ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Static Analysis of Virtualization-Obfuscated Binaries

Johannes Kinder

School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Virtualization Obfuscation

- Obfuscation
  - *Make code hard to understand for humans and tools*
  - *Popular for protecting benign and malicious code*
- Virtualization Obfuscation
  - *Hide code inside self-contained VM*
  - *Considered one of the strongest obfuscation schemes*
  - *Commercial tools (CodeVirtualizer, VMProtect) and research obfuscator*

Static Analysis vs. Virtualization

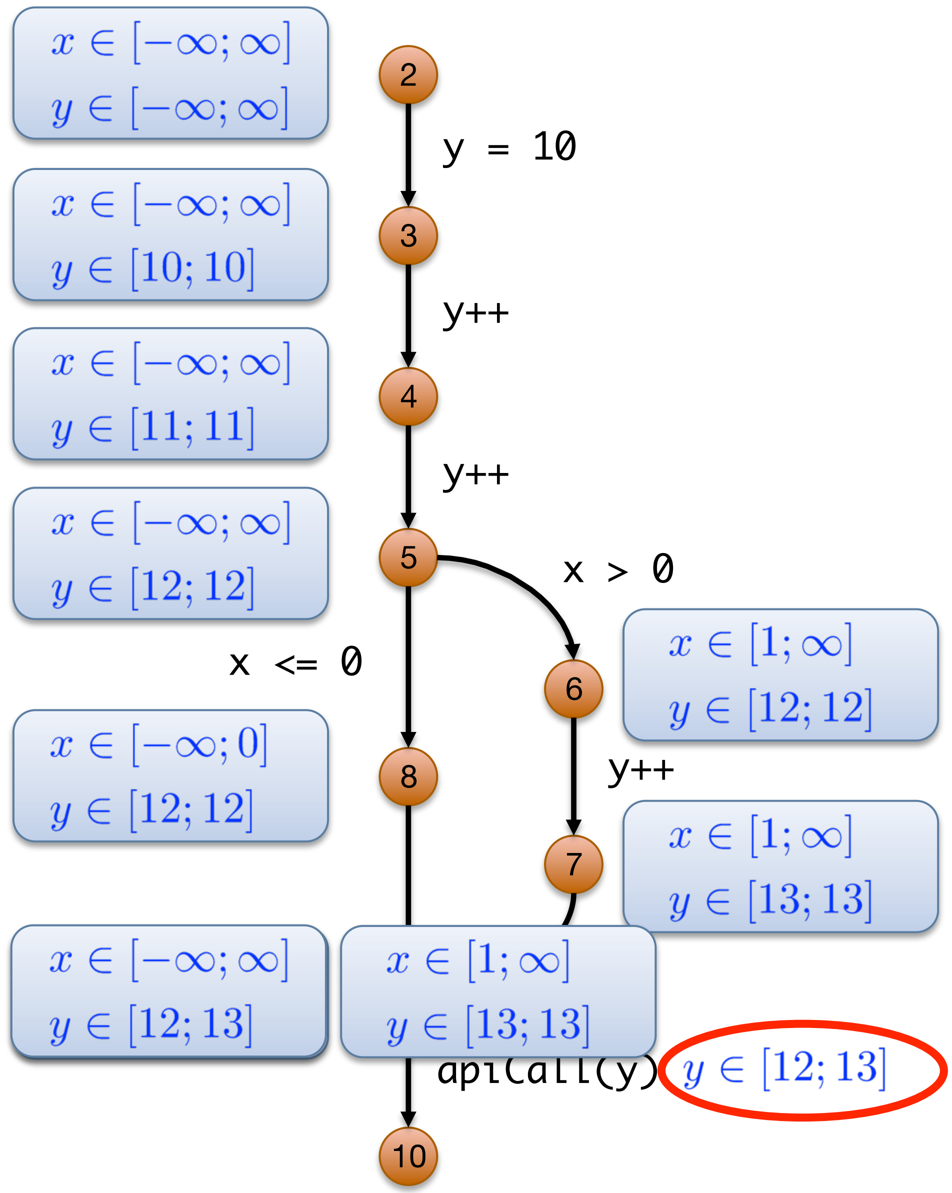
- Static Analysis
  - *Computes program invariants*
  - *Safety proofs, bug finding, malware detection*

What makes static analysis of virtualization-obfuscated code fundamentally hard?

Can we still make it work and use it for deobfuscation?

Static Analysis

```c
void foo (int x) {
  int y = 10;
  y++;
  y++;
  if (x > 0) {
    y++;
  }
  else {}
  apiCall(y);
}
```



