



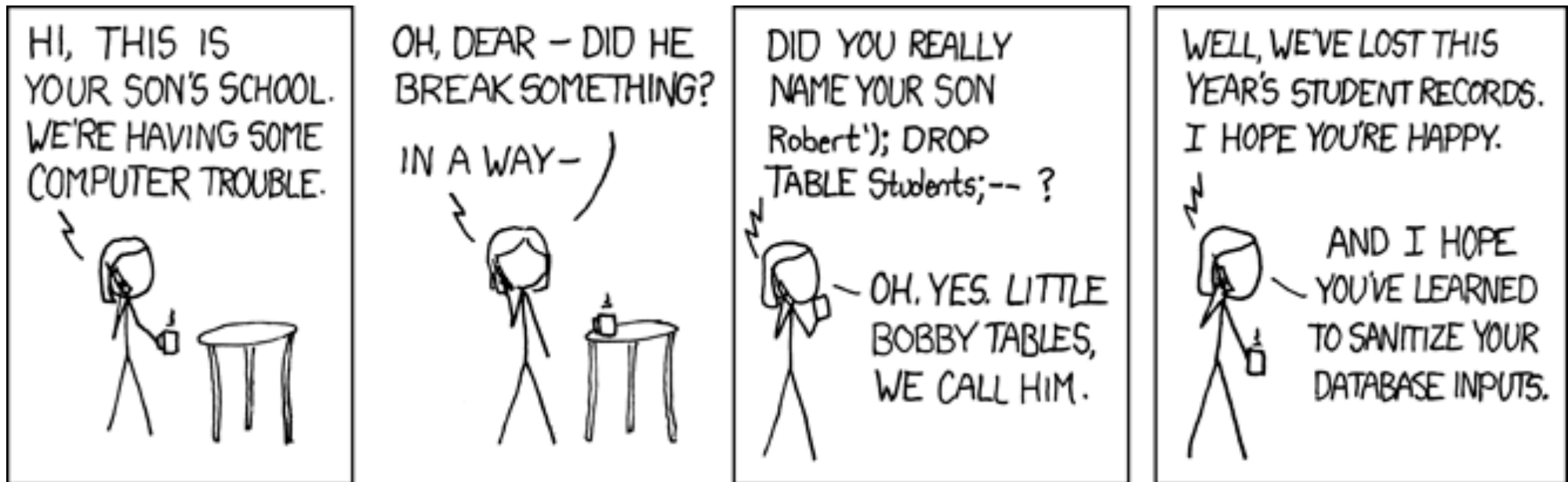
TAINTED FLOW ANALYSIS

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

An Example is Worth Many Words



- 1) What is the person behind the phone complaining about?
- 2) Have you ever seen such a problem before?

Bobby Tables has got a Car



What is the problem with the plate of this car?

Information Flow

- Programs manipulate information.
- Some information should not leave the program.
 - Example: an unencrypted password.
- Other information should not reach sensitive parts of the program.
 - Example: a string too large to fit into an array.

In the cartoon we just saw, what is the problem: information entering the program, or information leaving the program?

Information flow vulnerabilities

- If the user can read sensitive information from the program, we say that the program has an *information disclosure vulnerability*.
- If the user can send harmful information to the program, we say that the program has a *tainted flow vulnerability*.

- 1) Which tainted flow vulnerabilities can you think about?
- 2) Can you think about information disclosure vulnerabilities?
- 3) Can you think about a way to find out if a program has such a vulnerability. Try to be creative!

Tainted Flow Vulnerabilities

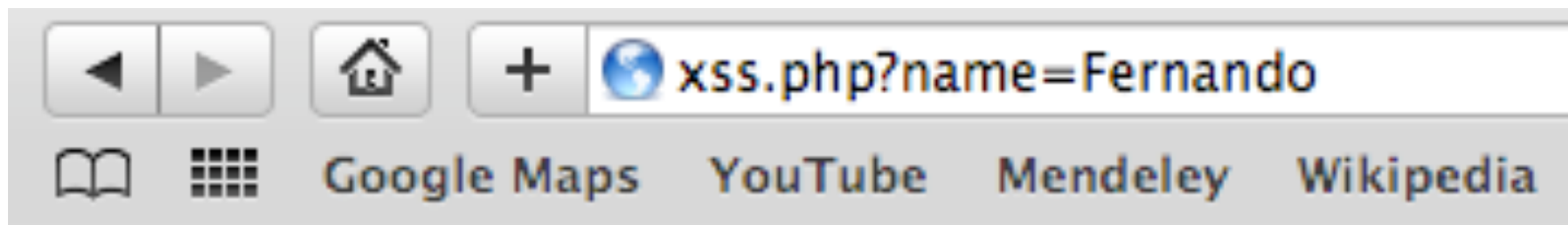
- An adversary can compromise the program by sending malicious information to it.
 - This type of attack is very common in web servers.

An example of security bug

```
1 <?php
2     init_session();
3     echo "Hello " . $_GET['name'];
4 ?>
```

What is the vulnerability of this program?

`http://localhost/xss.php?name=Fernando`



Hello Fernando

An example of security bug

```
1 <?php
2   init_session();
3   echo "Hello " . $_GET['name'];
4 ?>
```

http://localhost/xss.php?name=fernando

How to steal a cookie:

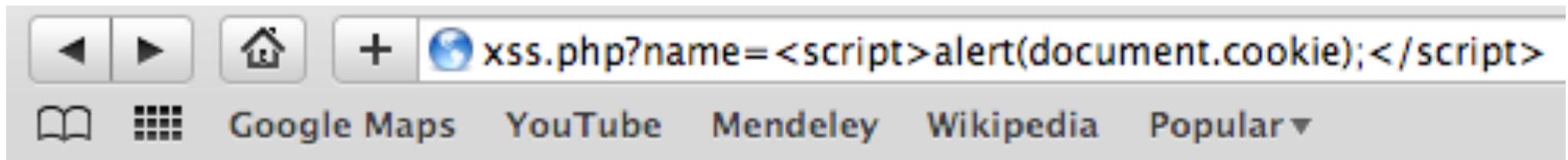
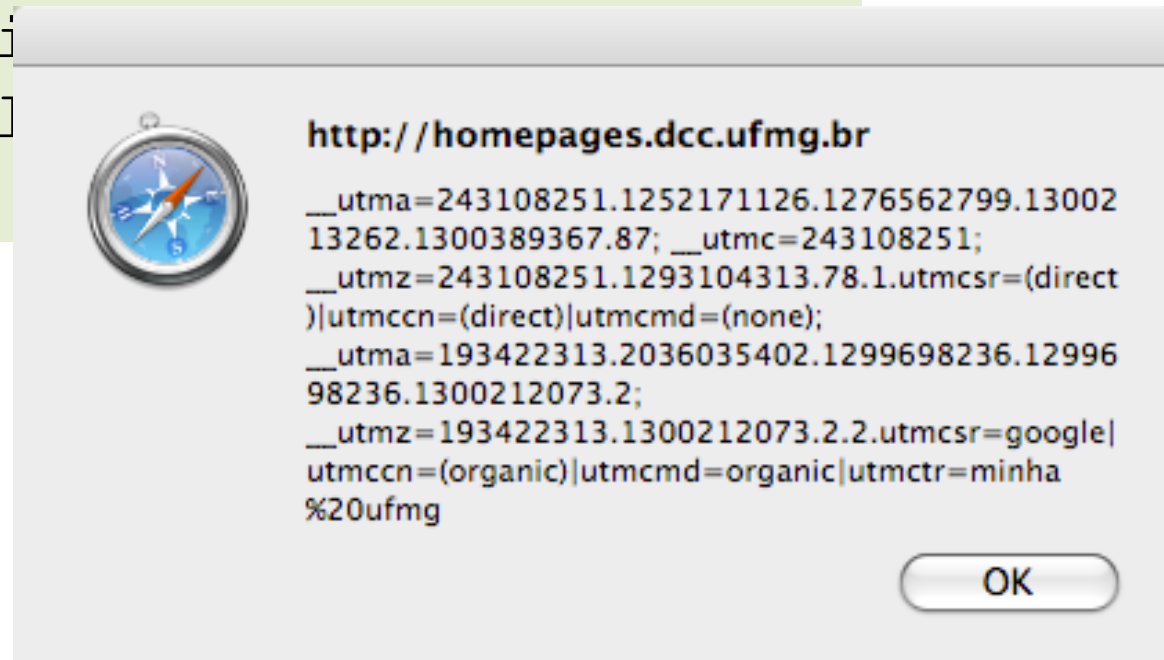
http://localhost/xss.php?
name=<script>alert(docume
nt.cookie);</script>

Hello fernando

Wikipedia

An example of security bug

```
1 <?php
2     init_session_start();
3     echo "Hello";
4 ?>
```



Hello

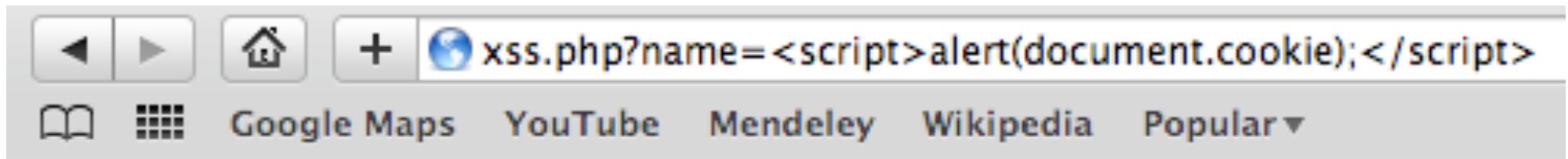
An example of security bug

```
1  <?php
2      init_session();
3      echo "Hello " . $_GET['name'];
4  ?>
```

How to clean this program?

An example of security bug

```
1  <?php
3  init_session();
4  echo "Hello " .
      htmlentities($_GET['name']);
4  ?>
```



Hello \"<script>alert(document.cookie);</script>\"

There are several ways to break a program

```
$id = $_GET["user"];
```

```
if ($id == '') {  
    echo "Invalid user: $id"  
} else {  
    $getuser = $DB->query  
        ("SELECT * FROM `table` WHERE id='$id'");  
    echo $getuser;  
}
```

What is the
vulnerability of
this program?

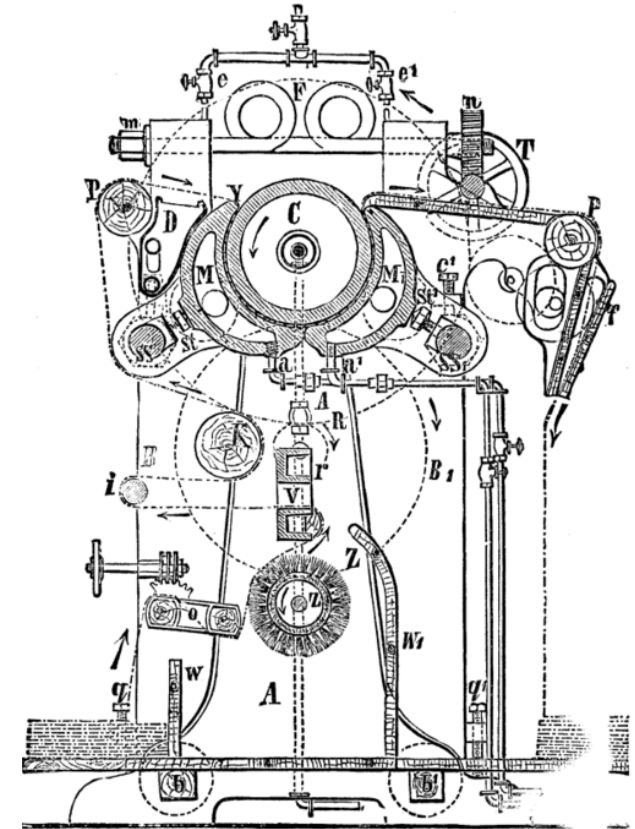
There are several ways to break a program

```
$id = $_GET["user"];  
  
if ($id == '') {  
    echo "Invalid user: $id"  
} else {  
    $getuser = $DB->query  
        ("SELECT * FROM `table` WHERE id='$id'");  
    echo $getuser;  
}
```

What if the name of the student is "Robert" ;
drop table
Students ; --"?

The Tainted Flow Problem

- Instance: a tuple $T = (P, SO, SI, SA)$
 - $P \rightarrow$ Program
 - $SO \rightarrow$ Sources
 - $SI \rightarrow$ Sinks
 - $SA \rightarrow$ Sanitizers



```
<?php  
    echo htmlentities($_GET['name']);  
?>
```

The Tainted Flow Problem

- Instance: a tuple $T = (P, \mathbf{SO}, SI, SA)$
 - $P \rightarrow$ Program
 - **SO** \rightarrow Sources
 - $SI \rightarrow$ Sinks
 - $SA \rightarrow$ Sanitizers



```
<?php
    echo htmlentities($_GET['name']);
?>
```

The Tainted Flow Problem

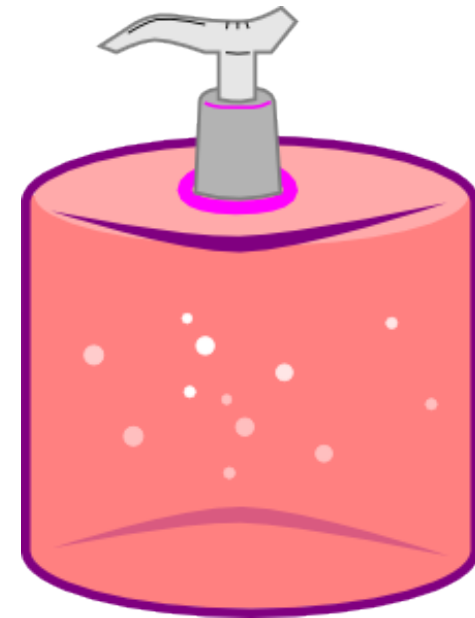
- Instance: a tuple $T = (P, SO, \mathbf{SI}, SA)$
 - $P \rightarrow$ Program
 - $SO \rightarrow$ Sources
 - **$SI \rightarrow$ Sinks**
 - $SA \rightarrow$ Sanitizers



```
<?php  
    echo htmlentities($_GET['name']);  
?>
```

The Tainted Flow Problem

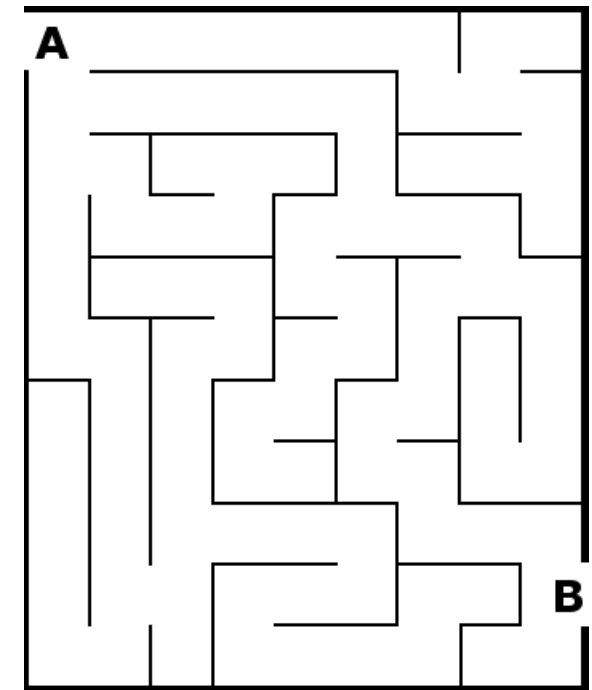
- Instance: a tuple $T = (P, SO, SI, \mathbf{SA})$
 - $P \rightarrow$ Program
 - $SO \rightarrow$ Sources
 - $SI \rightarrow$ Sinks
 - **$SA \rightarrow$ Sanitizers**



```
<?php
    echo htmlentities($_GET['name']);
?>
```

The Tainted Flow Problem

- Instance: a tuple $T = (P, SO, SI, SA)$
 - $P \rightarrow$ Program
 - $SO \rightarrow$ Sources
 - $SI \rightarrow$ Sinks
 - $SA \rightarrow$ Sanitizers
- Problem: find a path from a source to a sink that does not go across a sanitizer



What do you think
I mean by path?

Example: cross-site scripting (XSS)

- Instance: a tuple $T = (P, SO, SI, SA)$
 - $P \rightarrow$ Program
 - $SO \rightarrow$ Sources
 - $SI \rightarrow$ Sinks
 - $SA \rightarrow$ Sanitizers

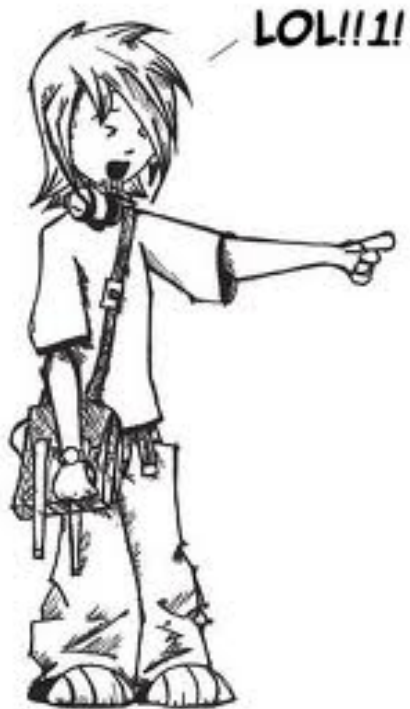
Cross-site Scripting (XSS)

SO: `$_GET, $_POST, ...`

SI: `echo, print, printf, ...`

SA: `htmlentities, strip_tags, ...`

Example: XSS



```
<?php  
$name = $_GET['name'];  
echo $name;  
?>
```



```
<?php  
$name = htmlentities($_GET['name']);  
echo $name;  
?>
```

Example: SQL injection

- Instance: a tuple $T = (P, SO, SI, SA)$
 - $P \rightarrow$ Program
 - $SO \rightarrow$ Sources
 - $SI \rightarrow$ Sinks
 - $SA \rightarrow$ Sanitizers

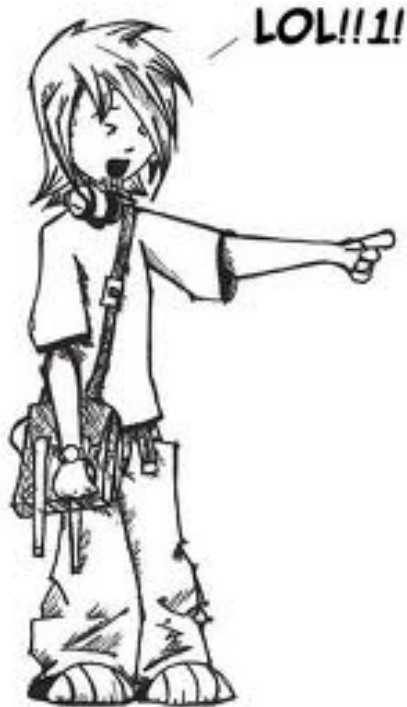
SQL injection:

SO: `$_GET, $_POST, ...`

SI: `mysql_query, pg_query, ...`

SA: `addslashes, pg_escape_string, ...`

Example: SQL Injection



LOL!!1!

```
<?php
  $userid = $_GET['userid'];
  $passwd = $_GET['passwd'];
  ...
  $result = mysql_query("SELECT userid FROM users
    WHERE userid=$userid AND
    passwd='$passwd'");
?>
```

Can you find a **string** that goes around the password guard?



```
<?php
  $userid = (int) $_GET['userid'];
  $passwd = addslashes($_GET['passwd']);
  ...
  $result = mysql_query("SELECT userid FROM
    users WHERE userid=$userid AND
    passwd='$passwd'");
?>
```

Example: Command Execution

- Instance: a tuple $T = (P, SO, SI, SA)$
 - $P \rightarrow$ Program
 - $SO \rightarrow$ Sources
 - $SI \rightarrow$ Sinks
 - $SA \rightarrow$ Sanitizers



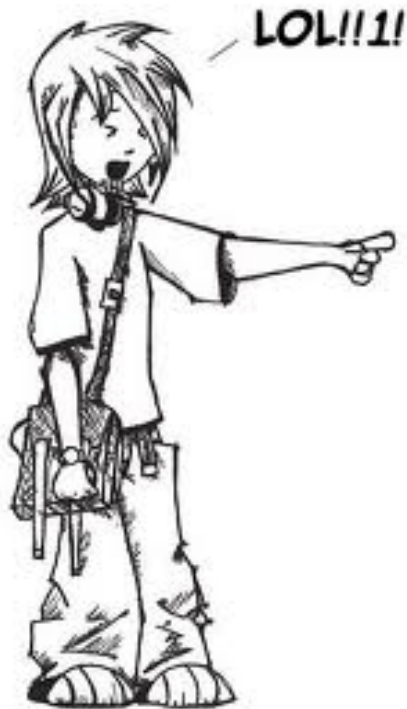
Command Execution:

SO: `$_GET, $_POST, ...`

SI: `exec, system, passthru, ...`

SA: `escapeshellcmd, escapeshellarg, ...`

Example: Command Execution



LOL!!1!

```
<?php
  $filename = $_GET['filename'];
  system("/usr/bin/file $filename");
?>
```

1) Do you know what the `file` command does?

2) Can you do anything evil with **this** program?



```
<?php
  $filename = escapecmdshell
    ($_GET['filename']);
  system("/usr/bin/file $filename");
?>
```

Example: Remote File Inclusion

- Instance: a tuple $T = (P, SO, SI, SA)$
 - $P \rightarrow$ Program
 - $SO \rightarrow$ Sources
 - $SI \rightarrow$ Sinks
 - $SA \rightarrow$ Sanitizers

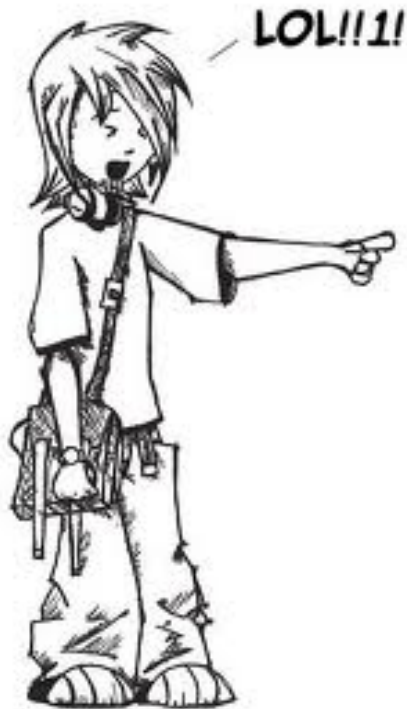
Remote File Inclusion:

SO: `$_GET, $_POST, ...`

SI: `include, include_once, require, require_once, ...`

SA: ?

Example: Remote File Inclusion



```
<?php  
  $file = $_GET['filename'];  
  include($file);  
?>
```



```
<?php  
  $file_incs = array("file1", ..., "fileN");  
  $file_id = $_GET['file_id'];  
  
  $inc = $file_incs[$file_id];  
  if (isset($inc))  
    include($inc);  
  else  
    echo "Error...";  
?>
```

Example: File System Access

- Instance: a tuple $T = (P, SO, SI, SA)$
 - $P \rightarrow$ Program
 - $SO \rightarrow$ Sources
 - $SI \rightarrow$ Sinks
 - $SA \rightarrow$ Sanitizers



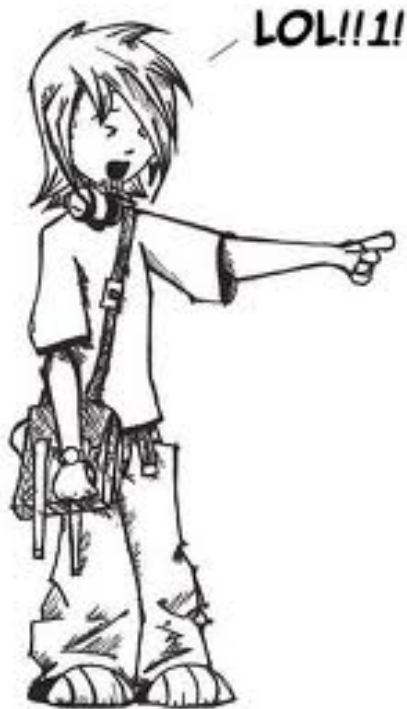
File System Access:

SO: `$_GET, $_POST, ...`

SI: `chdir, mkdir, rmdir, rename, copy, chgrp, chown, chmod, unlink, ...`

SA: ?

Example: File System Access



```
<?php  
$filename = $_GET['filename'];  
unlink($filename);  
?>
```



The developer can use the same methods used to guard against remote file inclusion attacks.

Example: Malicious Evaluation

- Instance: a tuple $T = (P, SO, SI, SA)$
 - $P \rightarrow$ Program
 - $SO \rightarrow$ Sources
 - $SI \rightarrow$ Sinks
 - $SA \rightarrow$ Sanitizers



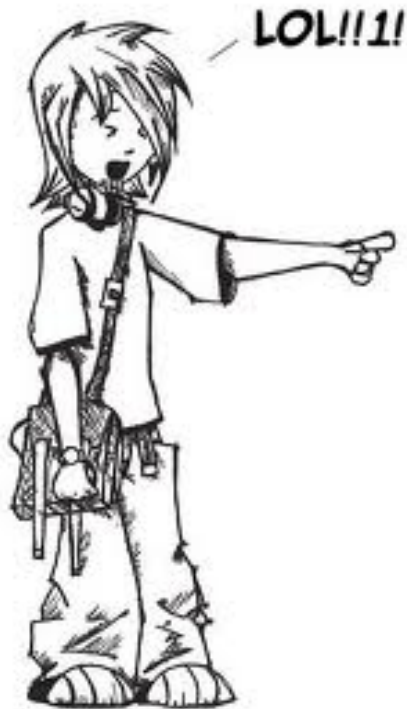
Malicious Evaluation:

SO: `$_GET, $_POST, ...`

SI: `eval, preg_replace, ...`

SA: `?`

Example: Malicious Evaluation



```
<?php  
$code = $_GET['code'];  
eval($code);  
?>
```

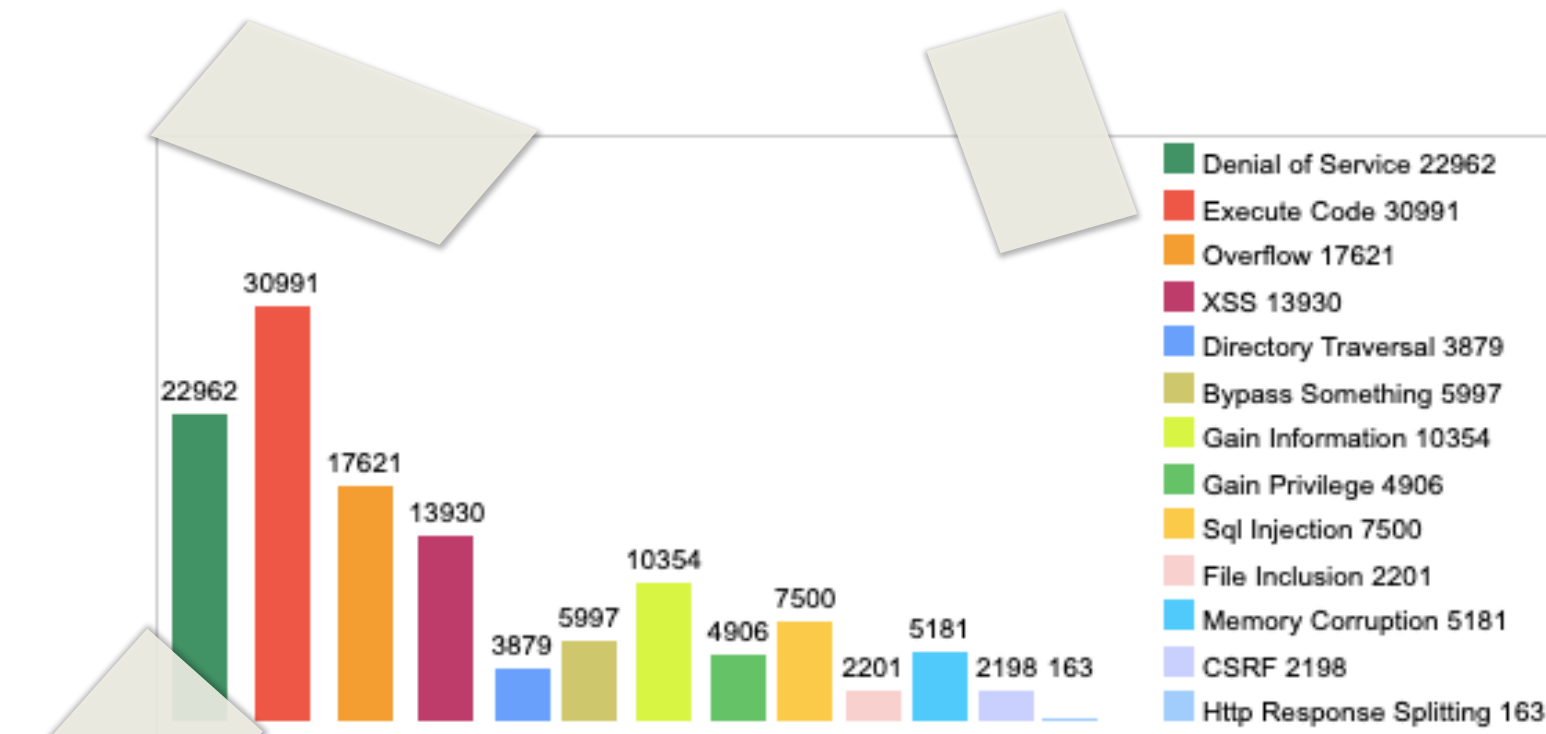
There is not a systematic way of checking statically that dynamic code is safe...



It is a serious problem...

The cost for a minor SQL injection attack exceeds \$196,000.

