Taint Analysis … so far

- Taint analysis is a data flow analysis tracking how Information flows.
- **Goal: determine what data an attacker can control.**
- It requires knowing where information enters the program and how it moves through the program.
- **Ingredients:**
  - **Insert some a tag or label for data we are interested in**
  - **Track the influence of the tainted object along the execution of the program.**
  - **Obverse if it flows to sensitive functions.**

Taint Analysis … so far

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

α string name = getsFromNetwork(….)
β string x;
x = name
x = “ciao”;
printfun(x)

tainted $<=$ α
α $<=$ β
untainted $<=$ β
β $<=$ untainted

variable x is overridden

Constraints are unsolvable: illegal flow

Flow sensitivity

- The analysis we developed is **Flow Insentive**
  - The qualifier of each variable ***abstracts the taintness of all values*** it ever contains
- A **flow sensitive** analysis accounts for variables whose values may change
  - Each assignment has a different qualifier
    - The two assignment at x in our example would have two different qualifiers
  - Idea: **static single assignment (SSA)**

Static Single Assignment (SSA)

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

α **string name getsFromNetwork(....)**

β **string x1;**

γ **string x2**

x1 = name

x2 = "ciao";

printfun(x2)

tainted $<= \alpha$

$\alpha <= \beta$

untainted $<= \gamma$

$\gamma <=$ **untainted**

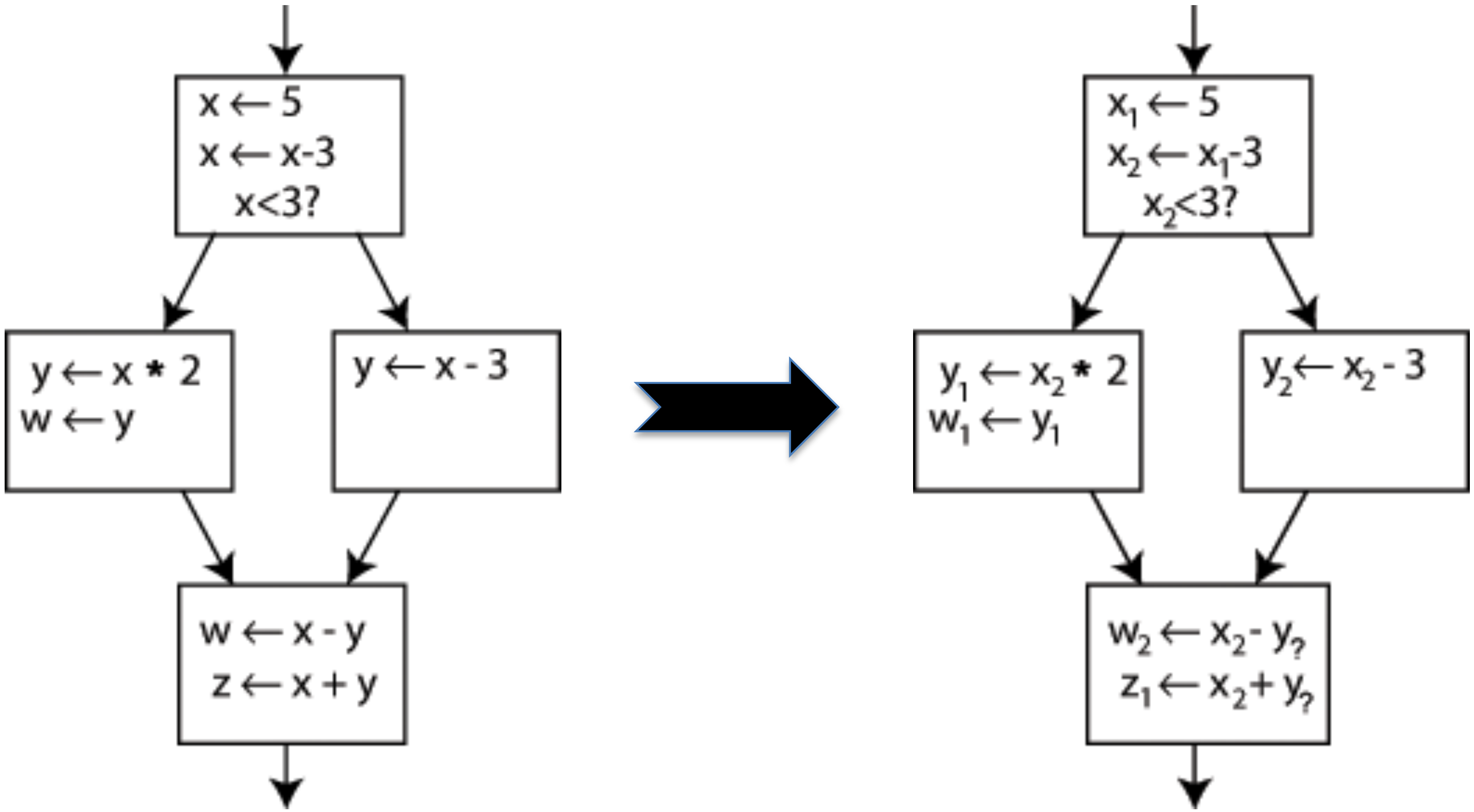
NO ALARM

Constraints are solvable: γ = **untainted** $\alpha = \beta$ = **tainted**

SSA Review

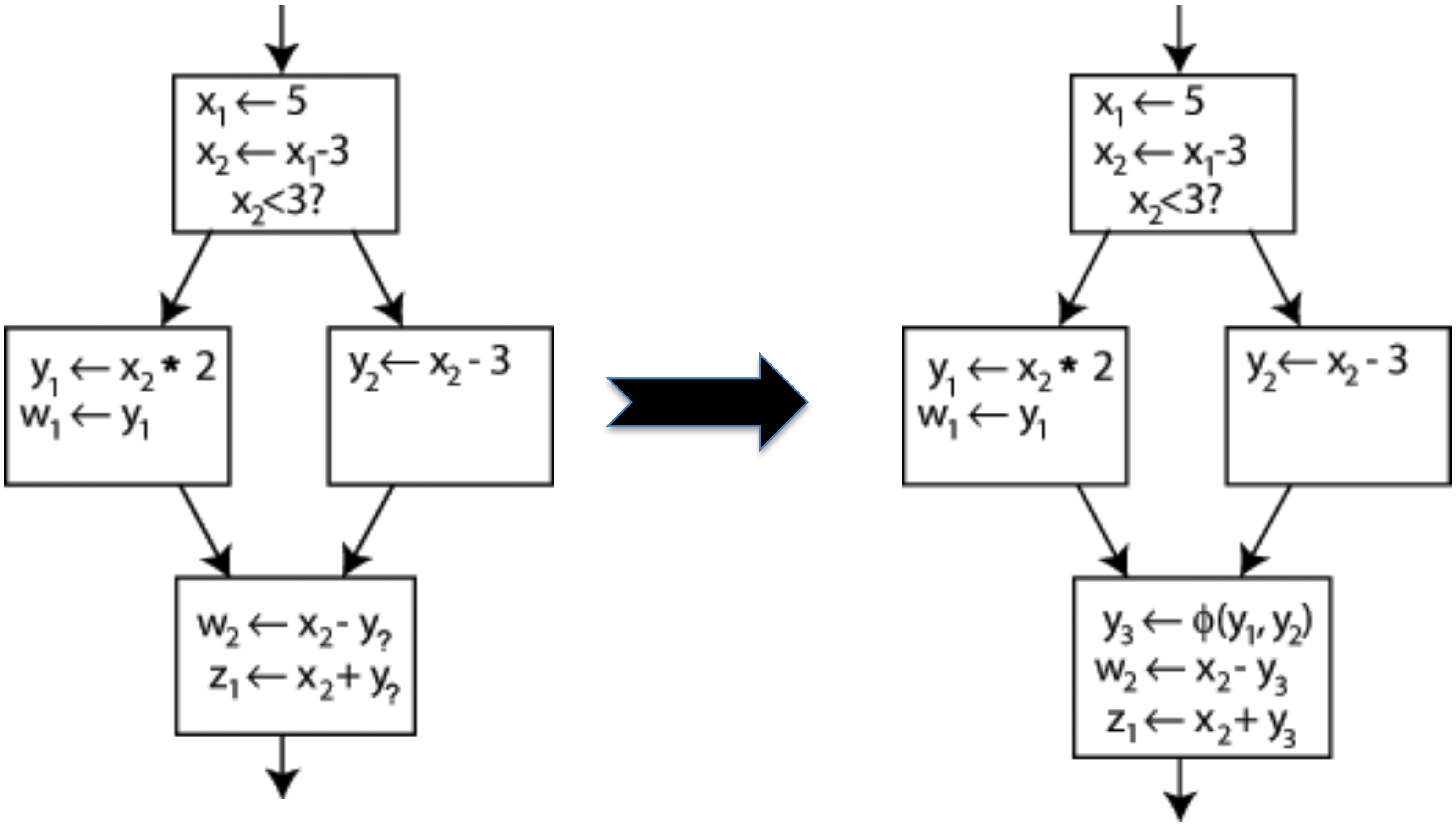
- SSA is a way of structuring the intermediate representation (IR) of programs so that **every variable is assigned exactly once** and and **every variable is defined before it is used**
- Intuition: Existing variables in the original IR are split into versions, new variables typically indicated by the 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) {
    α string y;
    If (x) y = ”ciao”
    else  y = getsFromNetwork()
    if (x)  printfun(y);
}
      untainted <= α
```

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);
}
```