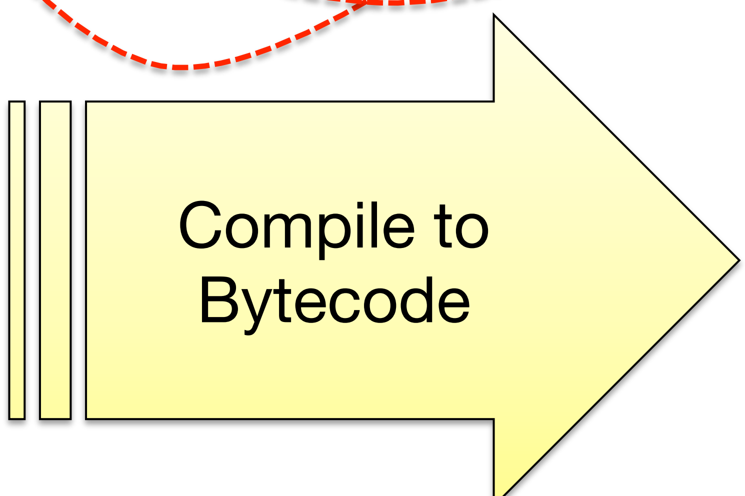
Virtualization Obfuscation

```
void foo (int x) {
  int y = 10;
  y++;
  y++;
  if (x > 0) {
     y++;
  }
  else {}
  apiCall(y);
}
```

```
code = { 52, 01, 02, 03, 01, 03, 01, 08,
         00, 03, 03, 01, 18, 01, 00  }

data = { 00, 00, 10, 05 }
```

x y

conditional jump distance

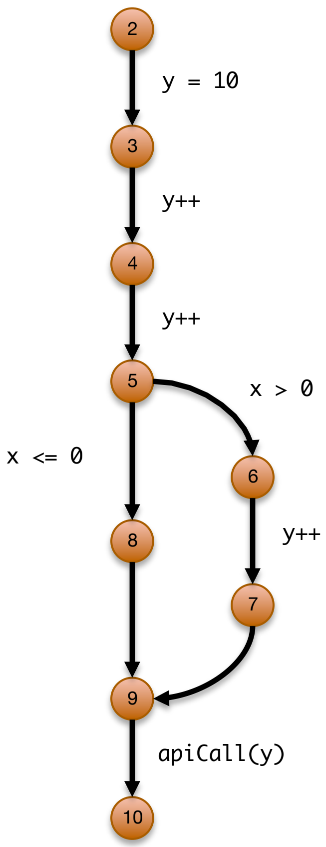


```
int vpc = 0, op1, op2;
while (true) {
  switch(code[vpc]) {
    case 03: // increment
      op1 = code[vpc + 1];
      data[op1]++;
      vpc += 2;
      break;
    case 08: // conditional jump
      op1 = code[vpc + 1];
      op2 = code[vpc + 2];
      if (data[op1] <= 0)
        vpc += data[op2]
      else
        vpc += 3;
      break;
    case 18: // call function
      op1 = code[vpc + 1];
      apiCall(data[op1]);
      vpc += 2;
      break;
    case 52: // assignment
      op1 = code[vpc + 1];
      op2 = code[vpc + 2];
      data[op1] = data[op2];
      vpc += 3;
      break;
    default: // halt
      return;
  } // end switch
} // end while
```

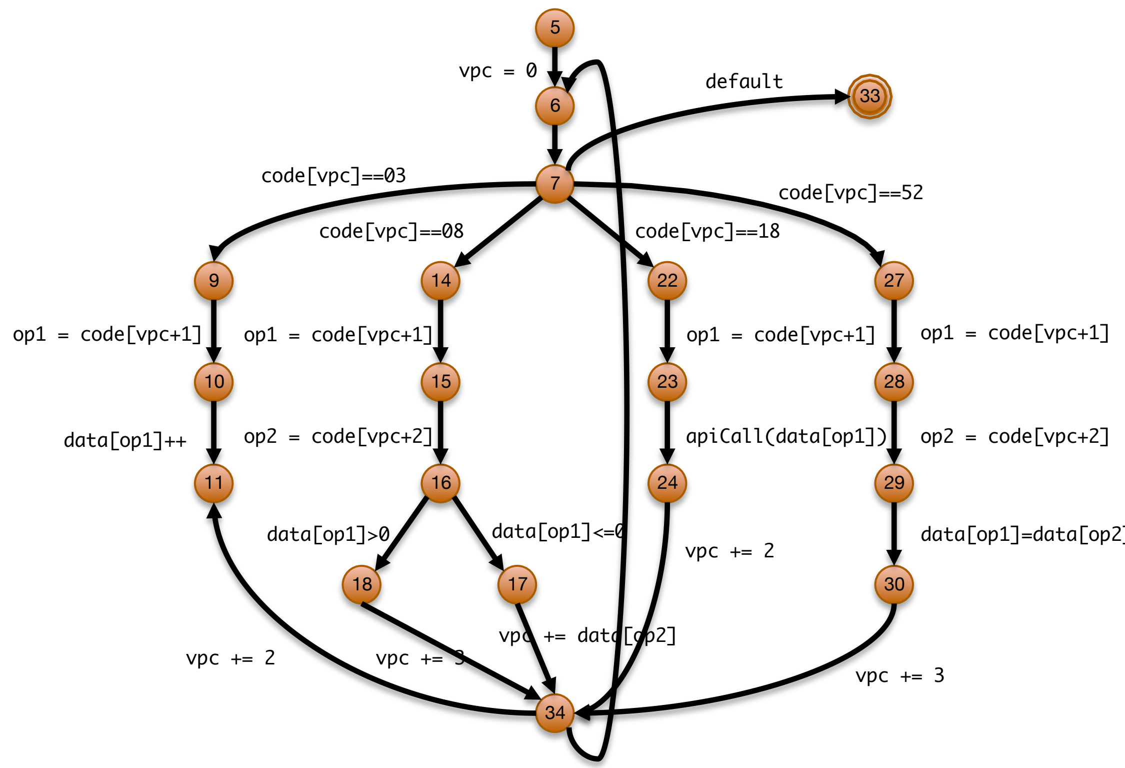
```
code = { 52, 01, 02, 03, 01, 03, 01, 08,
         00, 03, 03, 01, 18, 01, 00  }

data = { 00, 00, 10, 05 }
```

