PIFT: Predictive Information-Flow Tracking

Man-Ki Yoon

University of Illinois at Urbana-Champaign

Negin Salajegheh, Yin Chen, Mihai Christodorescu

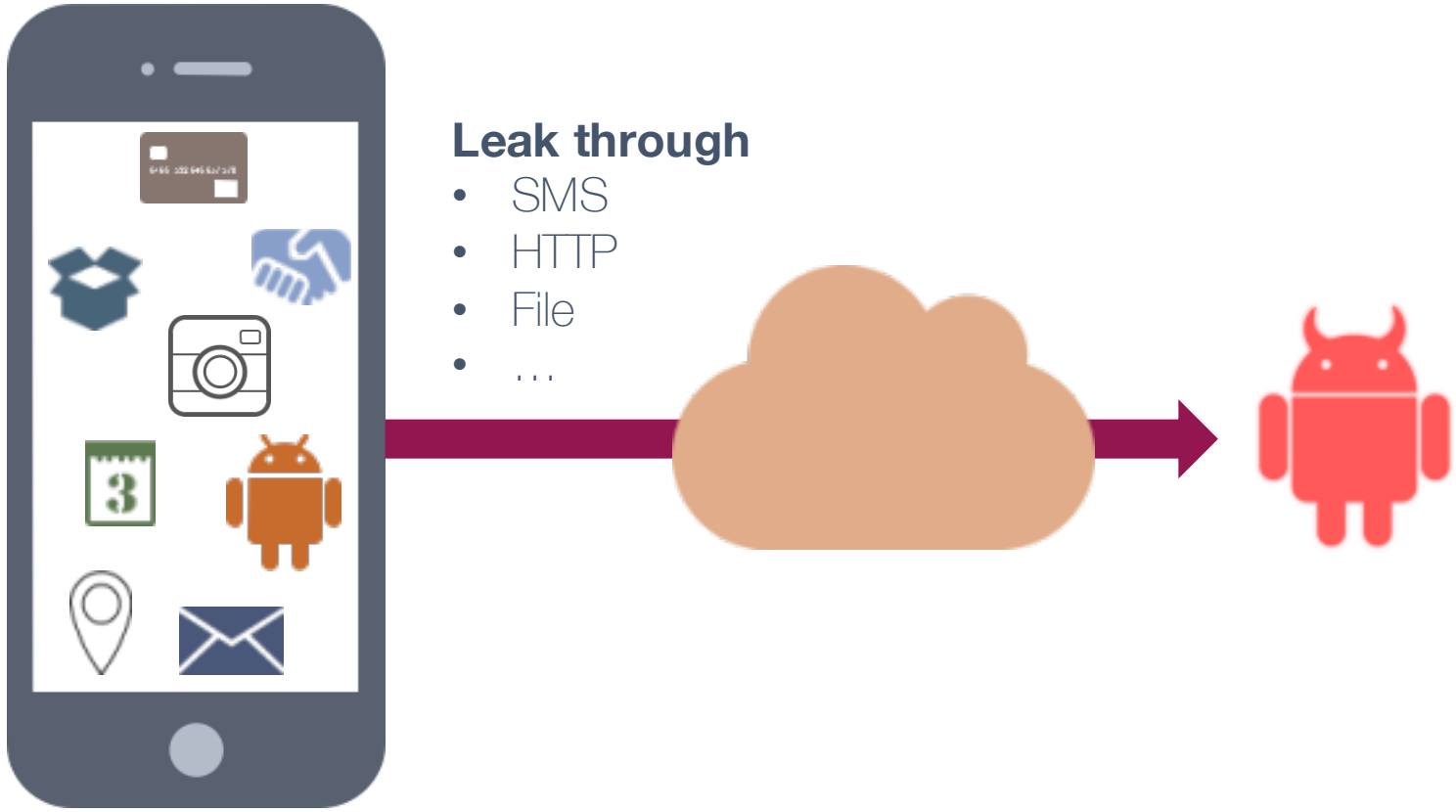
Qualcomm Research Silicon Valley

April 6, 2016

Information Leakage

Sensitive Information

- Location
- Phone number
- Device ID
- Health data
- Contact list
- Messages
- Photos
- …

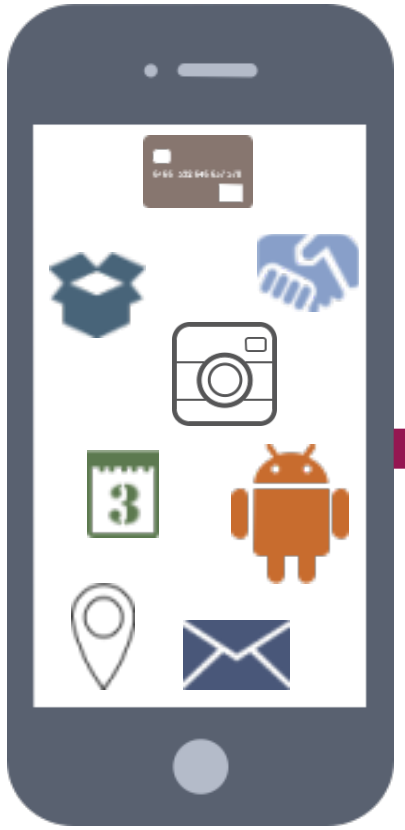


Leak through

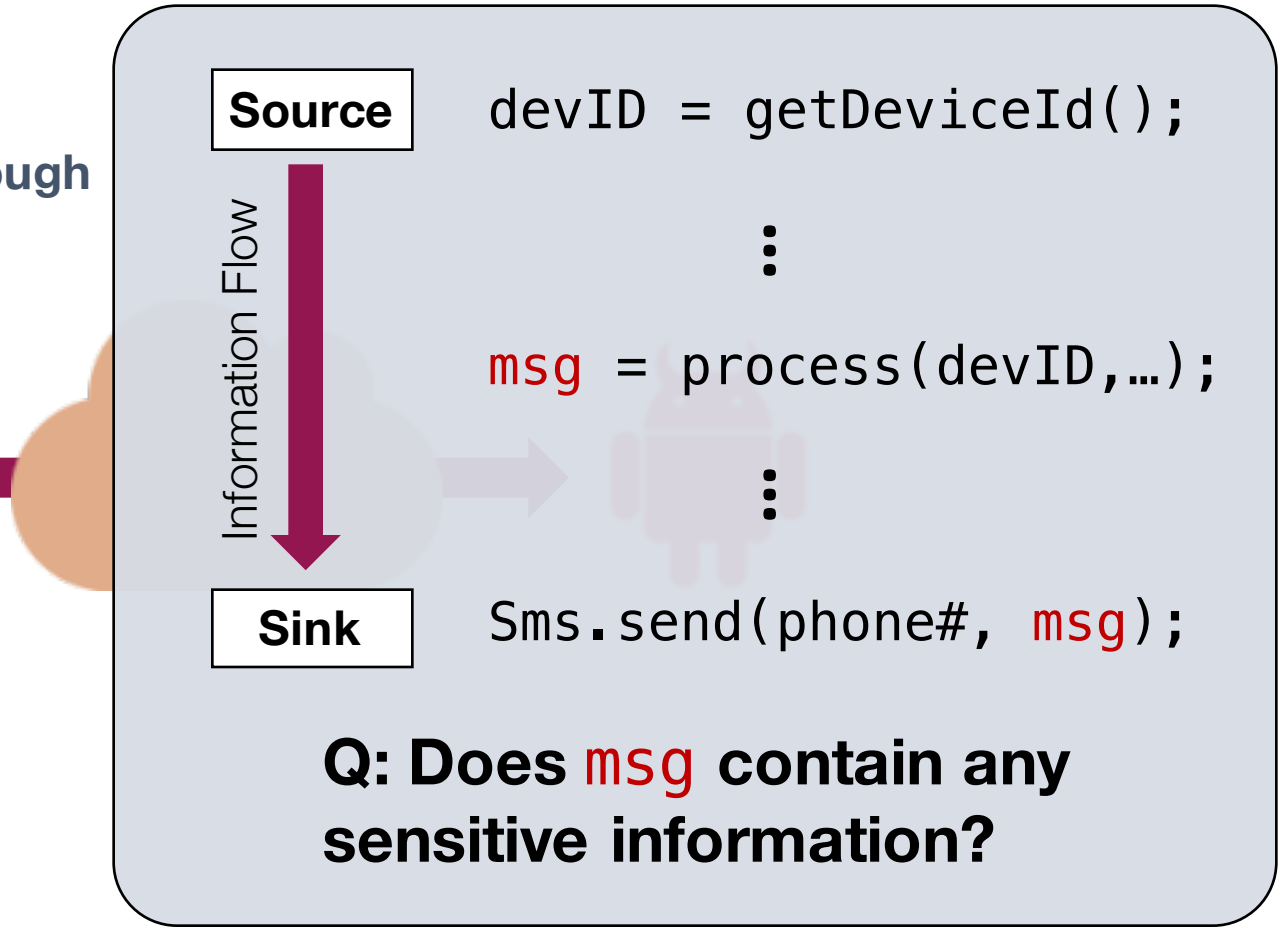
- SMS
- HTTP
- File
- …

Information Flow Tracking

Sensitive Information
- Location
- Phone number
- Device ID
- Health data
- Contact list
- Messages
- Photos
- …