untainted <= α

tainted <= α

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);
}
```

untainted <= α

tainted <= α

α <= 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);
}
```

untainted <= α

tainted <= α

α <= untainted

tainted <= α <= untainted

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)
}
```

tainted <= α <= untainted

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*

Path Sensitivity

The analysis considers execution path sensitity

```
void f (int x) {
    bool y;
    (1) If (x) (2) y = "ciao"
    else  (3) y = getsFromNetwork()
    (4) if (x)  (5) printfun(y)
(6)}
```

Execution paths

1-2-4-5-6 when x = true

1-3-4-6 when x = false

Path

1-3-4-5-6 is infeasible

Path Sensitivity

The analysis considers execution path sensitity

```
void f (int x) {
    string y;
    (1) If (x) (2) y = "ciao"
    else  (3) y = getsFromNetwork()
    (4) if (x)  (5) printfun(y)
(6)}
```

Execution paths
1-2-4-5-6 when x = true
1-3-4-6 when x = false

Path
1-3-4-5-6 is infeasible

Path senstitive analysis estends with a path condition the constraints

x = true => untainted <= α \\path segment 1-2
x = false => tainted <= α \\path segment 1-3
x = true => α <= untainted \\path segment 4-5

Path sensitity

- Path sensitivity makes the analysis more precise (**good new!!!**)
- Path sensitivity makes the constraint solver more difficult (**bad new!!!)**
  - Increase the number of nodes in the constraint graphs
  - Require a more general solver to handle path conditions
- Issue: precision vs scalability

Nest step

We focused on tracking tainted flows through blocks of code of normal statements.

Now we consider how we handle tainted flows to function calls.

Handling function calls

```
string a = gestFromNetwork();
string b = id(a)
```

```
  string id(string x) {
  return x ;
}
```

A client program takes a value from the network, passes it to the a server function (the identity server function)
The server function returns the result into the variable b.

Our goal: to see wether or not there is a tainted flow in this program; need to track the flow into the id function and back out again

Handling function calls

```
α string a = gestFromNetwork();
β string b = id(a)
```

```
δ char id(γ string x) {
  return x ;
}
```

Methodological step: we need to give flow qualifiers to the argument (γ) and the return value of the function (δ).

Why?

Handling function calls

(1)