x y conditional jump distance

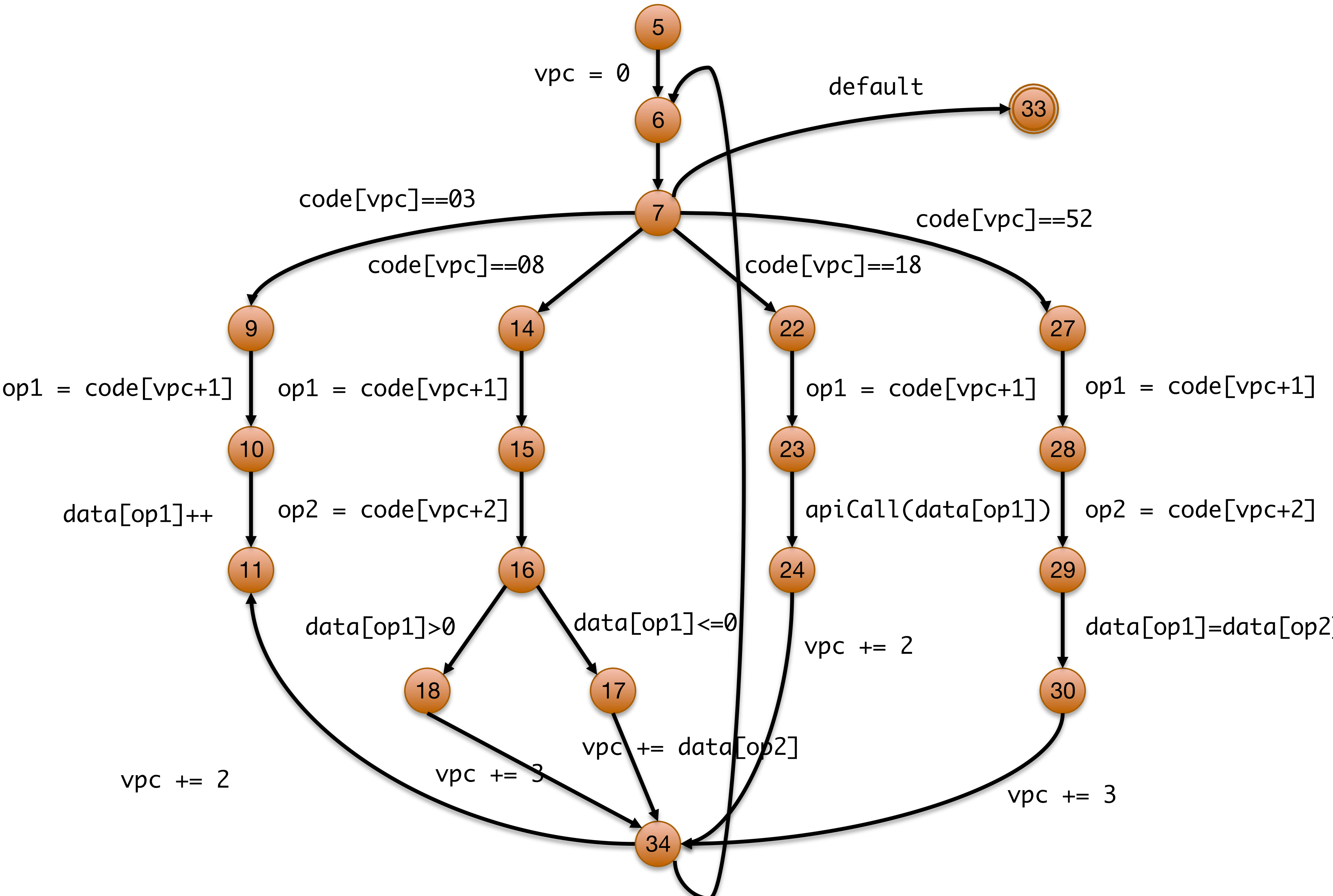
Control Flow Graphs



Original CFG

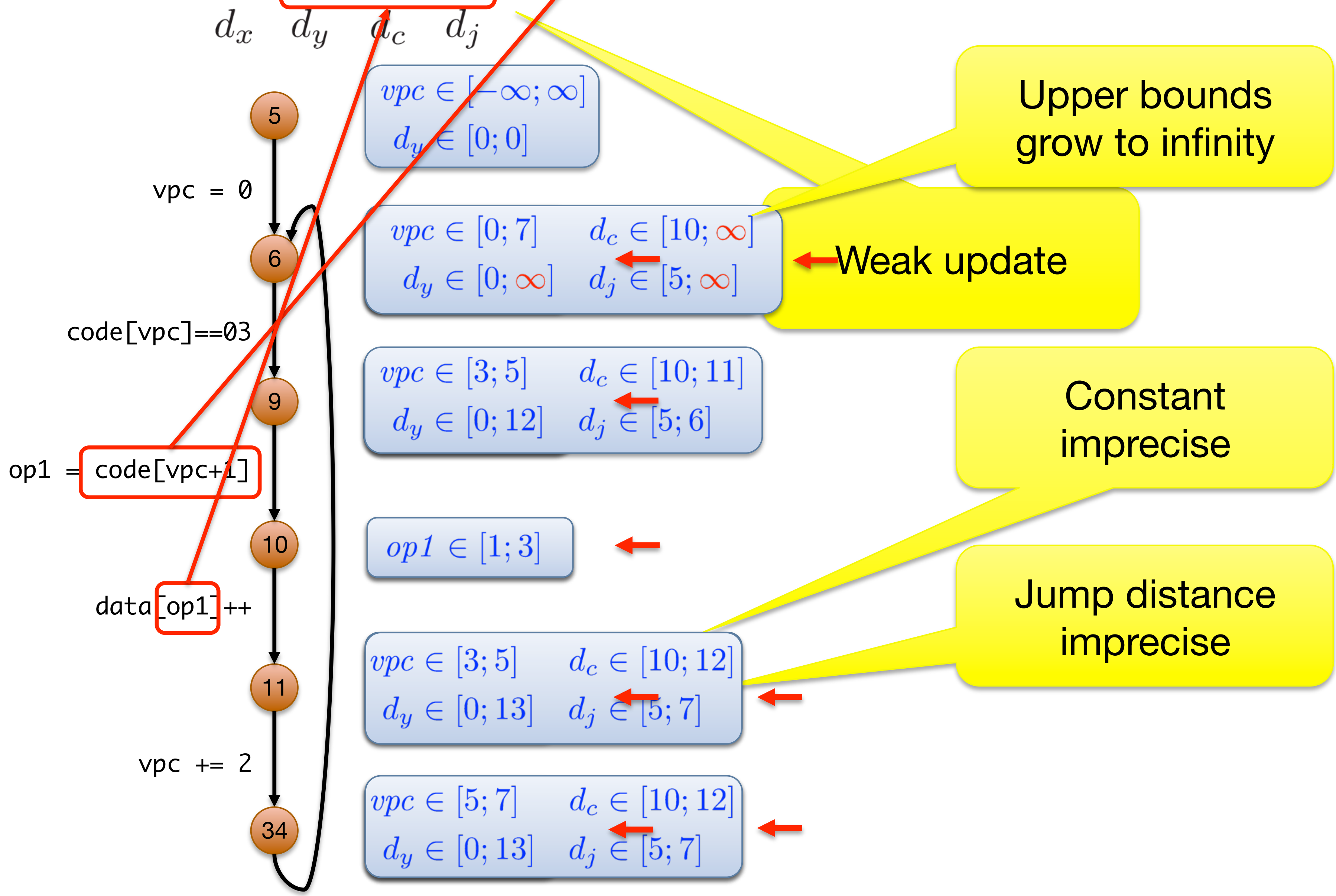


Obfuscated CFG

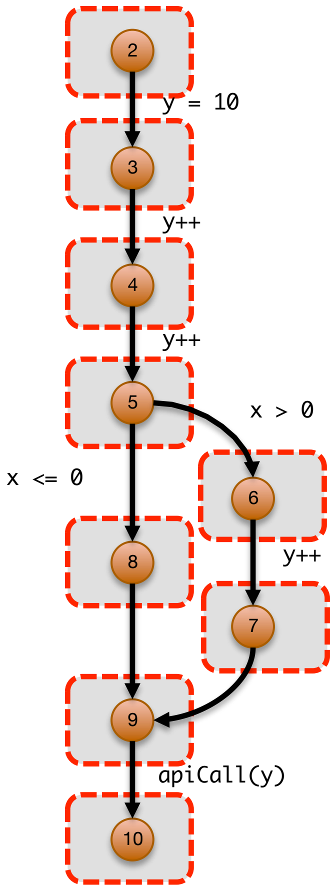


5
vpc = 0
6
default
33
7
code[vpc]==03
code[vpc]==52
code[vpc]==08
code[vpc]==18
9
14
22
27
op1 = code[vpc+1]
op1 = code[vpc+1]
op1 = code[vpc+1]
op1 = code[vpc+1]
10
15
23
28
data[op1]++
op2 = code[vpc+2]
apiCall(data[op1])
op2 = code[vpc+2]
11
16
24
29
data[op1]>0
data[op1]<=0
vpc += 2
data[op1]=data[op2]
18
17
30
vpc += data[op2]
vpc += 2
vpc += 3
vpc += 3
34