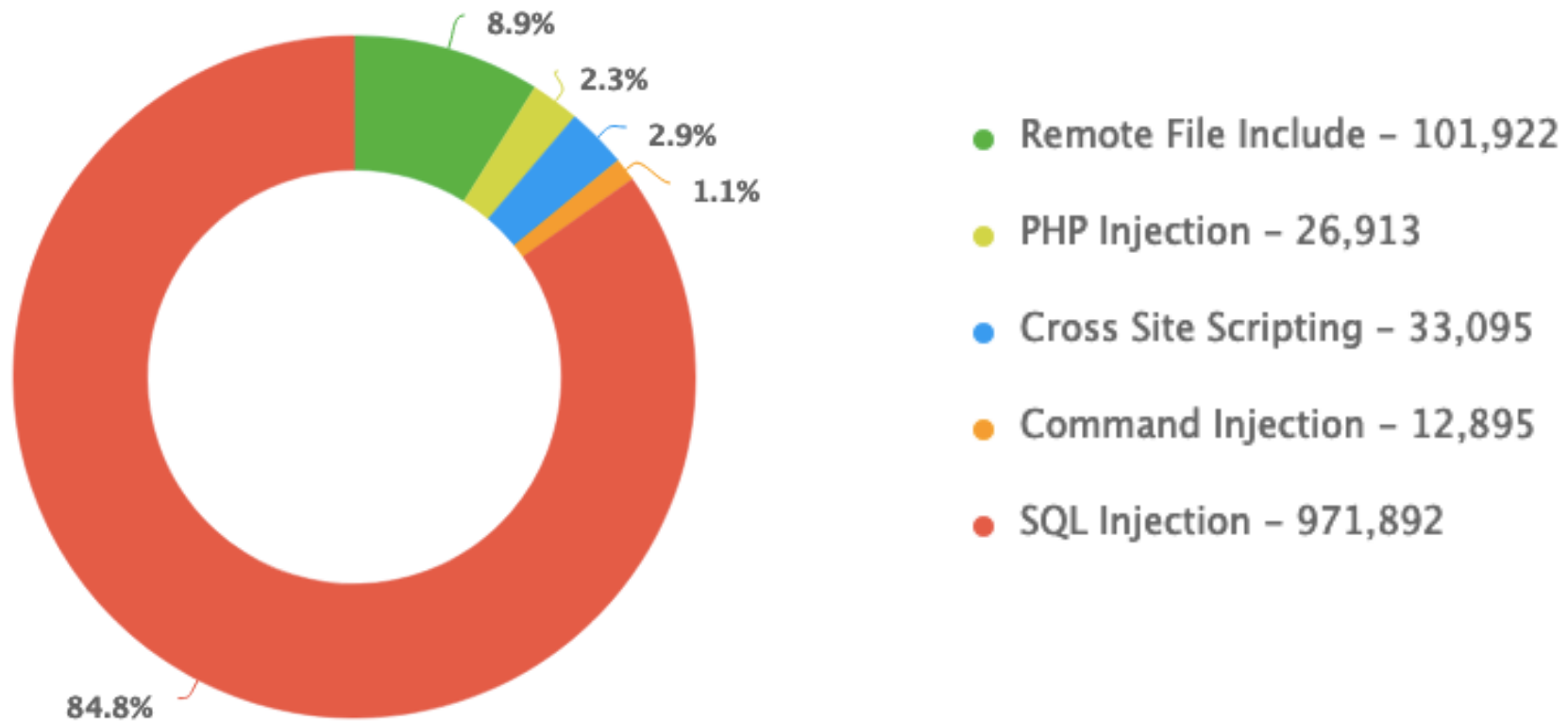
NTT Global Threat Intelligence Report - 2014



Vulnerabilities by category (1999-2019)
The CVE Report 2019



Web Attack Visualization - Akamai.com



1,146,717 observed attacks

A Market with Many Players

VERACODE



sonarqube



AppScan
IBM Security



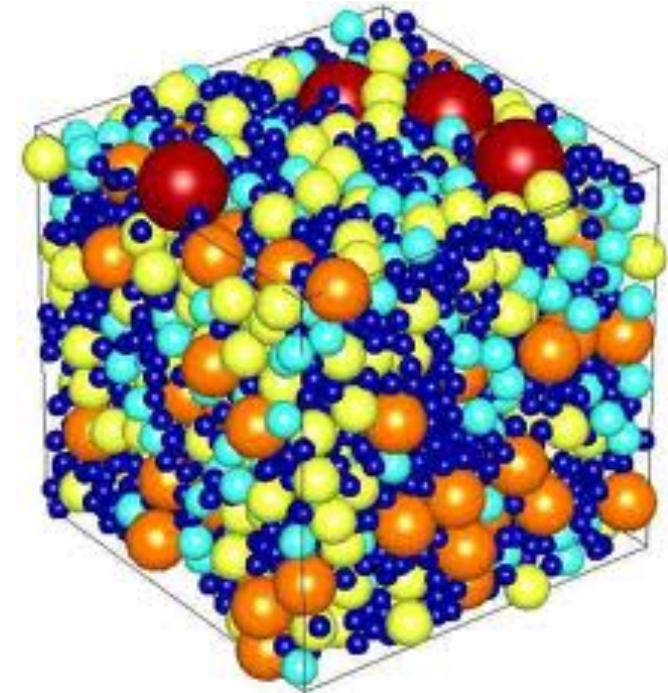
Have you ever
heard of any of
these companies
and tools?

Discovering Tainted Flow Vulnerabilities

- How can we find if a program contains a tainted flow vulnerability?
 - Is your algorithm decidable?
 - Is it fast?
 - Is it too **conservative**? In other words, can it output a false positive?
 - Is it **sound**? In other words, can it produce false negatives?
 - Does it work for every kind of tainted flow vulnerability?



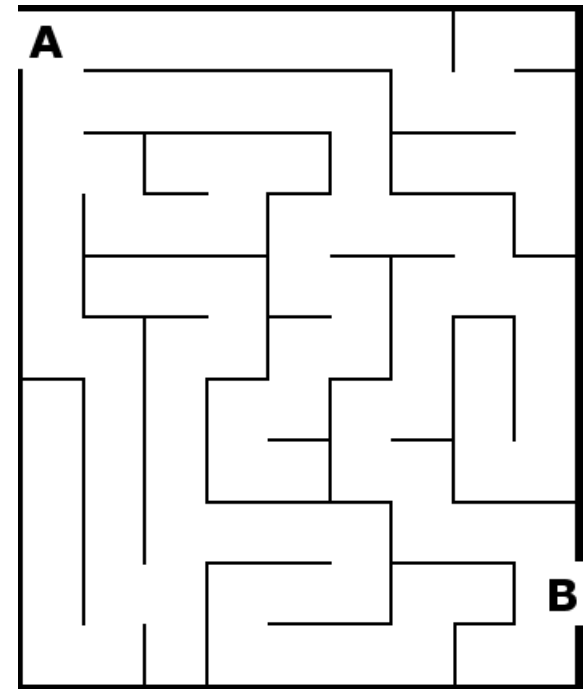
DENSE TAINTED FLOW ANALYSIS



How to detect tainted flow vulnerabilities?

- Problem: find a path from a source to a sink that does not go across a sanitizer.

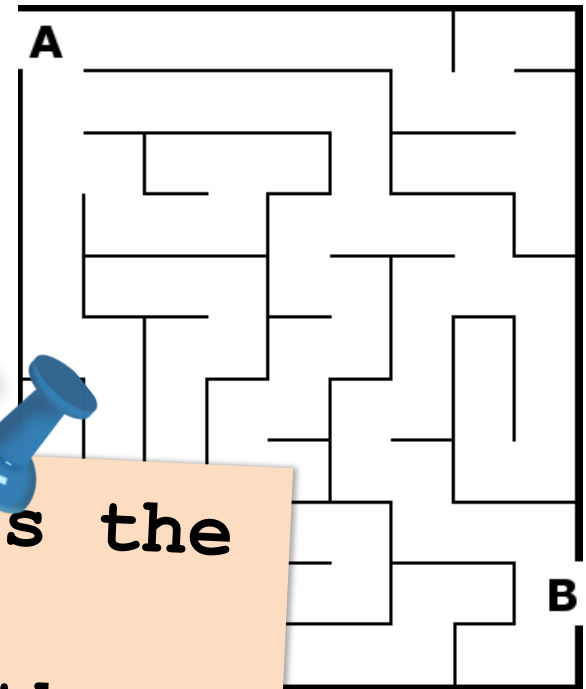
The very old question: what is a path inside the program?



How to detect tainted flow vulnerabilities?

- Problem: find a path from a source to a sink that does not go across a sanitizer.

The very old question: what is a path inside the program?



A path is a chain of data dependences in a program

A path is the flow of control that exists in the program

Nano-PHP: our toy language

- Often it is useful to define a static analysis on a small programming language.
- We can prove properties for this programming language.
- These proofs not only build up confidence on the algorithm, but they also help the reader to understand how the algorithm works.
- The toy language should be simple, so that the entire exposition of ideas is not complex.
- Yet, it should be complex enough to include every aspect of the algorithm that will be important in the real world.



Nano-PHP: our toy language

Name	Instruction	Example
Assignment from source	$v = \bigcirc$	<code>\$x = \$_POST['content']</code>
Assignment to sink	$\bullet = v$	<code>echo (\$v)</code>
Simple assignment	$x = \otimes(x_1, \dots, x_n)$	<code>\$a = \$t1 * \$t2</code>
Branch	<code>bra l_1, \dots, l_n</code>	<code>if (...) {...} else {...}</code>
Filter	<code>x = filter</code>	<code>\$a = htmlentities(\$t1)</code>
Validator	<code>validate x, l_c, l_t</code>	<code>if(is_num(\$t1)) {...}</code>

```

$v = DB.get($_GET[ 'child' ] );
$x = "";
if (DB.isMember($v)) {
    while (DB.hasParent($v)) {
        echo($x);
        $x = $_POST[ '$v' ];
        $v = DB.getParent($v);
    }
    echo($v);
}

```

```

 $l_0: v = \bigcirc$ 
 $l_1: x = \text{filter}$ 
 $l_2: \text{validate}(v, l_3, l_9)$ 
 $l_3: \text{bra } l_4, l_8$ 
 $l_4: \bullet = x$ 
 $l_5: v = \bigcirc$ 
 $l_6: v = \otimes(v)$ 
 $l_7: \text{bra } l_4$ 
 $l_8: \bullet = x$ 
 $l_9: \text{bra } l_4$ 

```

Nano-PHP: our toy language

Name	Instruction	Example
Assignment from source	$v = \bigcirc$	<code>\$x = \$_POST['content']</code>
Assignment to sink	$\bullet = v$	<code>echo (\$v)</code>
Simple assignment	$x = \otimes(x_1, \dots, x_n)$	<code>\$a = \$t1 * \$t2</code>
Branch	<code>bra l_1, \dots, l_n</code>	<code>if (...) {...} else {...}</code>
Filter	$x = \text{filter}$	<code>\$a = htmlentities(\$t1)</code>
Validator	<code>validate x, l_c, l_t</code>	<code>if(is_num(\$t1)) {...}</code>

```

$v = DB.get($_GET[ 'child' ] );
$x = "" ;
if (DB.isMember($v)) {
    while (DB.hasParent($v)) {
        echo ($x);
        $x = $_POST[ '$v' ];
        $v = DB.getParent($v);
    }
    echo ($v);
}

```

```

l0: v =  $\bigcirc$ 
l1: x = filter
l2: validate(v, l3, l9)
l3: bra l4, l8
l4:  $\bullet = x$ 
l5: v =  $\bigcirc$ 
l6: v =  $\otimes(v)$ 
l7: bra l3
l8:  $\bullet = x$ 
l9: bra l9

```

Nano-PHP: our toy language

Name	Instruction	Example
Assignment from source	$v = \bigcirc$	<code>\$x = \$_POST['content']</code>
Assignment to sink	$\bullet = v$	<code>echo (\$v)</code>
Simple assignment	$x = \otimes(x_1, \dots, x_n)$	<code>\$a = \$t1 * \$t2</code>
Branch	<code>bra l_1, \dots, l_n</code>	<code>if (...) {...} else {...}</code>
Filter	<code>x = filter</code>	<code>\$a = htmlentities(\$t1)</code>
Validator	<code>validate x, l_c, l_t</code>	<code>if (is_num(\$t1)) {...}</code>

```

$v = DB.get($_GET[ 'child' ] );
$x = "";
if (DB.isMember($v)) {
    while (DB.hasParent($v)) {
        echo ($x);
        $x = $_POST[ '$v' ];
        $v = DB.getParent($v);
    }
    echo ($v);
}

```

```

l0: v =  $\bigcirc$ 
l1: x = filter
l2: validate(v, l3, l9)
l3: bra l4, l8
l4:  $\bullet = x$ 
l5: v =  $\bigcirc$ 
l6: v =  $\otimes(v)$ 
l7: bra l3
l8:  $\bullet = x$ 
l9: bra l9

```

Nano-PHP: our toy language

Name	Instruction	Example
Assignment from source	$v = \bigcirc$	<code>\$x = \$_POST['content']</code>
Assignment to sink	$\bullet = v$	<code>echo (\$v)</code>
Simple assignment	$x = \otimes(x_1, \dots, x_n)$	<code>\$a = \$t1 * \$t2</code>
Branch	bra l_1, \dots, l_n	if (...) {...} else {...}
Filter	$x = \text{filter}$	<code>\$a = htmlentities(\$t1)</code>
Validator	validate x, l_c, l_t	<code>if(is_num(\$t1)) {...}</code>

```

$v = DB.get($_GET[ 'child' ] );
$x = "";
if (DB.isMember($v)) {
    while (DB.hasParent($v)) {
        echo ($x);
        $x = $_POST[ '$v' ];
        $v = DB.getParent($v);
    }
    echo ($v);
}

```

```

l0: v =  $\bigcirc$ 
l1: x = filter
l2: validate(v, l3, l9)
l3: bra l4, l8
l4:  $\bullet = x$ 
l5: v =  $\bigcirc$ 
l6: v =  $\otimes(v)$ 
l7: bra l3
l8:  $\bullet = x$ 
l9: bra l9

```

Nano-PHP: our toy language

Name	Instruction	Example
Assignment from source	$v = \bigcirc$	<code>\$x = \$_POST['content']</code>
Assignment to sink	$\bullet = v$	<code>echo (\$v)</code>
Simple assignment	$x = \otimes(x_1, \dots, x_n)$	<code>\$a = \$t1 * \$t2</code>
Branch	<code>bra l_1, \dots, l_n</code>	<code>if (...) {...} else {...}</code>
Filter	<code>x = filter</code>	<code>\$a = htmlentities(\$t1)</code>
Validator	<code>validate x, l_c, l_t</code>	<code>if(is_num(\$t1)) {...}</code>

```

$v = DB.get($_GET[ 'child' ] );
$x = "";
if (DB.isMember($v)) {
    while (DB.hasParent($v)) {
        echo ($x) ;
        $x = $_POST[ '$v' ];
        $v = DB.getParent($v);
    }
    echo ($v) ;
}

```

```

 $l_0$ :  $v = \bigcirc$ 
 $l_1$ : x = filter
 $l_2$ : validate(v,  $l_3, l_9$ )
 $l_3$ : bra  $l_4, l_8$ 
 $l_4$ :  $\bullet = x$ 
 $l_5$ :  $v = \bigcirc$ 
 $l_6$ :  $v = \otimes(v)$ 
 $l_7$ : bra  $l_3$ 
 $l_8$ :  $\bullet = x$ 
 $l_9$ : bra  $l_9$ 

```

Nano-PHP: our toy language

Name	Instruction	Example
Assignment from source	$v = \bigcirc$	<code>\$x = \$_POST['content']</code>
Assignment to sink	$\bullet = v$	<code>echo (\$v)</code>
Simple assignment	$x = \otimes(x_1, \dots, x_n)$	<code>\$a = \$t1 * \$t2</code>
Branch	<code>bra l_1, \dots, l_n</code>	<code>if (...) {...} else {...}</code>
Filter	<code>x = filter</code>	<code>\$a = htmlentities(\$t1)</code>
Validator	<code>validate x, l_c, l_t</code>	<code>if(is_num(\$t1)) {...}</code>

```

$v = DB.get($_GET[ 'child' ] );
$x = "";
if (DB.isMember($v)) {
    while (DB.hasParent($v)) {
        echo($x);
        $x = $_POST[ '$v' ];
        $v = DB.getParent($v);
    }
    echo($v);
}

```

```


 $l_0: v = \bigcirc$ 
 $l_1: x = \text{filter}$ 
 $l_2: \text{validate}(v, l_3, l_9)$ 
 $l_3: \text{bra } l_4, l_8$ 
 $l_4: \bullet = x$ 
 $l_5: v = \bigcirc$ 
 $l_6: v = \otimes(v)$ 
 $l_7: \text{bra } l_3$ 
 $l_8: \bullet = v$ 
 $l_9: \text{bra } l_9$ 

```

Path as Control Flow

```
$v = DB.get($_GET['child']);  
$x = "";  
if (DB.isMember($v)) {  
    while (DB.hasParent($v)) {  
        echo($x);  
        $x = $_POST['$v'];  
        $v = DB.getParent($v);  
    }  
    echo($v);  
}
```

- 1) Does this program contain a vulnerability?
- 2) What is the control flow graph of this program?



**A path is the
flow of
control that
exists in the
program**

Path as Control Flow

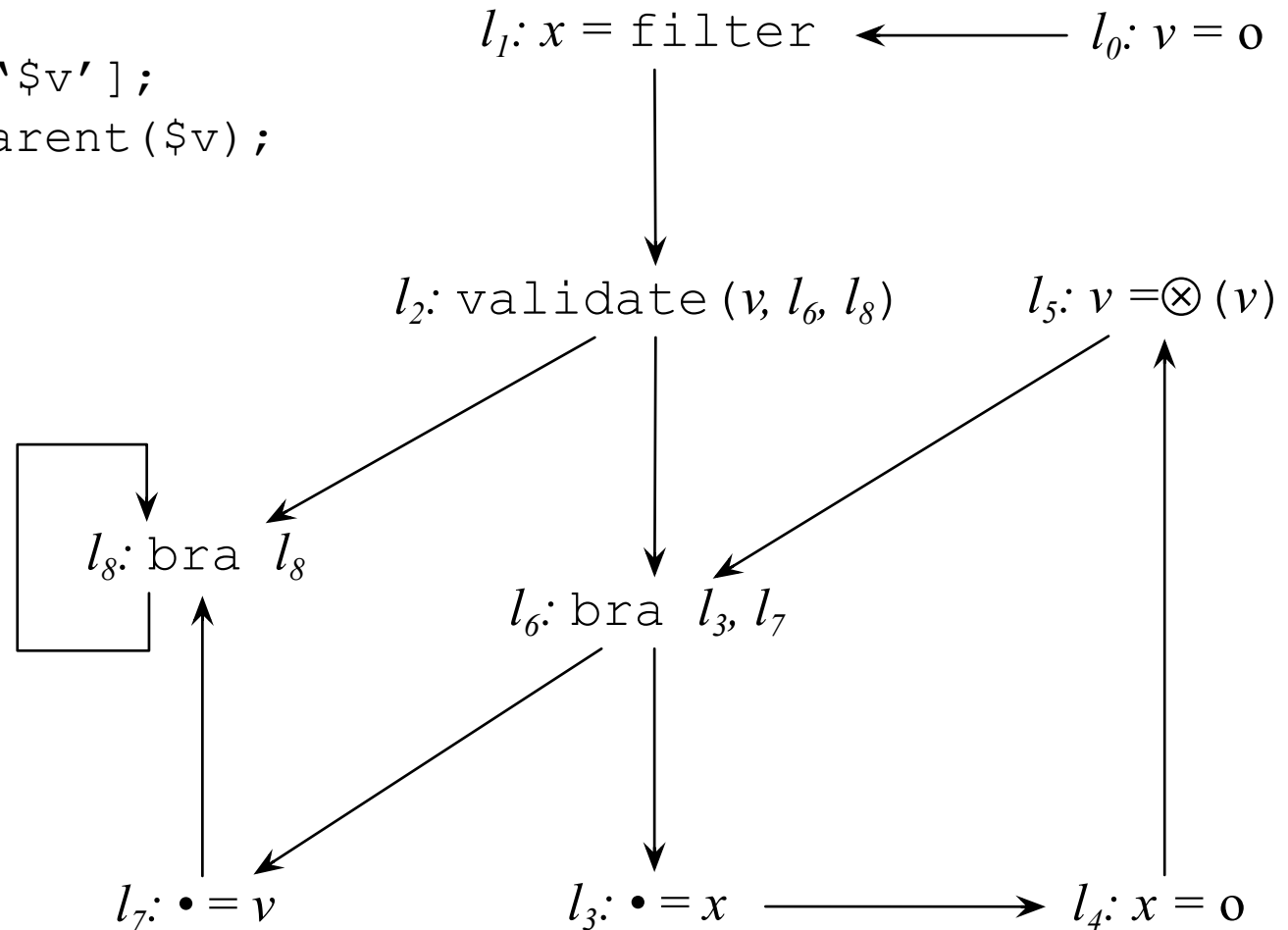
```

$v = DB.get($_GET['child']);
$x = "";
if (DB.isMember($v)) {
    while (DB.hasParent($v)) {
        echo($x);
        $x = $_POST['$v'];
        $v = DB.getParent($v);
    }
    echo($v);
}

```

Can you find a vulnerable path along the control flow graph of this program?

$l_0: v = \bigcirc$
 $l_1: x = \text{filter}$
 $l_2: \text{validate}(v, l_3, l_9)$
 $l_6: \text{bra } l_3, l_7$
 $l_3: \bullet = x$
 $l_4: x = \bigcirc$
 $l_5: v = \otimes(v)$
 $l_7: \bullet = v$
 $l_8: \text{bra } l_8$

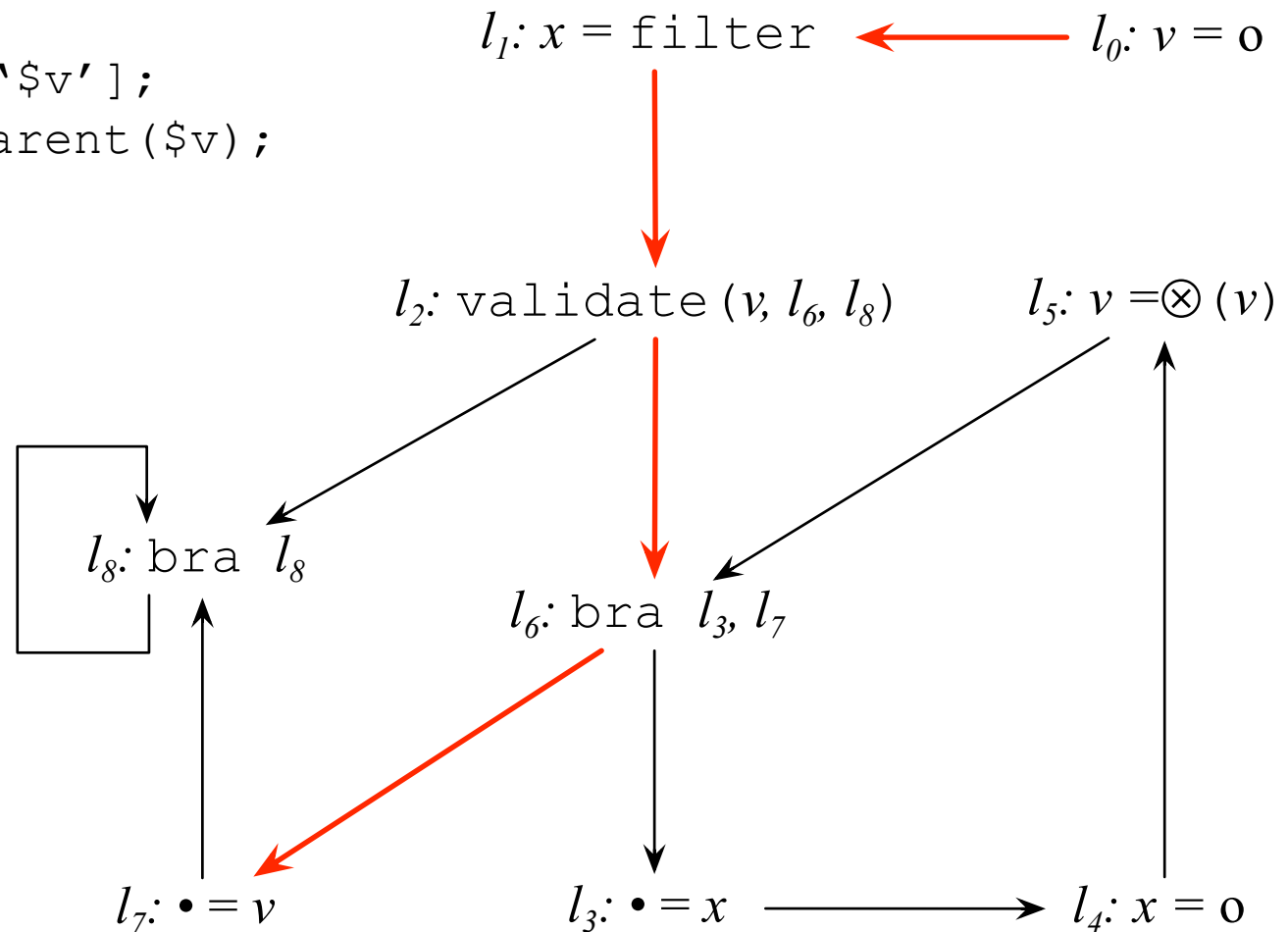


Path as Control Flow

```

$v = DB.get($_GET['child']);
$x = "";
if (DB.isMember($v)) {
  while (DB.hasParent($v)) {
    echo($x);
    $x = $_POST['$v'];
    $v = DB.getParent($v);
  }
  echo($v);
}

```



Tracking the Program Data Flow

```

$v = DB.get($_GET['child']);
$x = "";
if (DB.isMember($v)) {
  while (DB.hasParent($v)) {
    echo ($v);
    $x = $_POST['svl'];
    $v = DB.getParent($v);
  }
  echo ($v);
}

```

The mission:

Provide an algorithm that determines if the program contains a vulnerability.

- * Does it terminate?
- * What is the complexity?
- * Is it sound?

$l_7: \bullet = v$

$l_3: \bullet = x$

$l_4: x = 0$

$l_1: x = \text{filter}$

$l_0: v = 0$

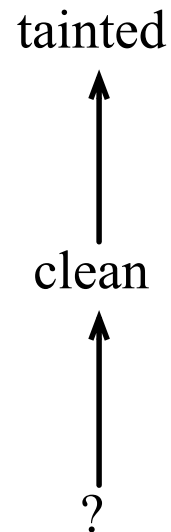
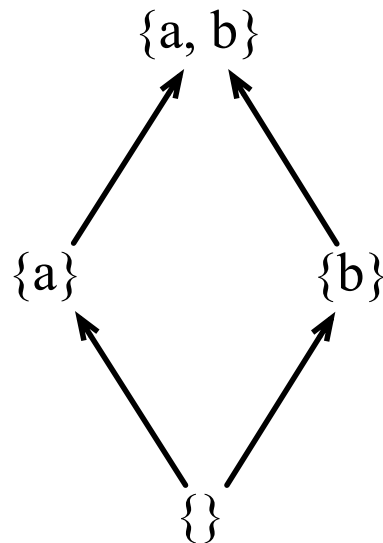
$l_5: v = \otimes(v)$

$l_6: \text{bra } l_3, l_7$

$l_8: \text{bra } l_8$

Data Flow Analysis

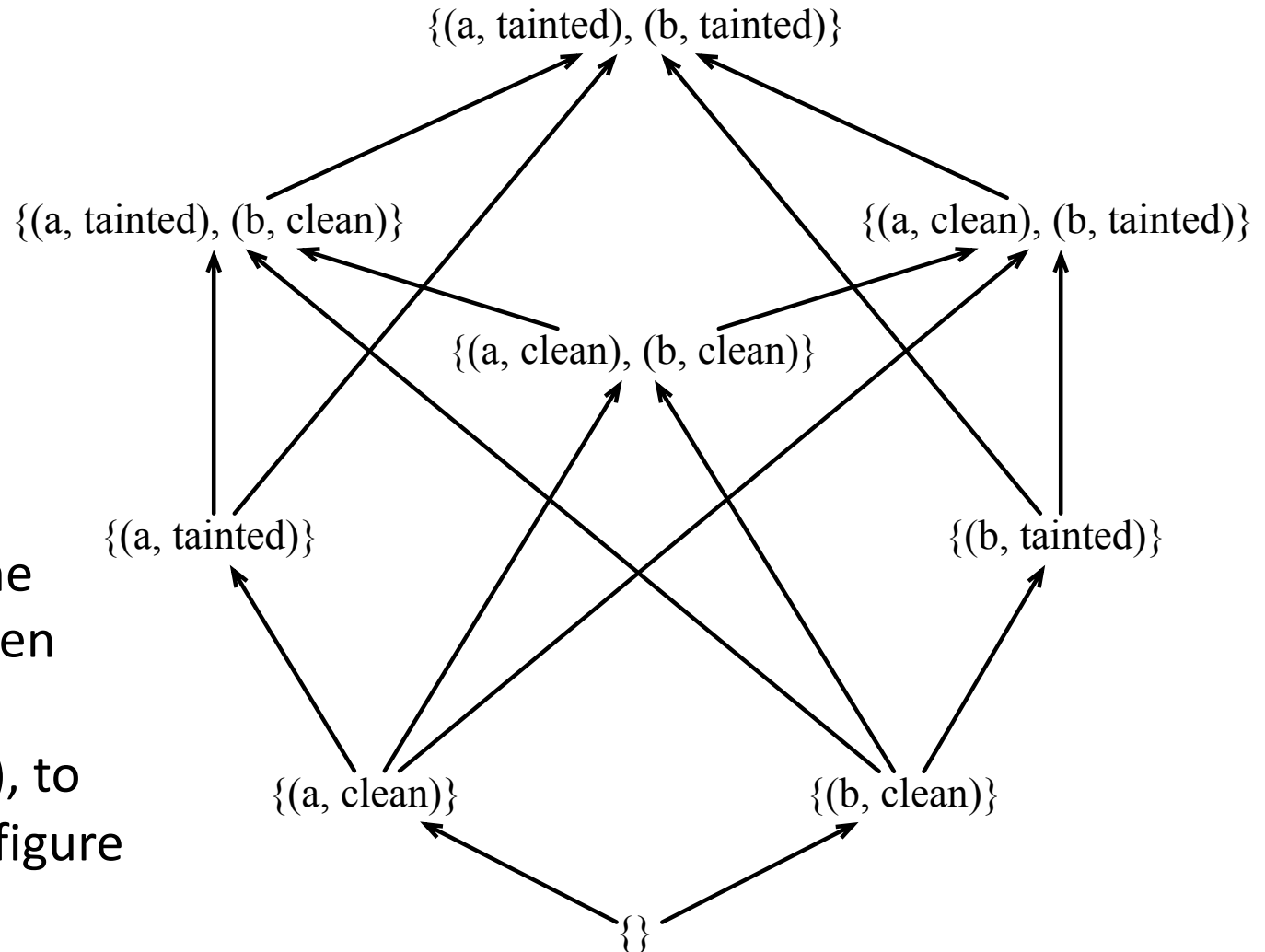
- A program point is any point between two consecutive instructions.
- Let's associate with each program point a function that maps variables to either clean or tainted.
- This function is, indeed, a point in the product lattice of the two lattices below, assuming two variables, a and b:



Which lattice do we obtain from the product of these two lattices?

