these formulas are:
- **SAT** (satisfiable)
  - at least one assignment makes the formula true (for boolean variables)
- **UNSAT** (unsatisfiable)
  - no assignment satisfies the formula
- **UNKNOWN**
  - can't determine (e.g., due to undecidable theory or timeout)

SAT: Boolean Satisfiability Problem

What is the SAT Problem?

Input: A Boolean formula

Question: *Is there an assignment of true/false values to variables that makes the formula true?*

The problem proven **NP-complete** (Cook, 1971)

(x ∨ ¬y) ∧ (¬x ∨ z)

SAT if x = true, y = false, z = true Satisfied if ...

- No polinomial solution up to now

SAT is hard to solve!!

- The search space is **exponential**: 2^n possible assignments for n variables
- SAT is **NP-complete**
  - No known polynomial-time algorithm for all inputs
- Many real-world problems reduce to SAT:
  - Circuit verification
  - Program analysis static analysis
  - Scheduling, planning
- Even **small changes** to input can change the solution space drastically

Techniques to address SAT

How Modern SAT Solvers Behave

- **DPLL algorithm** and its modern variants
- **Heuristics** for variable selection
- **Preprocessing and simplification**

- Improvements done, modern solvers efficient and work well.
- To make several cases more tractable

- Solvers like **MiniSAT**, **Z3**, and **CryptoMiniSat** make SAT feasible in many cases

However … DLLP first

The **DPLL algorithm** (Davis–Putnam–Logemann–Loveland) is the foundation of most modern **SAT solvers**.

DPLL is a **complete backtracking-based search algorithm** for solving the **SAT problem**. It determines whether a **Boolean formula in CNF (conjunctive normal form)** is **satisfiable**.

DLLP improves brute-force search by introducing **smart pruning and inference techniques**.

In CNF, a formula is made up of **ANDs of ORs**. Think of it like this: you've got a bunch of clauses, and each clause is a group of statements connected by ORs, and then all the clauses are tied together with ANDs.

For example:

$$(A \vee \neg B) \wedge (B \vee C \vee \neg D) \wedge (\neg A \vee D)$$

How DLLP Works

Unit Propagation

- If a clause has **only one literal unassigned** (a *unit clause*), assign it in a way that satisfies the clause.
  - Clause: (x) the set x=true

Pure Literal Elimination (optional)

- If a literal appears with **only one polarity** (always positive or always negative) in all clauses, assign it to satisfy all such clauses.
  - If x appears only as x (never as ¬x), set x = true.

Variable Assignment (Decision Step)

- Pick an unassigned variable and try assigning true or false.

Backtracking

- If the current assignment leads to a **conflict** (unsatisfiable clause), **backtrack** and try the other assignment.
- If both fail, backtrack further.

1. Unit Propagation

This is like saying: *"Hey, there's a clause that has only one variable left unassigned. So we've got no choice—assign it in a way that satisfies the clause."*

Example:

- Clause: (x)
- Only literal is x, so to make this clause true, we **must** set x = true.

It's a kind of forced move. You're not making a decision here—you're following the only path that avoids contradiction.

2. Pure Literal Elimination (optional step)

Now you look at all the literals. If a variable only shows up **positively** (like x, x, x) or only **negatively** (¬x, ¬x, ¬x), you can safely assign it in a way that satisfies every clause it's in.

Example:

- If x appears only as **x** (never ¬x), set x = true and all those clauses become true.
- If x appears only as **¬x**, set x = false.

This step simplifies things without risk, but not all SAT solvers do it.

3. Variable Assignment (Decision Step)

Okay, now you make a real choice. You pick an unassigned variable and just try setting it to true or false. You're kind of guessing, but you'll use logic and backtracking to correct any wrong turns later.

---

4. Backtracking

If your guess (plus the consequences it triggered) ends up creating a conflict—like a clause that's now impossible to satisfy—you backtrack.

- First, try the **opposite** value of the variable you chose.
- If that also leads to failure, you **backtrack even further**, undoing earlier decisions, until you find a path that works—or until you've tried everything and proven it's unsatisfiable.