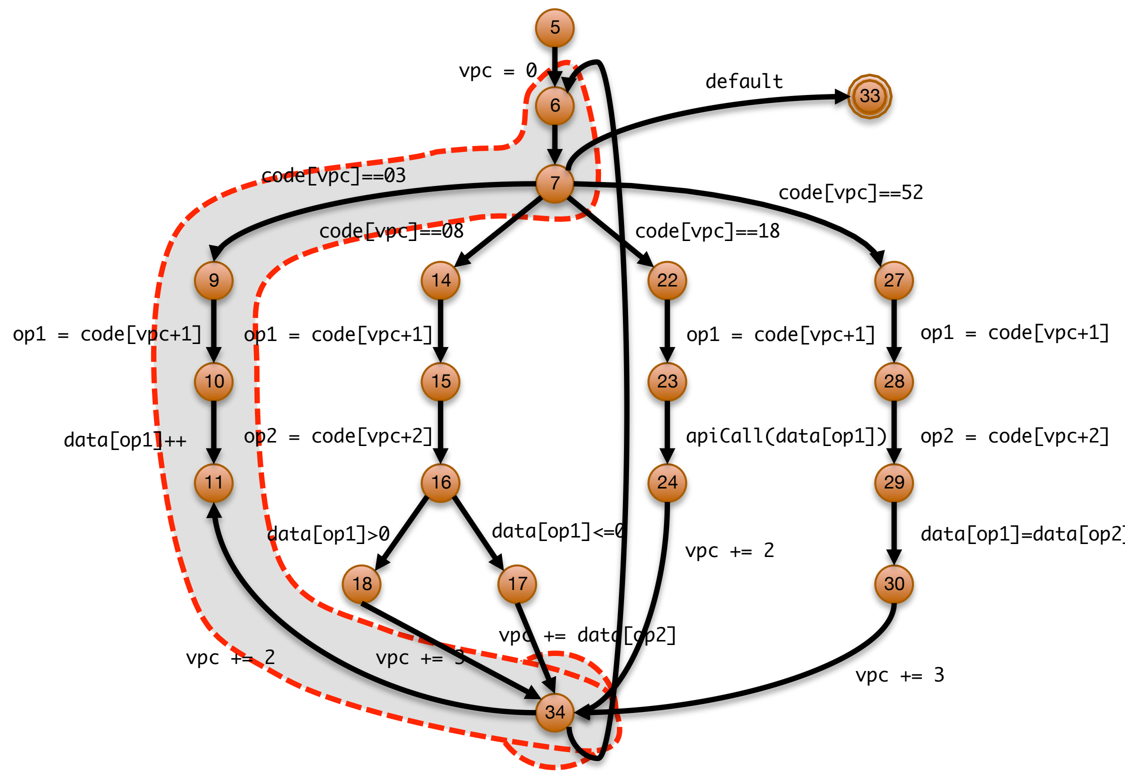
```
code = { 52, 01, 02, 03, 01, 03, 01, 08, 00, 03, 03, 01, 18, 01, 00 }

data = { 00, 00, 10, 05 }
```



- 1 interpreter case = many original locations
- Interpreter loop head shared among all