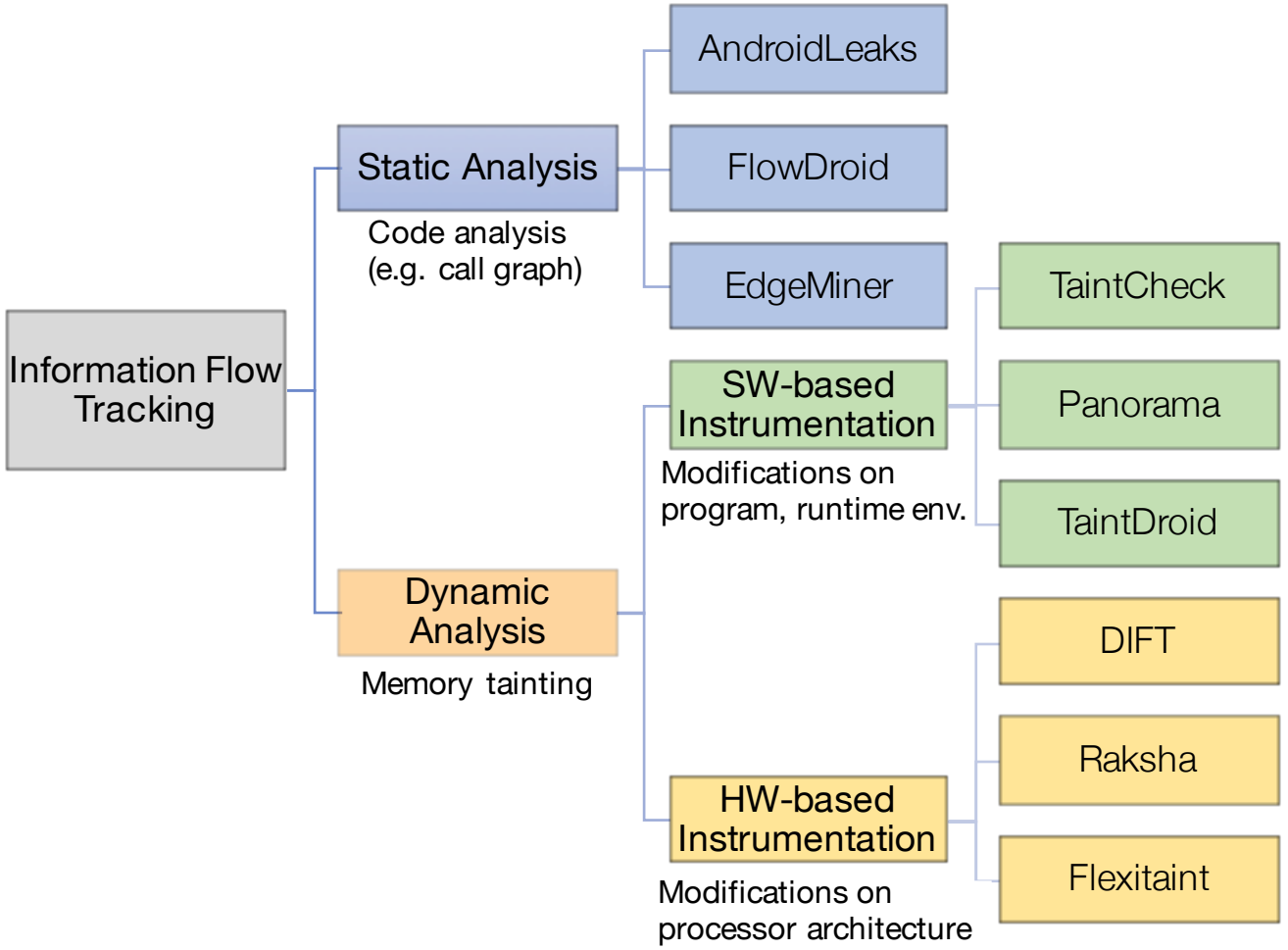


Leak through
- SMS
- HTTP
- File
- …

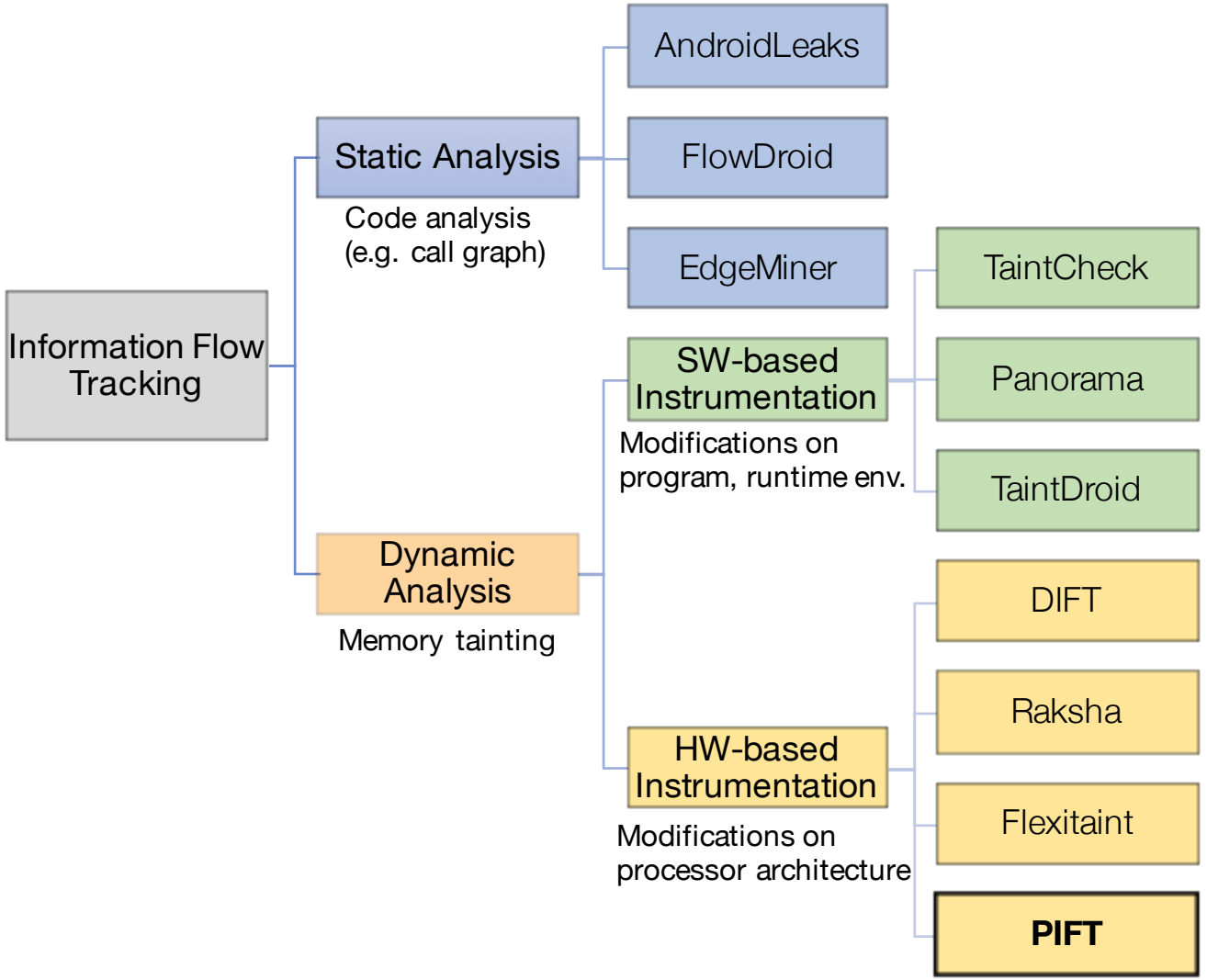


Q: Does `msg` contain any sensitive information?

Information Flow Tracking



Information Flow Tracking



System and Threat Models

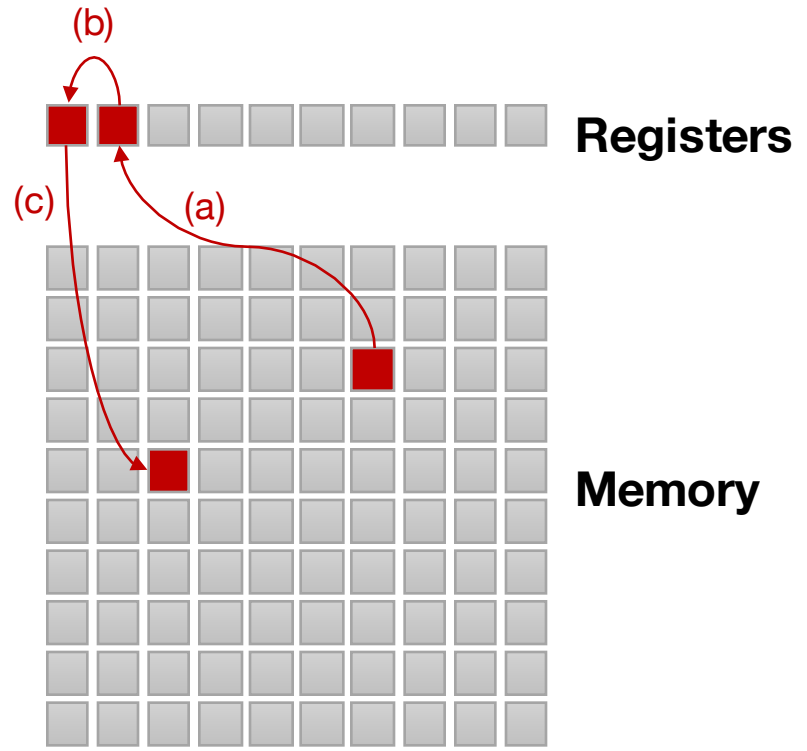
- Android-on-ARM
- Malware that steals and leaks out sensitive data
- Direct information flow

Fine-grained Taint Tracking

- Tags memory and registers with *taint flags*
  - Tainted: contains sensitive data
- Taints propagation
  - From source operand to destination operand

```
mov   r3, r7, LSR #12
ubfx  r9, r7, #8, #4
ldr   r1, [r5, r3 LSL #2]   (a)
ldr   r0, [r5, r9 LSL #2]
ldrh  r7, [r4, #2]!
mul   r0, r1, r0            (b)
and   r12, r7, #255
str   r0, [r5, r9 LSL #2]   (c)
add   pc, r8, r12, LSL #6
```