```
α string a = gestFromNetwork();
β string b = id(a)
```

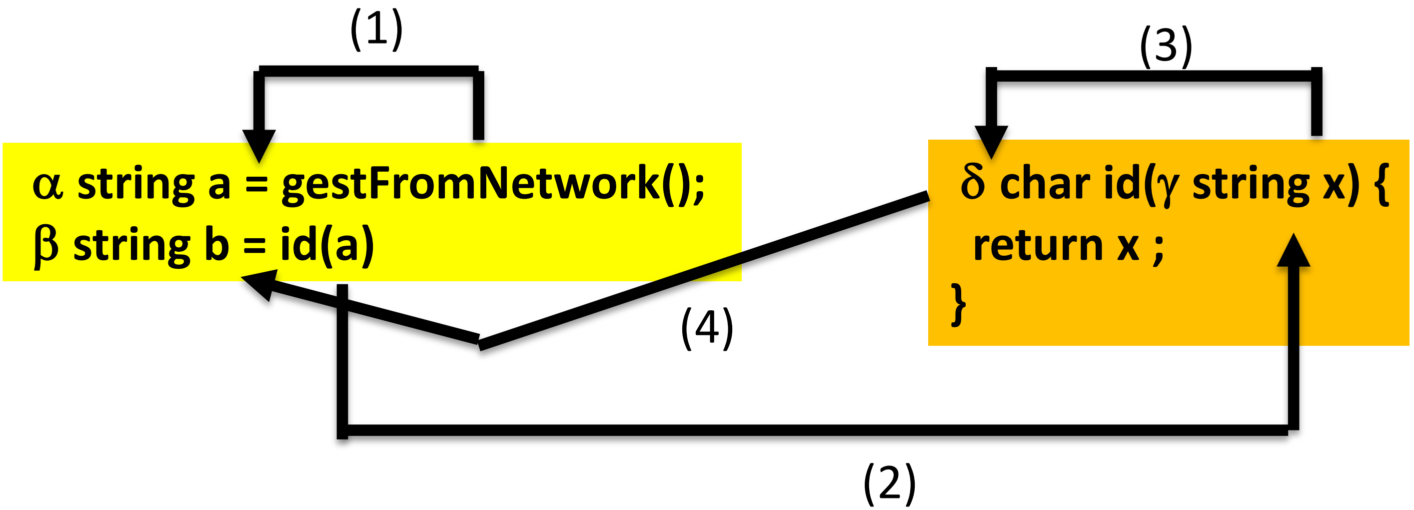
```
δ char id(γ string x) {
  return x ;
}
```

(2)

(1) tainted <= α

(2) α <= γ

Handling function calls



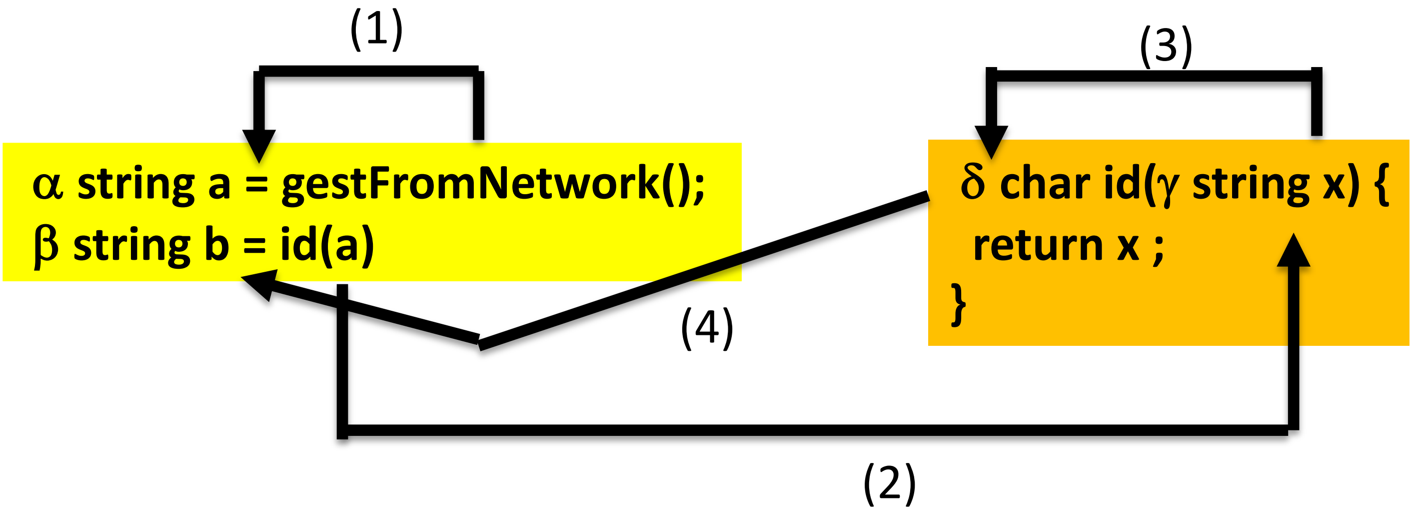
(1) **tainted** <= α

(2) α <= γ

(3) γ <= δ

(4) δ <= β

Handling function calls



(1) tainted <= α

(2) α <= γ

(3) γ <= δ

(4) δ <= β

MIMIKING THE CONTROL FLOW GRAPH!!!

Variable b is tainted!!

function calls