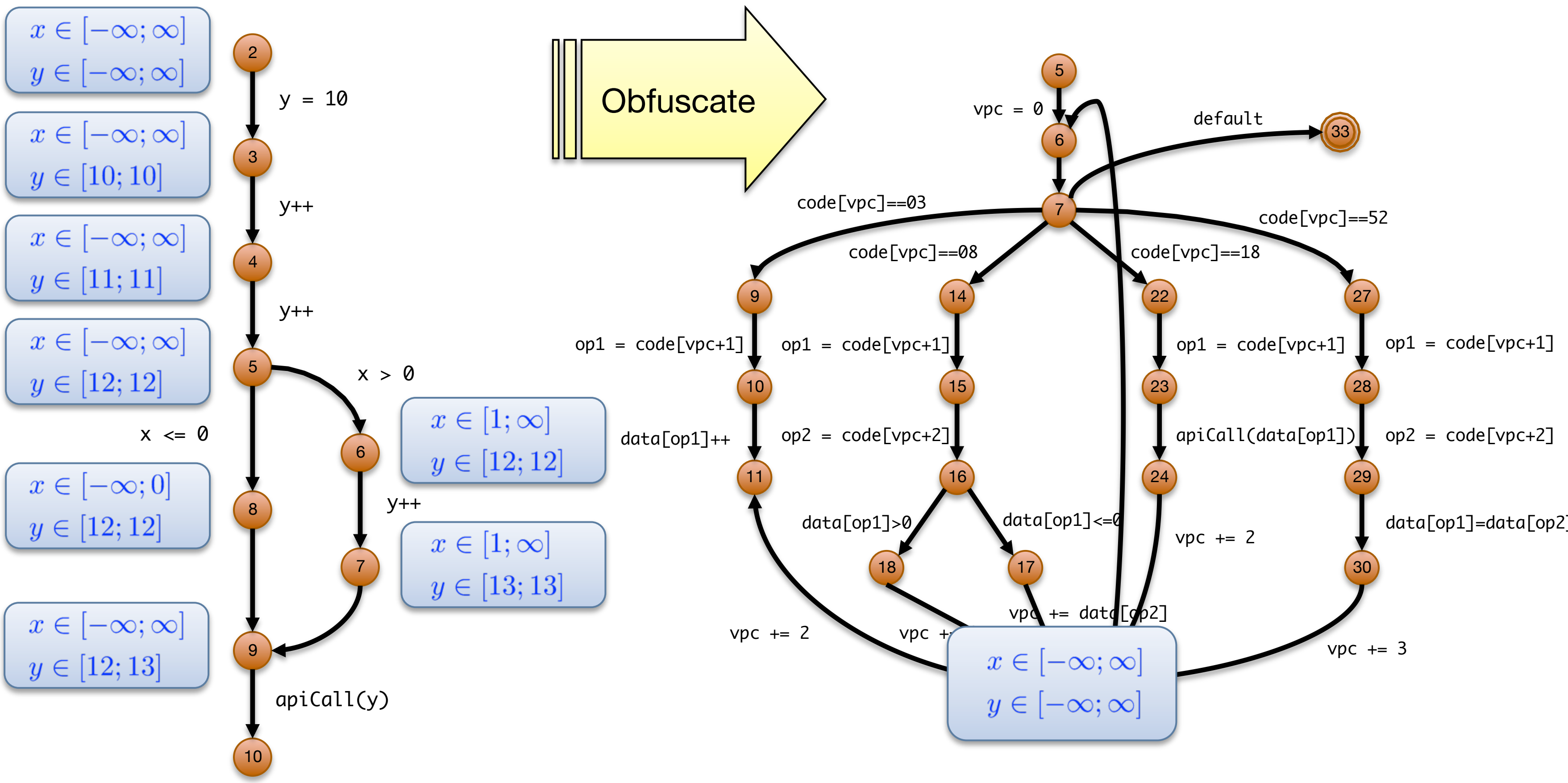


Original CFG



Obfuscated CFG

Domain Flattening



Location Sensitive Analysis → Location Insensitive Analysis

VPC Lifting

- Virtualization flattens one dimension of location
- Idea: track VPC and use as additional dimension
  - *Separate states with differing VPC values*

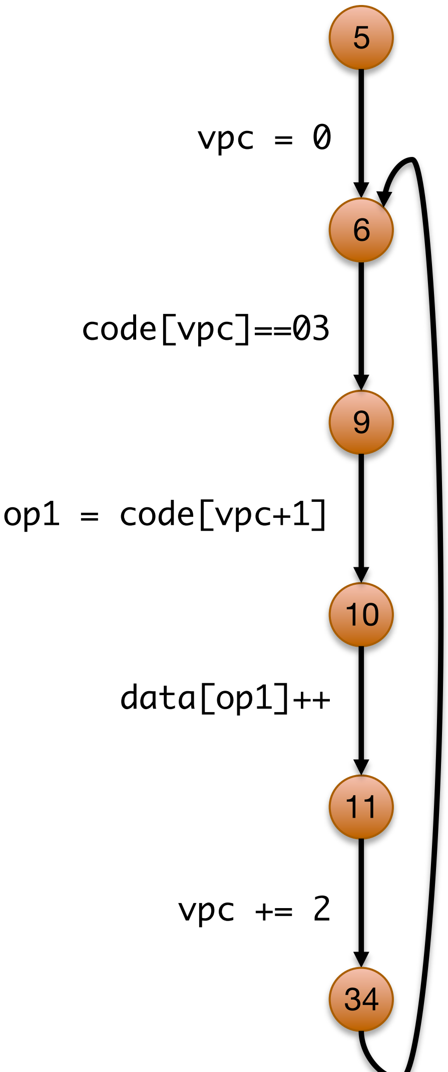
```
int y = 10;     y++;
```

$vpc \in [0;0]$, $d_y \in [0;0]$ $\quad$ $vpc \in [3;3]$, $d_y \in [10;10]$

  - *Join states with equal VPC values*

```
if (…) {}     else {}
```

$vpc \in [12;12]$, $d_y \in [12;13]$ $\quad$ $vpc \in [12;12]$, $d_y \in [13;13]$



`y++`	`y++`	`{ y++ }`	`apiCall(y)`
$vpc \in [3;3]$	$[5;5]$	$[10;10]$	$[12;12]$
$d_x \in [-\infty;\infty]$ $d_y \in [10;10]$	$d_x \in [-\infty;\infty]$ $d_y \in [11;11]$	$d_x \in [1;\infty]$ $d_y \in [12;12]$	$d_x \in [-\infty;\infty]$ $d_y \in [12;13]$
$d_x \in [-\infty;\infty]$ $d_y \in [10;10]$	$d_x \in [-\infty;\infty]$ $d_y \in [11;11]$	$d_x \in [1;\infty]$ $d_y \in [12;12]$	
$d_x \in [-\infty;\infty]$ $d_y \in [10;10]$	$d_x \in [-\infty;\infty]$ $d_y \in [11;11]$	$d_x \in [1;\infty]$ $d_y \in [12;12]$	
$d_x \in [-\infty;\infty]$ $d_y \in [11;11]$	$d_x \in [-\infty;\infty]$ $d_y \in [12;12]$	$d_x \in [1;\infty]$ $d_y \in [13;13]$	
$d_x \in [-\infty;\infty]$ $d_y \in [11;11]$	$d_x \in [-\infty;\infty]$ $d_y \in [12;12]$	$d_x \in [1;\infty]$ $d_y \in [13;13]$	$d_x \in [-\infty;\infty]$ $d_y \in [12;13]$

