COMP5111 – Fundamentals of Software Testing and Analysis

Program Slicing and Taint Analysis

Shing-Chi Cheung

Computer Science & Engineering

HKUST

What is a program slice?

```
void main ( ) {
  int i = 0;
  int sum = 0;
  while (i < N) {
     sum = add(sum, i);
     i = add(i, 1);
  }
  printf ("sum=%d\n", sum);
  printf("i=%d\n", i); // find the slice of variable i
}
```

Adapted from the notes by Xiangyu Zhang (Purdue)

What is a Program Slice?

```
void main ( ) {
   int i = 0;
   int sum = 0;
   while (i < N) {
       sum = add(sum, i);
       i = add(i, 1);
    }
    printf ("sum=%d\n", sum);
    printf("i=%d\n", i); // find the slice of variable i
}
```

A program slice of variable *i* at statement *S* is the set of statements involved in computing *i*'s value at *S*.

[Mark Weiser, 1982]

Adapted from the notes by Xiangyu Zhang (Purdue)

Why Slicing?

- Debugging
- Testing
- Differencing
- Regression testing
- Program understanding
- Complexity measurement
- Program integration
- Reverse engineering

```
void main ( ) {
  int i = 0;
  int sum = 0;
  while (i < N) {
      sum = add(sum, i);
      i = add(i, 1);
  }
  printf (“sum=%d\n”, sum);
  printf(“i=%d\n”, i);
}
```

Some analysis tasks may require an alternative or even finer behavioral view other than execution traces

Emerging Applications of Slicing

- Security
  - Malware detection
  - Information privacy
- Software Transactional Memory
- Architecture
  - Value speculation
- Program optimization
- …

An access control mechanism to shared memory in concurrent transactions

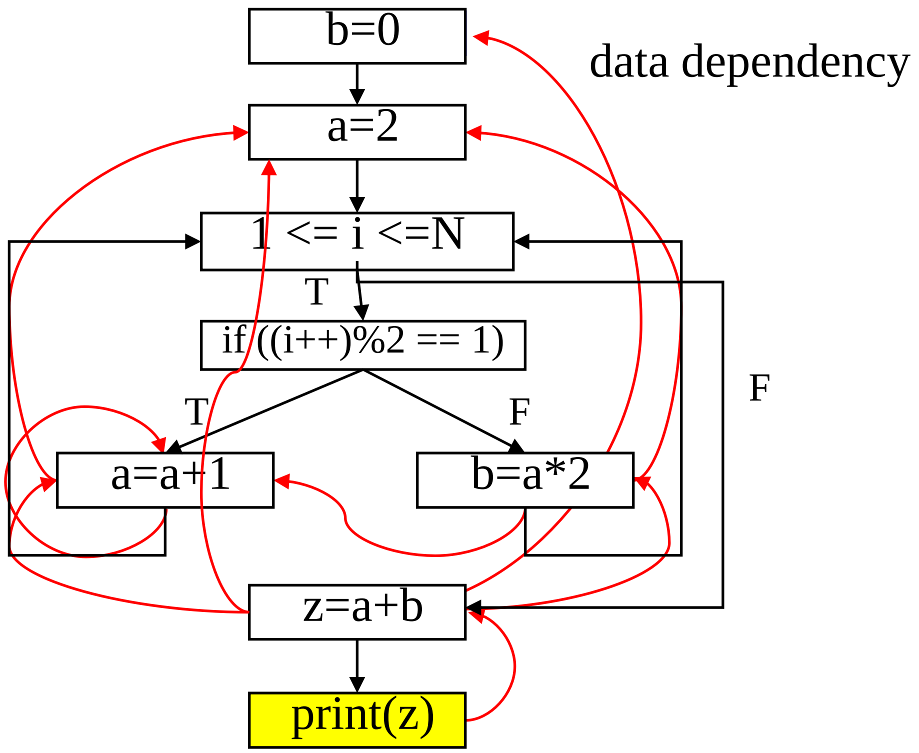
Outline

- Slicing Overview
- Dynamic slicing
  - Efficiency
  - Effectiveness
  - Challenges

Slicing Classification

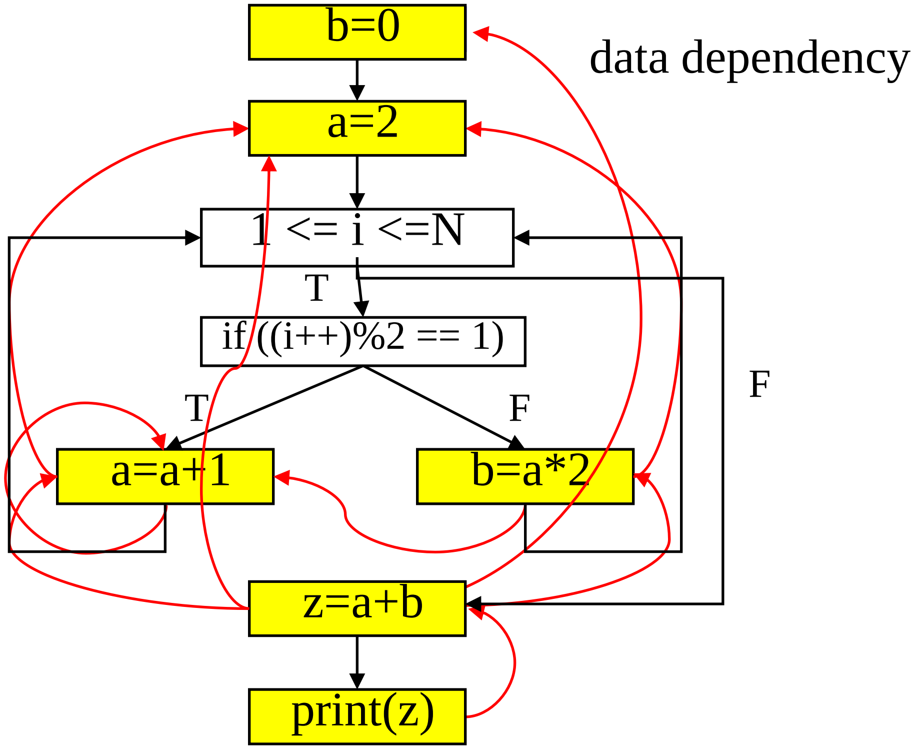
- Static vs. Dynamic
- Backward vs. Forward
- Executable vs. Non-Executable
- More …

How to Slice?