```
α string a = gestFromNetwork();
β string b = id(a);
ρ string c = “ciao”;
printfun(c)
```

```
δ string id(γ string x) {
  return x ;
}
```

tainted $\le \alpha$

$\alpha \le \gamma$

$\gamma \le \delta$

$\delta \le \beta$

untainted $\le \rho$

$\rho \le$ untainted

function calls

```
α string a = gestFromNetwork();
β string b = id(a);
ρ string c = "ciao";
printfun(c)
```

```
δ string id(γ string x) {
  return x ;
}
```

tainted $<=$ α

$\alpha <= \gamma$

$\gamma <= \delta$

$\delta \quad <= \beta$

untainted $<=$ ρ

$\rho <=$ untainted

No Alarm

Solution

$\rho =$ untainted \\c

$\alpha = \beta = \gamma = \delta =$ tainted \\b

Two calls to the same function

```
α string a = gestFromNetwork();
β string b = id(a);
ρ string c = id("ciao");
printfun(c)
```

```
δ string id(γ string x) {
  return x ;
}
```

If we were to run this program and the prior program, they would have exactly the same outcome.

Two calls to the same function

```
α string a = gestFromNetwork();
β string b = id(a);
ρ string c = id(“ciao”);
printfun(c)
```

```
 δ string id(γ string x) {
   return x ;
}
```

If we were to run this program and the prior program, they would have exactly the same outcome.

But … what about the analysis?

Two calls to the same function

```
α string a = gestFromNetwork();
β string b = id(a);
ρ string c = id("ciao");
printfun(c)
```

```
δ string id(γ string x) {
  return x ;
}
```

tainted <= α

α <= γ

γ <= δ

δ <= β

Two calls to the same function

```
α string a = gestFromNetwork();
β string b = id(a);
ρ string c = id("ciao");
printfun(c)
```

```
δ string id(γ string x) {
  return x ;
}
```

tainted <= α

α <= γ

γ <= δ

δ <= β

---

untainted <= γ

γ <= δ

δ <= ρ

ρ <= untainted

Two calls to the same function

```
α string a = gestFromNetwork();
β string b = id(a);
ρ string c = id("ciao");
printfun(c)
```

```
δ string id(γ string x) {
  return x ;
}
```

tainted <= α

$\alpha \le \gamma$

$\gamma \le \delta$

$\delta \le \beta$

untainted <= γ

$\gamma \le \delta$

$\delta \le \rho$

ρ <= untainted

tainted <= α <= γ <= δ <= ρ <= untainted

FALSE ALARM

No solution but yet no true tainted flow!!

Two calls to the same function

```
α string a = gestFromNetwork();
β string b = id(a);
ρ string c = id("ciao");
printfun(c)
```

```
δ string id(γ string x) {
  return x ;
}
```

tainted <= α <= γ <= δ <= ρ <= untainted

FALSE ALARM

No solution but yet no true tainted flow!!

The constraints represent an infeasible execution path

The analysis is imprecise: consider a call into which a tainted value is passed, and the return into which we pass the untainted value

Discussion

- The problem: **context insensitivity**.
- The two calls are **conflated** in the constraint graph.
- A **context sensitive** analysis solves this problem by **distinguishing** calls
  - we do not allow a function call to return its value to another, different, call.

Handling context sensitivity

- We associate a **different label to each call** (e.g. correlate the label with the line number in the program at which the call occurs).
- We match up **calls with corresponding returns**, only when the labels on flow edges match.
- We add **polarities** to distinguish **calls from returns**.
  - minus for argument passing, and plus for return values.

Two calls to the same function

```
α string a = gestFromNetwork();
β string b = id₁(a);
ρ string c = id₂("ciao");
printfun(c)
```

```
δ string id(γ string x) {
  return x ;
}
```

tainted <= α

α <= -1 γ

γ <= δ

δ <= $+1$ β

untainted <= -2 γ

γ <= δ

δ <= $+2$ ρ

ρ <= untainted

Two calls to the same function