VPC Values as Locations

- Location-sensitive analysis over domain A has domain

$$L \to A$$

- VPC-sensitive analysis over domain A, for VPC domain V, has domain

$$\underbrace{L \to V}_{\text{"VPC Location"}} \to A$$

Reconstructing CFGs

- A program CFG is the set of all feasible transitions between *locations*

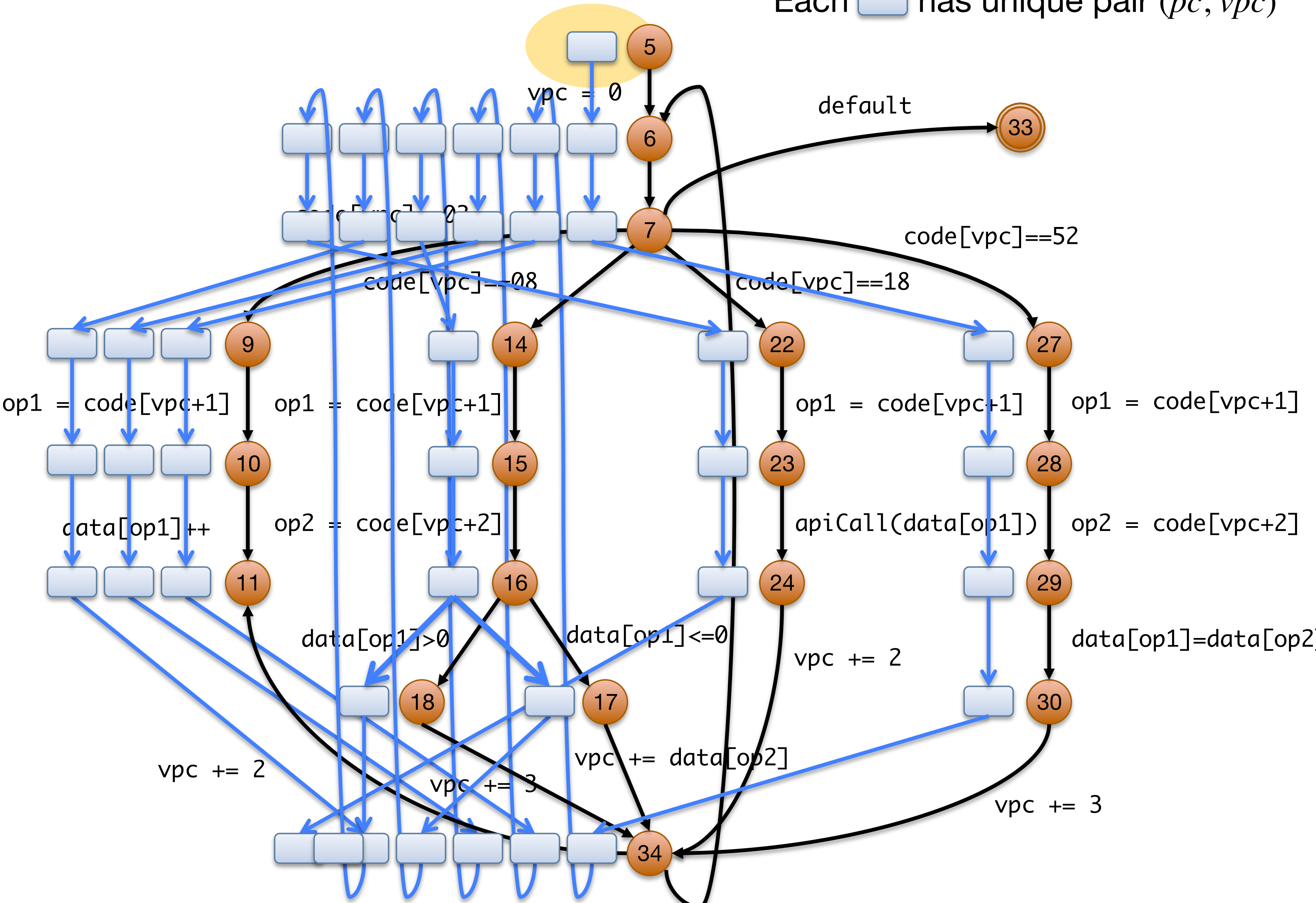
$$CFG = \{(\ell, \ell') \mid (\ell, a) \mapsto (\ell', a'), \ell, \ell' \in L, a, a' \in A\}$$

Abstract transition relation

- A *VPC-CFG* is the set of all feasible transitions between *VPC locations*

$$\begin{aligned} CFG = \{((\ell, v), (\ell', v')) \mid (\ell, v, a) \mapsto (\ell', v', a'), \\ \ell, \ell' \in L, a, a' \in A, v, v' \in V\} \end{aligned}$$