We can do step 1 (setting unassigned variable to true to make clause true) and step 2 (assigning all x to true to make all clauses true) because clauses are just ORs of variables, so one true makes all true, correct?

Exactly, Giovanni—nailed it.

Clauses in CNF are **ORs of literals**, so as long as **at least one** literal in the clause is true, the whole clause is satisfied. That's the key idea behind both **unit propagation** and **pure literal elimination**.

The CNF formula

Let's use this one (in CNF form, of course):

(¬A ∨ B) ∧ (¬B ∨ C) ∧ (¬C ∨ D) ∧ (¬D) ∧ (A ∨ ¬B ∨ C)

This one has 5 clauses and variables A, B, C, D.

Let's break it down clause by clause for clarity:

1. (¬A ∨ B)
2. (¬B ∨ C)
3. (¬C ∨ D)
4. (¬D)
5. (A ∨ ¬B ∨ C)

Step 1: Unit Propagation

Check if any clause is a unit clause (only one literal):

Clause 4 is: **(¬D)** → That's a unit clause → We *must* assign D = false.

Now that D = false, we propagate it.

Clause 3: (¬C ∨ D)
→ becomes (¬C ∨ **false**) → simplifies to (¬C)
Now **(¬C)** is a unit clause → so **C = false**

Now that C = false, propagate it:

Clause 2: (¬B ∨ C)
→ becomes (¬B ∨ **false**) → simplifies to (¬B)
→ Unit clause → **B = false**

Now propagate B = false:

Clause 1: (¬A ∨ B)
→ becomes (¬A ∨ **false**) → simplifies to (¬A)
→ Unit clause → **A = false**

Now all variables are assigned:

- A = false
- B = false
- C = false
- D = false

Let's now check all clauses:

1. (¬A ∨ B) → (true ∨ false) → **true**
2. (¬B ∨ C) → (true ∨ false) → **true**
3. (¬C ∨ D) → (true ∨ false) → **true**
4. (¬D) → (true) → **true**
5. (A ∨ ¬B ∨ C) → (false ∨ true ∨ false) → **true**

All clauses are satisfied!

We didn't even need pure literal elimination or decision steps or backtracking—**unit propagation alone** solved the whole thing for us. That's kinda cool, actually.

Want to try another one that *does* need a decision and backtracking?

> Can A, B etc be other CNFs or are they usualoy variables?

Nice question—so, Giovanni, in **CNF**, **A, B, C, etc. are typically variables** (or more precisely, **literals**, meaning either a variable or its negation, like A or ¬A).

Each clause in CNF is made of **ORs of literals**, and then the full formula is the **AND** of those clauses. That's the standard setup for SAT solvers and most logical systems using CNF.

No, they cannot

Z3 SAT SOLVER

- **Z3** is an **SMT** solver developed by Microsoft Research
  - https://github.com/Z3Prover/z3
- **Z3 Input**: A set of **declarative constraints**, often expressed in logic over various domains:
  - Integers, reals, booleans, bitvectors, arrays, strings, etc.
- Converts the problem into a combination of **Boolean SAT solving** + **theory solvers** (for strings, integers, arrays, etc.)
- Uses **efficient heuristics and decision procedures** to prune infeasible choices

Can take a lot of different constraints in terms of

Z3 in Action

Suppose you want to check if a tainted value might reach a sink **only if a certain condition is true**.