```
α string a = gestFromNetwork();
β string b = id₁(a);
ρ string c = id₂("ciao");
printfun(c)
```

```
δ string id(γ string x) {
  return x ;
}
```

tainted <= α
α <= -1 γ
γ <= δ

δ <= $+2$ ρ
ρ <= untainted

Indexes of the calls do not match!!
Infeasible flow not allowed

NO ALARM

Discussion

- Context sensitivity is a tradeoff, favoring precision over scalability.
  - the context insensitive algorithm takes roughly time O(n), where n is the size of the program,
  - the context sensitive algorithm will take time $O(n^3)$
- The added precision actually helps performance. By eliminating infeasible paths it can reduce the size of the constraint graph by a constant factor.
- The general trend is that **greater precision means lower scalability**

What about pointers?

```
α char *a ="ciao";
(β char *) *p = &a;
(γ char *) *q = p;
δ char * v =getFromNetwork();
*q = v;
printf(*p)
```

untainted <= α
α <= β
β <= γ
tainted <= δ
δ <= γ
β <= untainted

```
α char *a ="ciao";
(β char *) *p = &a;
(γ char *) *q = p;
δ char * v =getFromNetwork();
*q = v;
printf(*p)
```

SOLUTION EXISTS

$\text{untainted} \le \alpha$
$\alpha \le \beta$
$\beta \le \gamma$
$\text{tainted} \le \delta$
$\delta \le \gamma$
$\beta \le \text{untainted}$

$\alpha = \beta = \text{untainted}$
$\gamma = \delta = \text{tainted}$

```
α char *a ="ciao";
(β char *) *p = &a;
(γ char *) *q = p;
δ char * v =getFromNetwork();
*q = v;
printf(*p)
```

p and q are aliases

~~SOLUTION EXISTS~~

untainted <= α
α <= β
β <= γ
tainted <= δ
δ <= γ
β <= untainted

α = β = untainted
γ = δ = tainted

tainted <= δ <= γ <= β <= untainted

```
α char *a ="ciao";
(β char *) *p = &a;
(γ char *) *q = p;
δ char * v =getFromNetwork();
*q = v;
printf(*p)
```

untainted <= α

α <= β

$\boxed{\begin{array}{l}\beta \text{ <= } \gamma \\ \gamma \text{ <= } \beta\end{array}}$

tainted <= δ

δ <= γ

β <= untainted

IDEA: ADDING ALIASING CONSTRAINTS
ASSIGNMENT VIA POINTERS
FLOW GOES IN BOTH WAYS

DATA STRUCTURES

Array Ops

```
void copy (tainted char[ ] src, untainted char[ ] dst), int len) {
  int untainted i;
  for (i=0; i<len; i++) {
     dst[i] = src[i];
   }
}
```

Tainted Flow

```
void copy (tainted char[ ] src, untainted char[ ] dst), int len) {
  int untainted I;
  for (i=0; i<len; i++) {
     dst[i] = src[i];
   }
}
```

untainted ← **tainted**

ILLEGAL FLOW

Implicit Flow

```
void copy (tainted char[ ] src, untainted char[ ] dst), int len) {
  int untainted i;
  int untainted j;
  for (i=0; i<len; i++) {
     for (j=0; j < sizeof(char)*256, j++) {
         if src[i] = (char) j
                 dst[i] = (char) j // Is legal?
          }
    }
}
```

Implicit Flow

```
void copy (tainted char[ ] src, untainted char[ ] dst), int len) {
  int untainted i;
  int untainted j;
  for (i=0; i<len; i++) {
    for (j=0; j < sizeof(char)*256, j++) {
        if src[i} = (char) j
              dst[i] = (char) j  // Is legal?
        }     untained     untained
    }
}
```

MISSED FLOW

the char value was not directly assigned from src

but the value itself was.

The contents of src is certainly copied to dst: the information is leaked.

Data did not flow, but the information did

Information flow

- The analysis needs to be **more precise**:
  - We add a **taint constraint** affecting the **current flow position** abstracted by the **pc**
- Idea the assignment x = y (i.e the flow from y to x) now produces two constraints
  1. as expected the contraint between the taint labels of y and x
  2. the pc flow label bounds the label of x

Flow equation (revisited)

The pc flow label represents the taint value affecting the current execution

The assignment **x = y** results in two constraints

1. **TaintLabel(y) <= TaintLabel(x)**
2. **TaintLabel(x) <= pc**

Example