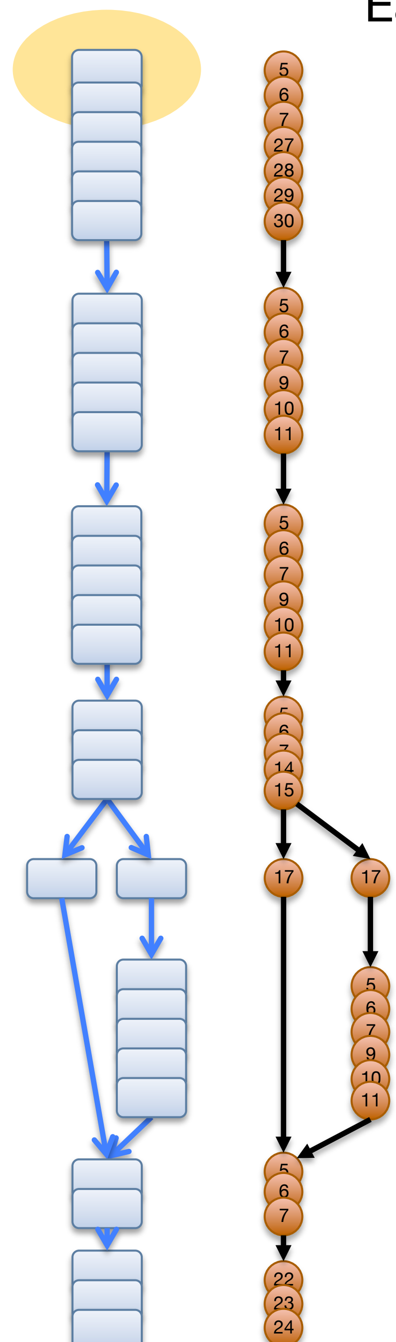
Each has unique pair (pc, vpc)

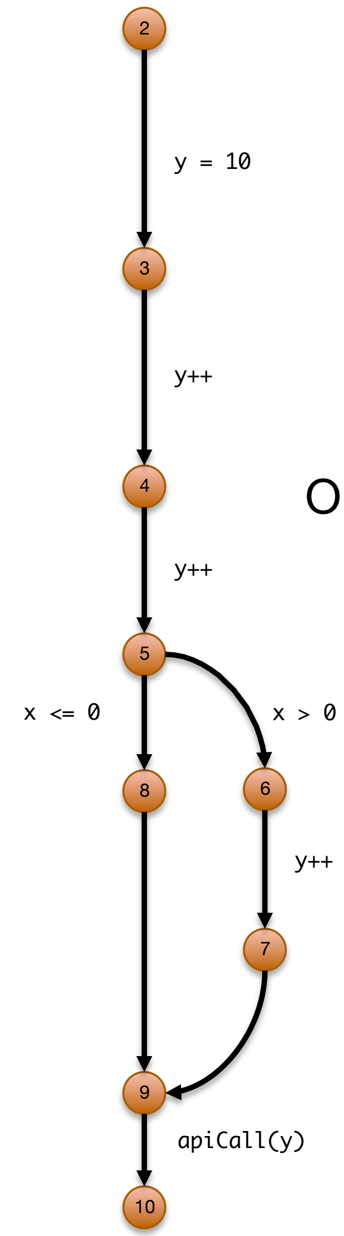


Each [box] has unique pair (pc, vpc)

```
switch(code[vpc]) {
     case 03:
       op1 = code[vpc + 1];
       data[op1]++;
       vpc += 2;
       break;
```

- Constant propagation
- Dead code elimination
- Jump threading





Original CFG

Implementation

- Implemented in Jakstab [CAV'08]
  - *Processes obfuscated binaries*
  - *Reconstructs CFGs in presence of indirect jumps*
- Analysis
  - *VPC-lifted Bounded Address Tracking* [FMCAD'10]
  - *Sets of memory states up to per-variable bound*

$$\left\{ \begin{pmatrix} x = 5 \\ y = 20 \end{pmatrix}, \begin{pmatrix} x = 6 \\ y = 20 \end{pmatrix}, \begin{pmatrix} x = 7 \\ y = 24 \end{pmatrix} \right\} \cup \left\{ \begin{pmatrix} x = 8 \\ y = 24 \end{pmatrix}, \begin{pmatrix} x = 9 \\ y = 28 \end{pmatrix} \right\}$$

k = 3

$$\left\{ \begin{pmatrix} x = 5 \\ y = 20 \end{pmatrix}, \begin{pmatrix} x = 6 \\ y = 20 \end{pmatrix}, \begin{pmatrix} x = \top \\ y = 24 \end{pmatrix}, \begin{pmatrix} x = \top \\ y = 28 \end{pmatrix} \right\}$$

VPC Discovery

- Intuition: VPC changes frequently
  - *Each instruction will have a separate VPC value*
  - *VPC is the busiest variable of the interpreter*
- Detect VPC on the fly
  - *The variable that hits the value bound per variable first is promoted to VPC*
  - *Heuristic – not guaranteed to identify it correctly*

Preliminary Results

Benchmark	Baseline	Similarity	Time
tamperproof guard	0%	81%	13s
search tree	0%	100%	312s
matrix multiply	0%	100%	311s
stuxnet	0%	87%	319s

- Targets
  - *Created with research obfuscator (C. Collberg)*
- Similarity
  - *Graph edit distance using basic block markers*
  - *Baseline: similarity of obfuscated code to original*

Conclusion

- Virtualization causes domain flattening
  - *Strips one level of location sensitivity*
- VPC-lifting reintroduces location sensitivity
- CFG reconstruction by tracing VPC values
- Ongoing work
  - *Improve VPC discovery*
  - *Apply to real world obfuscators*

`http://www.jakstab.org`