- Static analysis
  - Input insensitive
  - May analysis
- Dependence Graph
- Characteristics
  - Very fast
  - Very imprecise

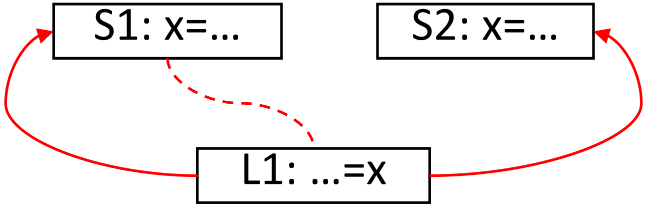
How to Slice?



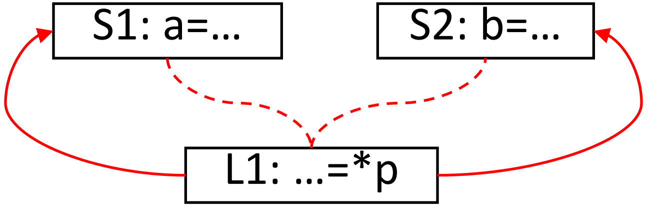
- Static analysis
  - Input insensitive
  - May analysis
- Dependence Graph
- Characteristics
  - Very fast
  - Very imprecise

Why is a Static Slice Imprecise?

- All possible program paths



- Use of pointers – static alias analysis is very imprecise



All possible variables referenced by p

Static Slicing is Imprecise

- Static slicing of s at line 15
- Give the whole program

```
1 n = read();
2 a = read();
3 x = 1;
4 b = a + x;
5 a = a + 1;
6 i = 1;
7 s = 0;
8 while (i <= n) {
9     if (b > 0)
10        if (a > 1)
11            x = 2;
12    s = s + x;
13    i = i + 1;
14 }
15 write(s);
```

Dynamic Slicing

- Korel and Laski, 1988
- Dynamic slicing makes use of all information about a particular execution of a program and computes the slice based on an execution history (trace)
  - Trace consists of control flow trace and memory reference trace
- A dynamic slice query is a triple
  - <Var, Input, Execution Point>
- Smaller, more precise, more useful

Dynamic Slicing

Dynamic slice <z, 2, 8>?

```
func(N) {
1: b=0
2: a=2
3: for i= 1 to N do
4:   if (i%2==1)then
5:         a = a+1
     else
6:         b = a*2
     endif
   done
7: z = a+b
8: print(z)
}
```

For input N=2,

1_1:	b=0	[b=0]
2_1:	a=2	
3_1:	for i = 1 to N do	[i=1]
4_1:	if (i%2 == 1) then	[i=1]
5_1:	a=a+1	[a=3]
3_2:	for i=1 to N do	[i=2]
4_2:	if (i%2 == 1) then	[i=2]
6_1:	b=a*2	[b=6]
7_1:	z=a+b	[z=9]
8_1:	print(z)	[z=9]

Issues about Dynamic Slicing

- Precision
  - Perfect
- Running history
  - Very big (GB)
- Algorithm to compute dynamic slice
  - Slow and large space requirement

Backward vs. Forward

An Example Program & its static forward slice w.r.t. <3, sum>

Useful to information privacy analysis

```
1 main( )
2 {
3     int i, sum;
4     sum = 0;
5     i = 1;
6     while (i <= 10)
7     {
8         sum = sum + 1;
9         ++i;
10    }
11    cout << sum;
12    cout << i;
13 }
```

Summary …

- Want to know more?
  - Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages 3* (1995), 121-189
- Static slicing is a useful tool for static analysis
  - Code transformation, program understanding, etc.
  - Points-to analysis is the key challenge
  - Not as useful in reliability as dynamic slicing

- We use dynamic slicing more often
  - Precise
  - Much longer trace → solution space is much larger → more expensive
  - There exist hybrid techniques.

Taint Analysis

adapted from the slides by David Brumley (CMU) and by Yepang Liu (HKUST)

What is Taint Analysis?

- Keep track of which variables are affected by predefined tainted values, usually inputs from users or internet
- A dataflow analysis: static or dynamic
- Mostly used for flow security analysis
- Similar to forward slicing, but focusing on the **impact of inputs** on **variable values** instead of program statements.

Forward Slicing vs Taint Analysis

```
void foo(int x) {
  int y = 3;
  int p = x + y;
  int z = y * 3;
  if (p == 0)
    z = y – 1;
}
```

- Taint analysis:
  - variables p and z are tainted after execution.
  - if x's value is non-trustable, p's and z's value are also non-trustable.
- Forward slicing:

```
int p = x + y;
if (p == 0)
z = y – 1;
```

A Lattice Model for Secure Information Flow

- A ***secure information flow model*** is defined by a lattice $< SC, \preccurlyeq >$
  - SC : a set of **security classes** of logical information storage objects (e.g., files, program variables)
  - $\preccurlyeq$: a **partial order** on SC that specifies legitimate information flows among the security classes

Prof. Dorothy E. Denning
ACM Fellow (1995)

Dorothy E. Denning. 1976. A lattice model of secure information flow. *Communications of the ACM*, vol. 19, no. 5 (May 1976), pp. 236-243.