Data Flow Analysis

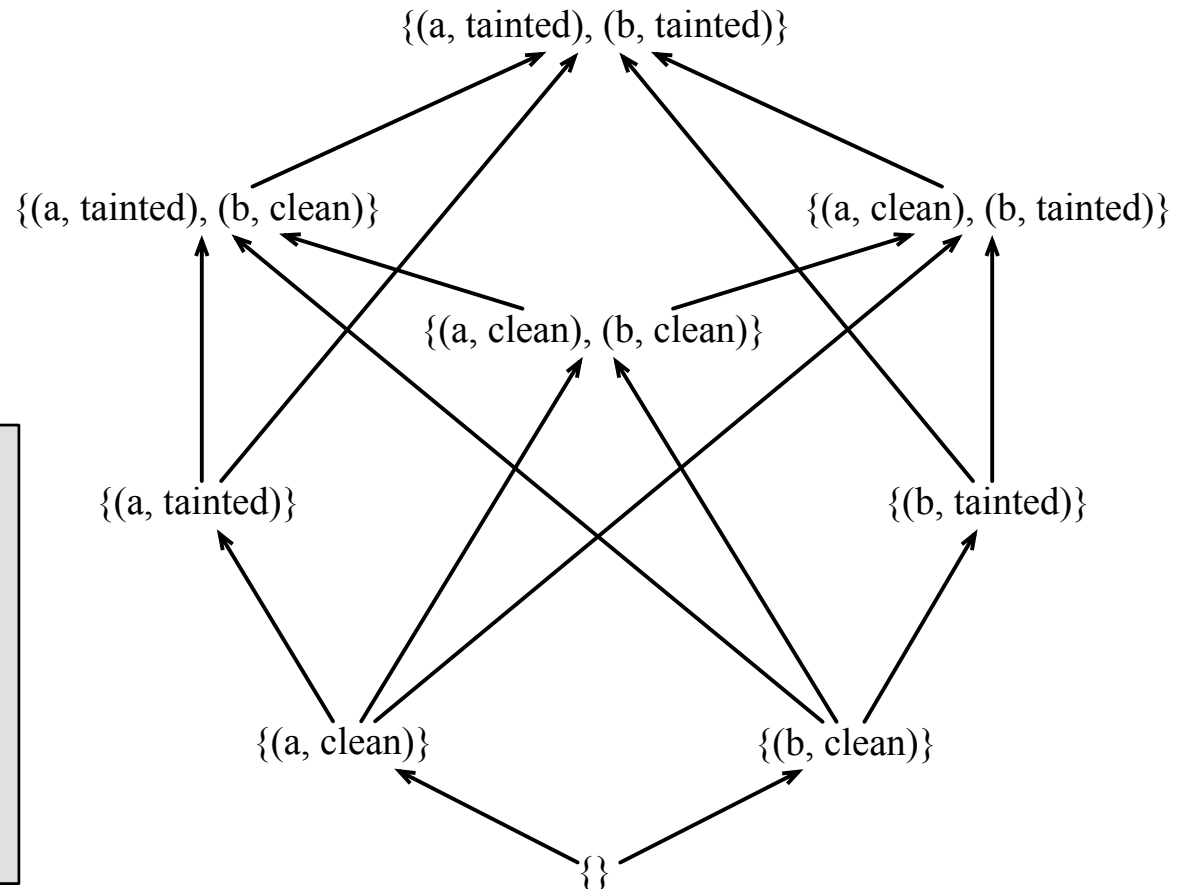
- We will use this product lattice, whose diagram is seen below:



We did not draw the associations between variables and the undefined value (?), to avoid clogging the figure too much.

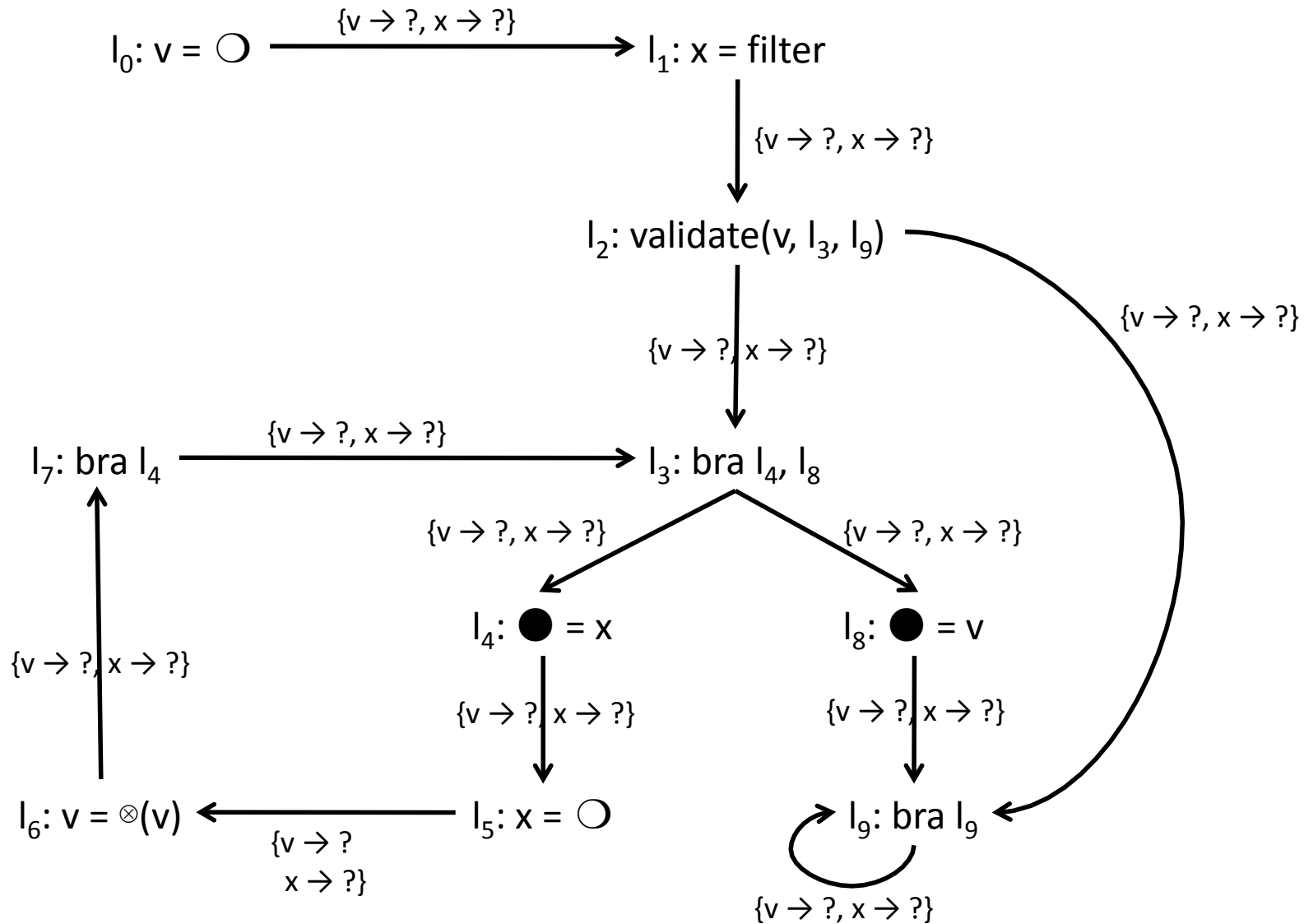
Dense Data Flow Analysis

- Notice that our analysis will be *dense*:
 - We are associating each pair (p, v) with an abstract state, where p is a program point, and v is a variable.

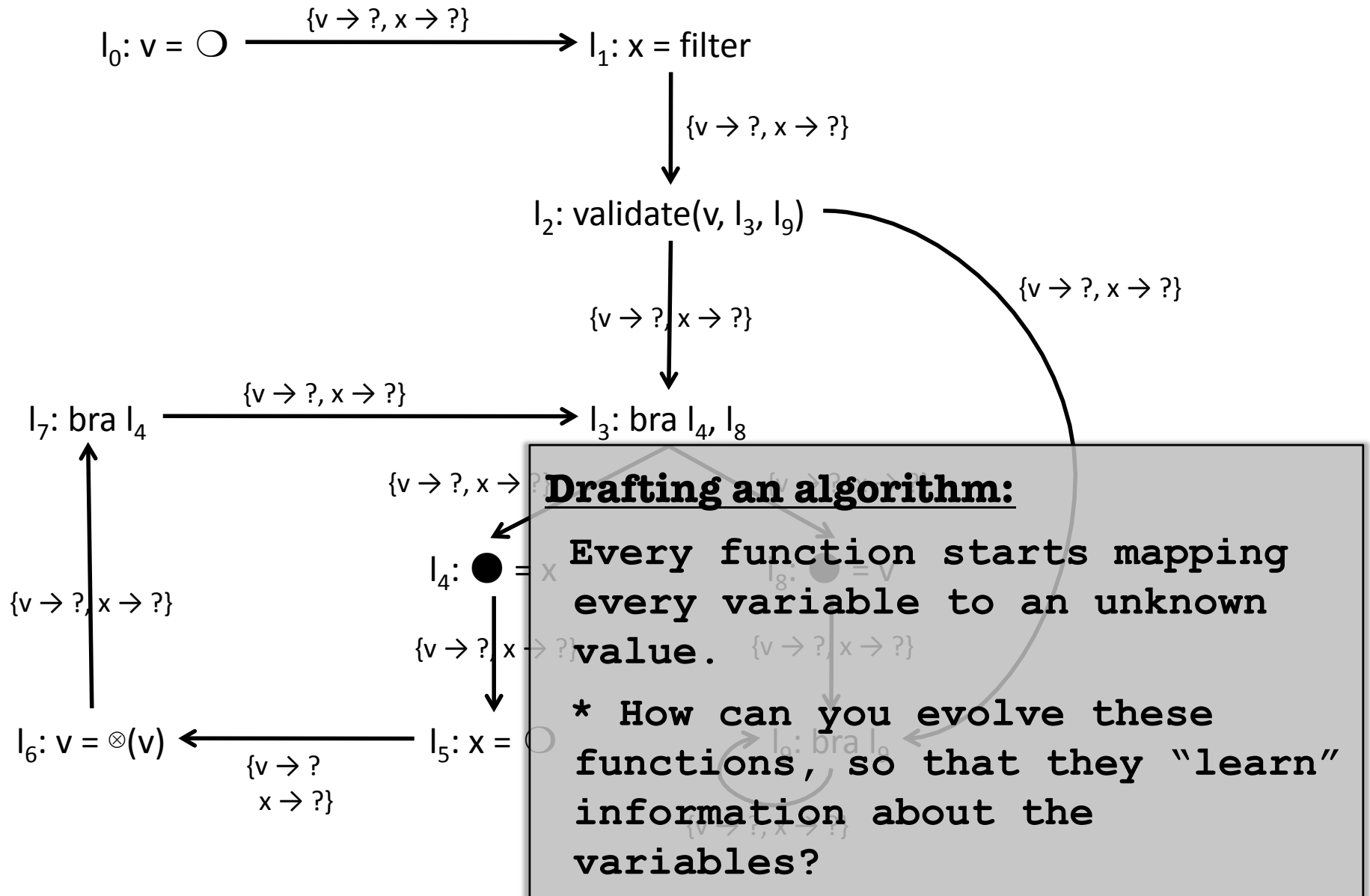


So, given a program with V variables, and P points, how many iterations we will need, in the worst case, to determine the abstract state of all the pairs $(\text{vars} \times \text{points})$?