```
tainted int src;
α int  dst;
if (src == 0)
   dst = 0;
else
   dst = 1;

dst +=0;
```

if the source (**src**) is zero, then **dst** will contain the same value as the source, otherwise it will contain one

Example

```
tainted int src;
α int  dst;
if (src == 0)
   dst = 0;            untainted <= α
else
   dst = 1;            untainted <= α

dst +=0;               untainted <= α
```

Example

	tainted int src;	
	α int dst;	
pc_1 = **untainted**	**if (src == 0)**	
pc_2 = **tainted**	**dst = 0;**	**untainted** <= α
	else	
pc_3 = **tainted**	**dst = 1;**	**untainted** <= α
pc_4 = **untainted**	**dst +=0;**	**untainted** <= α

Example

	tainted int src;	
	α int dst;	
pc_1 = **untainted**	**if (src == 0)**	
pc_2 = **tainted**	**dst = 0;**	**untainted** <= α pc_2 <= α
	else	
pc_3 = **tainted**	**dst = 1;**	**untainted** <= α pc_3 <= α
pc_4 = **untainted**	**dst +=0;**	**untainted** <= α pc_4 <= α

Example

	tainted int src;	
	α int dst;	
pc_1 = **untainted**	**if (src == 0)**	
pc_2 = **tainted**	**dst = 0;**	**untainted** <= α pc_2 <= α
	else	
pc_3 = **tainted**	**dst = 1;**	**untainted** <= α pc_3 <= α
pc_4 = **untainted**	**dst +=0;**	**untainted** <= α pc_4 <= α

The solution requires **tainted** = α

The analysis discover implicit flow

Example

	tainted int src;	
	α int dst;	
pc_1 = untainted	if (src == 0)	
pc_2 = tainted	dst = 0;	untainted <= α pc_2 <= α
	else	
pc_3 = tainted	dst = 1;	untainted <= α pc_3 <= α
pc_4 = untainted	dst +=0;	untainted <= α pc_4 <= α

The solution requires **tainted** = α

The analysis discover implicit flow

information is flowing from src to dst, though data is not: information about src can be recovered by looking at the value of dst after the program runs

More on Information flow

```
tainted int src;
α int dst;
if (src > 0)
    dst = 0;
    else dst = 0;
```

Tracking implicit flows can lead to **false alarms**

More on Information flow