A Lattice Model for Secure Information Flow

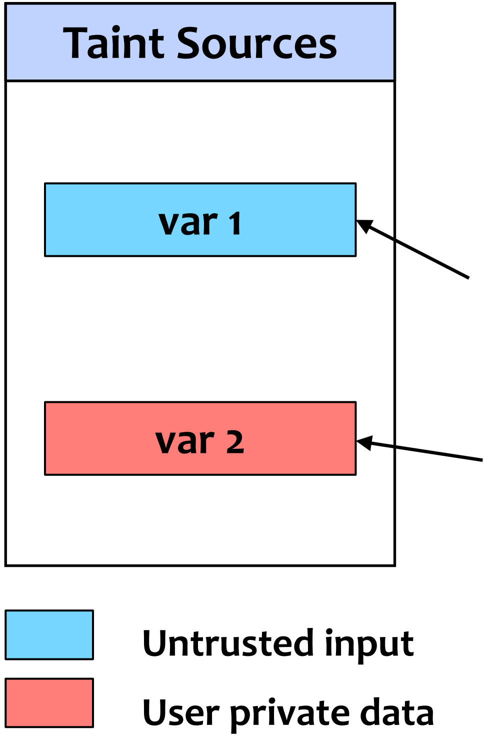
- An example with only two security classes:
  - $SC: \{\, public, secret \,\}$
  - Partial order: $public \preccurlyeq secret \ldots$

- $obj1\ (secret) \longmapsto obj2\ (public)$

insecure information flow

Prof. Dorothy E. Denning
ACM Fellow (1995)

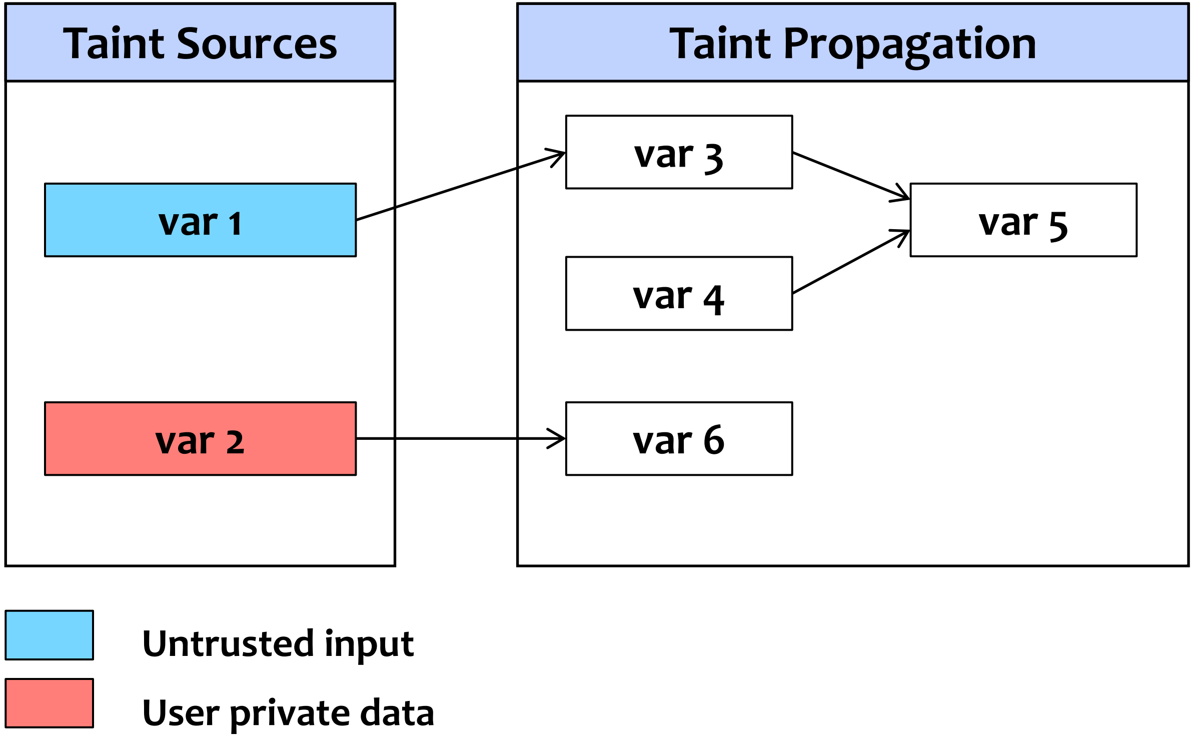
Basic Idea of Taint Analysis



Taint sources introduce untrusted or confidential data into the system under analysis

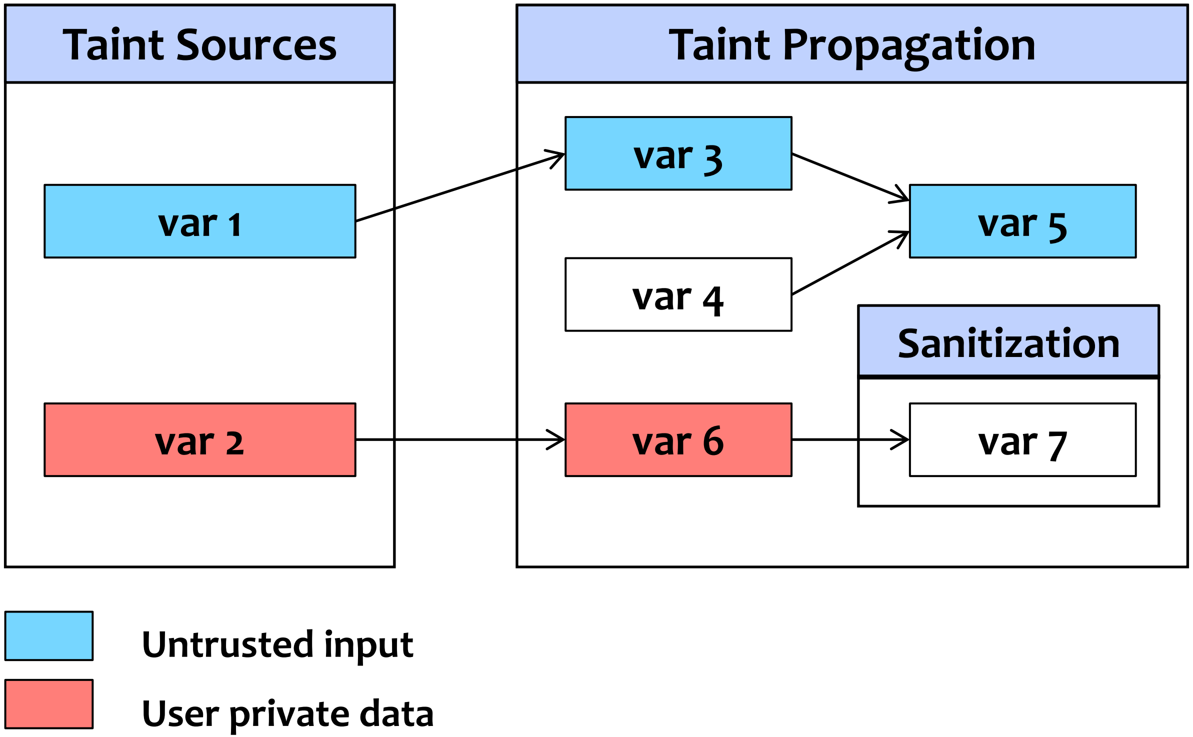
- Return value of `request.getParameter("q")`
- Return value of `getPassword()`