AND

Taint Fact = (x > 0 && x < 10)(x * 2 == 15) {x:int}

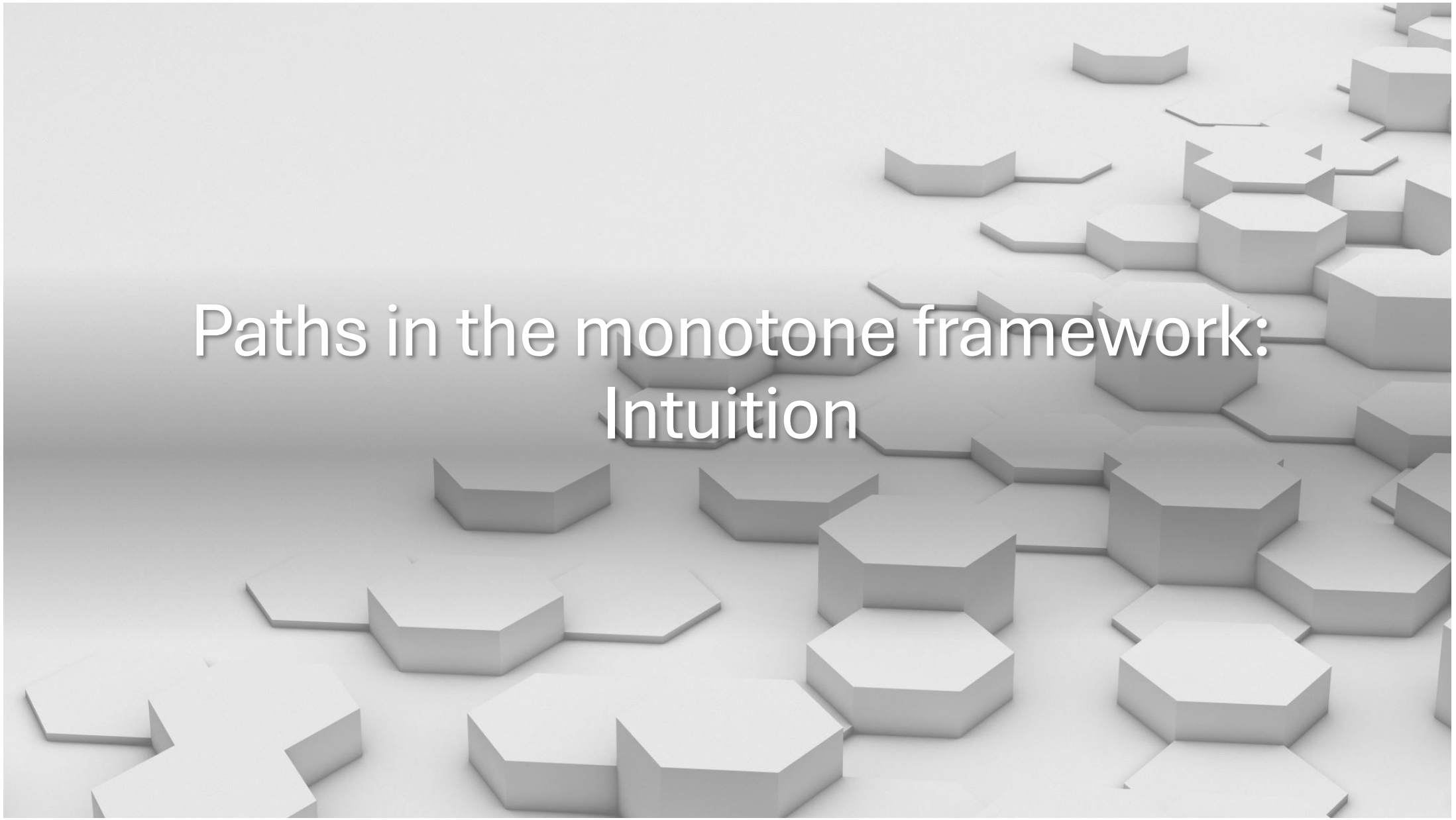
```
Z3>

(declare-const x Int)
     (assert (> x 0))
     (assert (< x 10))
     (assert (= (* x 2) 15))
(check-sat)
```

```
Z3 >

unsat
```

(S expressions)

The sink is unreachable under any input (unfeasible path)

The taint does not reach the sink


Paths in the monotone framework:
Intuition

The steps

We outline **how abstract states evolve** across **branches** in a **path-sensitive taint analysis**: **path predicates** and transfer functions are exploited to trace how taint information propagates and merges across control flow branches.

Three Steps:

1. **The Lattuce Model Setup**: Formal definitions
2. **Branch Evolution**: State splits, transfer, and joins
3. **Worked Example**: Formal trace of a branching program

The Model

Variables: $\boldsymbol{PVar = \{x, y, z \ \ldots.\}}$ is the set of program variables.