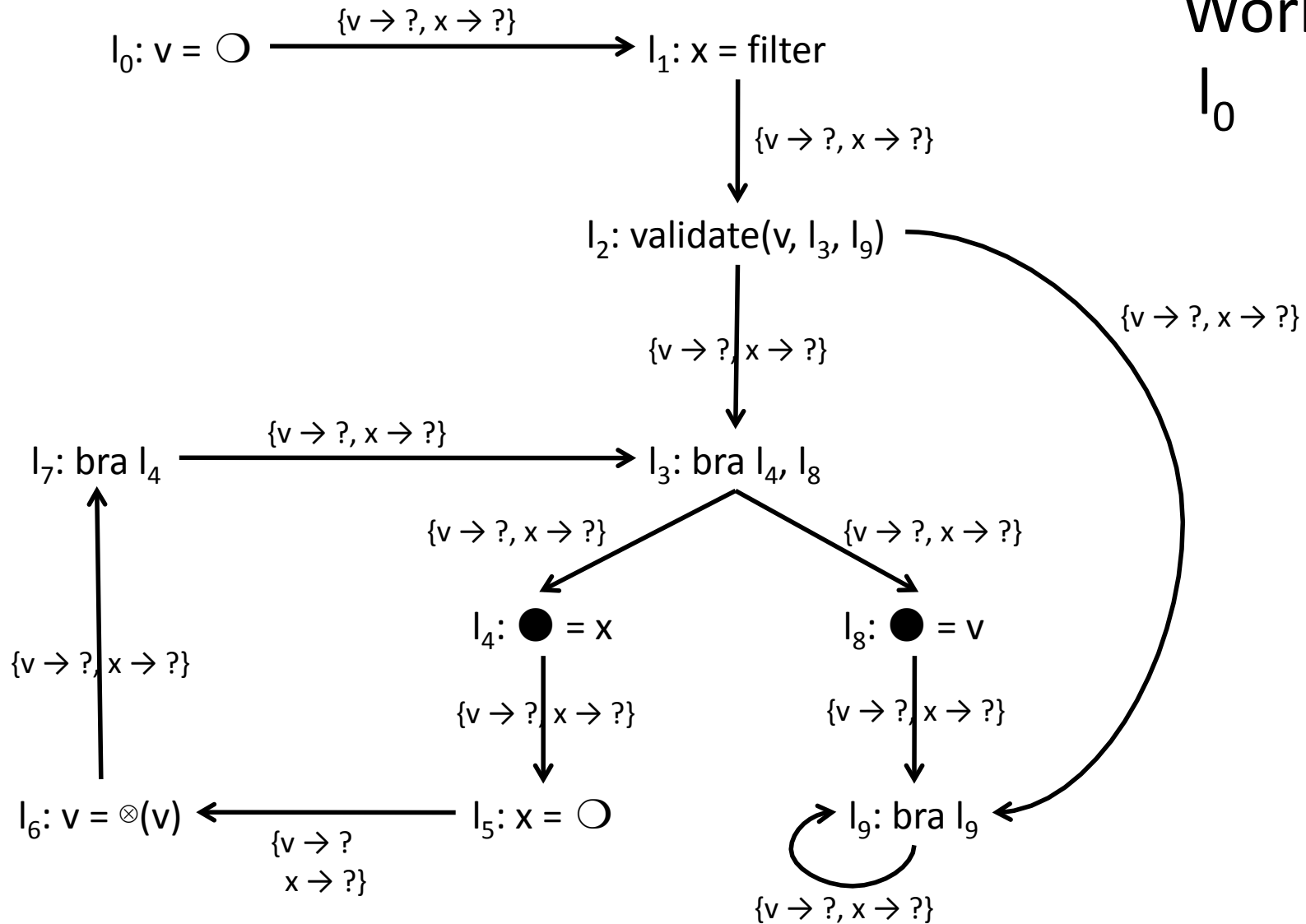
Example of dense approach



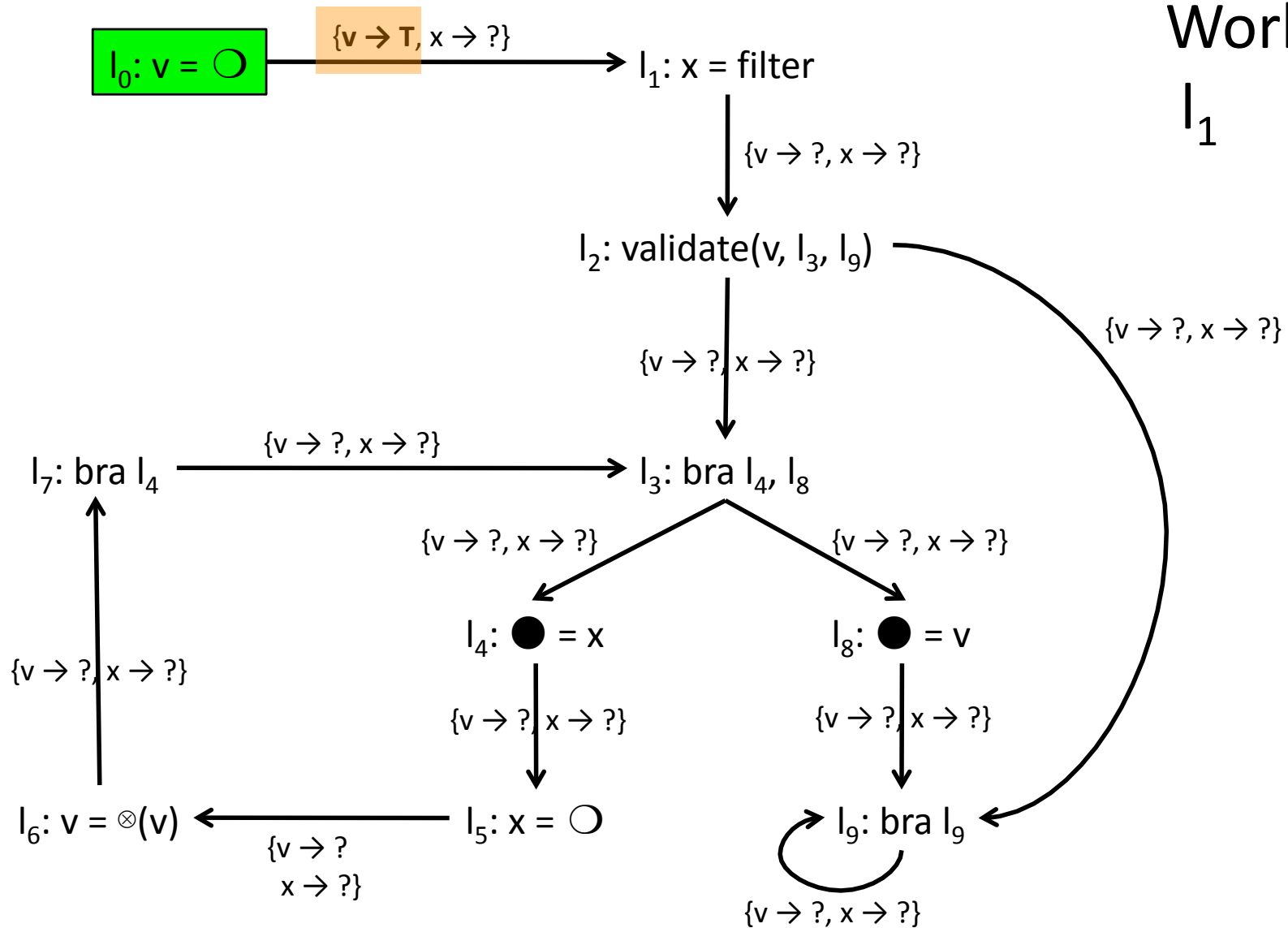
Example of dense approach



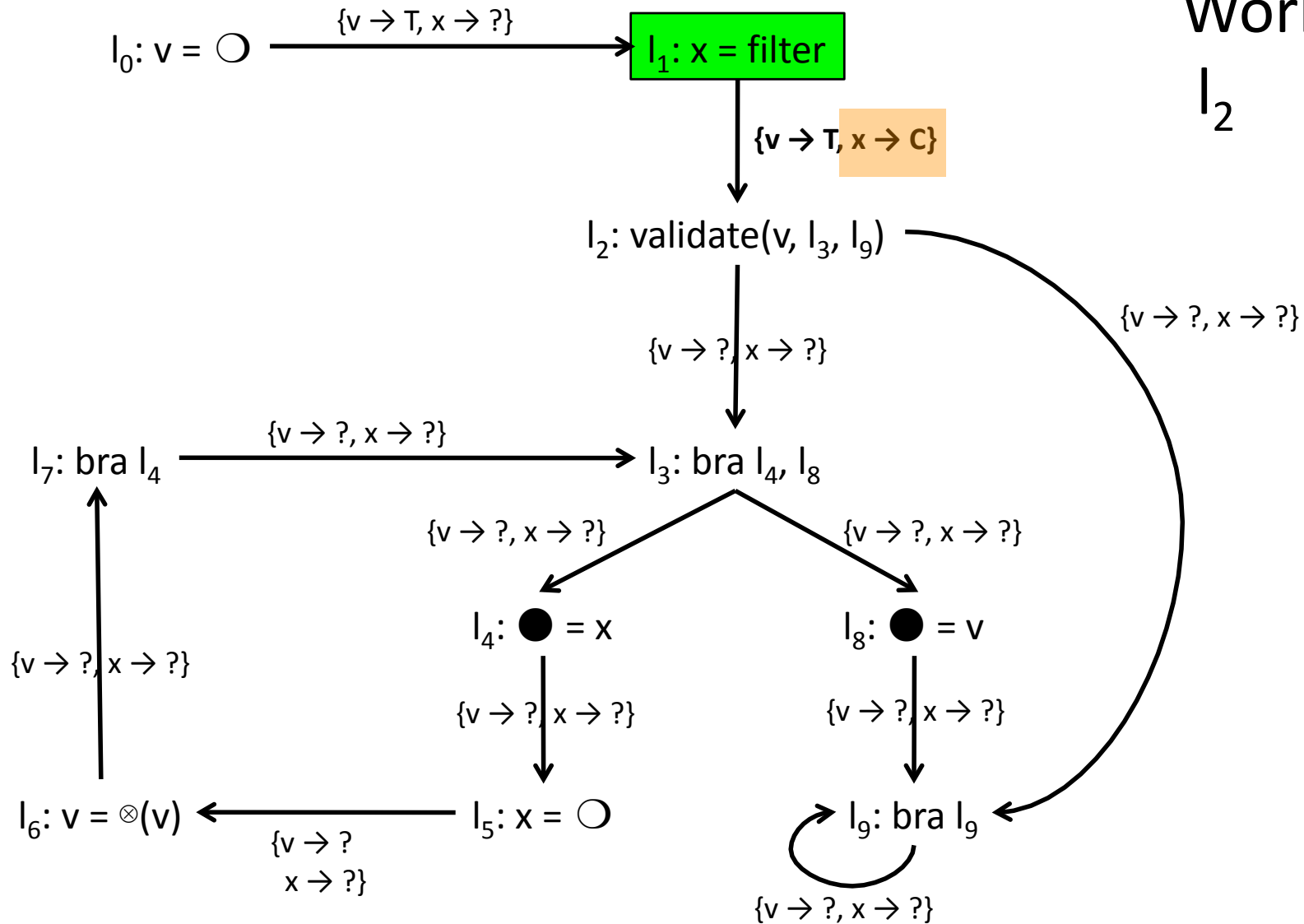
Example of dense approach



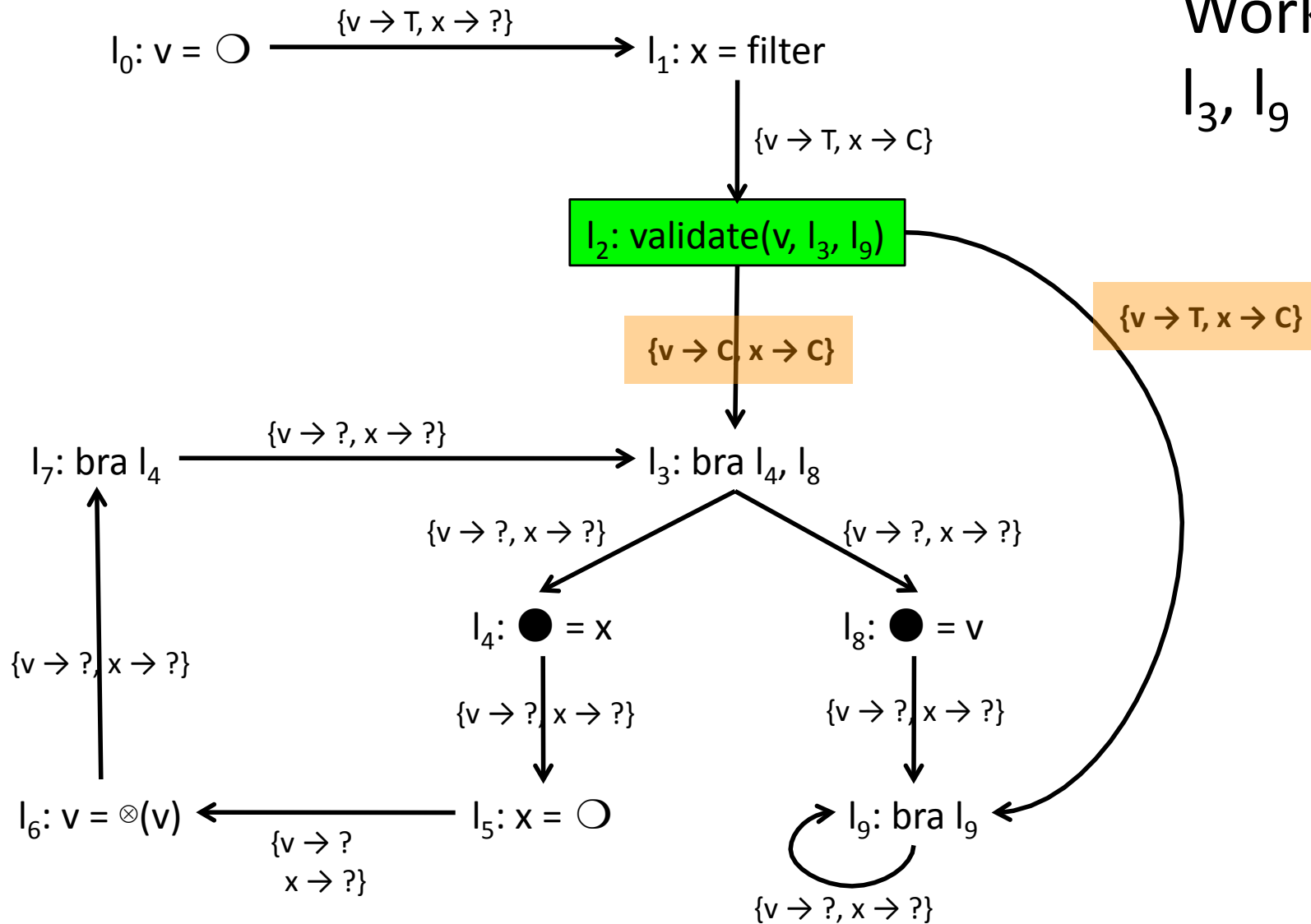
Example of dense approach



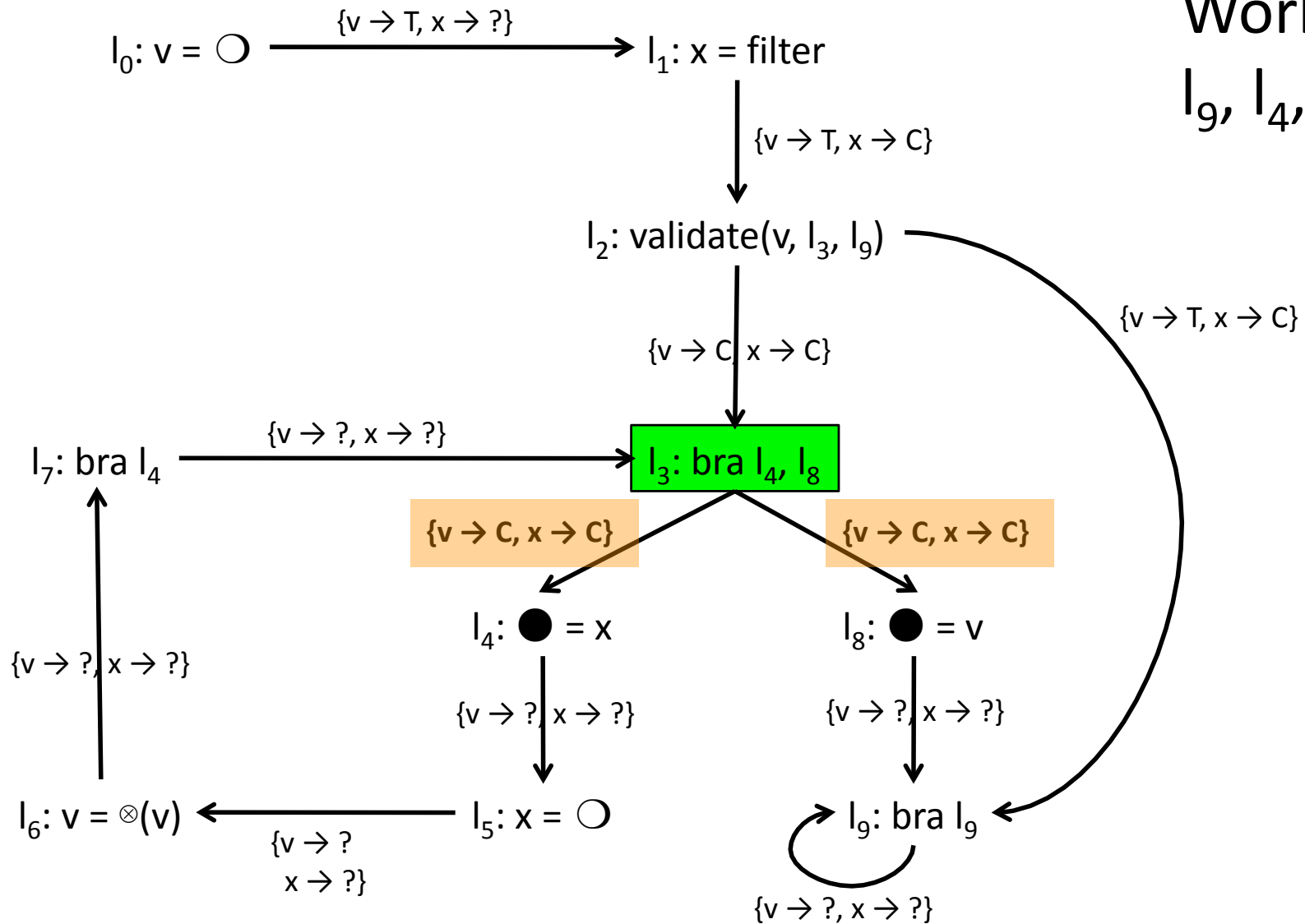
Example of dense approach



Example of dense approach

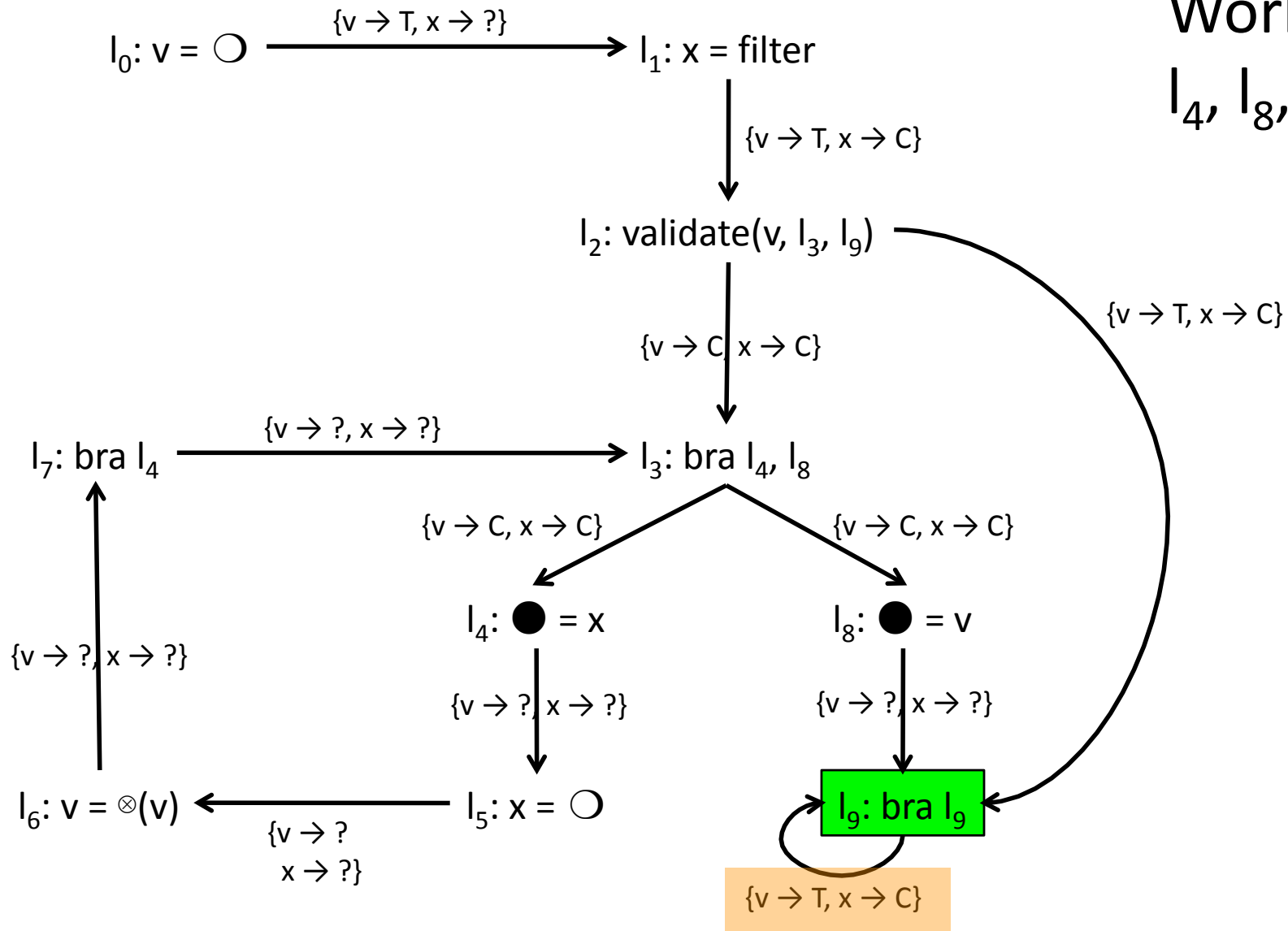


Example of dense approach

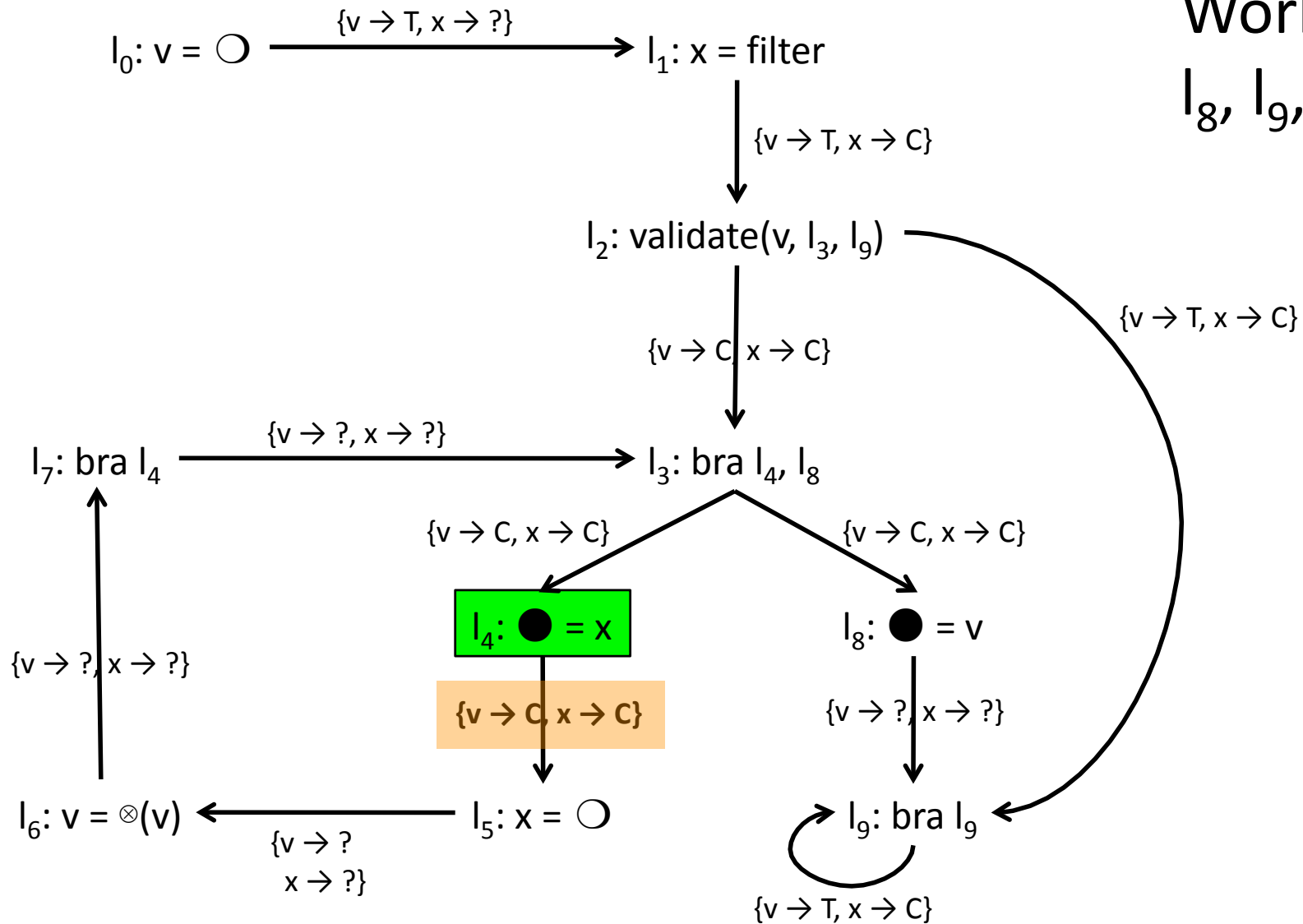


Worklist:
 l_9, l_4, l_8

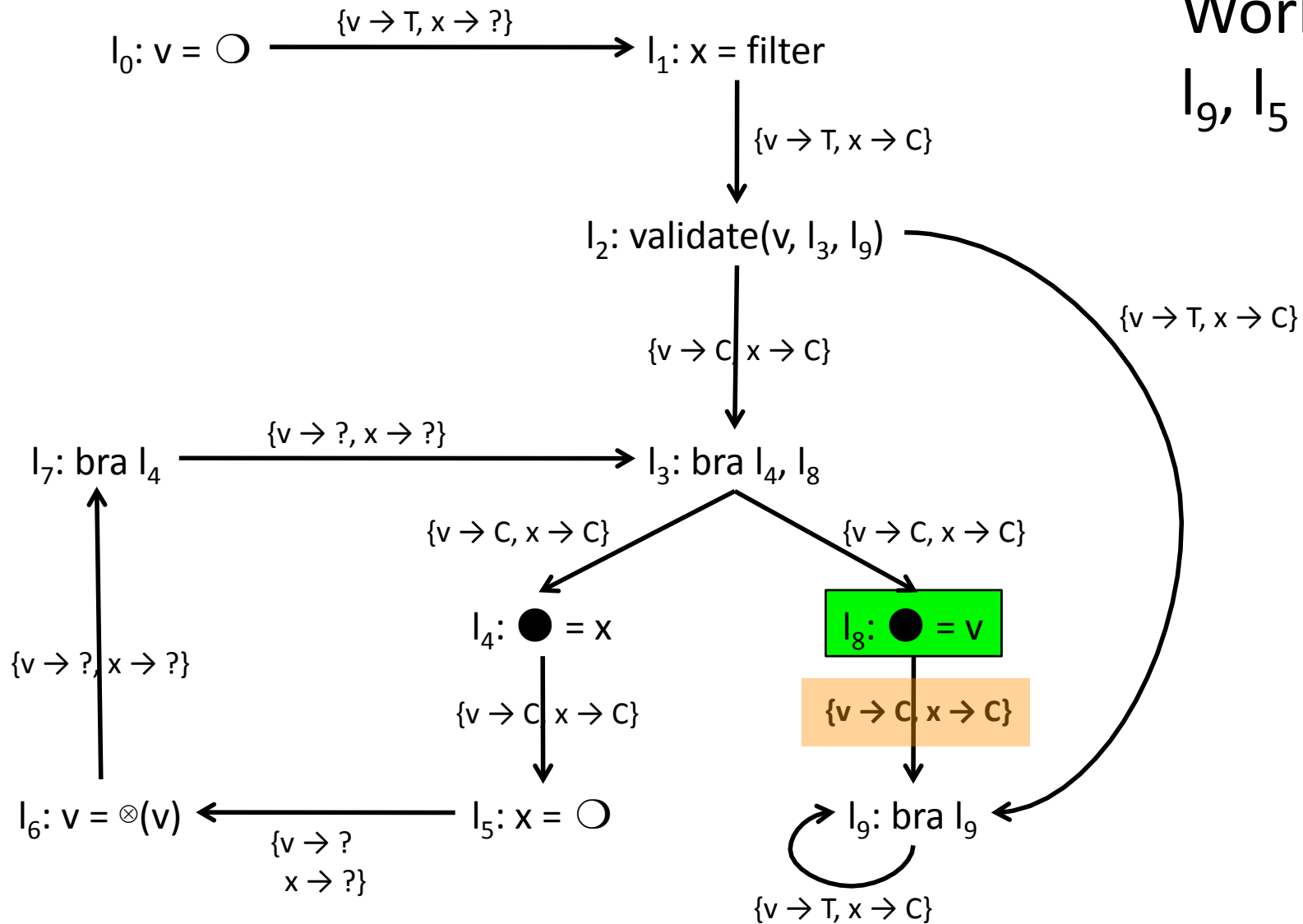
Example of dense approach



Example of dense approach



Example of dense approach



Example of dense approach

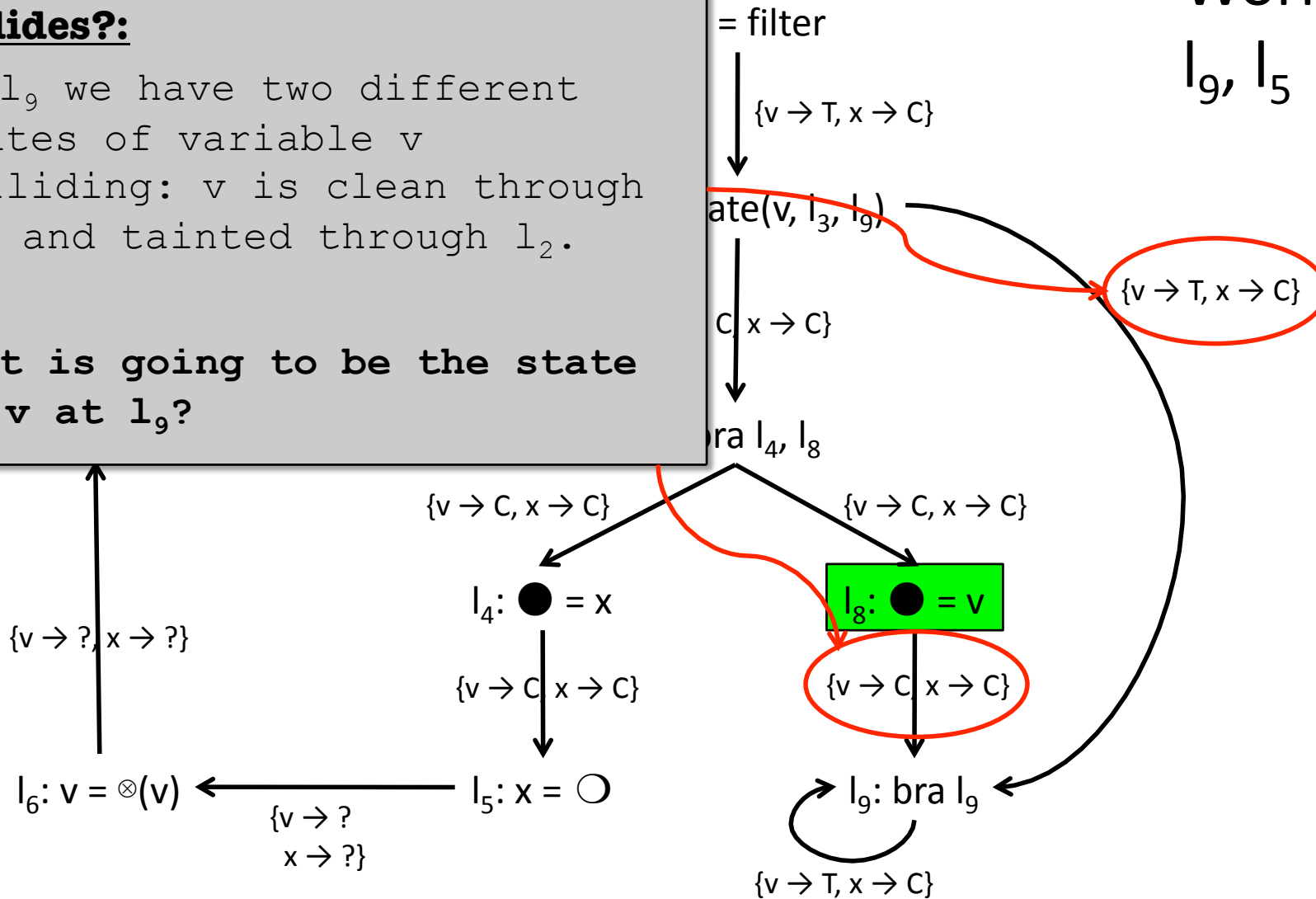
What happens when information collides?

At l_9 we have two different states of variable v colliding: v is clean through l_8 , and tainted through l_2 .

What is going to be the state of v at l_9 ?

Worklist:

l_9, l_5



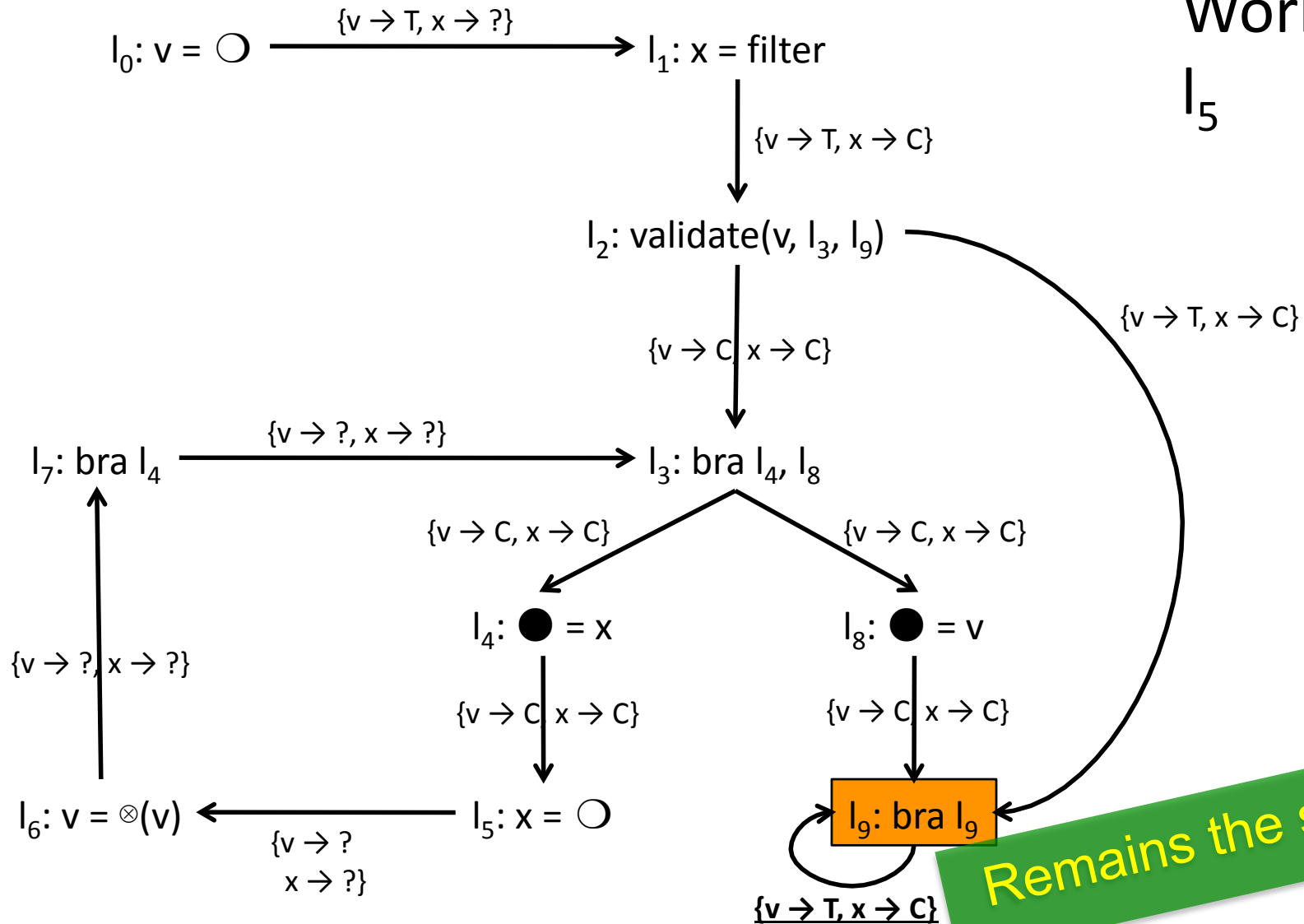
The Join Operator

\vee	Undefined	Clean	Tainted
Undefined	Undefined	Clean	Tainted
Clean	Clean	Clean	Tainted
Tainted	Tainted	Tainted	Tainted

- The join operator defines what happens at the joining points in the program.
- We can easily extend its definition to our product lattice, e.g., $\{(a, \text{tainted}), (b, \text{clean})\} \vee \{(a, \text{clean}), (b, \text{undefined})\} = \{(a, \text{tainted}), (b, \text{clean})\}$

Is this analysis
may or must?

Example of dense approach

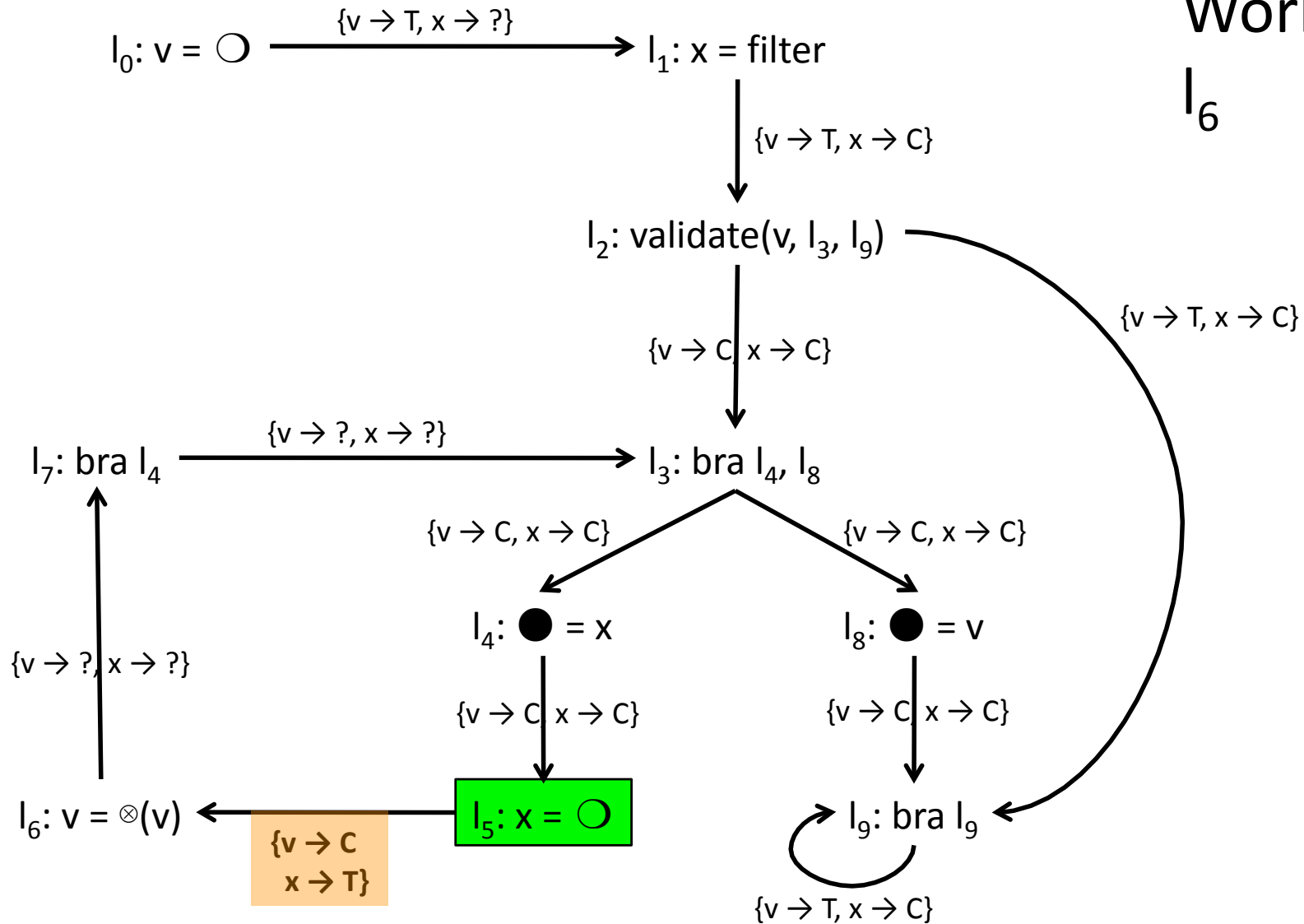


Worklist:

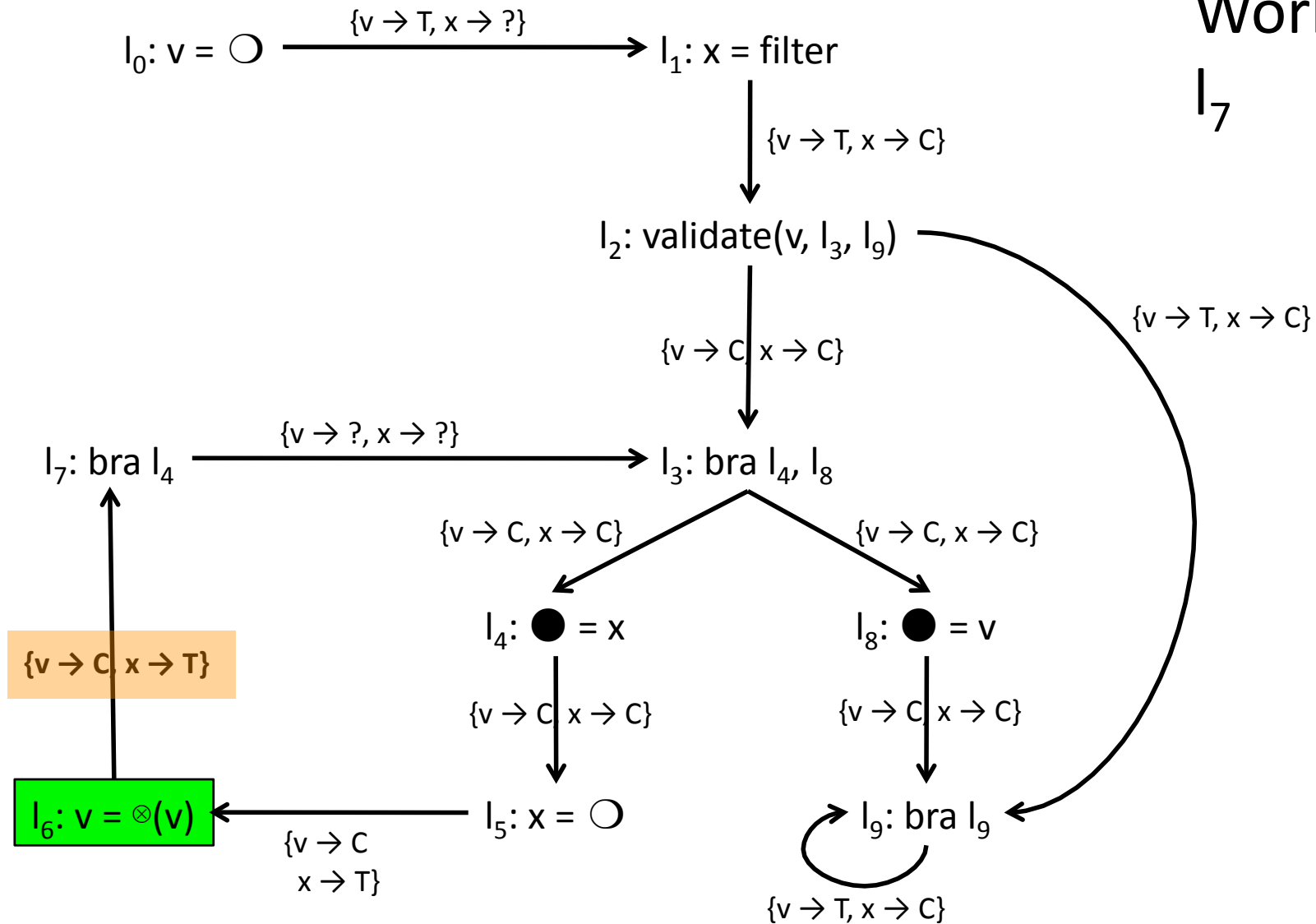
l_5

Remains the same!