Basic Idea of Taint Analysis



Values derived from tainted data should be tainted according to taint propagation rules

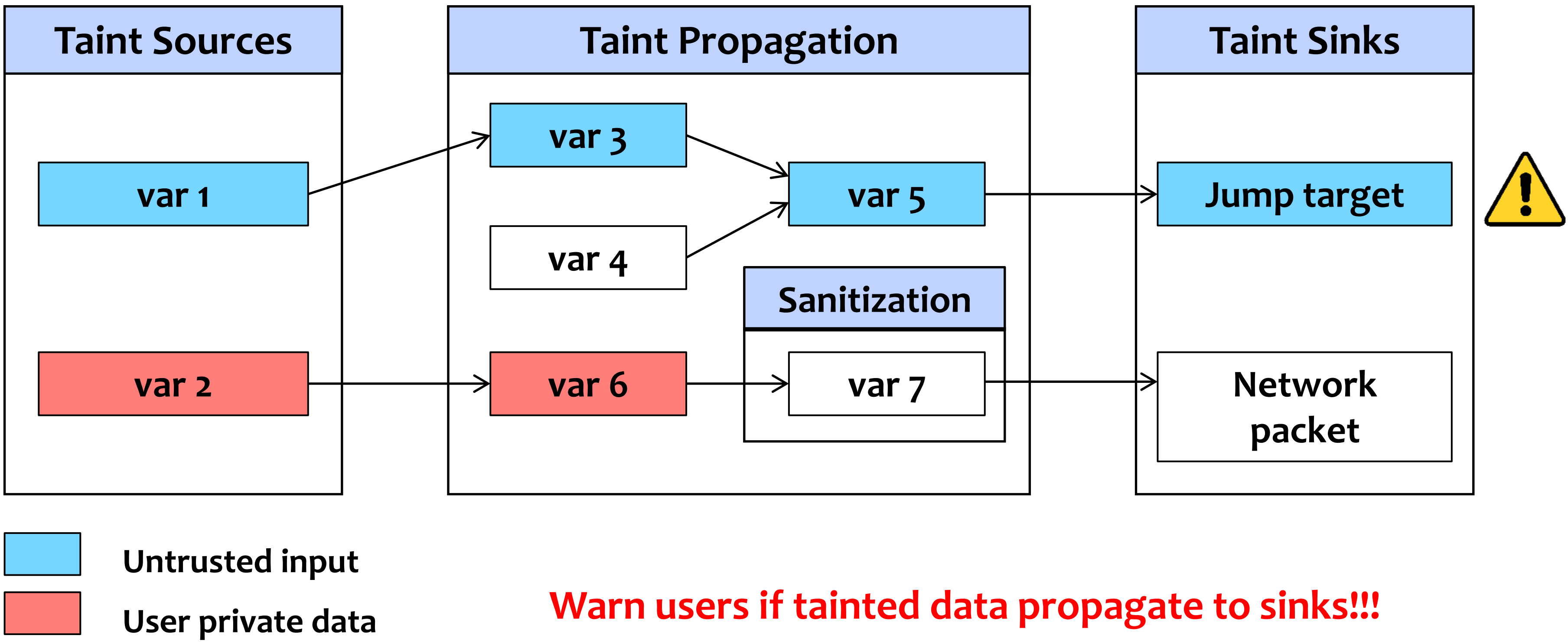
Basic Idea of Taint Analysis



After sanitization, taint should not be propagated further

e.g., if password gets encrypted

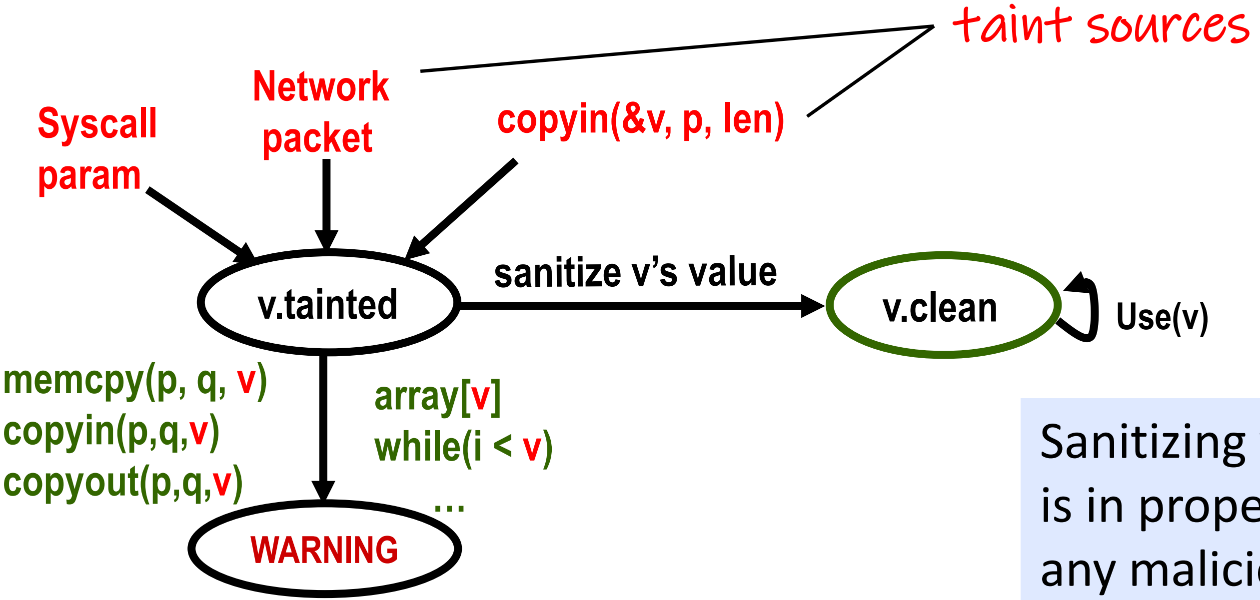
Basic Idea of Taint Analysis



Warn users if tainted data propagate to sinks!!!

Sanitize Integers Before Use

Warn when unchecked integers from taint (untrusted) sources reach trusted sinks



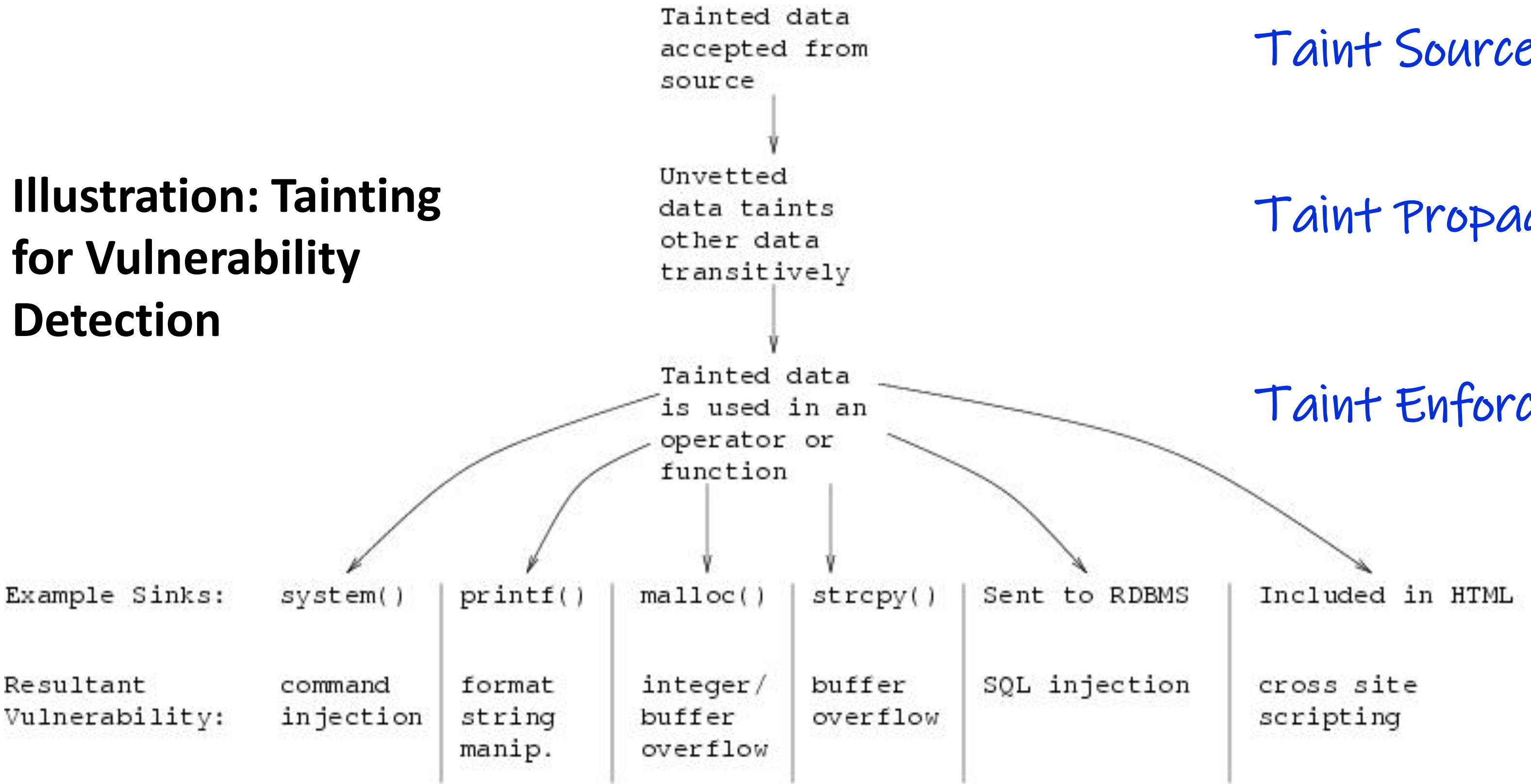
Sanitizing validates if v's value is in proper form and removes any malicious values

3 Components in Taint Analysis

- Taint source
  - At which is the predefined tainted value introduced (i.e., input)?
- Taint propagation
  - How may a tainted value propagate?
  - Default: Any value derived from tainted data is tainted.
- Taint enforcement (involves identification of taint sinks)
  - Property we want to assert
  - For example, *users should not be able to arbitrarily determine jump target address* OR *tainted values should not be outputted*