Integer multiplication in Java



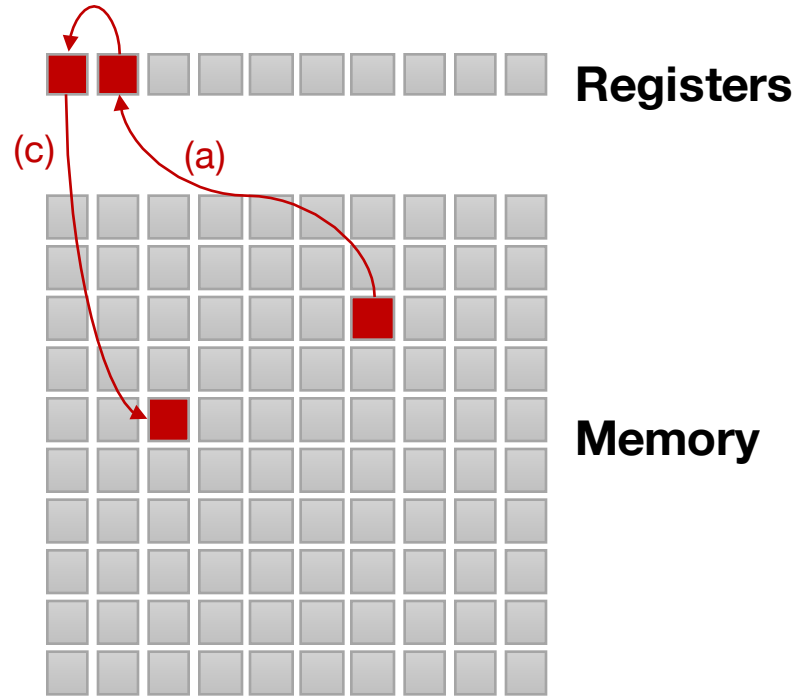
Fine-grained Taint Tracking

- Tags memory and registers with *taint flags*
  - Tainted: contains sensitive data
- Taints propagation
  - From source operand to destination operand

```
mov   r3, r7, LSR #12
ubfx  r9, r7, #8, #4
ldr   r1, [r5, r3 LSL #2]   (a)
ldr   r0, [r5, r9 LSL #2]
ldrh  r7, [r4, #2]!
mul   r0, r1, r0            (b)
and   r12, r7, #255
str   r0, [r5, r9 LSL #2]   (c)
add   pc, r8, r12, LSL #6
```

Integer multiplication in Java

High hardware-cost due to full register-level tracking

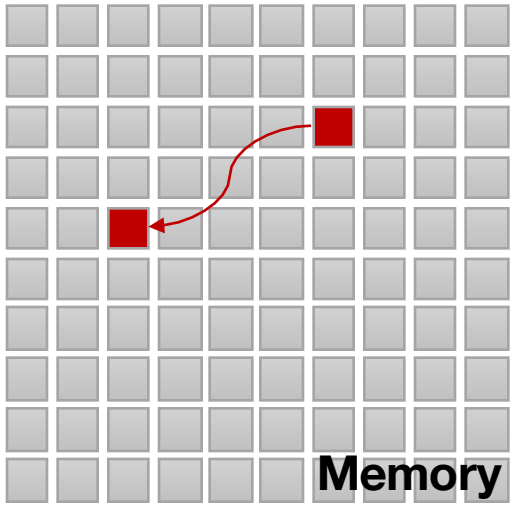


PIFT: Predictive Information Flow Tracking

- Not all instructions are tracked!
  - Monitors only memory load and store instructions
  - Because of “Load-Process-Store” structure
  - Propagate taints from sensitive load to close store(s)

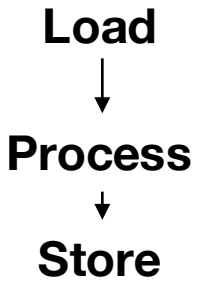
Load
↓
Process
↓
Store

```
mov   r3, r7, LSR #12
ubfx  r9, r7, #8, #4
ldr   r1, [r5, r3 LSL #2]
ldr   r0, [r5, r9 LSL #2]

str   r0, [r5, r9 LSL #2]
add   pc, r8, r12, LSL #6
```

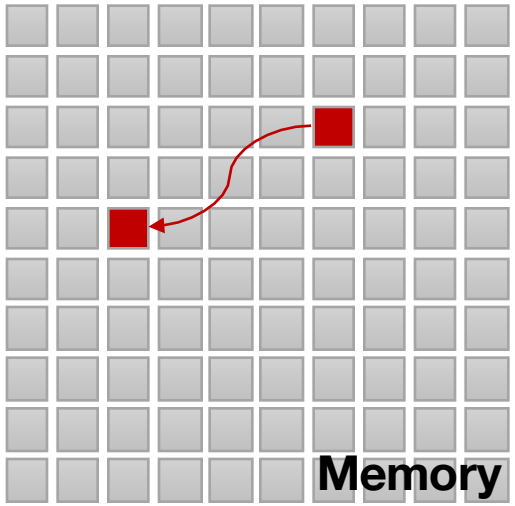


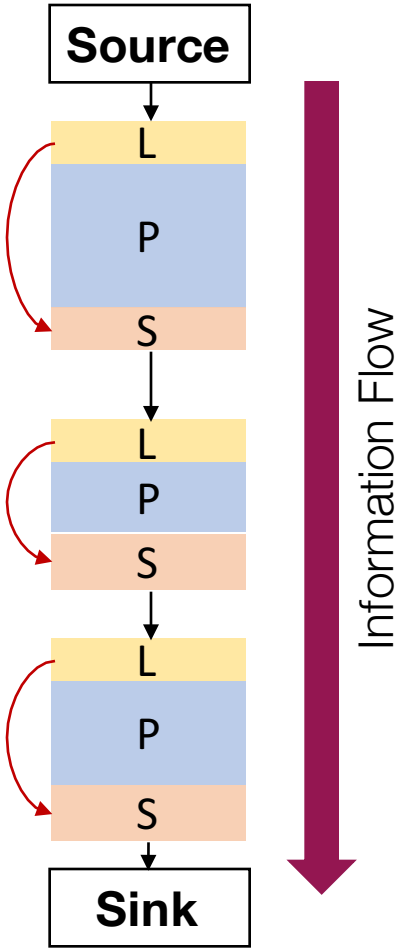
PIFT: Predictive Information Flow Tracking

- Not all instructions are tracked!
  - Monitors only memory load and store instructions
  - Because of "Load-Process-Store" structure
  - Propagate taints from sensitive load to close store(s)




```
mov   r3, r7, LSR #12
ubfx  r9, r7, #8, #4
ldr   r1, [r5, r3 LSL #2]
ldr   r0, [r5, r9 LSL #2]
str   r0, [r5, r9 LSL #2]
add   pc, r8, r12, LSL #6
```