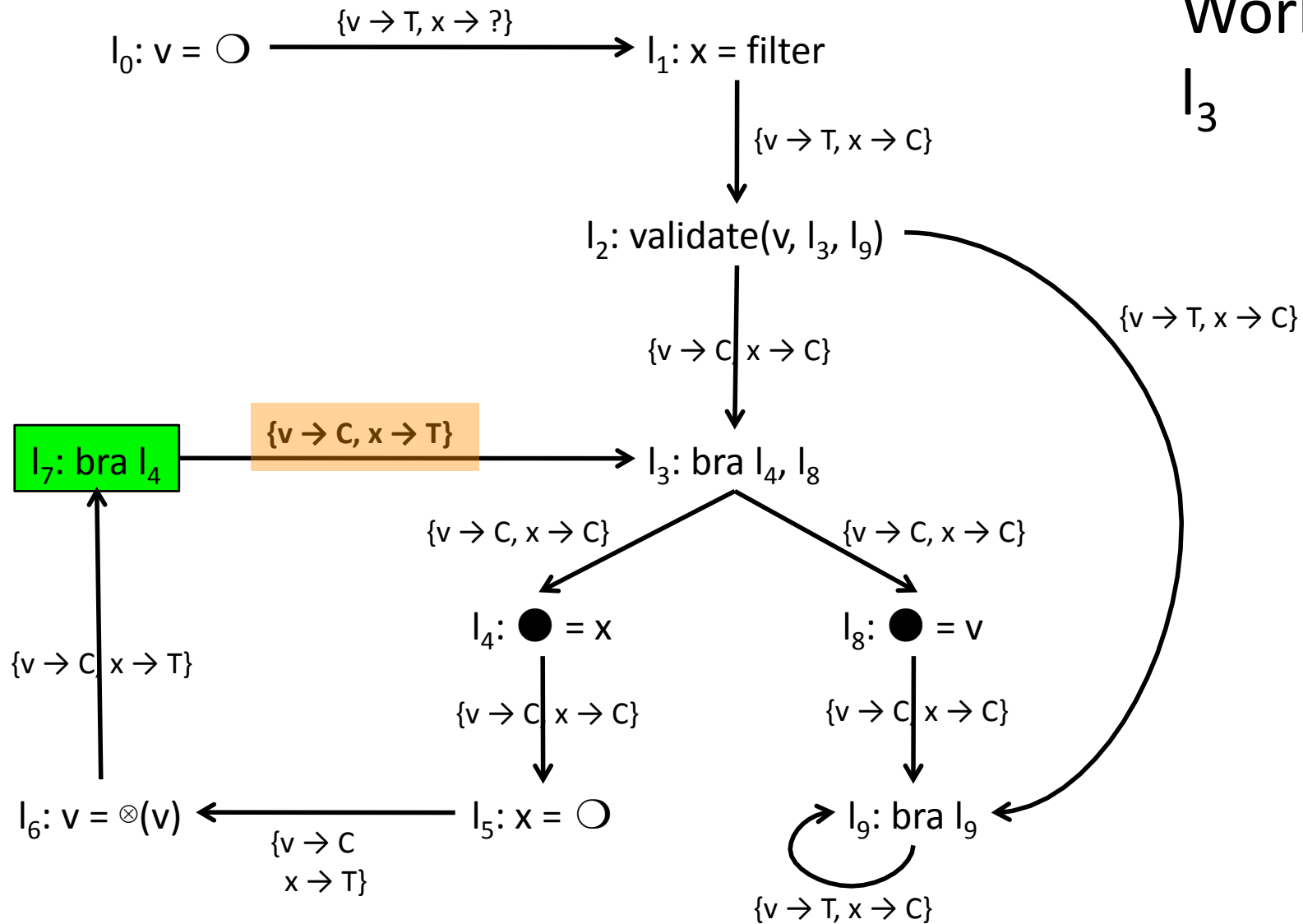
Example of dense approach



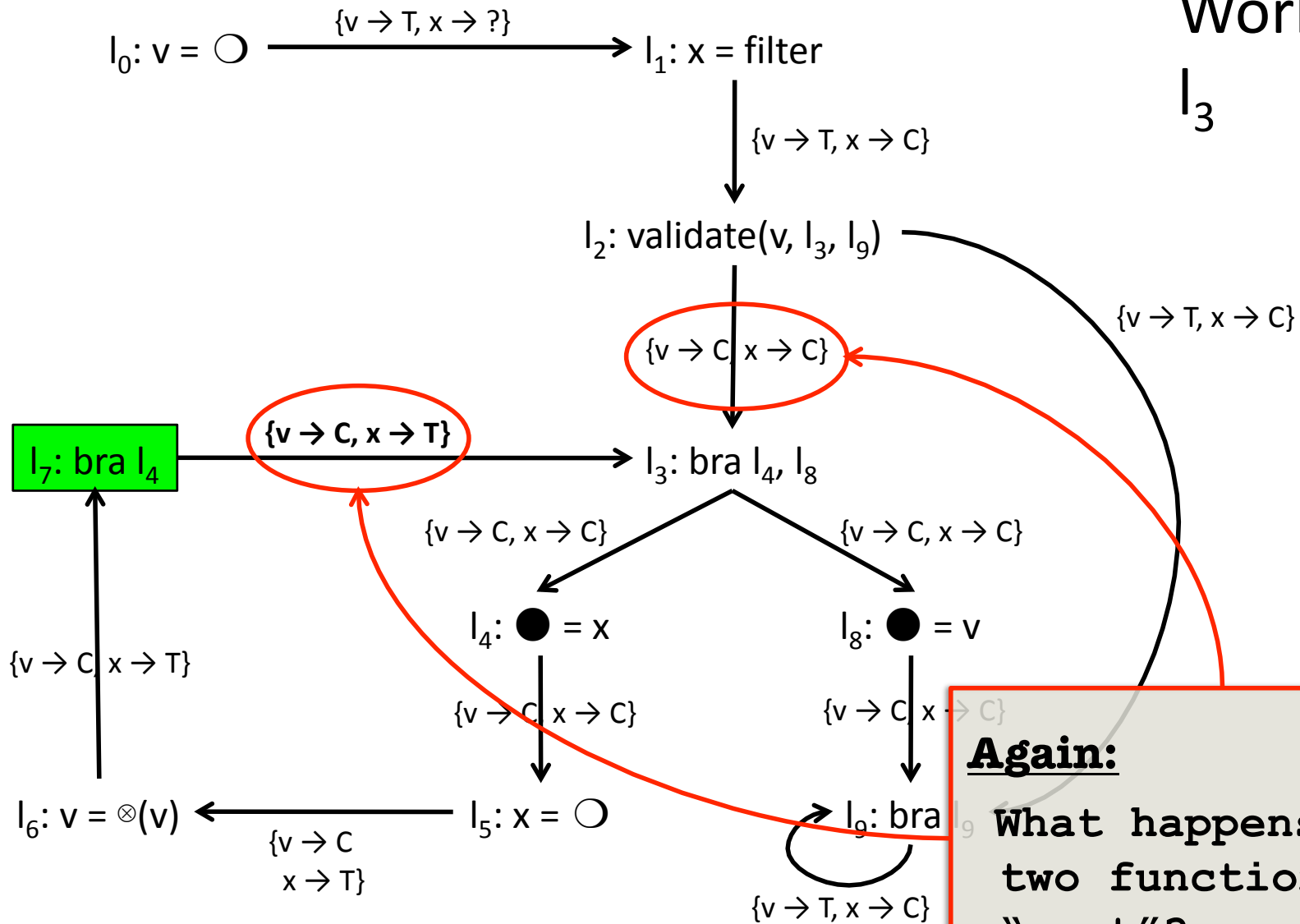
Example of dense approach



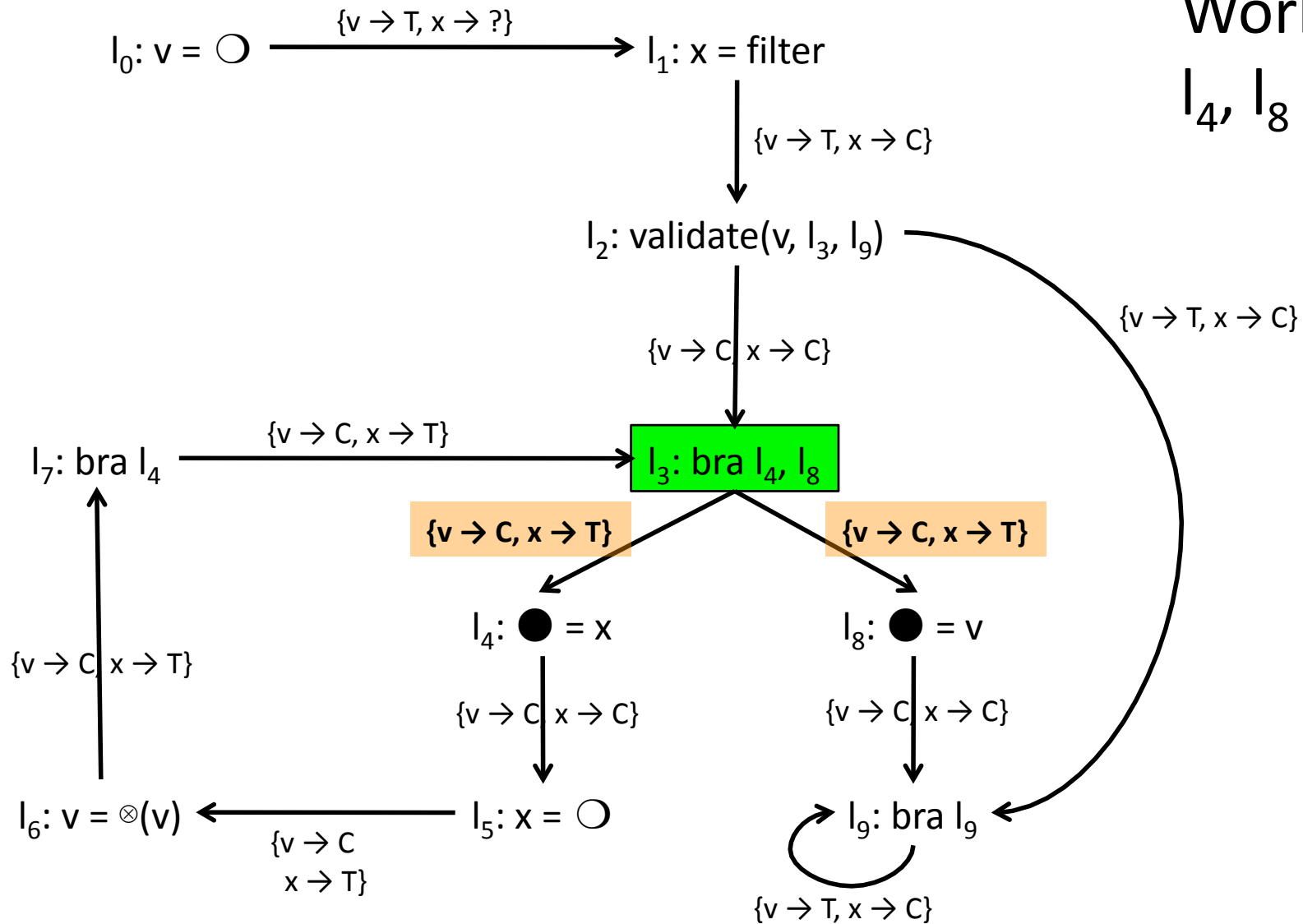
Example of dense approach



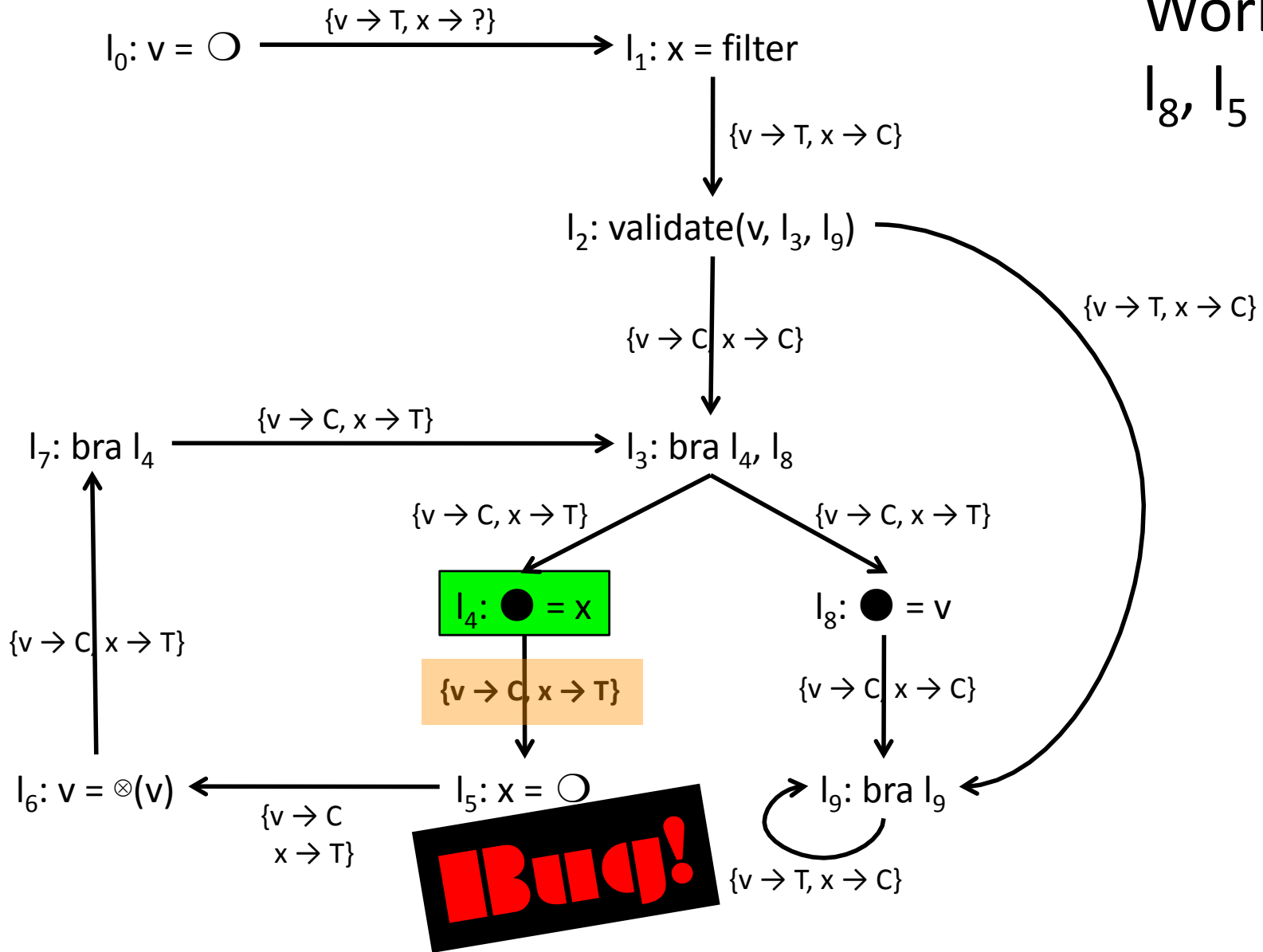
Example of dense approach



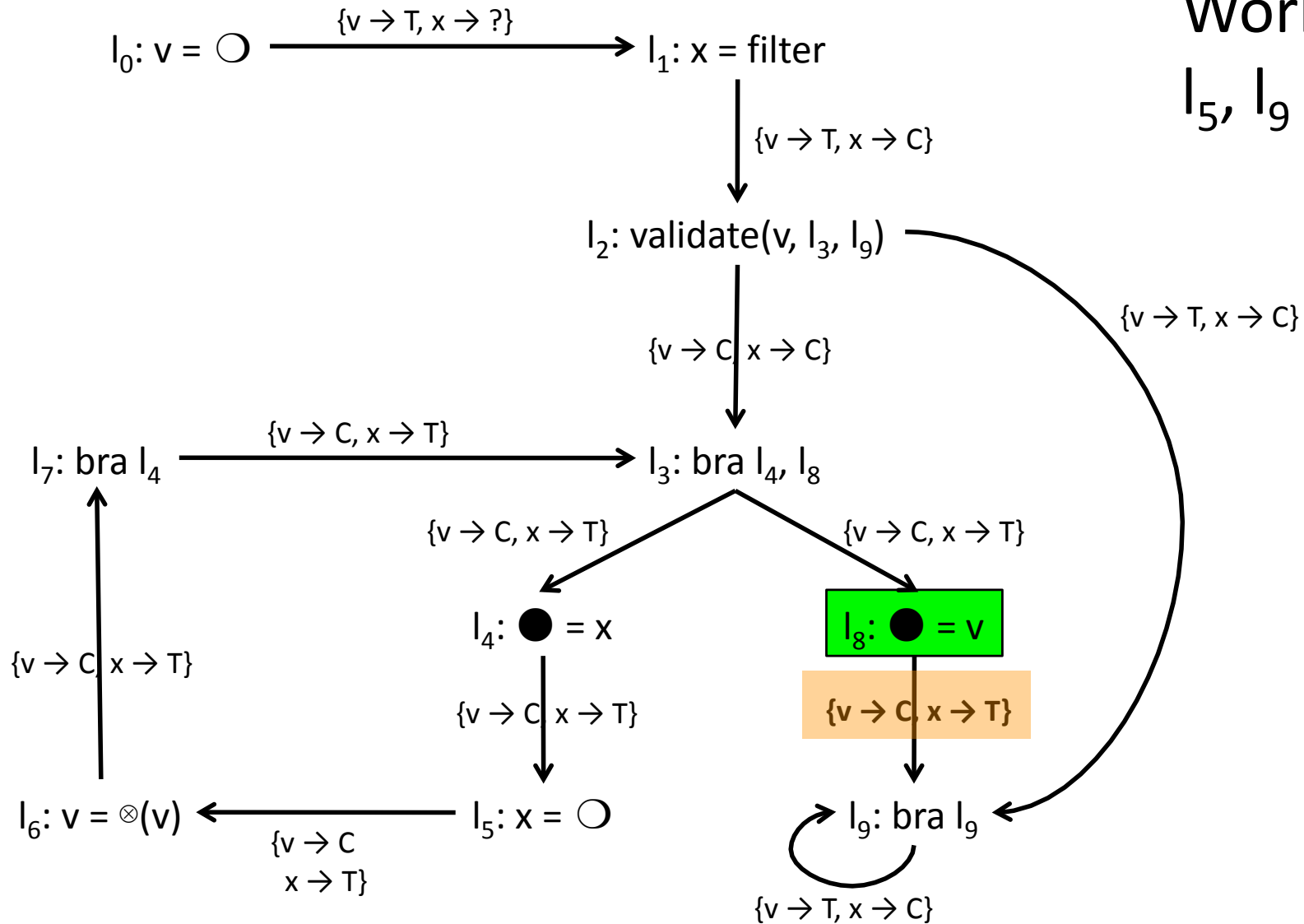
Example of dense approach



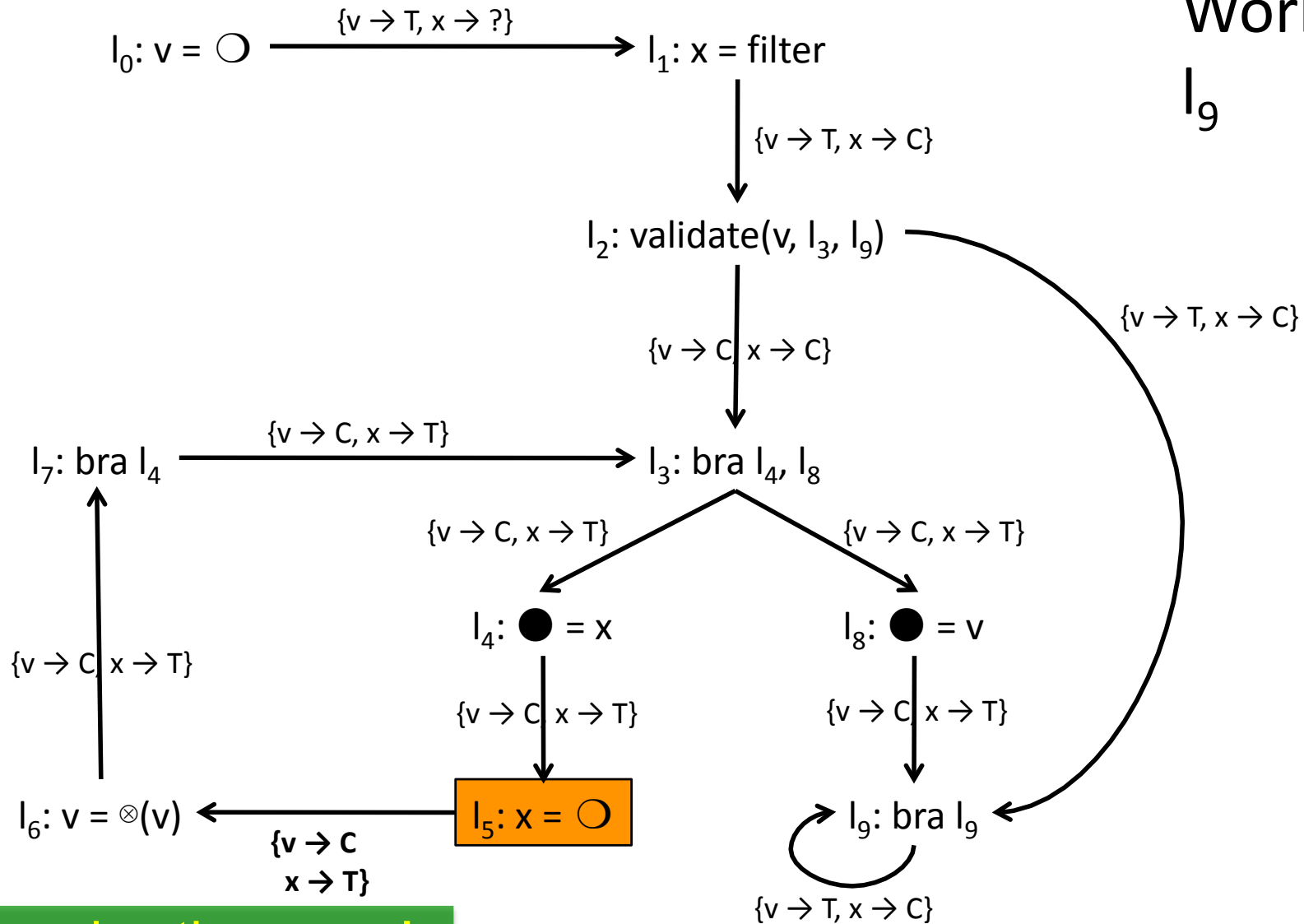
Example of dense approach



Example of dense approach

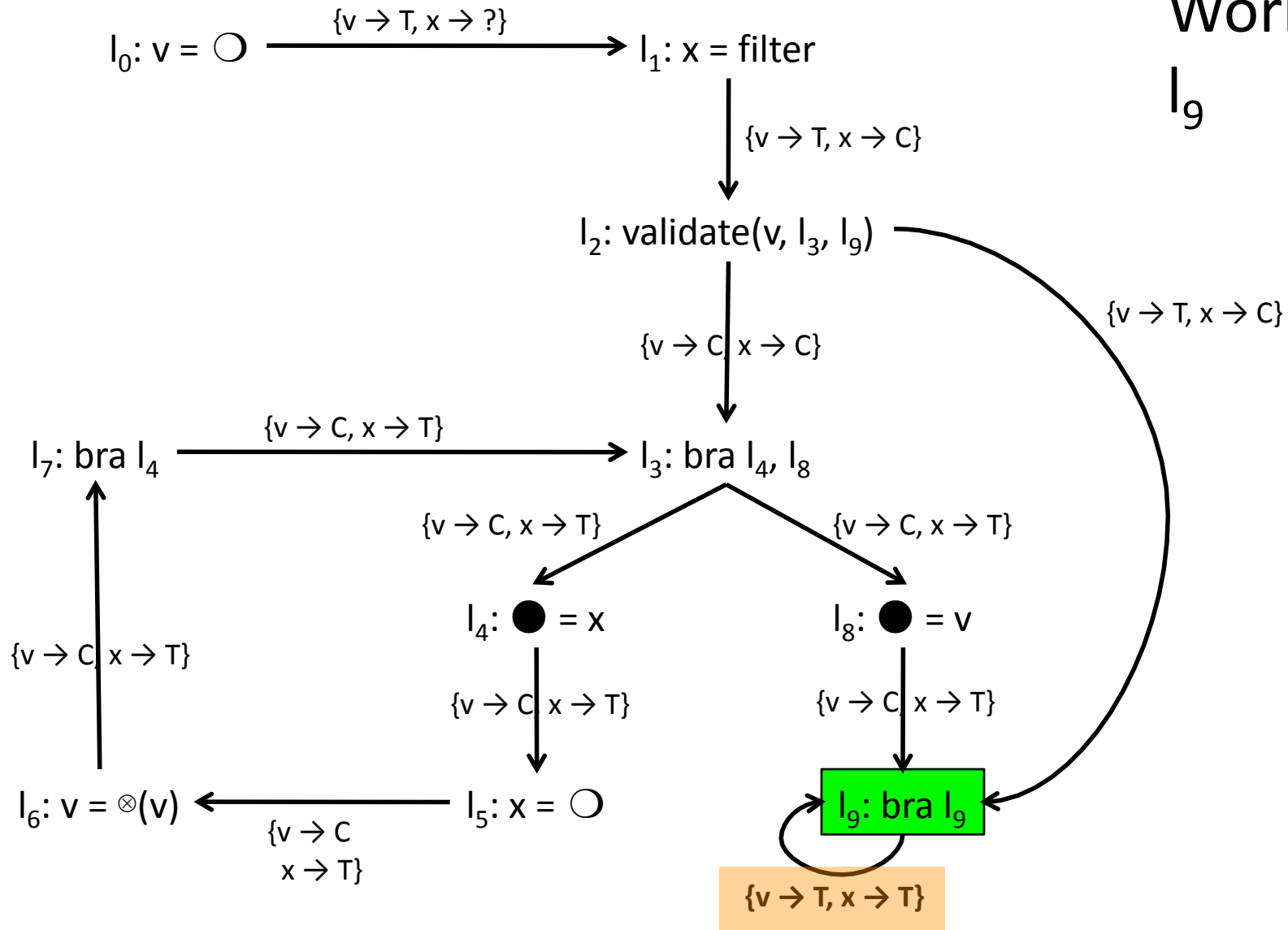


Example of dense approach

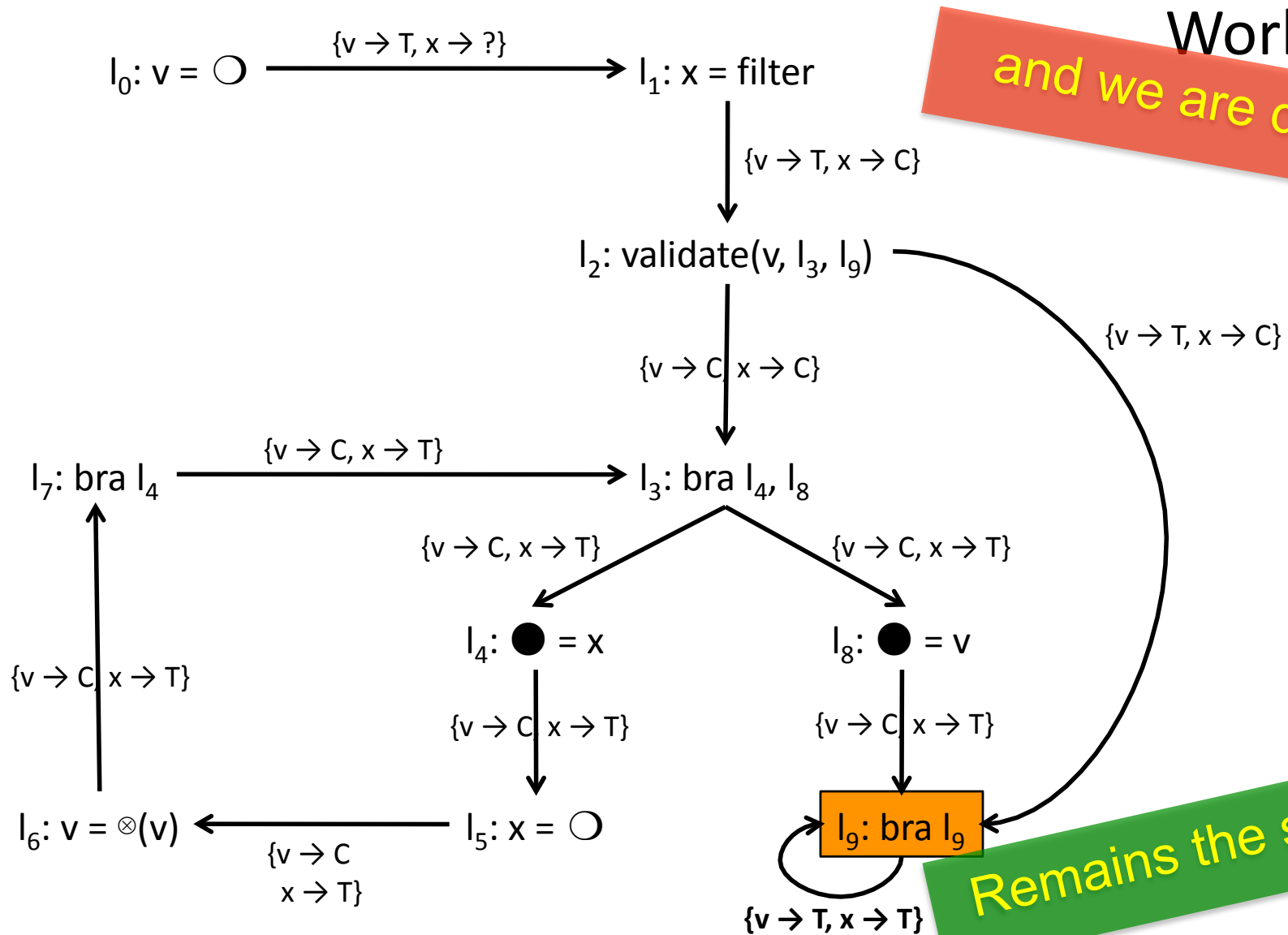


Remains the same!