3 Components in Taint Analysis

Illustration: Tainting for Vulnerability Detection



Taint Source

Taint Propagation

Taint Enforcement

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;    // variable l is also referred to as a taint sink.
```

Variable	Value (int)	Tainted Status

- Performed at each step of execution of a single run

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	6	T

label of this propagation rule

$$\frac{}{\text{get\_input} \underbrace{\Downarrow T}_{\text{receives a tainted status}}}\ \text{INPUT}$$

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	6	T
two	2	F

$$\frac{n \text{ is a constant}}{n \Downarrow F}\ \textsc{Const}$$

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	6	T
two	2	F

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	6	T
two	2	F
j	8	T

$$\frac{t_1 = \text{taint of } x_1 \quad t_2 = \text{taint of } x_2 \quad t = t_1 \vee t_2}{x_1 \Box x_2 \Downarrow t} \text{ OP}$$

Anything derived from tainted data is tainted

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	6	T
two	2	F
j	8	T
l	8	T

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	6	T
two	2	F
j	8	T
l	8	T

$$\frac{\textsc{taint status of } x \textsc{ is } f}{\text{jmp } x \text{ OK}}\ \text{JMP}$$

Jump target is unsafe!!!

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	6	T
two	2	F
j	8	T
l	8	T

Violate the enforcement that user input should not be used as jump target

Dynamic taint analysis would report this as an attack!

Another run …

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	7	T

$$\frac{}{\text{get\_input} \Downarrow T}\ \text{INPUT}$$

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	7	T
two	2	F

$$\frac{n \text{ is a constant}}{n \Downarrow F} \text{ CONST}$$

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	7	T
two	2	F

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	7	T
two	2	F
k	4	F

$$\frac{t_1 = \text{taint of } x_1 \quad t_2 = \text{taint of } x_2 \quad t = t_1 \vee t_2}{x_1 \Box x_2 \Downarrow t} \text{ OP}$$

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	7	T
two	2	F
k	4	F
l	4	F

Dynamic Taint Analysis

```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```

Variable	Value (int)	Tainted Status
i	7	T
two	2	F
k	4	F
l	4	F

$$\frac{\text{TAINT STATUS OF X IS } f}{\text{jmp } x \text{ OK}} \text{ JMP}$$

Dynamic taint analysis reports this is OK.

Current Trends

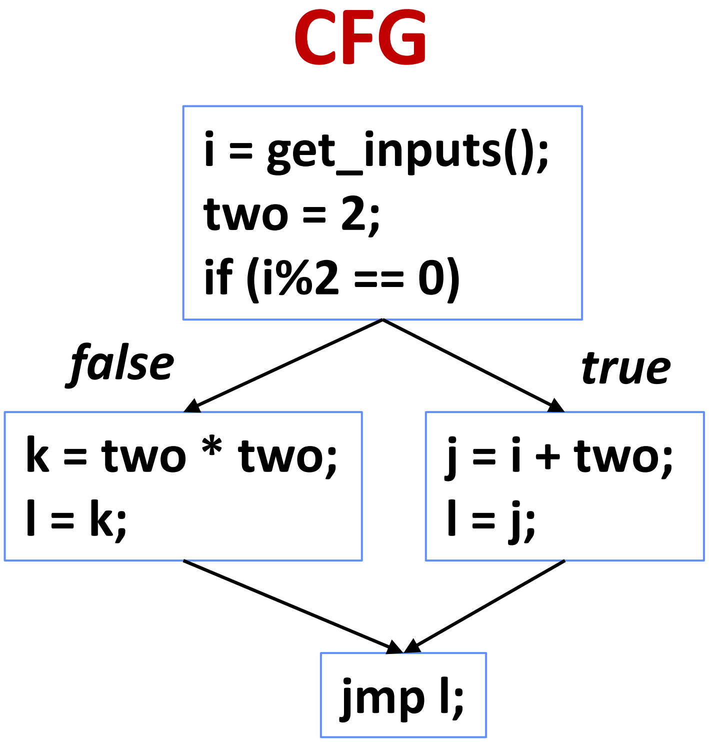
- Apply dynamic taint analysis to binary code
- Use emulator (e.g., TEMU) to inspect each instruction
- Most pressing issue: high overhead
  - Terribly CPU bound: e.g., 30x for gzip
  - IO bound not as serious

Static Taint Analysis

- Analysis performed over multiple paths of a program
- Typically performed on a control flow graph (CFG)
  - A statement is a node
  - A possible transfer of control is an edge between two nodes

Static Taint Analysis

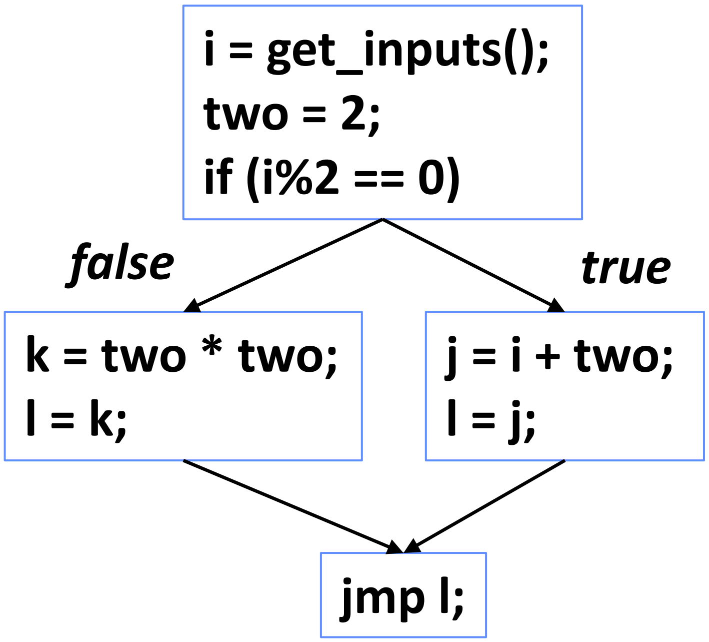
```
i = get_inputs();
two = 2;
if (i%2 == 0) {
  j = i + two;
  l = j;
} else {
  k = two * two;
  l = k;
}
jmp l;
```