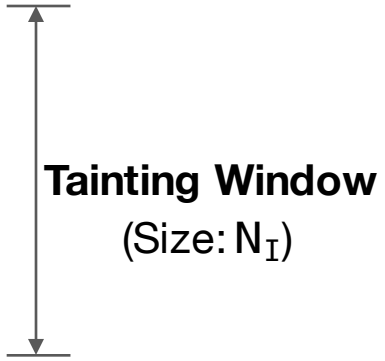
PIFT Taint-Propagation Algorithm

Time	Instruction
[k+0]	ldr reg_a, $addr_{L1}$ ⇢ **Tainted load**

PIFT Taint-Propagation Algorithm

Time **Instruction**

`[k+0] ldr reg`$_a$`, addr`$_{L1}$ ⇢ **Tainted load**



PIFT Taint-Propagation Algorithm

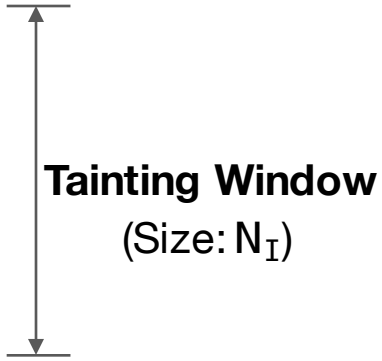
Time **Instruction**

`[k+0] ldr reg`$_a$`, addr`$_{L1}$ ⇢ **Tainted load**

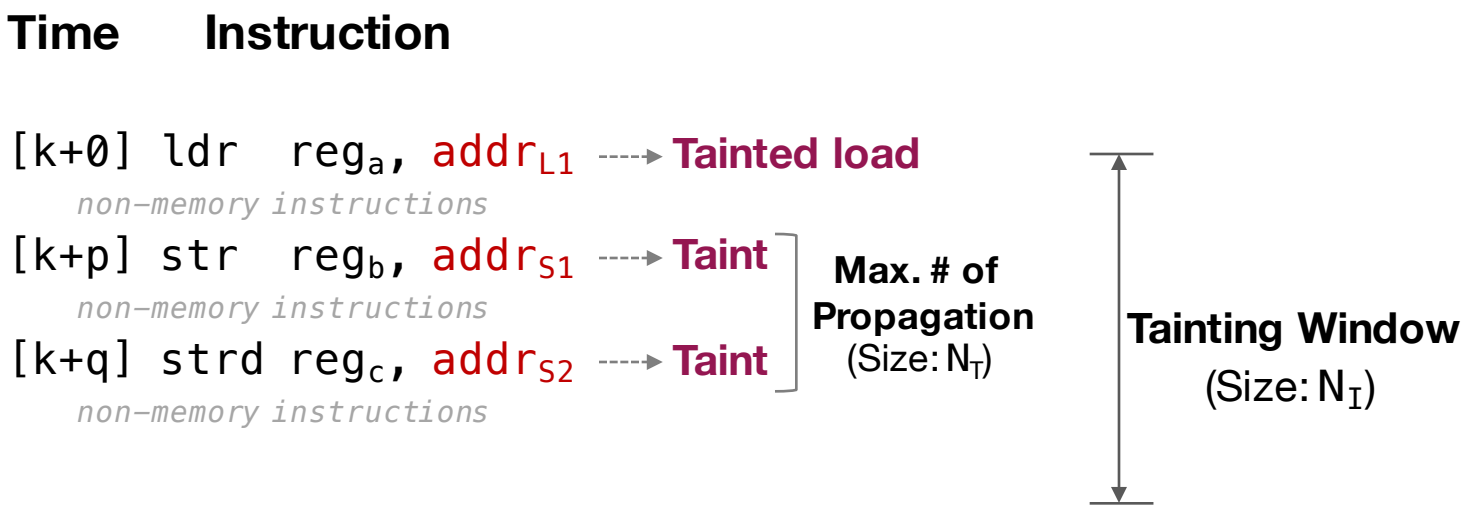
non-memory instructions

`[k+p] str reg`$_b$`, addr`$_{S1}$ ⇢ **Taint**

non-memory instructions

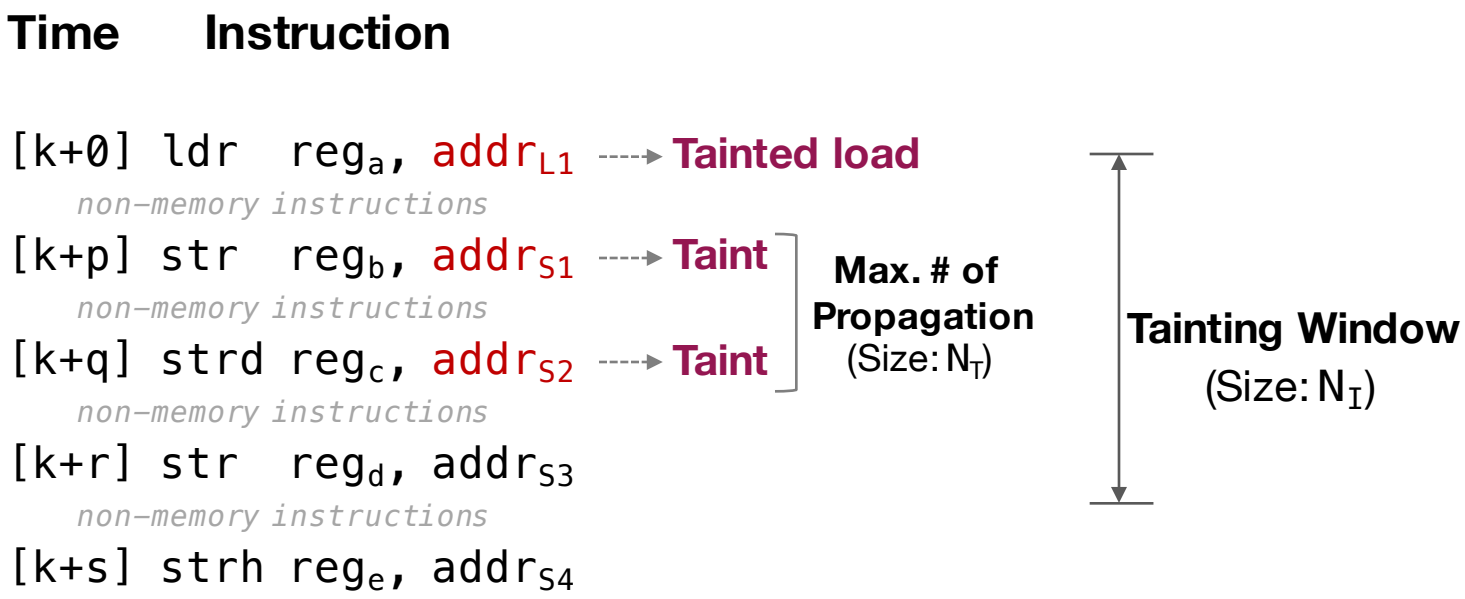


PIFT Taint-Propagation Algorithm

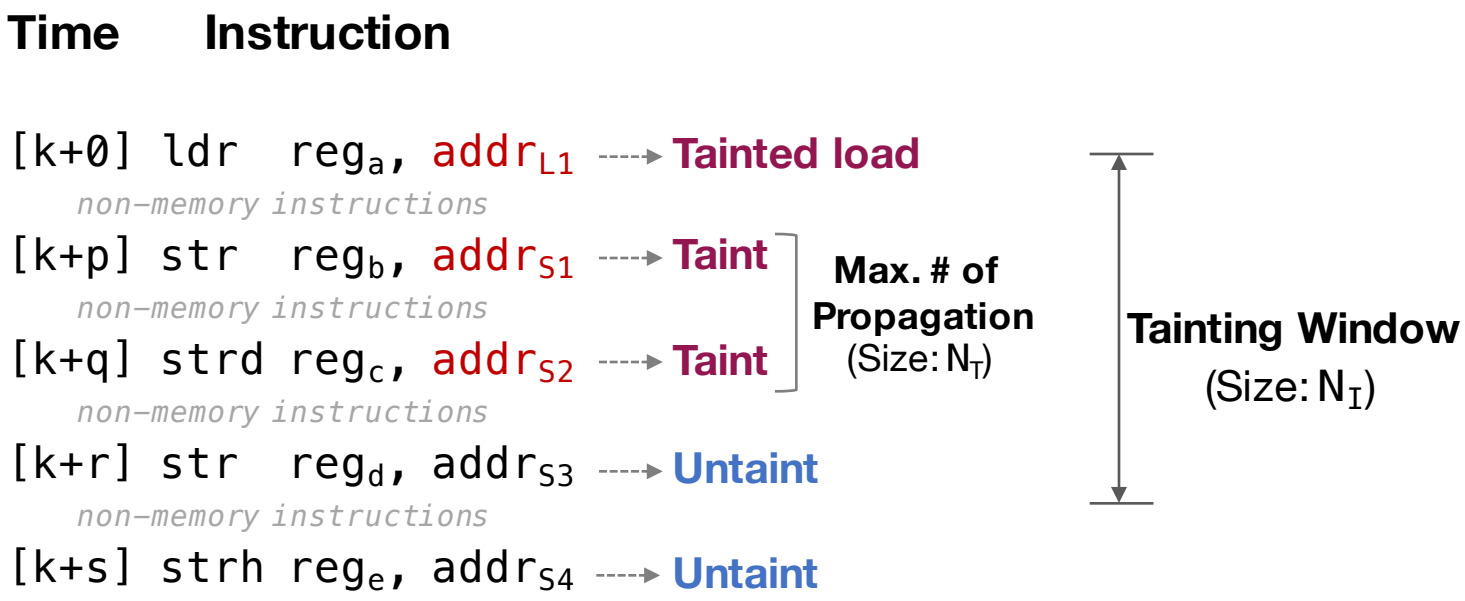


- Taints multiple stores
  - To increase the chance of tracking sensitive data flow
- But limit the number of propagations
  - To prevent taint explosion

PIFT Taint-Propagation Algorithm



PIFT Taint-Propagation Algorithm



- Untaint other stores if have been tainted
  - They are likely being overwritten by non-sensitive data
  - This further reduces taint region size

SW/HW Architecture

TelephonyManager (Source)

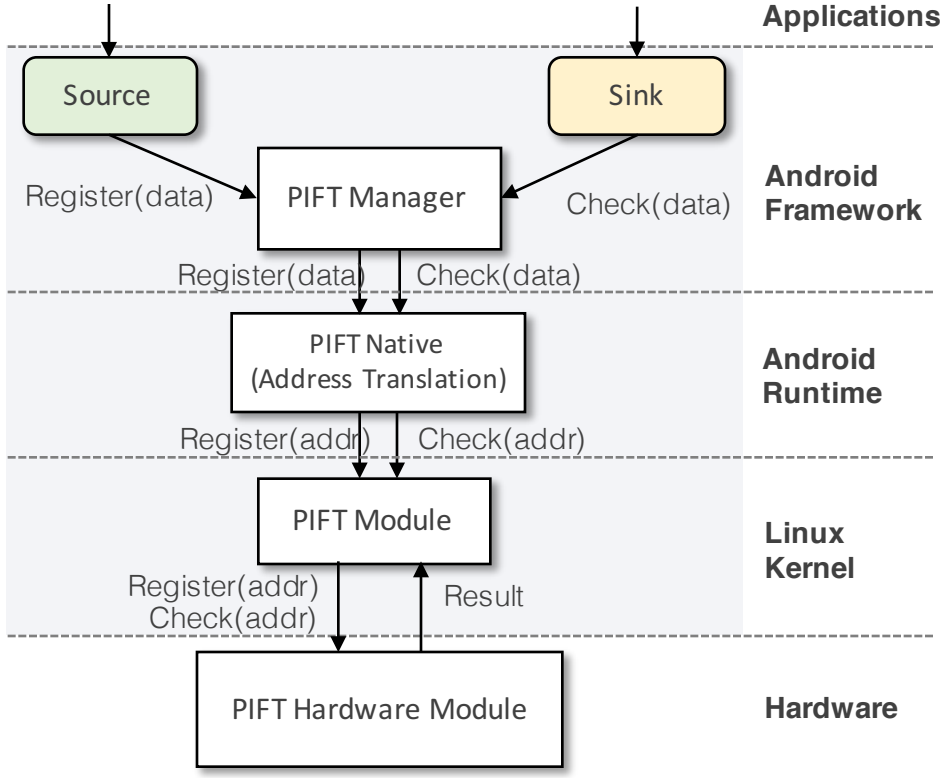
```
public String getDeviceId() {
    String IMEI = getSubscriberInfo().getDeviceId();
    PIFTManager.Register(IMEI);
    return IMEI;
}
```

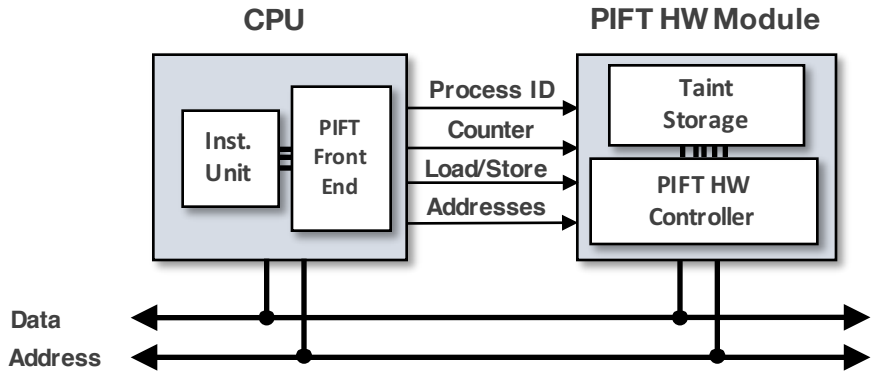
Application

```
String imei = telMan.getDeviceId();
...
String msg = "imei=" + imei + "&loc=" + ...
...
smsMan.sendTextMessage("+1217417xxxx", null, msg, null, null);
```