Example of dense approach



Example of dense approach



Ensuring Termination

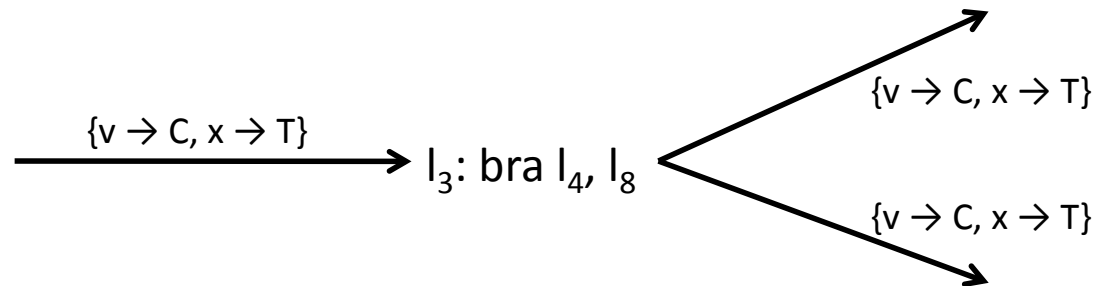
- Does this algorithm always terminates?
 - It does, because each variable can be mapped to either ?, clean or tainted. Once it reaches tainted, it does not change anymore. If every variable in a program point becomes tainted, that program point will not be added to the work list again.
- So, what is the complexity of this algorithm?
 - It is $O(P \times V \times M)$, where P is the number of program points, V is the number of variables, and M is the number of predecessors of a program point. Usually, $O(P) = O(V)$; and $O(M) = O(1)$. Thus, we have $O(V^2)$.

SPARSE TAINTED FLOW ANALYSIS



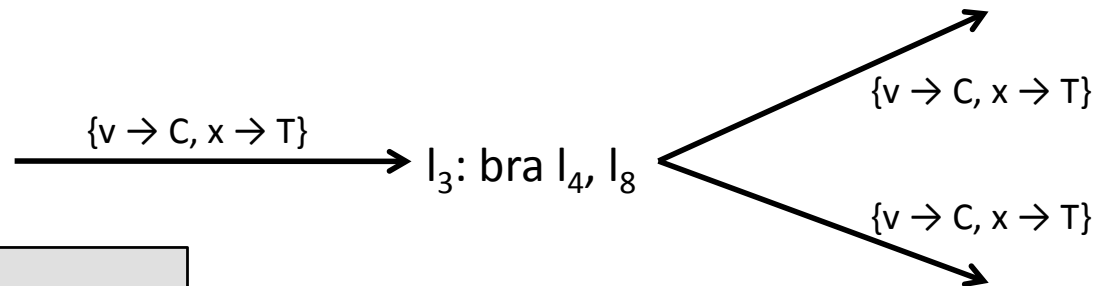
Problems with dense analyses

- Lots of redundant information
 - Some nodes just pass forward the information that they receive.
 - These nodes are bound to identity transfer functions:



Problems with dense analyses

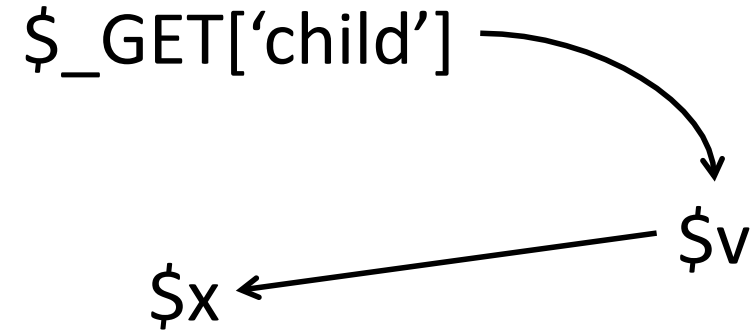
- Lots of redundant information
 - Some nodes just pass forward the information that they receive.
 - These nodes are bound to identity transfer functions:



We had two notions of "path". Do you remember the second?

Another notion of path

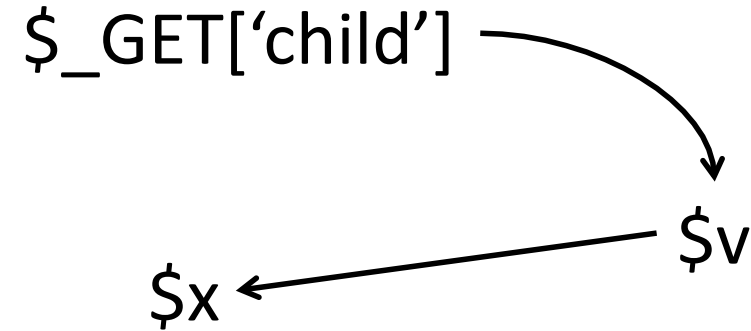
```
$v = DB.get($_GET['child']);  
$x = "";  
if (DB.isMember($v)) {  
    while (DB.hasParent($v)) {  
        echo($x);  
        $x = $_POST['$v'];  
        $v = DB.getParent($v);  
    }  
    echo($v);  
}
```



**A path is a
chain of data
dependences
in a program**

Sparse Analysis

```
$v = DB.get($_GET['child']);  
$x = "";  
if (DB.isMember($v)) {  
    while (DB.hasParent($v)) {  
        echo($x);  
        $x = $_POST['$v'];  
        $v = DB.getParent($v);  
    }  
    echo($v);  
}
```

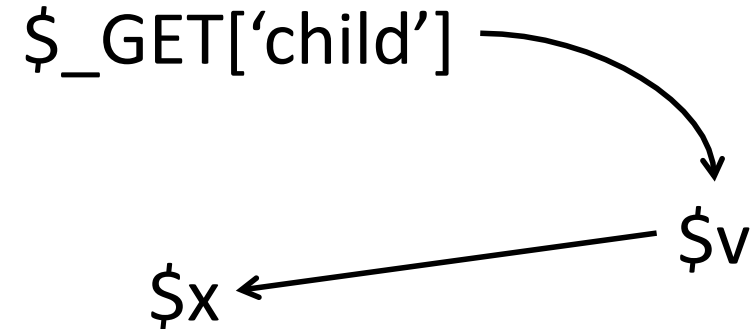


- If the “state” of a variable is always the same, we can bind this state, e.g., clean or tainted, to the variable itself.

Is the state of `$x` always the same in the program above?

Sparse Analysis

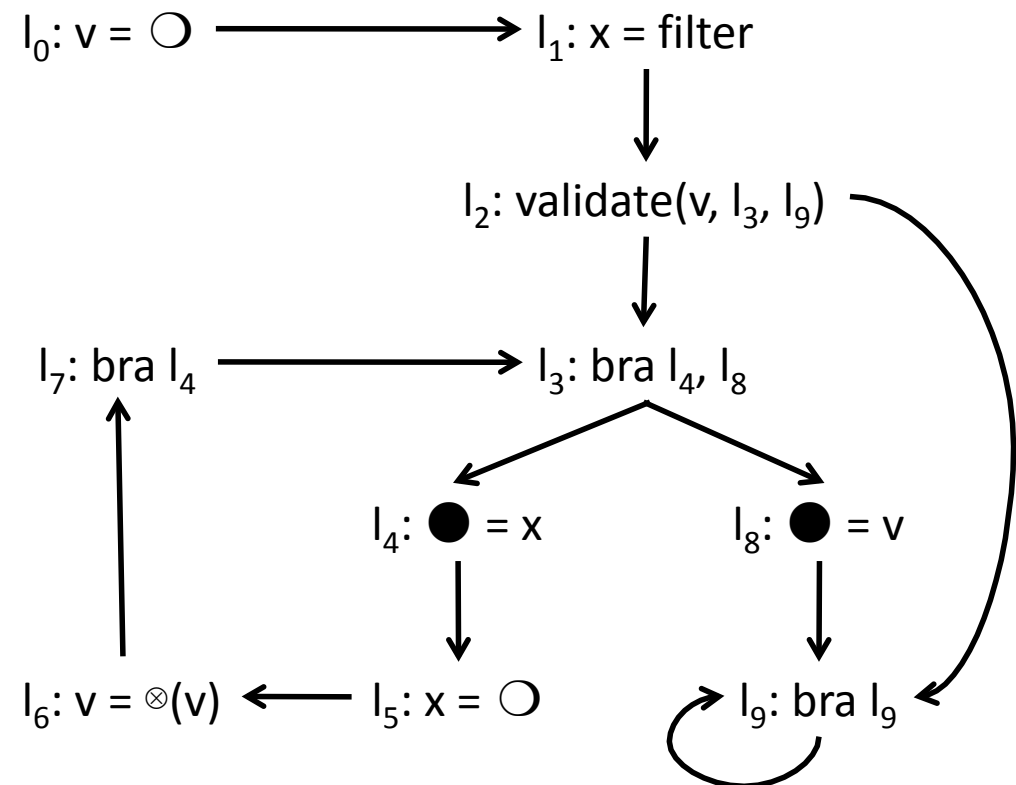
```
$v = DB.get($_GET['child']);  
$x = "";  
if (DB.isMember($v)) {  
    while (DB.hasParent($v)) {  
        echo($x);  
        $x = $_POST['$v'];  
        $v = DB.getParent($v);  
    }  
    echo($v);  
}
```



- 1) How can we ensure that the abstract state of a variable is always the same?
- 2) This property will give us a sparse analysis. Do you remember what is a sparse analysis?

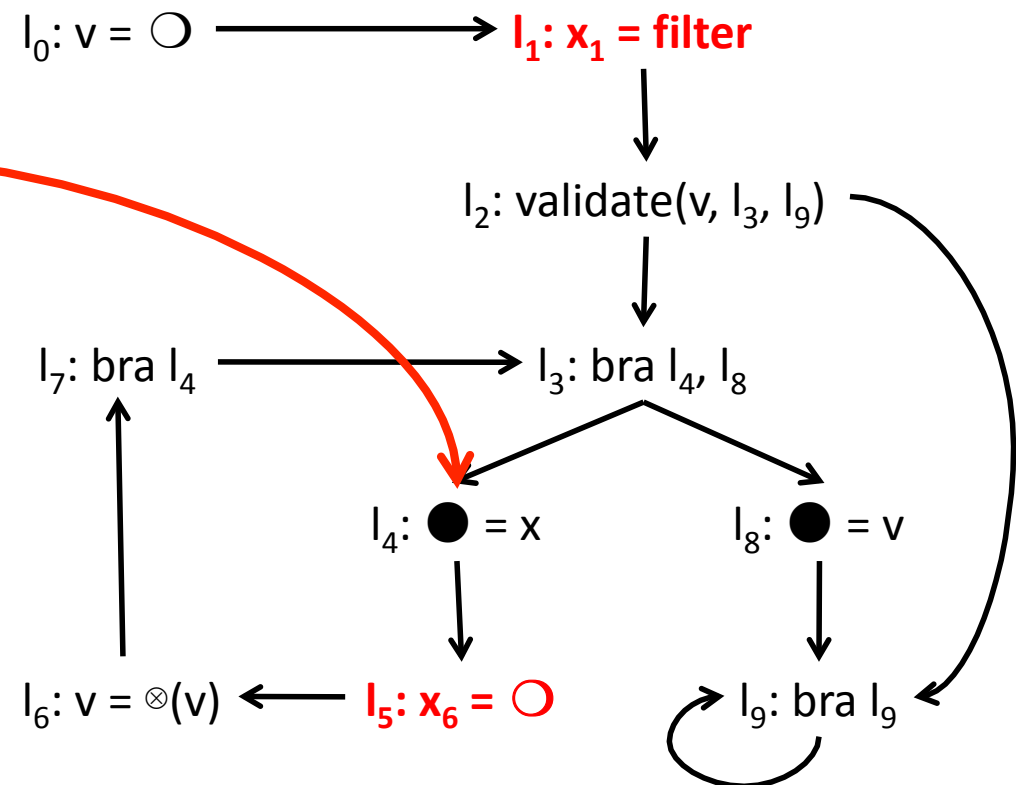
Static Single Assignment

- The key is to ensure that each variable is defined in at most one site inside the program text.
- The name x is defined twice. We need to rename it!



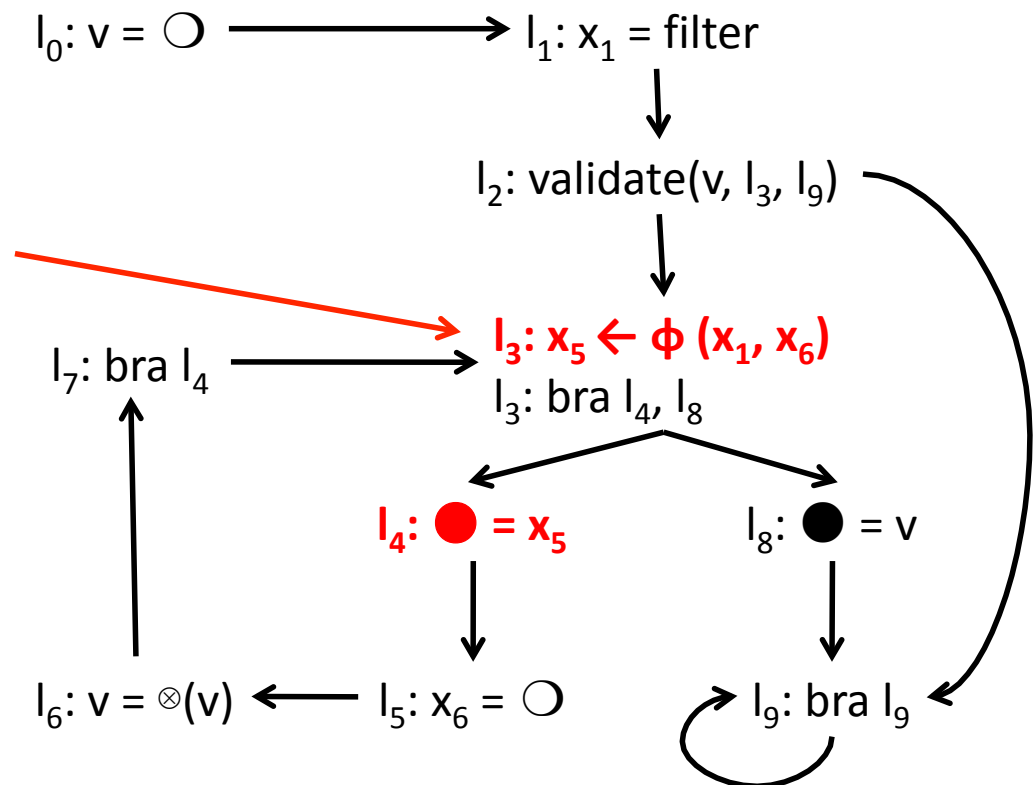
Static Single Assignment

- The key is to ensure that each variable is defined in at most one site inside the program text.
- The name x is defined twice. We need to rename it!
- But what is x at l_4 ?



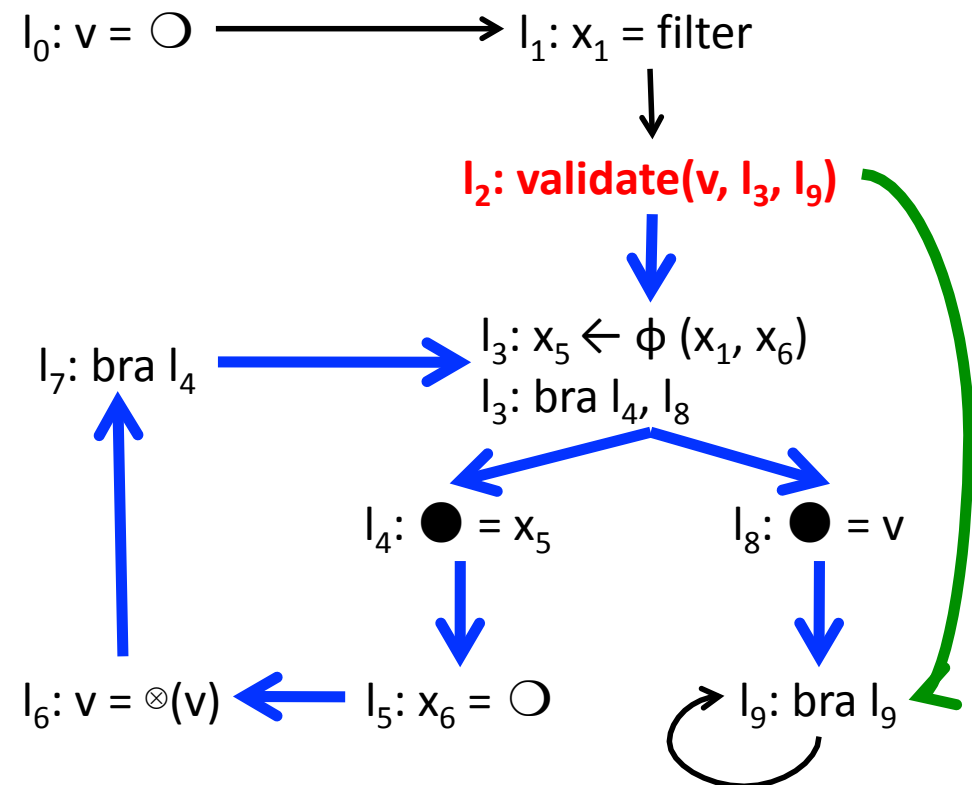
Static Single Assignment

- The key is to ensure that each variable is defined in at most one site inside the program text.
- The name x is defined twice. We need to rename it!
- But what is x at l_4 ?
- We use **phi-functions** to merge different variables into a single name.



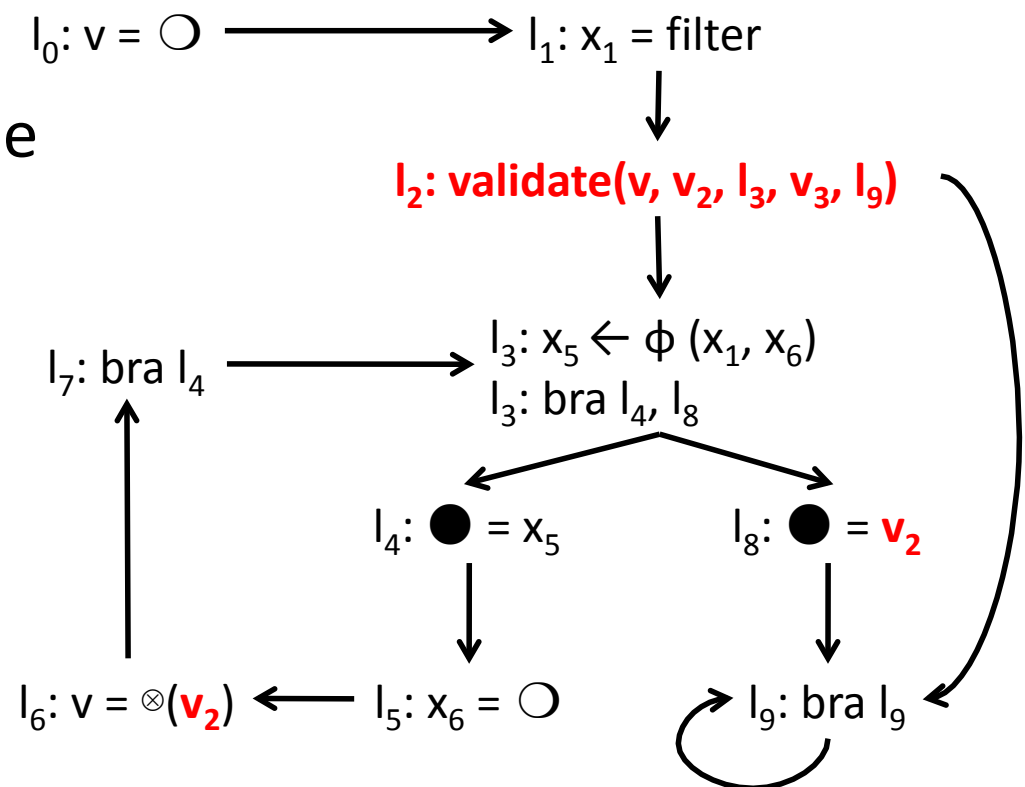
Static Single Assignment

- The variable v defined at l_0 is clean at some parts of the program, and tainted at others. Can you identify them?
- We can “learn” information about v from the *validator*.
- But we still would like to associate v with a single abstract state, clean or tainted. **How?**



Validators define new variable names

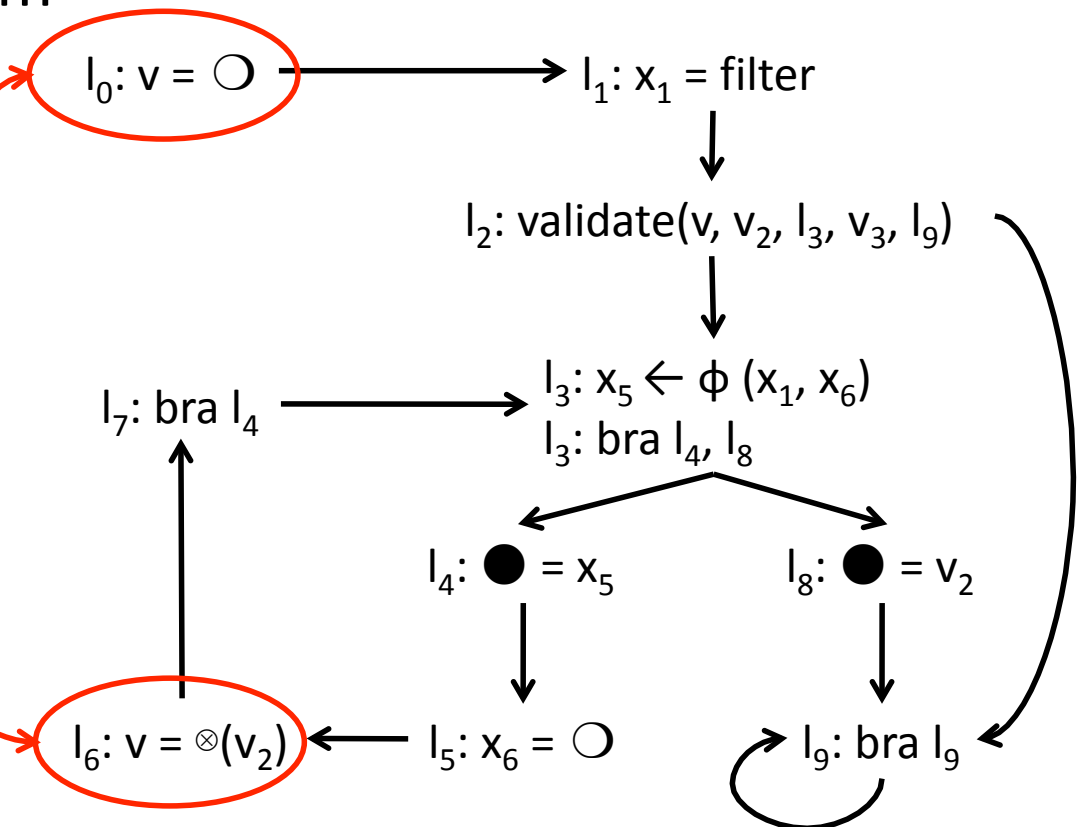
- The variable v defined at l_0 is clean in some parts of the program, and tainted at others. Can you identify them?
- We can “learn” information about v from the *validator*.
- Let’s rename each variable that is validated. In this way, each validator defines two new names.



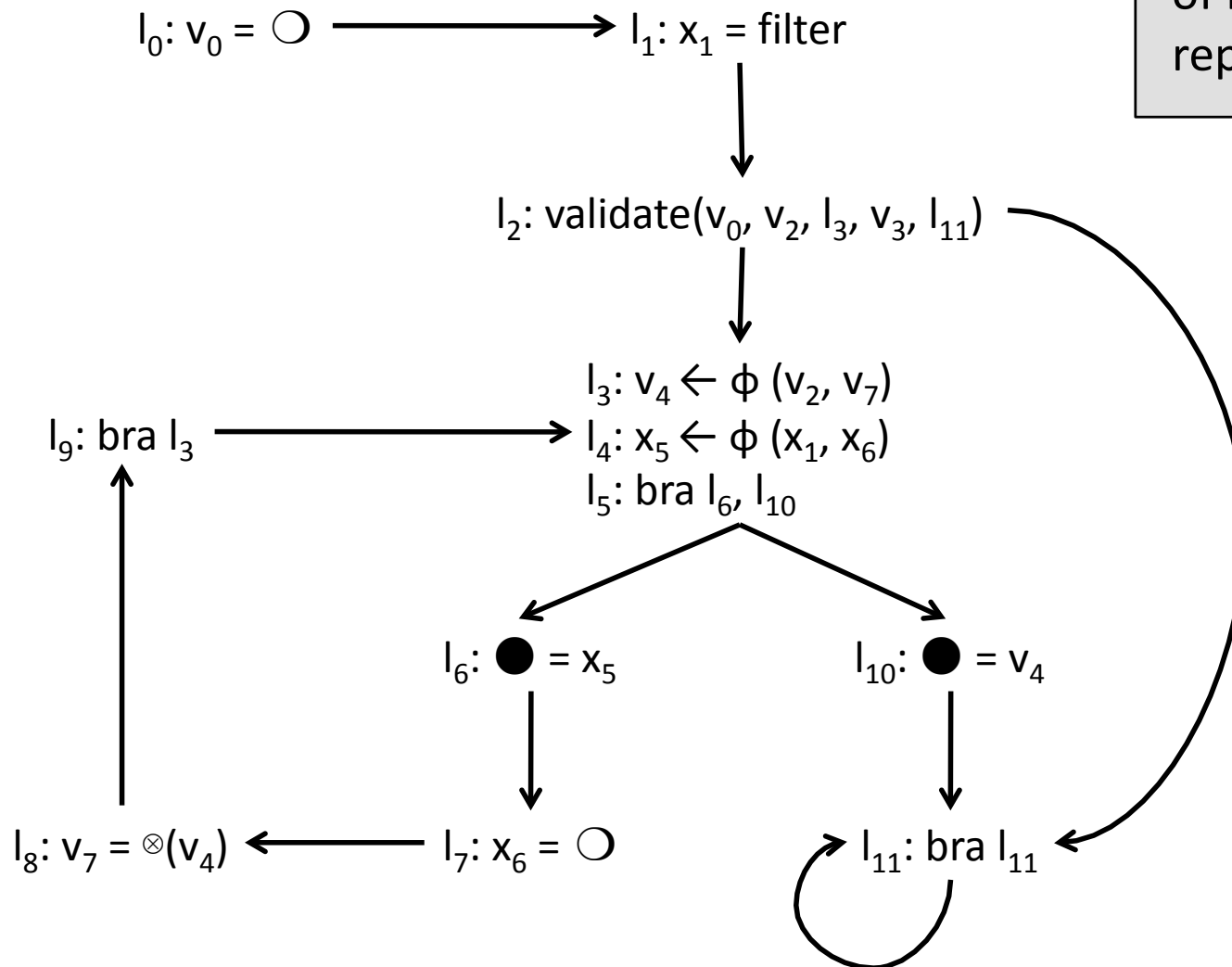
Validators define new variable names

- The variable v defined at l_0 is clean in some parts of the program, and tainted at others. Can you identify them?
- We can “learn” information about v from the *validator*.

And what should we do about the two definitions of variable v ? Remember: we can have only one definition per variable.



Example of Extended-SSA form program

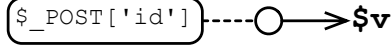
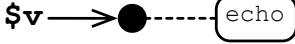
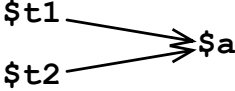
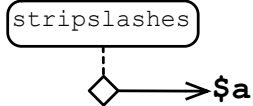
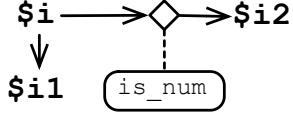
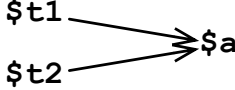


What is the complexity of building this program representation?

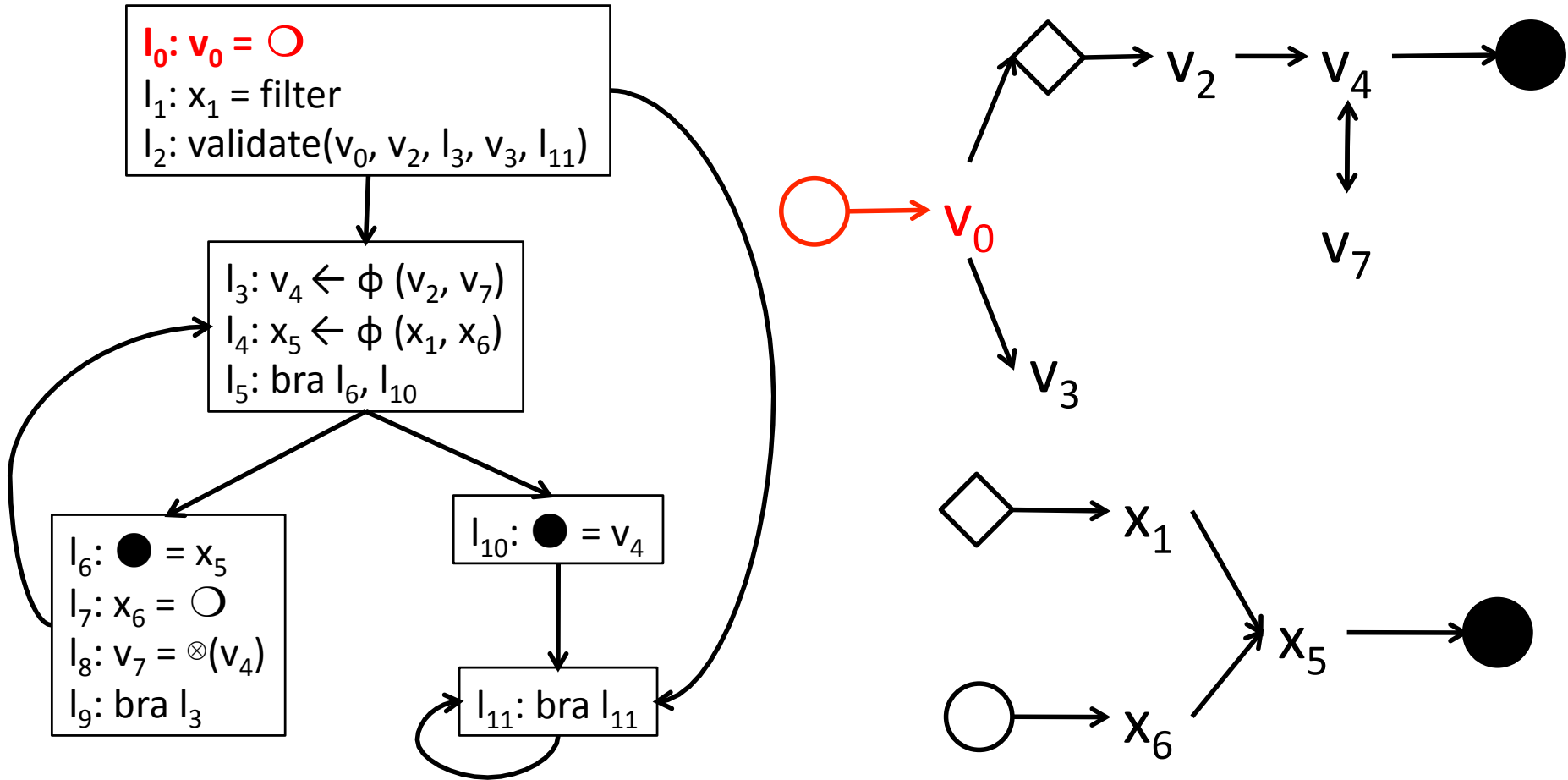
Sparse Analysis

- Algorithm in three steps:
 - Convert the program to e-SSA form $\rightarrow O(V^2)$
 - In practice, it is $\rightarrow O(V)$
 - Build the constraint graph $\rightarrow O(V^2)$
 - In practice, it is $\rightarrow O(V)$
 - Traverse the constraint graph $\rightarrow O(E) \rightarrow O(V^2)$
 - In practice, it is $\rightarrow O(V)$
- We are assuming that V is the number of variables in the program.