```
tainted int src;
α int dst;
if (src > 0)
     dst = 0;
     else dst = 0;
```

Tracking implicit flows can lead to **false alarms**

A different technique to manage information flow will be discussed later

Other challenges: Data Structures

Struct fields
Track taint for the whole struct, or each field?

Arrays:
Track taint per element or across whole array?

Other challenges: Data Structures

Struct fields

Track taint for the whole struct, or each field?

Arrays:

Track taint per element or across whole array?

No single correct answer!

(Tradeoffs: Soundness, completeness, performance)

Challenges

- We have considered how to analyze most of the key elements of a language. But not all of them.
  - A robust tool obviously has to handle them all

#1 Ops

Assignments transfer the taint from the source to the target

What happen if the source is an expression rather than a variable?

- **the taint of operators must be defined**.

#2
Pointers

Analyzing a function call using a function pointer

add constraints as if all possible targets were called rather than a single target

#3 Struct Objects

A precise analysis can track the taintedness of each field of a struct separately as if they were separate variables.

Such precision can be expensive.

Alternatives: tracks only some of its fields

Objects are much like a struct containing function pointers and so the trade offs we've just considered apply in the analysis of object oriented languages

#4 Abstract Values

Abstract Intepretation

Abstract interpretation is a static analysis abstricting over all possible concrete runs of a program.

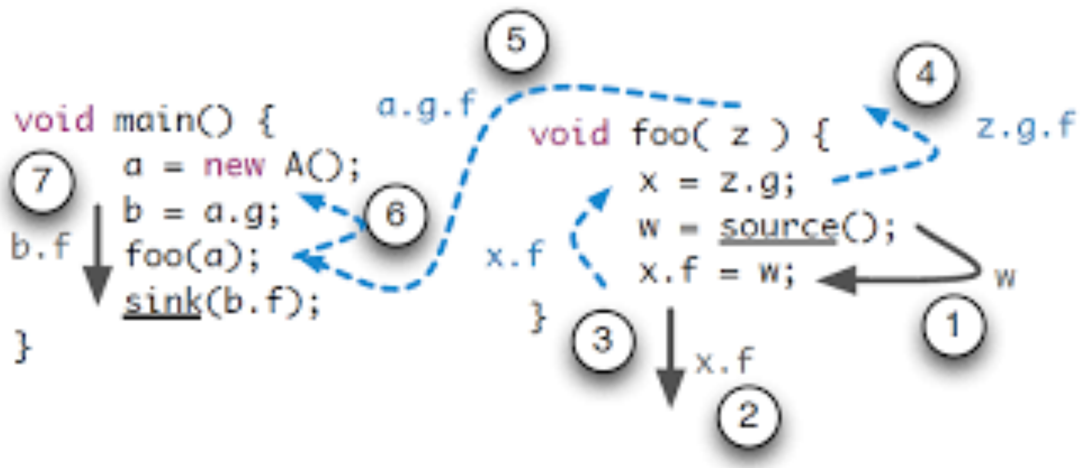
The key is to discard as much information as possible for purposes of scalability, while still being able to prove the property of interest.

Taint analysis in practice

- Taint analysis is limited but
  - **Eliminate** some categories of errors
  - Developers can concentrate on **deeper reasoning**
- Encourage **better development practices**
  - Programming models that avoid mistakes
  - Teach programmers to manifest their assumptions
  - Using **annotations** that improve tool precision
- **Increased commercial adoption**

FlowDroid:
static taint tracking on Android

- FlowDroid does static taint tracking for Android Applications
- It includes data flow tracking iuncluding pointer analysis as well as class and field refrences



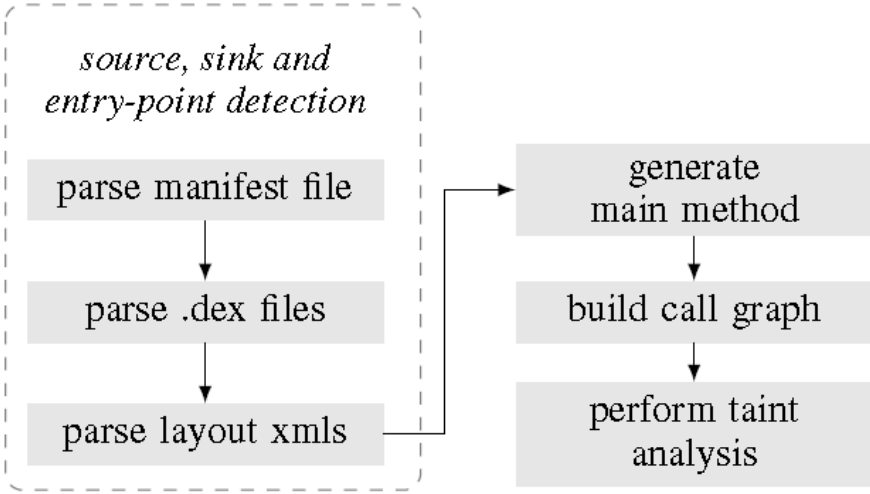
Case Study: FlowDroid

FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden
EC SPRIDE
Technische Universität Darmstadt
firstName.lastName@ec-spride.de

Alexandre Bartel, Jacques Klein, and Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
firstName.lastName@uni.lu

Damien Octeau, Patrick McDaniel
Department of Computer Science and Engineering
Pennsylvania State University
{octeau,mcdaniel}@cse.psu.edu



- Handle the challenge of Android applications, *e.g.,* callbacks invoked by the Android framework, aliasing.

Transfer Flow

- **Access path**: Taint the left-hand side if any of the operands on the right-hand side is tainted.
  - e.g., x.f includes taints x.f.g, x.f.h, x.f.g.h and so on.
- **Array**: Assignments to array elements are treated conservatively by tainting the entire array.
- **New expression**: Assigning a “new”-expression to a variable x erases all taints modeled by access paths rooted at x.

FlowDroid

- https://blogs.uni-paderborn.de/sse/tools/flowdroid/