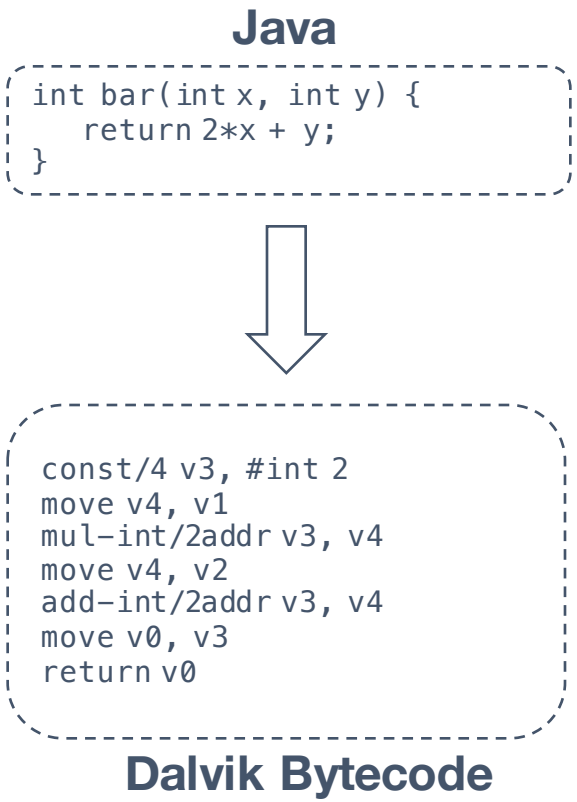
SmsManager (Sink)

```
public void sendTextMessage(..., String text, ...) {
    ...
    PIFTManager.Check(text);
    ...
}
```

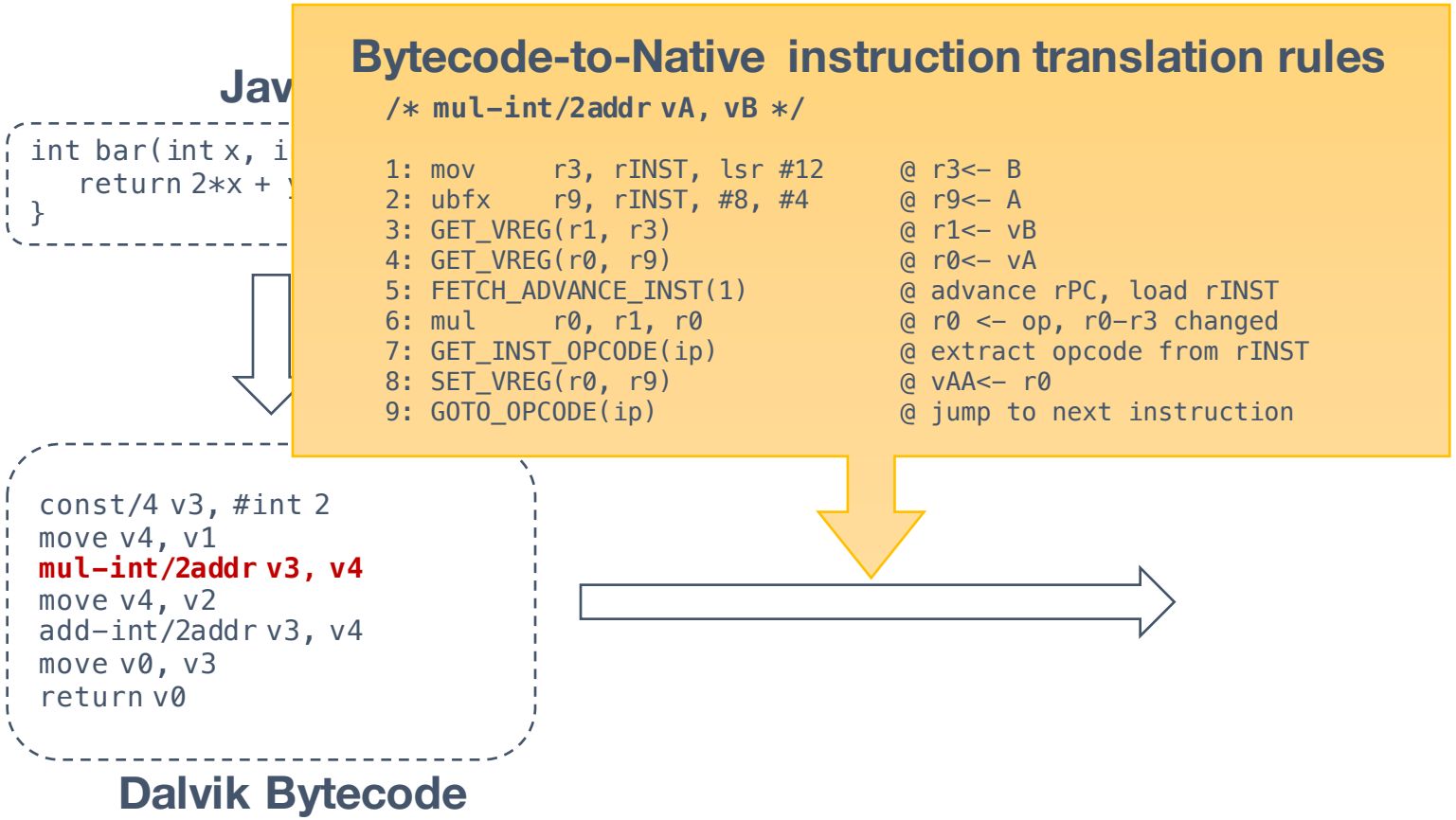




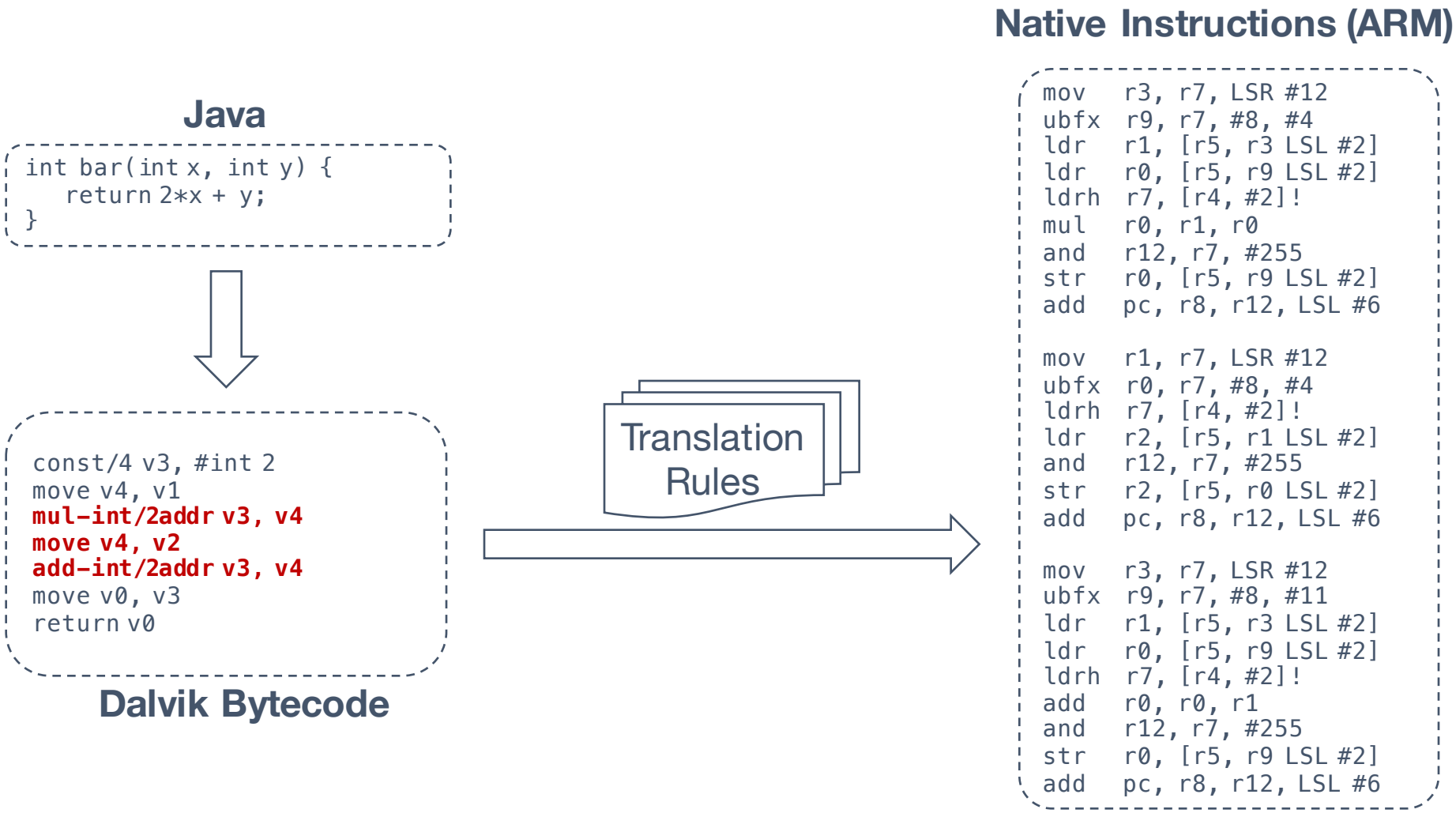
Java-Dalvik-Native Translation



Java-Dalvik-Native Translation



Java-Dalvik-Native Translation



Virtual Registers in Dalvik VM

- Operands are *virtual registers*

```
vA ← vA * vB
```

```
/* mul-int/2addr  vA, vB */

1: mov     r3, rINST, lsr #12     @ r3<- B
2: ubfx    r9, rINST, #8, #4      @ r9<- A
3: GET_VREG(r1, r3)               @ r1<- vB
4: GET_VREG(r0, r9)               @ r0<- vA
5: FETCH_ADVANCE_INST(1)          @ advance rPC, load rINST
6: mul     r0, r1, r0             @ r0 <- op, r0-r3 changed
7: GET_INST_OPCODE(ip)            @ extract opcode from rINST
8: SET_VREG(r0, r9)               @ vAA<- r0
9: GOTO_OPCODE(ip)                @ jump to next instruction
```