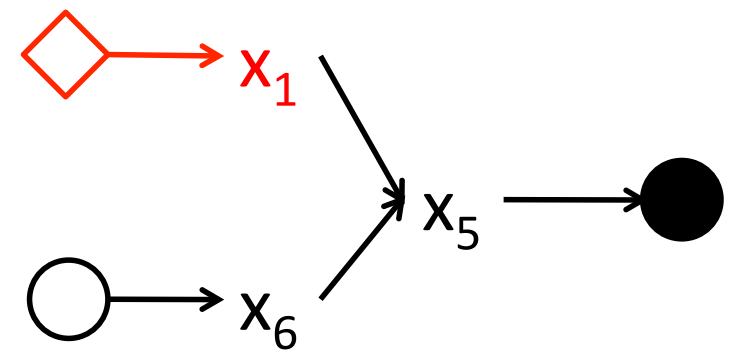
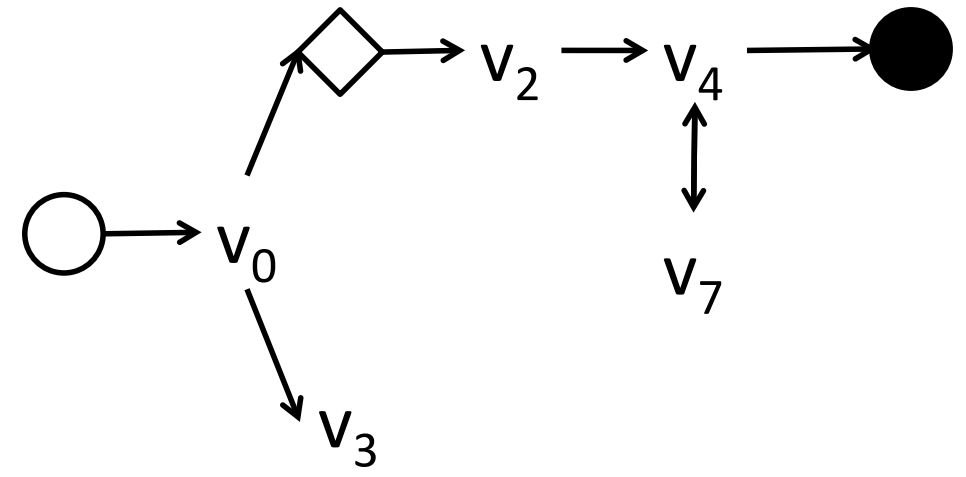
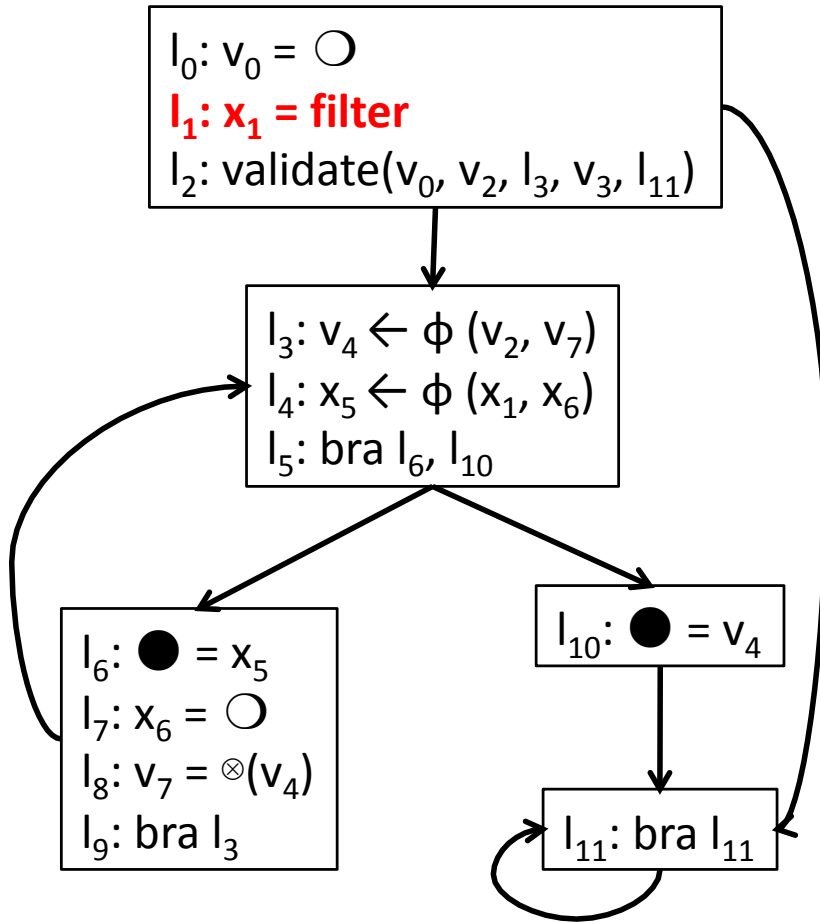
Building the Constraint Graph

Instruction	Example	Node
$v = \bigcirc$	<code>\$x = \$_POST['content']</code>	
$\bullet = v$	<code>echo(\$v)</code>	
$x = \otimes(x_1, \dots, x_n)$	<code>\$a = \$t1 * \$t2</code>	
$x = \text{filter}$	<code>\$a = htmlentities(\$t1)</code>	
<code>validate(v, v_c, l_c, v_t, l_t)</code>	<code>if(is_num(\$t1)) {...}</code>	
$v \leftarrow \phi(v_1, v_2)$	<code>\$v = phi(\$v1, \$v2)</code>	

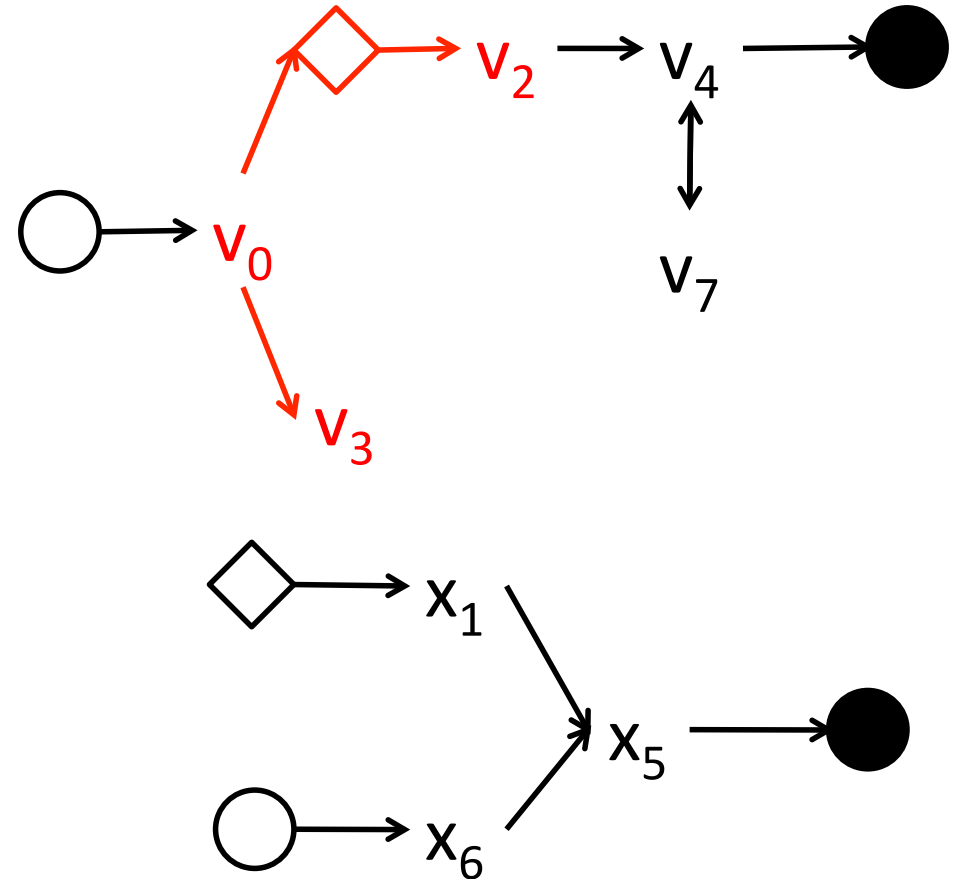
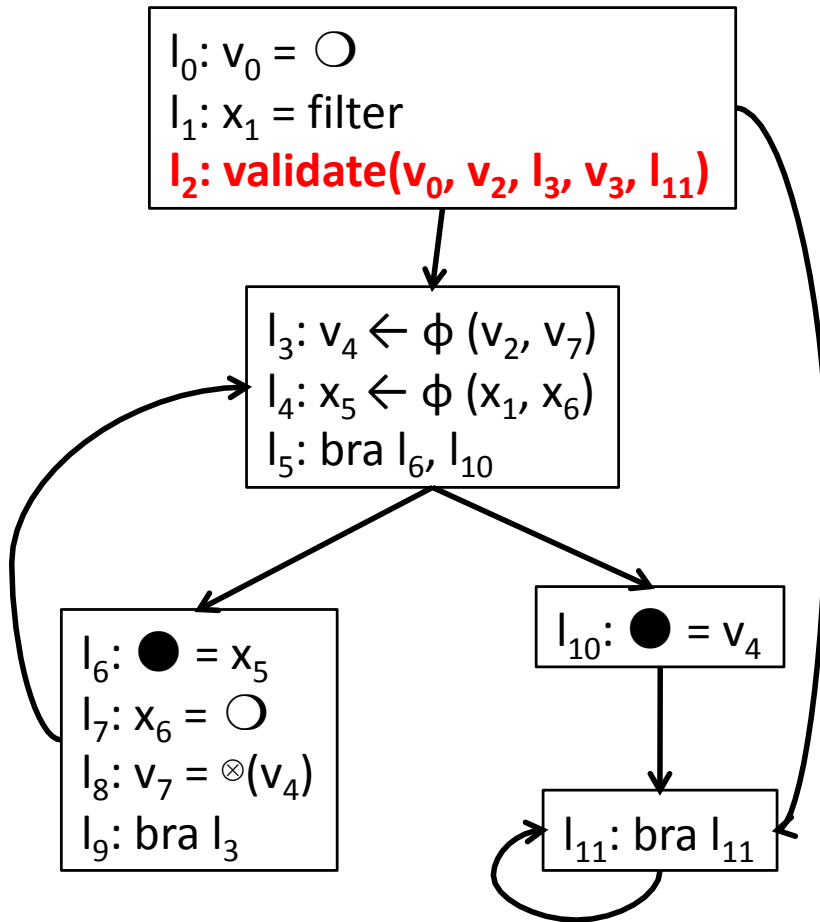
Example of constraint graph



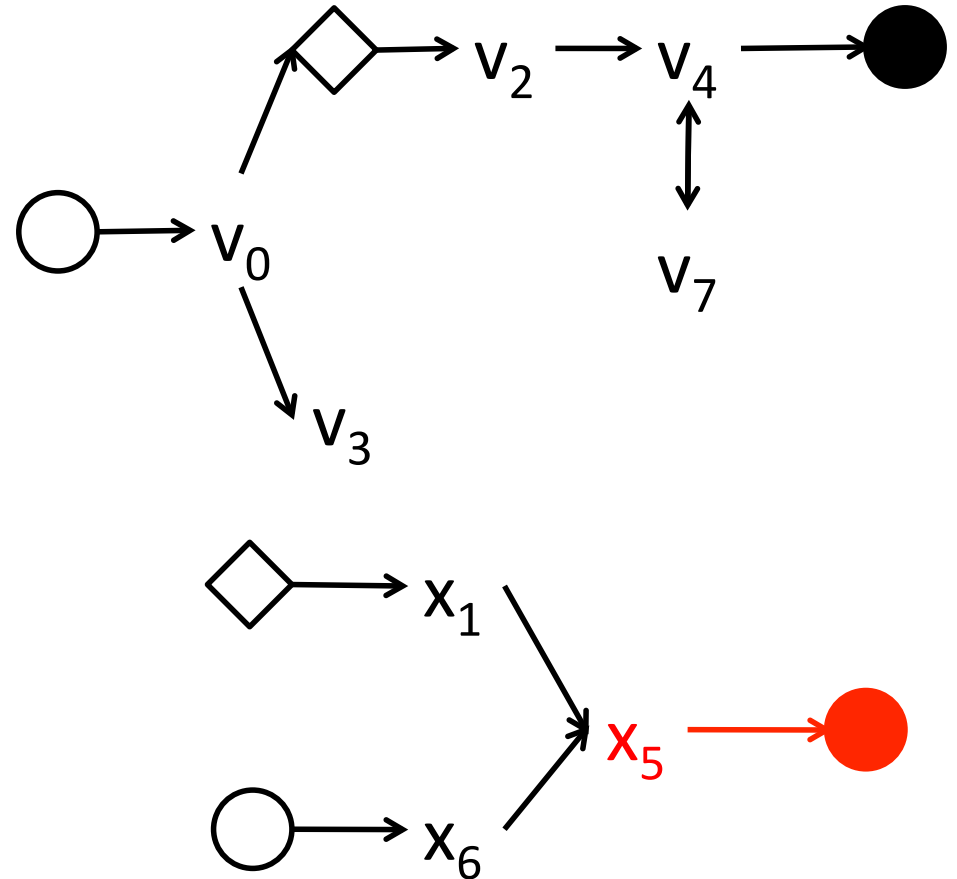
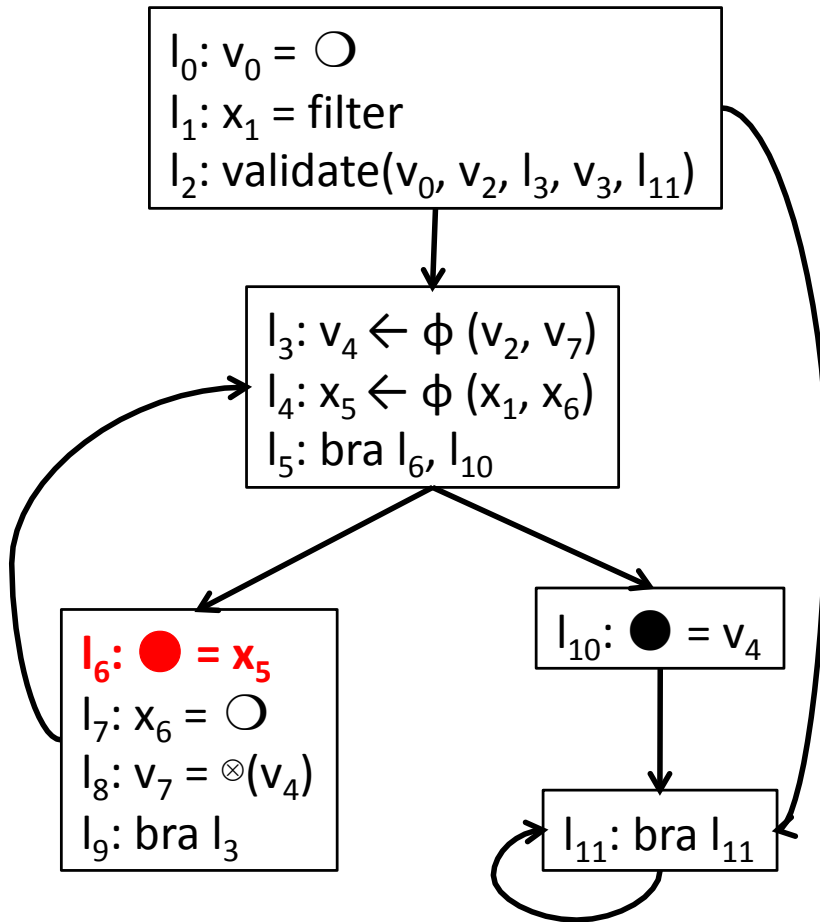
Example of constraint graph



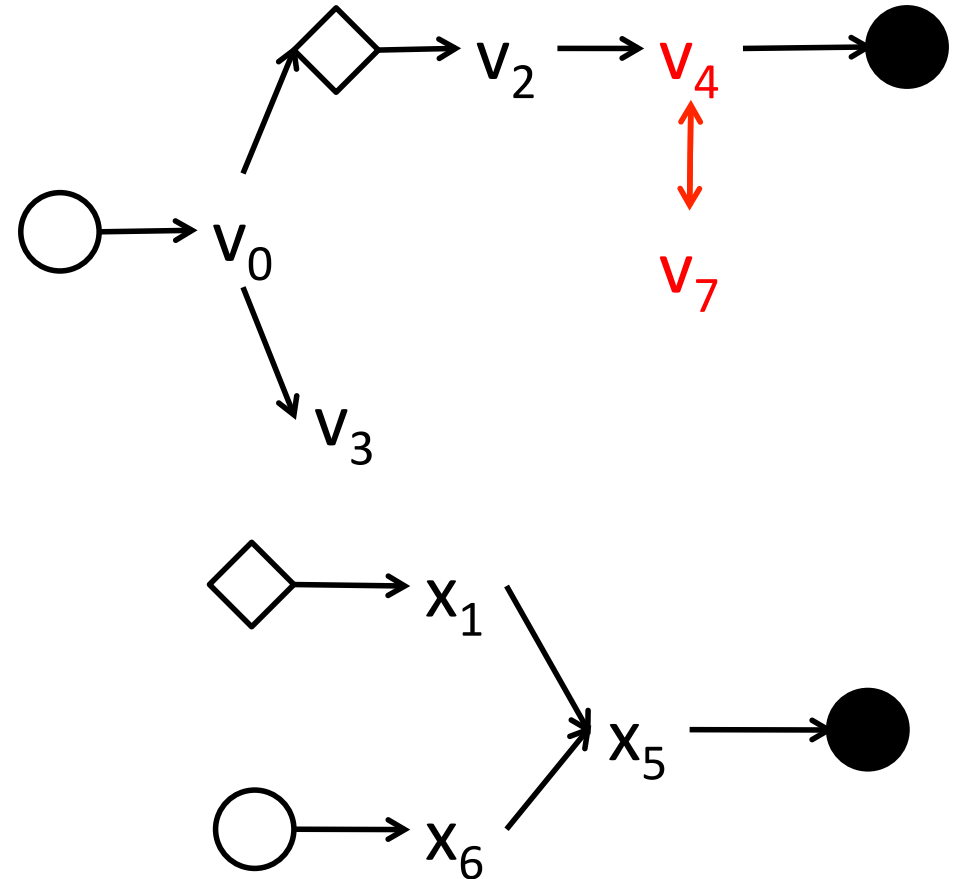
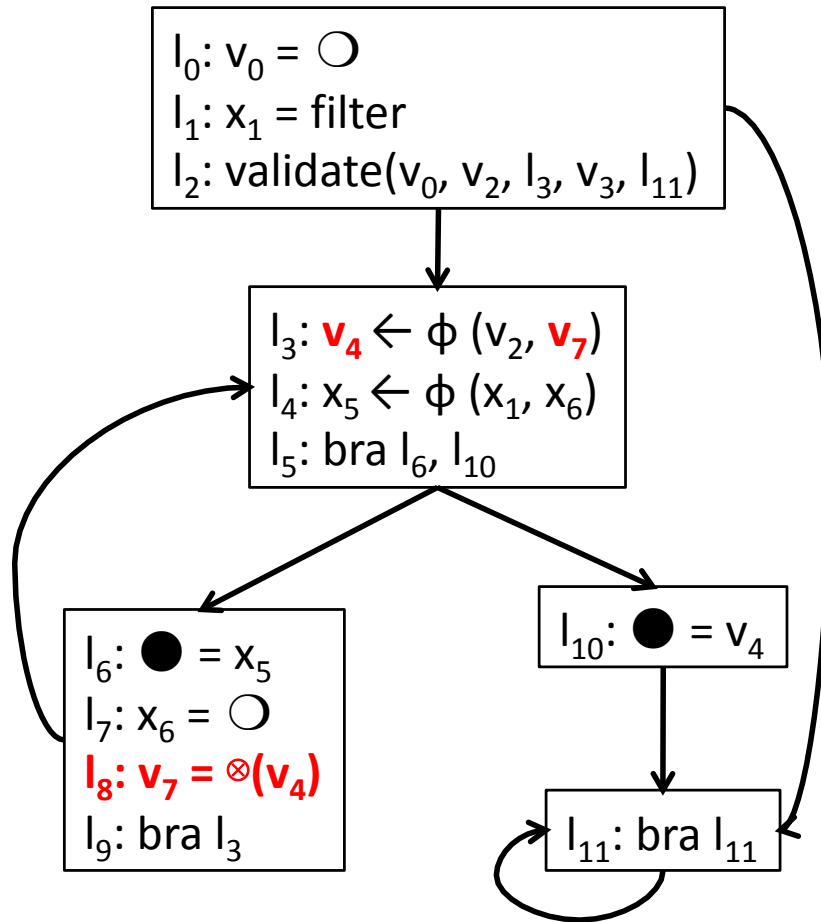
Example of constraint graph



Example of constraint graph

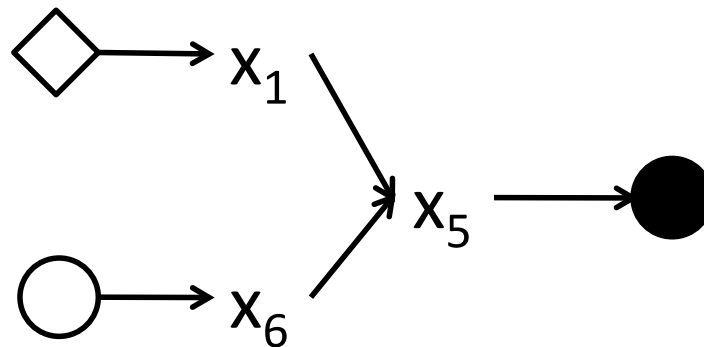
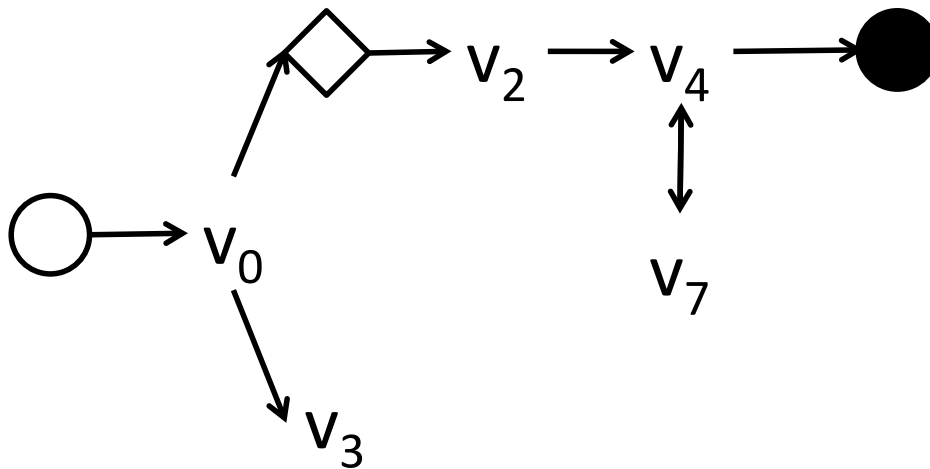


Example of constraint graph



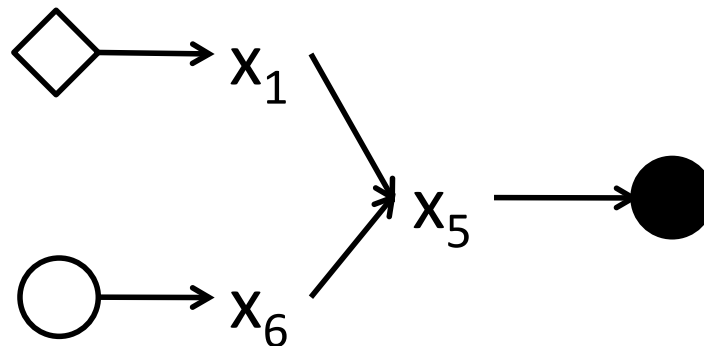
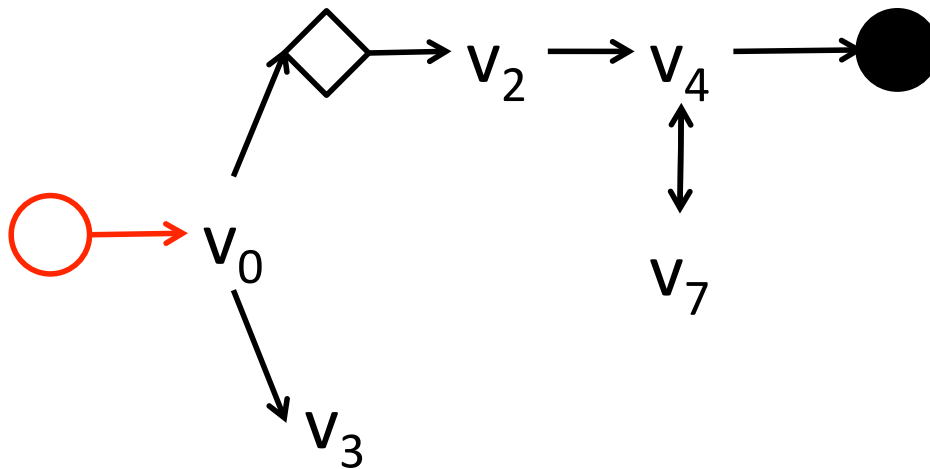
Traverse the Constraint Graph

Find a path from  to  without passing through 



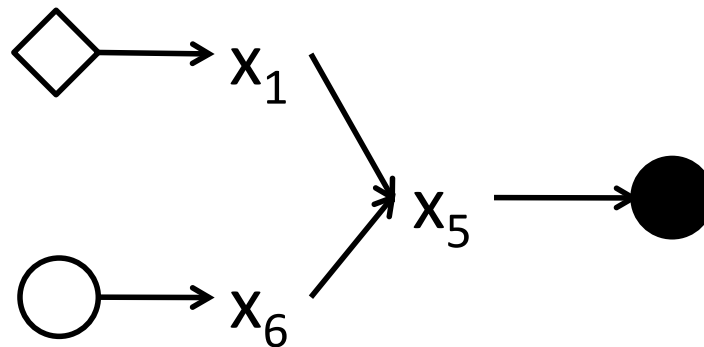
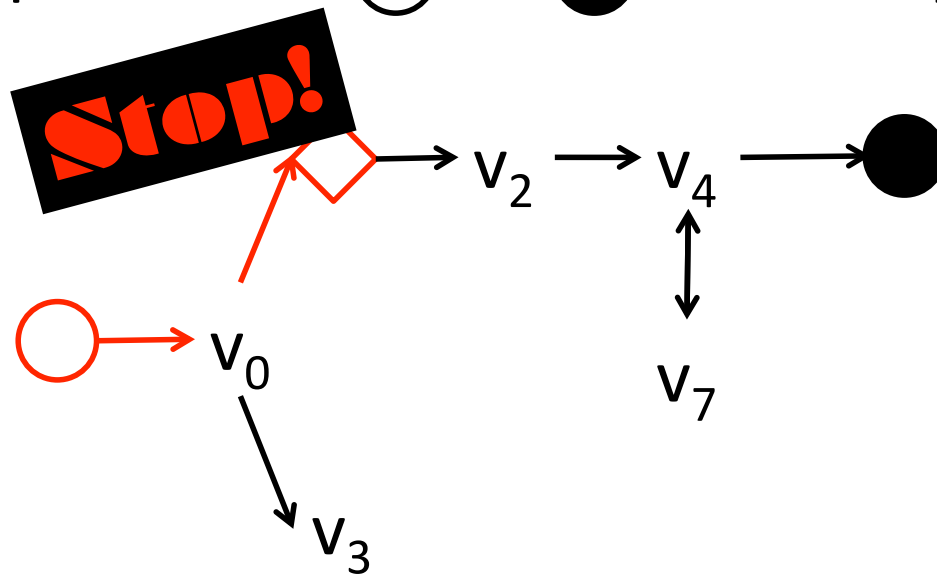
Traverse the Constraint Graph