Abstract Taint Domain

A taint state $\boldsymbol{t = PVar \rightarrow \{untaint, taint\}}$ (where $untaint \ \leq taint$)

Path Condition Each abstract state is annotated with a **path predicate** φ (a Boolean formula)

Path-sensitive state $\boldsymbol{s = (\varphi, t)}$

The overall abstract state $\boldsymbol{S \in AS}$ is a **set of such path states**:

$$\boldsymbol{S = \{s_i = (\varphi_1, t_i \ for \ i \in I\}}$$

ABSTRACT STATES

$$\boldsymbol{AS = PATH \rightarrow PVar \rightarrow \{untaint \leq taint\}}$$

TRANSFER FUNCTION

For each command (aka basic block) **cmd** we introduce the transfer function

$$f_{cmd} : AS \to AS$$

$$\boldsymbol{f_{cmd}}(\boldsymbol{S}) = \bigcup \{\boldsymbol{f_{cmd}}((\boldsymbol{\varphi}, \boldsymbol{t}) \mid (\boldsymbol{\varphi}, \boldsymbol{t}) \in \boldsymbol{S}\}$$

Conditional

```
if (cond) {
    cmd1;
} else {
    cmd2;
}
```

$\forall(\varphi, t) \in S$ where S is the input abstract state

Splitting paths

Then Branch: $(\boldsymbol{\varphi \wedge cond, t})$

Else Branch: $(\varphi \wedge \neg\, cond, t)$

Conditional Merge Point

$\boldsymbol{S' = \{(\varphi \wedge cond, t_1), (\varphi \wedge \neg cmd, t_2)\}}$

Transfer Funtions

$\boldsymbol{f_{cmd1}(\varphi \wedge cond, t) = (\varphi \wedge cond, t_1)}$

$\boldsymbol{f_{cmd2}(\varphi \wedge \neg cmd, t) = (\varphi \wedge \neg cmd, t_2)}$

Conditional

```
if (cond) {
    cmd1;
} else {
    cmd2;
}
```

$\forall(\varphi, t) \in S$ where S is the input abstract state

Splitting paths

Then Branch: $(\boldsymbol{\varphi \wedge cond, t})$

Else Branch: $(\varphi \wedge \neg\, cond, t)$

Transfer Funtions

$\boldsymbol{f_{cmd1}(\varphi \wedge cond, t) = (\varphi \wedge cond, t_1)}$

$\boldsymbol{f_{cmd2}(\varphi \wedge \neg cmd, t) = (\varphi \wedge \neg cmd, t_2)}$

Conditional Merge Point

$\boldsymbol{S' = \{(\varphi \wedge cond, t_1), (\varphi \wedge \neg cmd, t_2)\}}$

We do not join

$\boldsymbol{t_1, t_2}$

preserving path sensitivity

```
1: x = input();                  // Tainted source
2: if (x > 0) {
3:     y = sanitize(x);          // y = untaint
4: } else {
5:     y = x;                    // y = taint
6: }
7: print(y);                     // Sink
```

```
1: x = input();                 // Tainted source
2: if (x > 0) {
3:     y = sanitize(x);         // y = untaint
4: } else {
5:     y = x;                   // y = taint
6: }
7: print(y);                    // Sink
```

S_7 = {
  (x > 0, {x= taint, y = untaint}),
  (¬(x > 0), {x = taint, y = taint})
}

The analysis issues a **conditional warning** only on the path where **x ≤ 0.**

Line	Path Predicate	Taint Status
1	true	{x = taint}
2	x > 0	{x = taint}
3	x > 0	{ x = taint, y = untaint}
4	¬(x > 0)	{x = taint}
5	¬(x > 0)	{ x = taint, y = taint}
6	—	Path merge
7	Both predicates	{x = taint, y = untaint or taint}

Wht's next?

An analysis that models only a single function at a time is intra-procedural

- **An analysis that takes multiple functions into account is inter-procedural**
- **An inter-procedural analysis that considers the calling context when analyzing the target of a function call is context-sensitive**