Virtual Registers in Dalvik VM

- Operands are *virtual registers* that reside on the *memory*

vA ← vA * vB

```
/* mul-int/2addr  vA, vB */

1: mov     r3, rINST, lsr #12     @ r3<- B
2: ubfx    r9, rINST, #8, #4      @ r9<- A
3: GET_VREG(r1, r3)               @ r1<- vB
4: GET_VREG(r0, r9)               @ r0<- vA
5: FETCH_ADVANCE_INST(1)          @ advance rPC, load rINST
6: mul     r0, r1, r0             @ r0 <- op, r0-r3 changed
7: GET_INST_OPCODE(ip)            @ extract opcode from rINST
8: SET_VREG(r0, r9)               @ vAA<- r0
9: GOTO_OPCODE(ip)                @ jump to next instruction
```

Loading operand

```
GET_VREG(_reg, _vreg)
   := ldr _reg, [rFP, _vreg, lsl #2]
```

Storing resultant

```
SET_VREG(_reg, _vreg)
   := str _reg, [rFP, _vreg, lsl #2]
```

Virtual Registers in Dalvik VM

- Operands are *virtual registers* that reside on the *memory*
- Fixed translation rules → Load-Store distance cannot be arbitrary

vA ← vA * vB

```
/* mul-int/2addr  vA, vB */

1: mov     r3, rINST, lsr #12     @ r3<- B
2: ubfx    r9, rINST, #8, #4      @ r9<- A
3: GET_VREG(r1, r3)               @ r1<- vB
4: GET_VREG(r0, r9)               @ r0<- vA
5: FETCH_ADVANCE_INST(1)          @ advance rPC, load rINST
6: mul     r0, r1, r0             @ r0 <- op, r0-r3 changed
7: GET_INST_OPCODE(ip)            @ extract opcode from rINST
8: SET_VREG(r0, r9)               @ vAA<- r0
9: GOTO_OPCODE(ip)                @ jump to next instruction
```

TW

Loading operand

```
GET_VREG(_reg, _vreg)
   := ldr _reg, [rFP, _vreg, lsl #2]
```

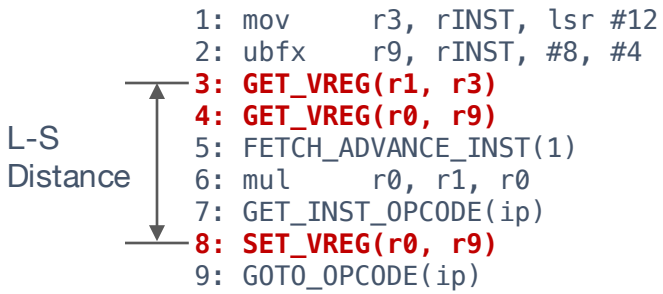
Storing resultant

```
SET_VREG(_reg, _vreg)
   := str _reg, [rFP, _vreg, lsl #2]
```

Can correctly propagate taint if Tainting Window size >= 5

Tainting Window Size

- Proper window size varies with bytecode



Load-Store Distance	Count	Example Bytecodes
1	3	return, return-wide, return-object
2	26	move-result, move/16, aget, aput, sput,iput-quick
3	19	move-object, sget-object, long-to-int, sget
4	11	iput-iput, neg-double, iget-quick, sget-volatile
5	46	iget, iget-object, int-to-long, add-int/lit8
6	21	int-to-char, sub-long, shl-int/lit8, iget-volatile
9-12	9	mul-long/2addr, aput-object, mul-long, shr-long
Unknown	47	double-to-int, rem-float, div-int/lit16

- Based on Android 4.2 on ARMv7-A
- 74 out of 256 do not move data.
- 10 out of 256 are unused.

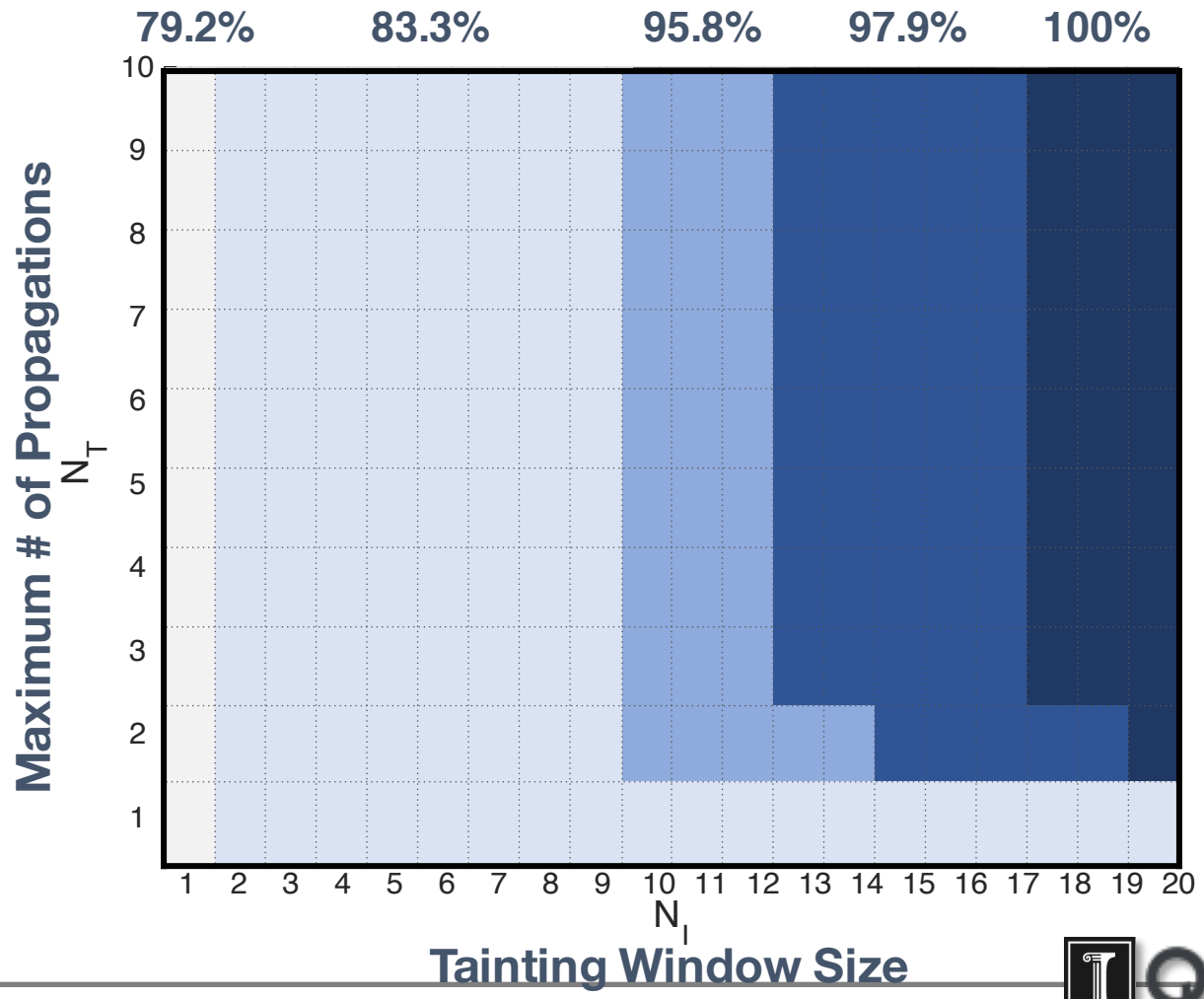
Evaluation Setup

- DroidBench 1.1
  - Sources
    - Device ID, serial number, phone number, GPS location
  - Sinks
    - SMS messages, HTTP queries, logging functions
  - Information flows through
    - arrays, lists, callbacks, exceptions, intents, method overriding, reflection, object inheritance
- Seven real-world malware apps
  - Sources
    - Device ID, phone number, GPS location
  - Sinks
    - SMS messages, HTTP queries
- Android 4.2 on gem5 simulator
  - Collected instruction streams

Accuracy

$$\text{Accuracy} = \frac{\text{True Positive} + \text{True Negative}}{\text{Total}}$$

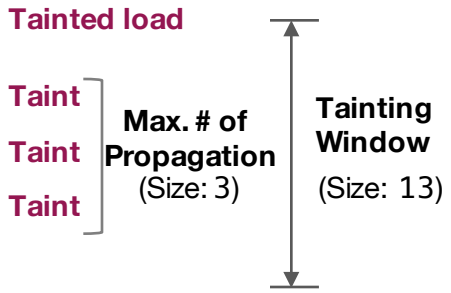
- DroidBench
  - 41 leaky and 16 benign apps