Static Taint Analysis

Variable	Tainted Status
i	⊥
two	⊥
k	⊥
l	⊥

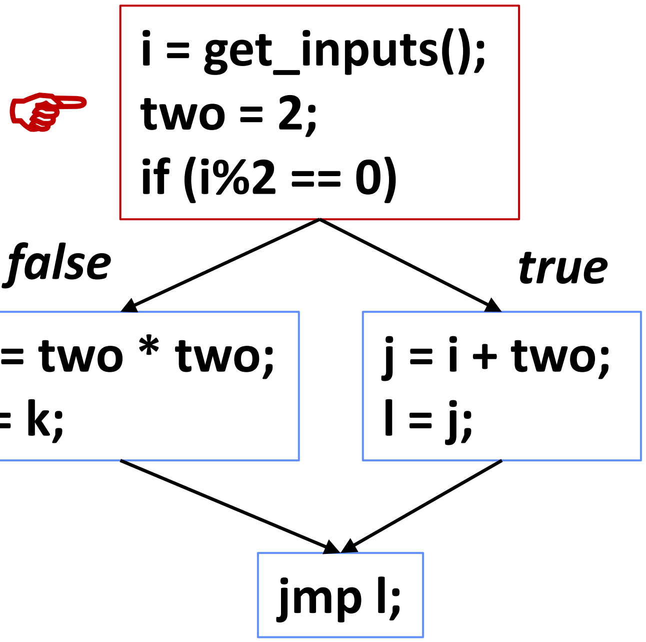
CFG



Static Taint Analysis

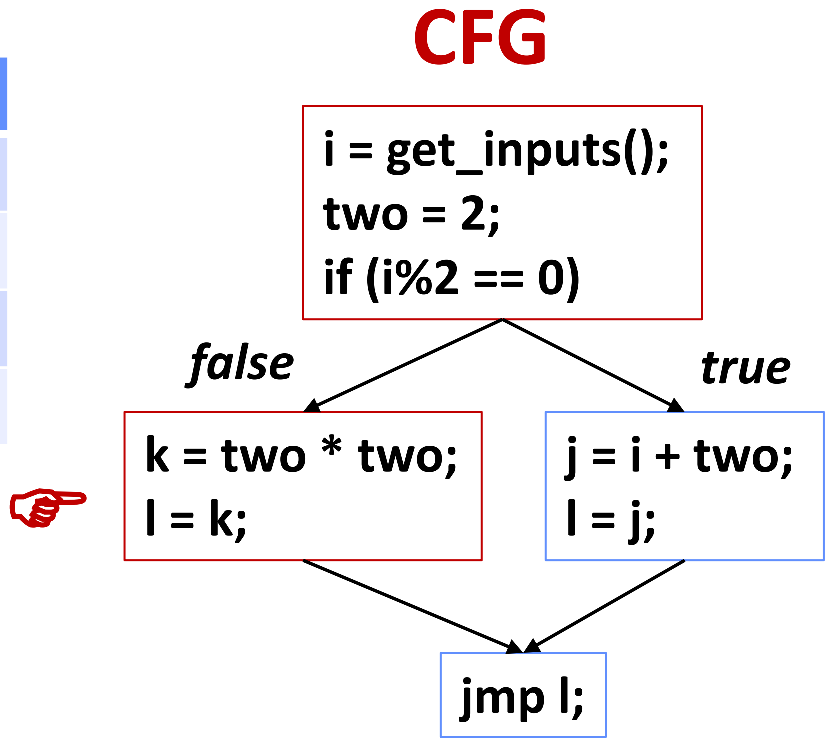
Variable	Tainted Status
i	T
two	F
k	⊥
l	⊥

CFG



Static Taint Analysis

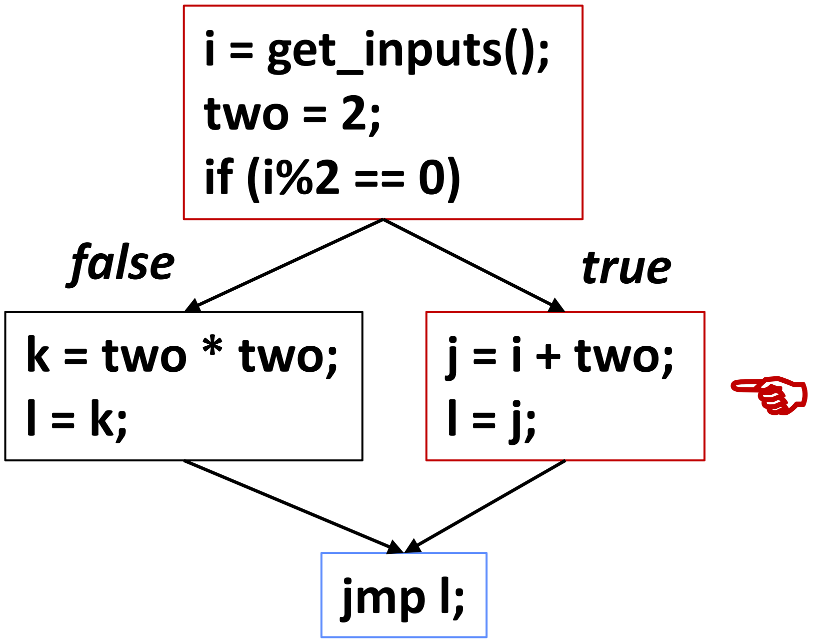
Variable	Tainted Status
i	T
two	F
k	F
l	F