Find a path from  to  without passing through 



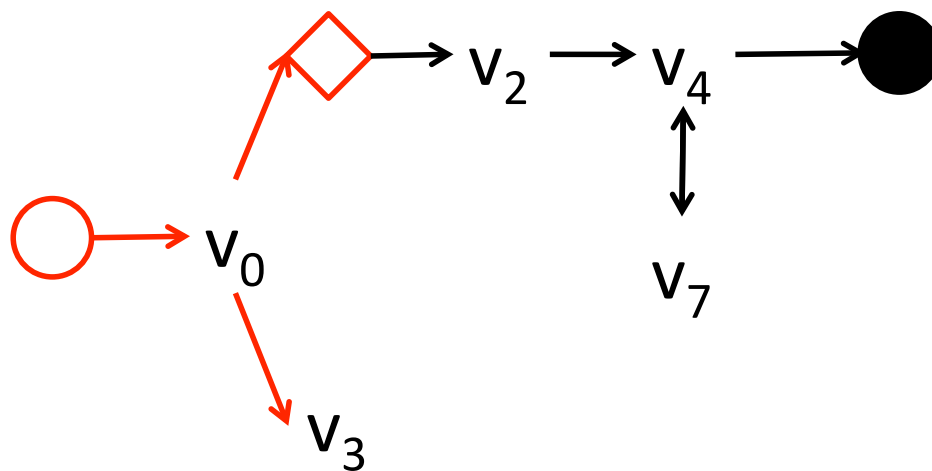
Traverse the Constraint Graph

Find a path from \bigcirc to \bullet without passing through \diamond



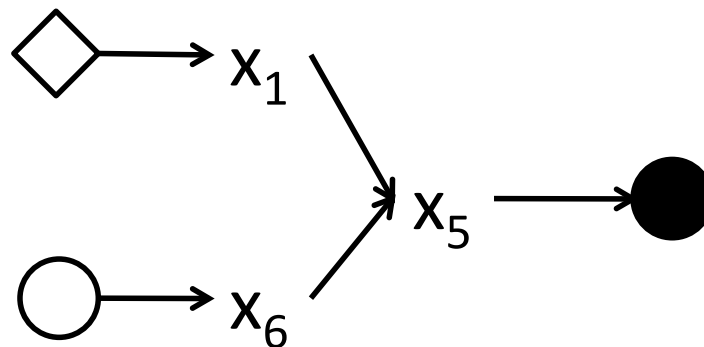
Traverse the Constraint Graph

Find a path from \bigcirc to \bullet without passing through \diamond



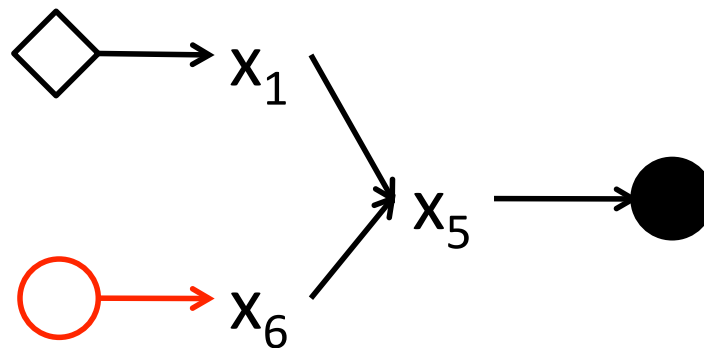
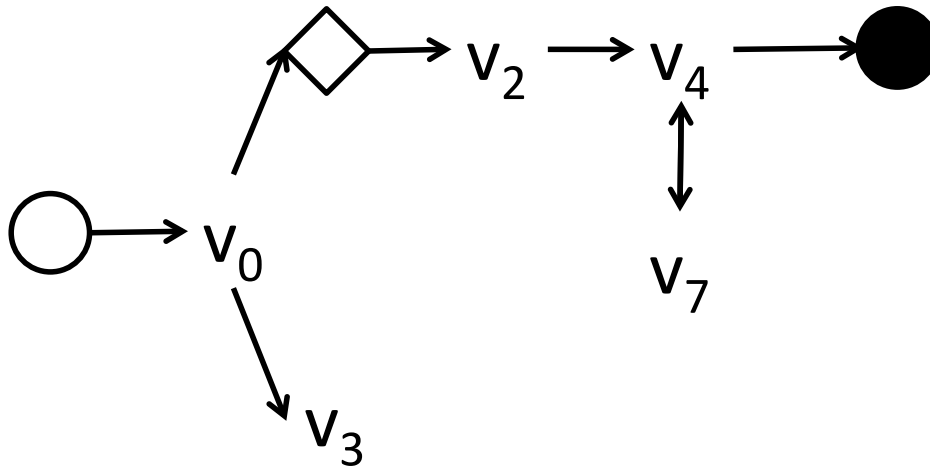
Report:

this program
slice is bug
free



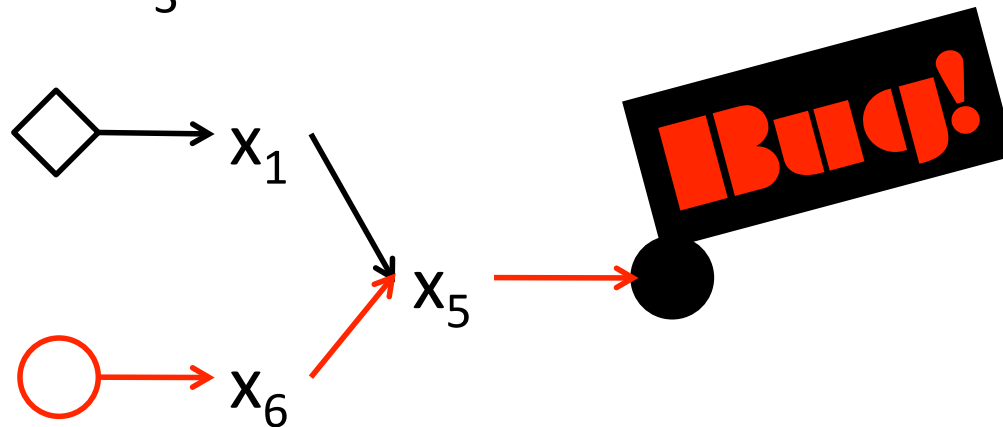
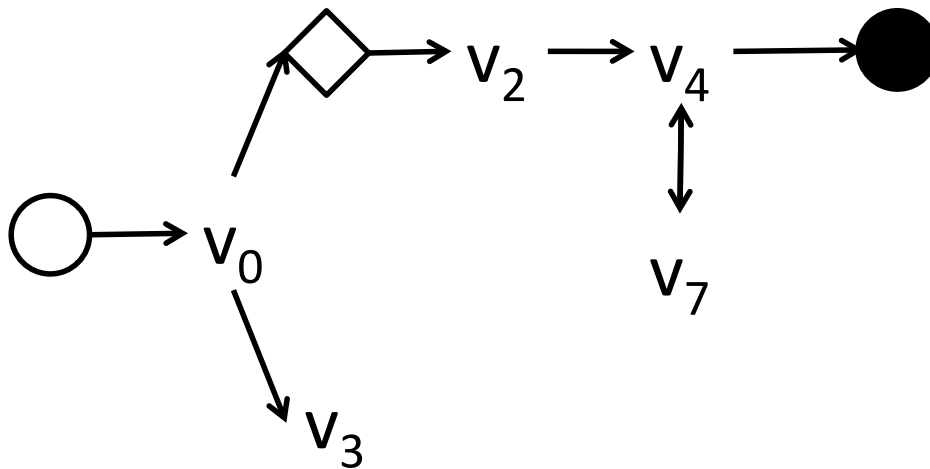
Traverse the Constraint Graph

Find a path from  to  without passing through 



Traverse the Constraint Graph

Find a path from  to  without passing through 

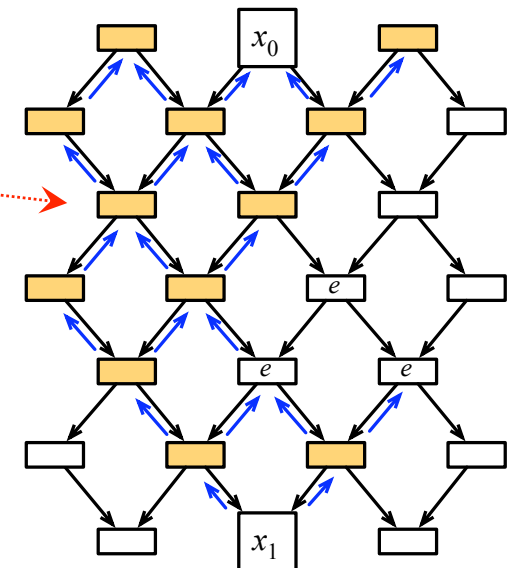
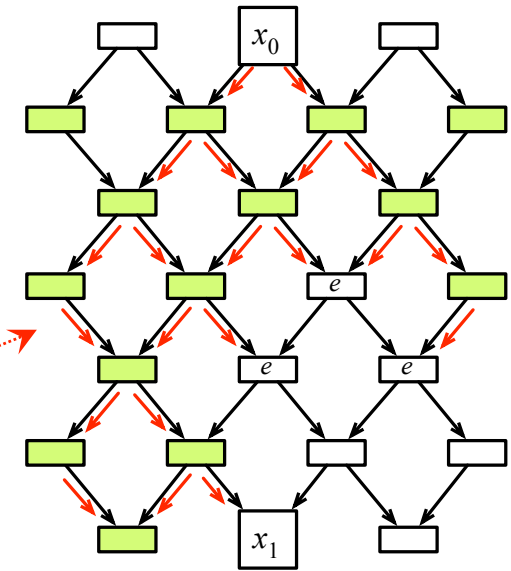


Data Flow analysis as Graph Reachability

- This data flow analysis reduces to a simple graph reachability problem because the lattice that is associated with each variable has height two: either a variable is clean, or it is tainted.
- In our formalism, any direct dependence from a variable v to a variable u transmits the abstract state from u to v .
- Therefore, we just want to know if there is a path from a source of malicious information, the original tainted data, to sensitive operations.
 - This path must not cross sanitizers, because sanitizers propagate clean information.

Program Slices

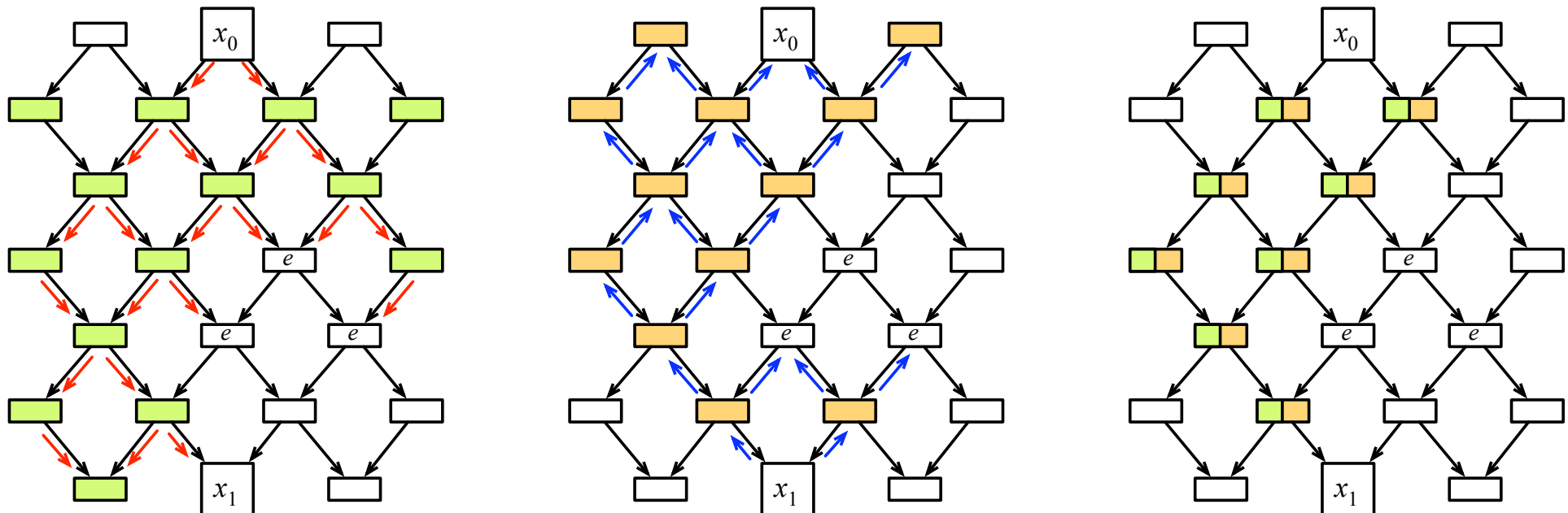
- Any subset of the dependence graph is called a *program slice*.
- The part of the dependence graph that depends on a variable x_0 is called the **forward slice** of x_0 .
- The part of the dependence graph on which a variable x_1 depends on is called the **backward slice** of x_1 .



Imagine that you must report a "buggy" part of the program back to the user. How can you use this notion of slices to determine which part of the program is buggy?

Program Slices

- Slices are useful to decrease the amount of information that we must report to the user, in case the program is vulnerable:
 - The dangerous part of the program is given by the intersection of the backward slice of a sink with the forward slice of a source, as long as there is a path from this source to this sink.



The Importance of Pointer Analysis

- In order to truly represent the dependencies in the program, the dependence graph must take pointers into consideration.

```
int main(int argc, char** argv){
    int a = 0;
    a++;
    int* p = (int*)malloc(sizeof(int));
    int* p2 = (int*)malloc(sizeof(int));
    *p = a;
    if (argc%2) {
        free(p2);
        p2 = p;
    } else {
        *p2 = *p;
    }
    return 0;
}
```



```
%0:
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i8**, align 4
%a = alloca i32, align 4
%p = alloca i32*, align 4
%p2 = alloca i32*, align 4
store i32 0, i32* %1
store i32 %argc, i32* %2, align 4
store i8** %argv, i8*** %3, align 4
store i32 0, i32* %a, align 4
%4 = load i32* %a, align 4
%5 = add nsw i32 %4, 1
store i32 %5, i32* %a, align 4
%6 = call i8* @malloc(i32 4)
%7 = bitcast i8* %6 to i32*
store i32* %7, i32** %p, align 4
%8 = call i8* @malloc(i32 4)
%9 = bitcast i8* %8 to i32*
store i32* %9, i32** %p2, align 4
%10 = load i32* %a, align 4
%11 = load i32** %p, align 4
store i32 %10, i32* %11
%12 = load i32* %2, align 4
%13 = srem i32 %12, 2
%14 = icmp ne i32 %13, 0
br i1 %14, label %15, label %19
```

T

F

```
%15:
```

```
%16 = load i32** %p2, align 4
%17 = bitcast i32* %16 to i8*
call void @free(i8* %17)
%18 = load i32** %p, align 4
store i32* %18, i32** %p2, align 4
br label %23
```

```
%19:
```

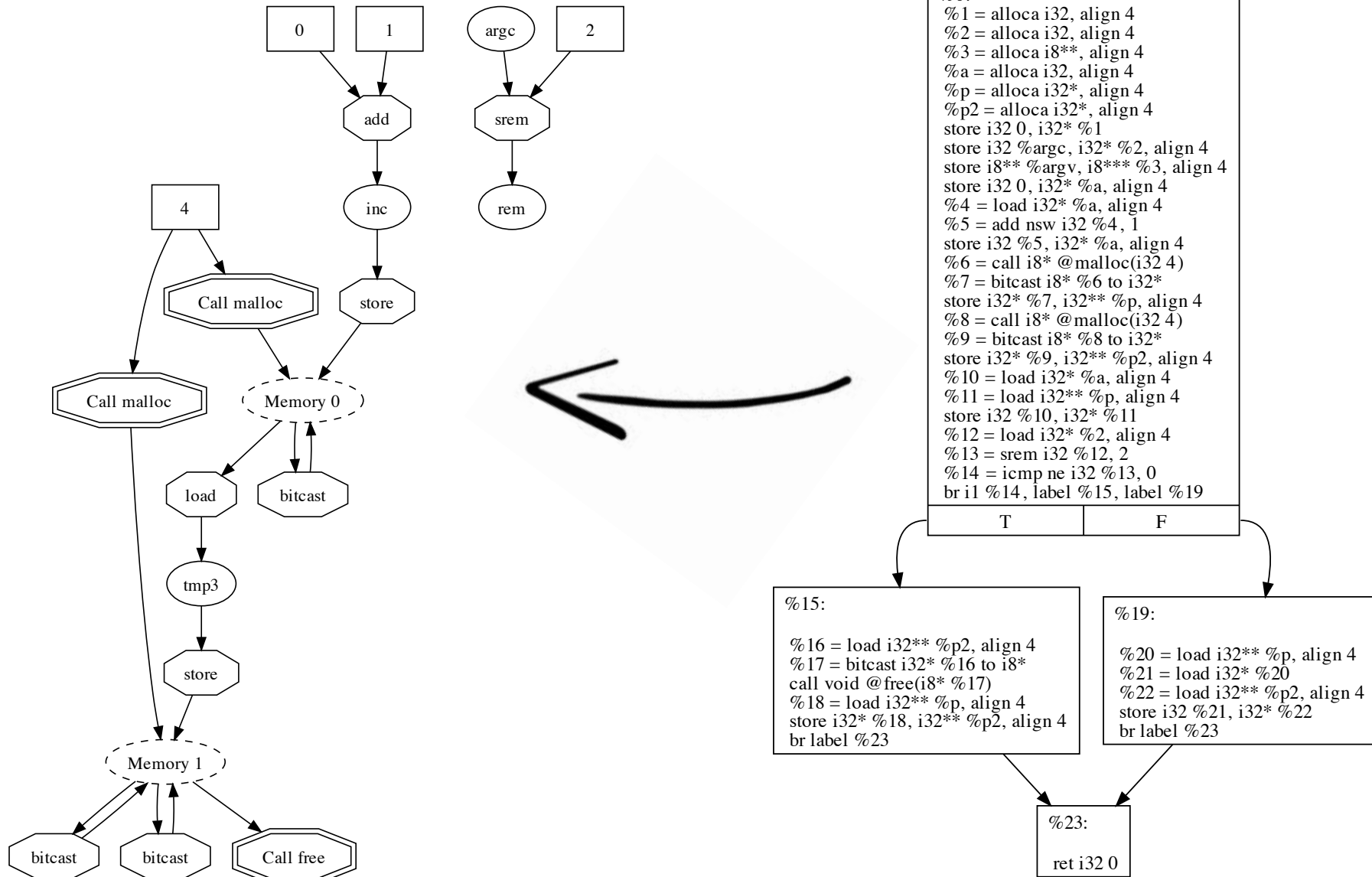
```
%20 = load i32** %p, align 4
%21 = load i32* %20
%22 = load i32** %p2, align 4
store i32 %21, i32* %22
br label %23
```

```
%23:
```

```
ret i32 0
```

This example is taken from an actual LLVM bytecode.

The Importance of Pointer Analysis



This example is taken from an actual LLVM bytecode.

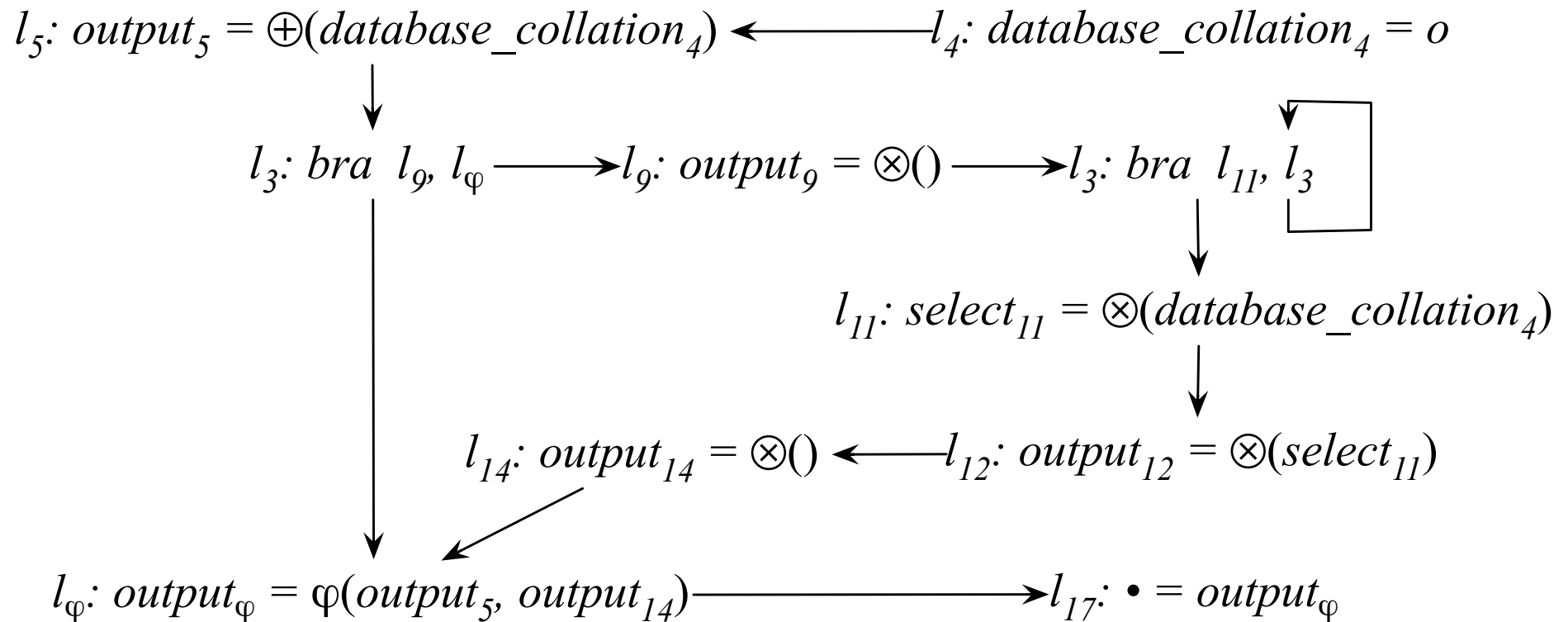
Real World Example

```
<?php
$host = $_POST['host'];    $uid = $_POST['uid'];    $pwd = $_POST['pwd'];
$database_collation = $_POST['database_collation'];
$output = '<select id="database_collation" name="database_collation"> <option
value="".'$database_collation.'" selected >'.'$database_collation.'</option></
select>';
if ($conn = @ mysql_connect($host, $uid, $pwd)) {
    $getCol = mysql_query("SHOW COLLATION");
    if (@mysql_num_rows($getCol) > 0) {
        $output = '<select id="database_collationse_collation"
                    name="database_collation">';
        while ($row = mysql_fetch_row($getCol)) {
            $selected = ( $row[0]==$database_collation ? ' selected' : " ");
            $output .= '<option value="".'$row[0]."'.'$selected.'>'.'$row[0].'</option>';
        }
        $output .= '</select>';
    }
}
echo $output;
?>
```

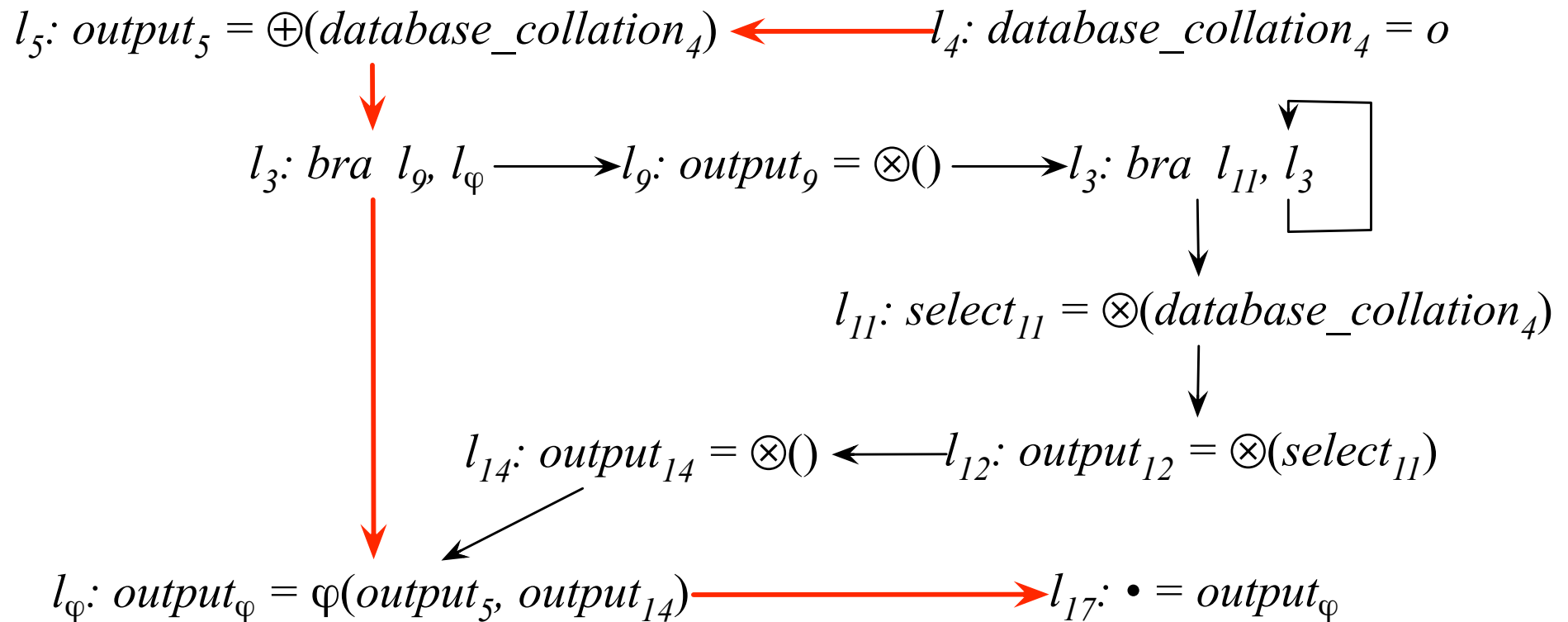
Can you identify the vulnerability of this program?

Real World Example

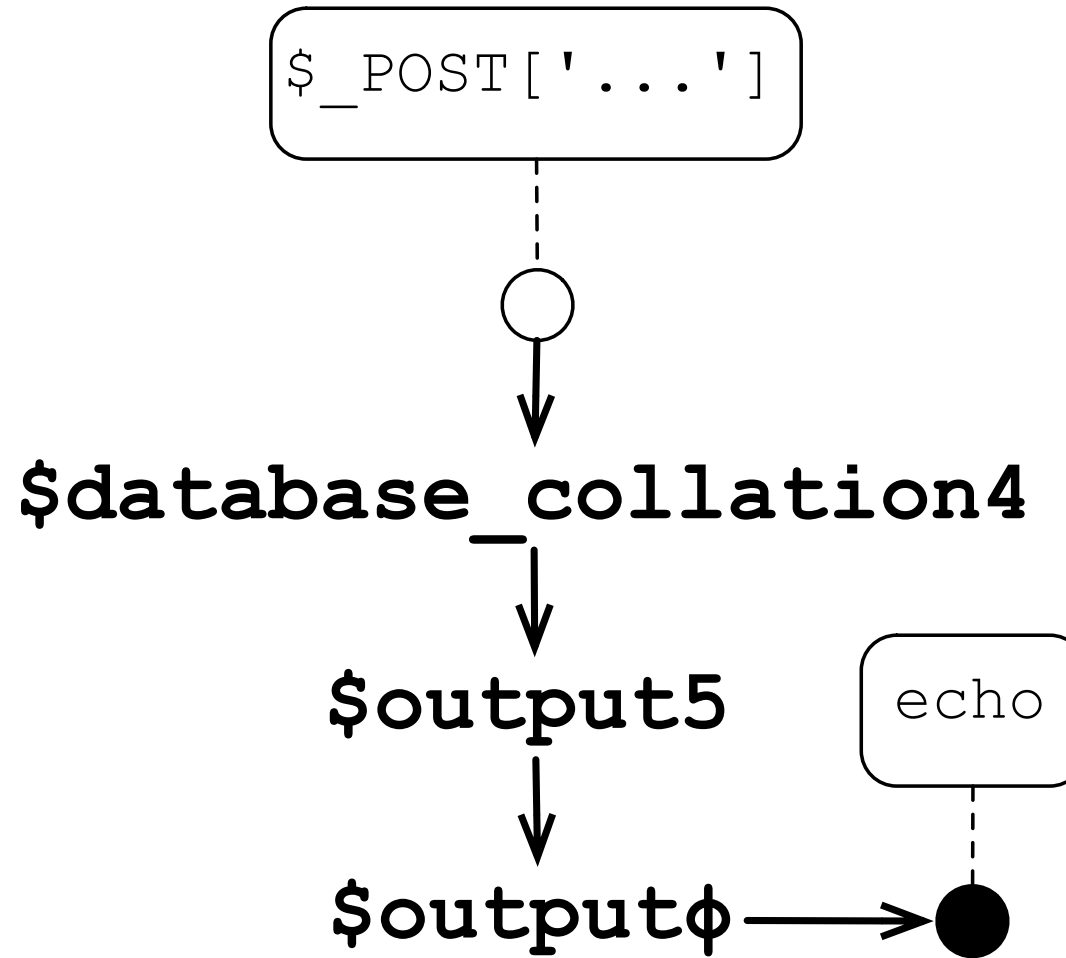
Where is
the bug?



Real World Example



Real World Example



A Bit of History

- The foundations of information flow were introduced by Denning and Denning in 1977
- The notion of program slice was introduced by Weiser.
- The dense algorithm that we saw in these slides is an adaptation of Orbaek and Palsberg's analysis.
- These slides follow the exposition given by Rimsa *et al.*

- Denning, D., and Denning, P., "Certification of programs for secure information flow", *commun. ACM* 20 p 504-513 (1977)
- Weiser, M., "Program Slicing", *ICSE*, p 439-449 (1981)
- Orbaek, P., and Palsberg, J., "Trust in the lambda-calculus", *Journal of Func. Programming* 7(6), p 557-591 (1997)
- Rimsa, A., Amorim, M., and Pereira, F., "Tainted Flow Analysis on e-SSA-form Programs", *CC*, p 124-143 (2011)