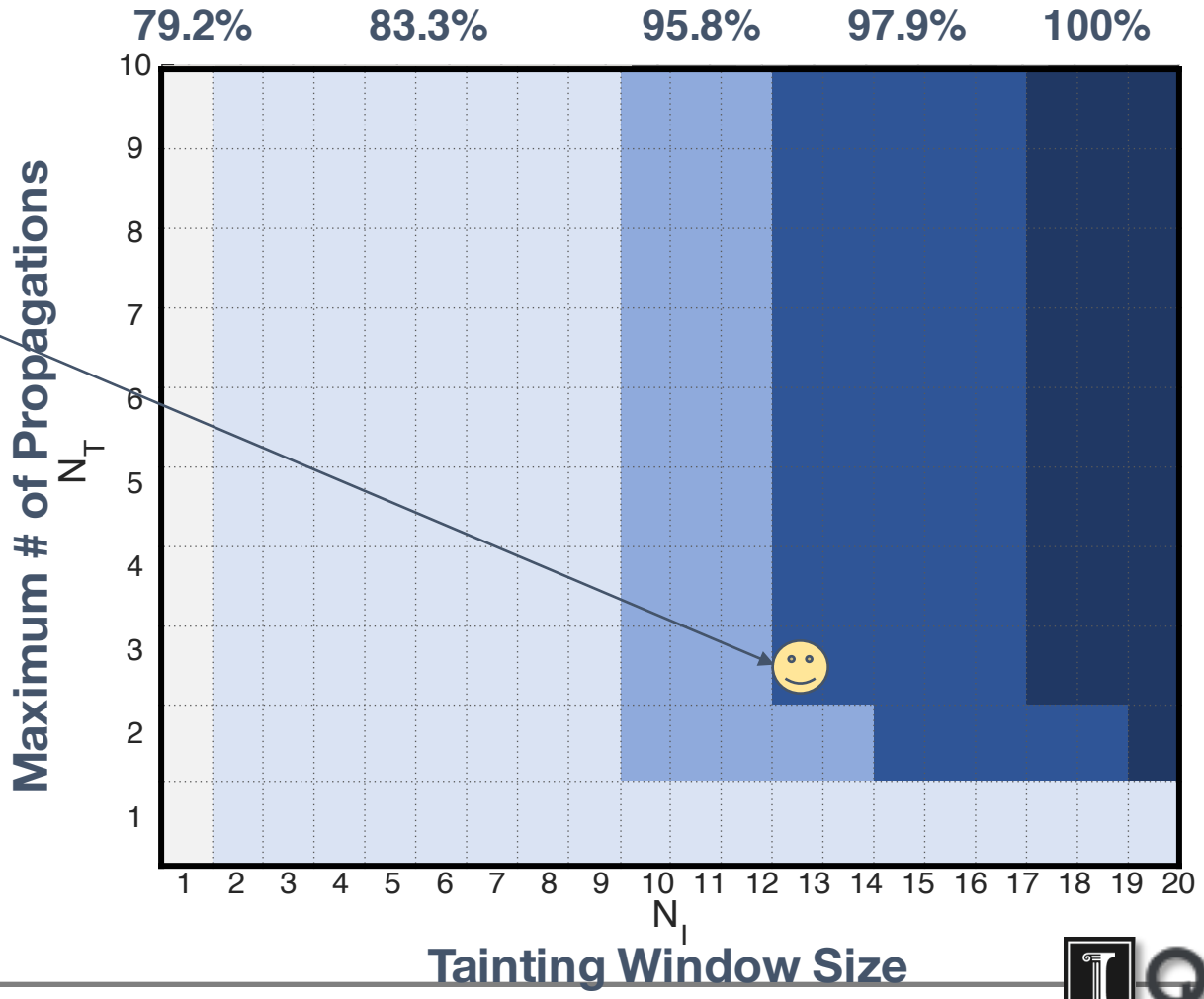


Accuracy

$$\text{Accuracy} = \frac{\text{True Positive} + \text{True Negative}}{\text{Total}}$$

- DroidBench
  - 41 leaky and 16 benign apps
  - 1 False Negative & 0 False Positive when

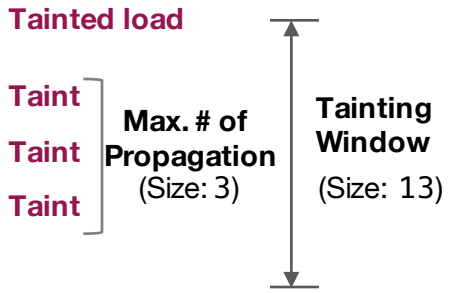




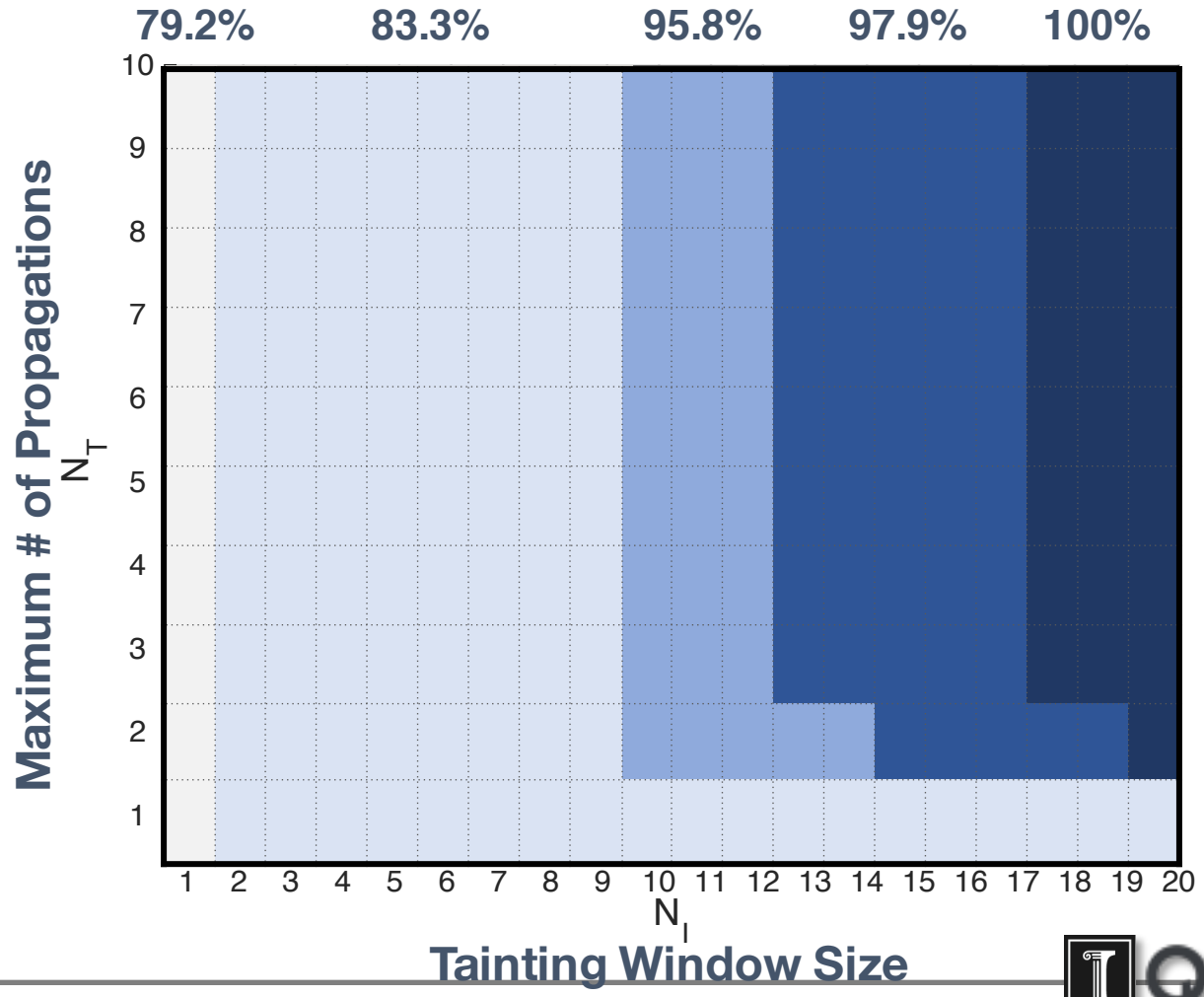
Accuracy

$$\text{Accuracy} = \frac{\text{True Positive} + \text{True Negative}}{\text{Total}}$$

- DroidBench
  - 41 leaky and 16 benign apps
  - 1 False Negative & 0 False Positive when



- Real-world malware
  - All are detected with N_I=3 and N_T=2
  - Most malware do not purposely evade PIFT (yet)