Static Taint Analysis

Variable	Tainted Status
i	T
two	F
j	T
l	T

CFG

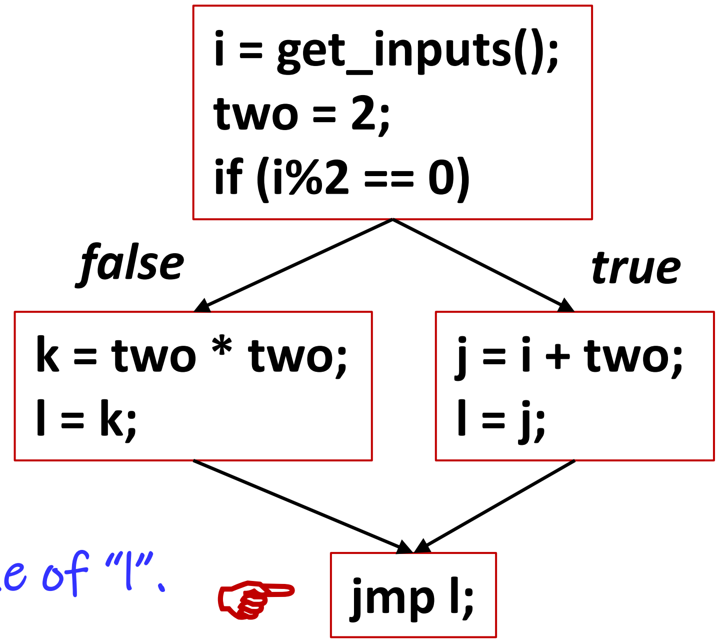


Static Taint Analysis

Variable	Tainted Status
i	T
two	F
k	F
j	T
l	T

Confluence of paths:
We take the most conservative value of "l".

CFG



Comparison

- Looks at a single path
- Determines exact taint values for run
- Must be run on each execution to detect attacks
- Combining multiple runs makes dynamic = static

Dynamic

- Looks at multiple paths
- Must either over- or under-approximate taint at confluence of paths
- Can be used to add monitoring code for only vulnerable paths

Static

Popular Static Tainting Techniques

Technique	Authors	Venue	Citations	Targeting programs	Main purpose
VulFinder*	M. Lam	USENIX Sec'05	680	Web apps (Java)	Vulnerability detection
Pixy (open-source)	E. Kirda	S&P'06	595	Web apps (PHP)	Vulnerability detection
XSSFinder*	Z. Su	ICSE'08	357	Web apps (PHP)	XSS detection
CHEX	W. Lee	CCS'12	431	Android apps (Dalvik bytecode)	Component hijacking vulnerability detection
FlowDroid	E. Bodden	PLDI'14	688	Android apps (Dalvik bytecode)	General data flow tracking

*** The original technique does not have a name**

Challenges of Tainting

- Memory addresses vs. values (same challenge applicable to dynamic tainting)

```
i = get_input();
a = arr[i];
goto a;
```

Address i is tainted, but the value at arr[i] may not need to be tainted ...

Shall we taint a? It is hard to decide ...

- No → undertainting (results in false negatives)
- Yes → overtainting (results in false positives)

Challenges of Tainting

- Container issues (containers are widely-used data structures)

```
i = get_input();
list.put(i);
…

j = list.get(idx);
goto j;
```

Tainted value flows into a container

Shall we taint j? Same dilemma …

- No → undertainting (results in false negatives)
- Yes → overtainting (results in false positives)

Challenges of Tainting

- Recognize sanitization code and determine if sanitization is sufficient

```
<?php
  $text = user_input();
  echo strip_tags($text);
?>
```

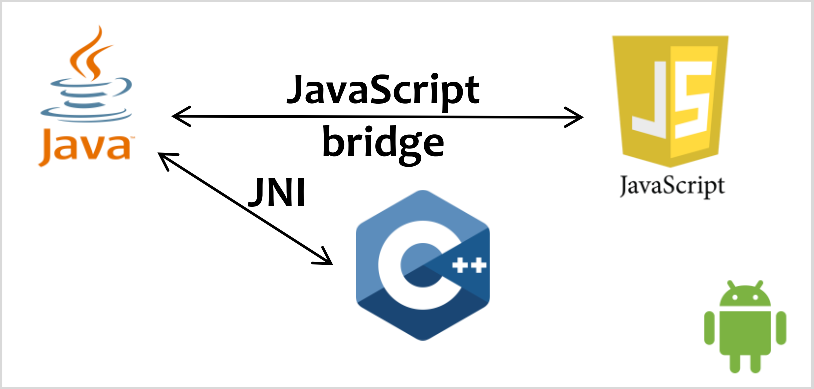
Standard lib call, this is easy …

```
$preg = '/(=|(U\s*R\s*L\s*\()\s*' .
'("|\')?[^>]*\s*' .
'S\s*C\s*R\s*I\s*P\s*T\s*:/i';
$data = preg_replace(
    $preg, 'HordeCleaned', $data);

/* Get all on<foo>="bar()".
 * NEVER allow these. */
$data = preg_replace(
    '/([\s"\']+on\w+)\s*=/i',
    'HordeCleaned=', $data);

/* Remove all scripts. */
$data = preg_replace(
    '|<script[^>]*>.*?</script>|is',
    '<HordeCleaned_script />', $data);
```

How about developers' own sanitization code?

Is the sanitization sufficient?

Future Trend of Taint Analysis



Cross-language analysis

Many projects are multi-lingual

Hybrid analysis

Conclusion

- Taint analysis is a classic (and old) data flow analysis technique
- Taint analysis is widely used to ensure software security
- Some issues with existing techniques
  - Overhead
  - Implicit flows
  - Sanitization recognition
  - Tainted address vs. value
  - Container issues

Online Resources

- Determination of the exploitability of bugs
- Malware investigation and defense
- Information flow security enforcement
- Static binary analysis for find vulnerabilities and backdoors
- Dynamic slicing for Python programs
- Information privacy and security (a series of 12 videos)