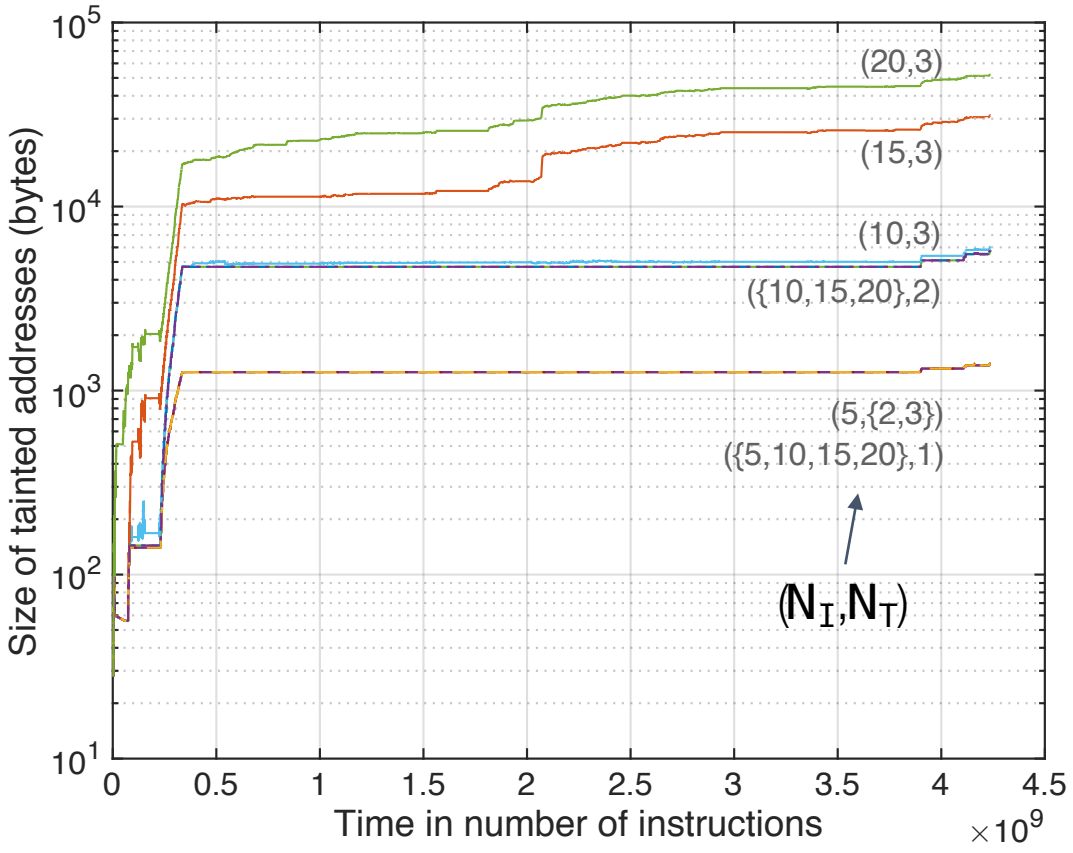


Overhead Evaluation

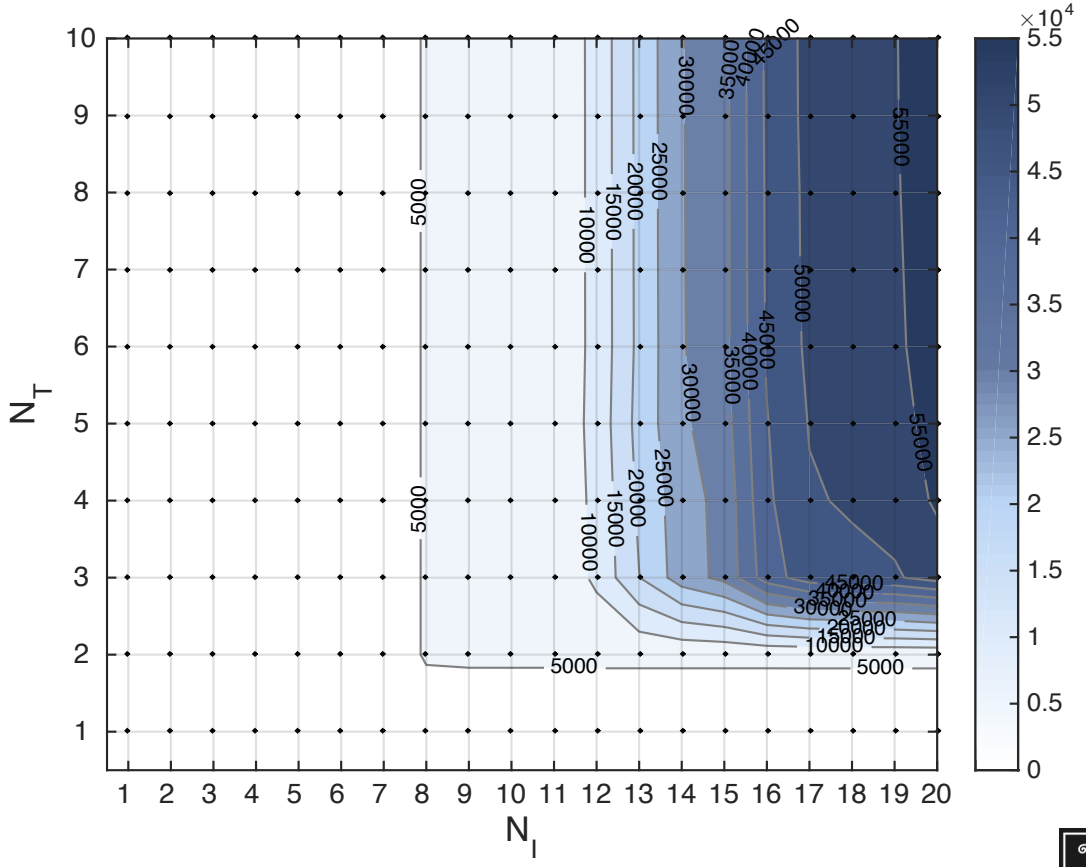
- Factors affecting run-time overhead
  - Size of tainted regions
  - Number of operations
- Detailed analysis on a real-world malware (LGRoot)

Overhead Evaluation

Size of tainted regions over time

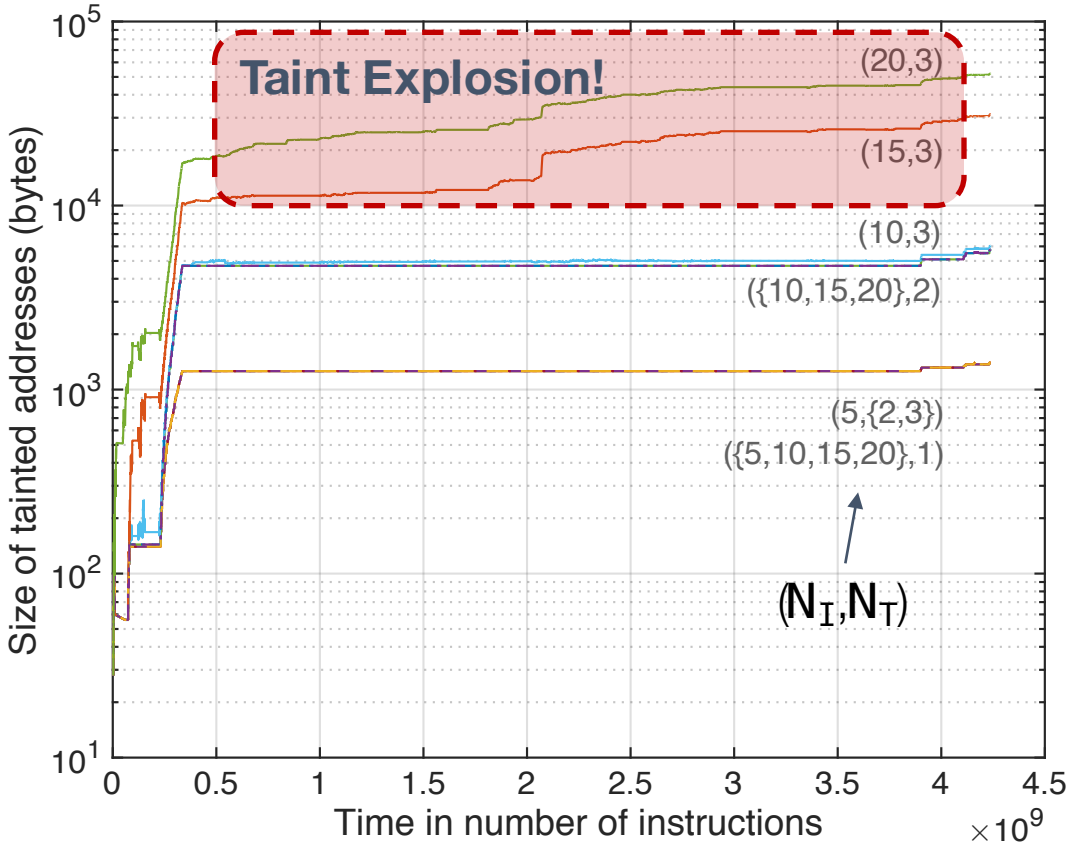


Maximum size of tainted regions

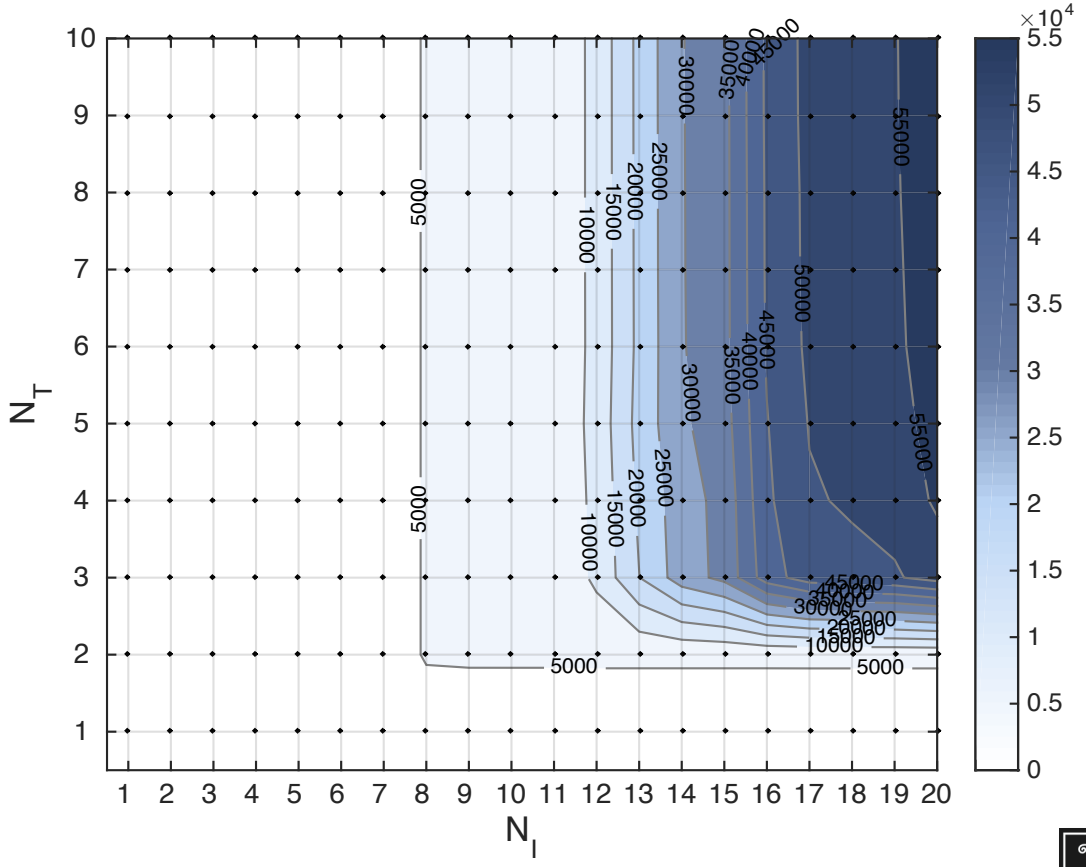


Overhead Evaluation

Size of tainted regions over time

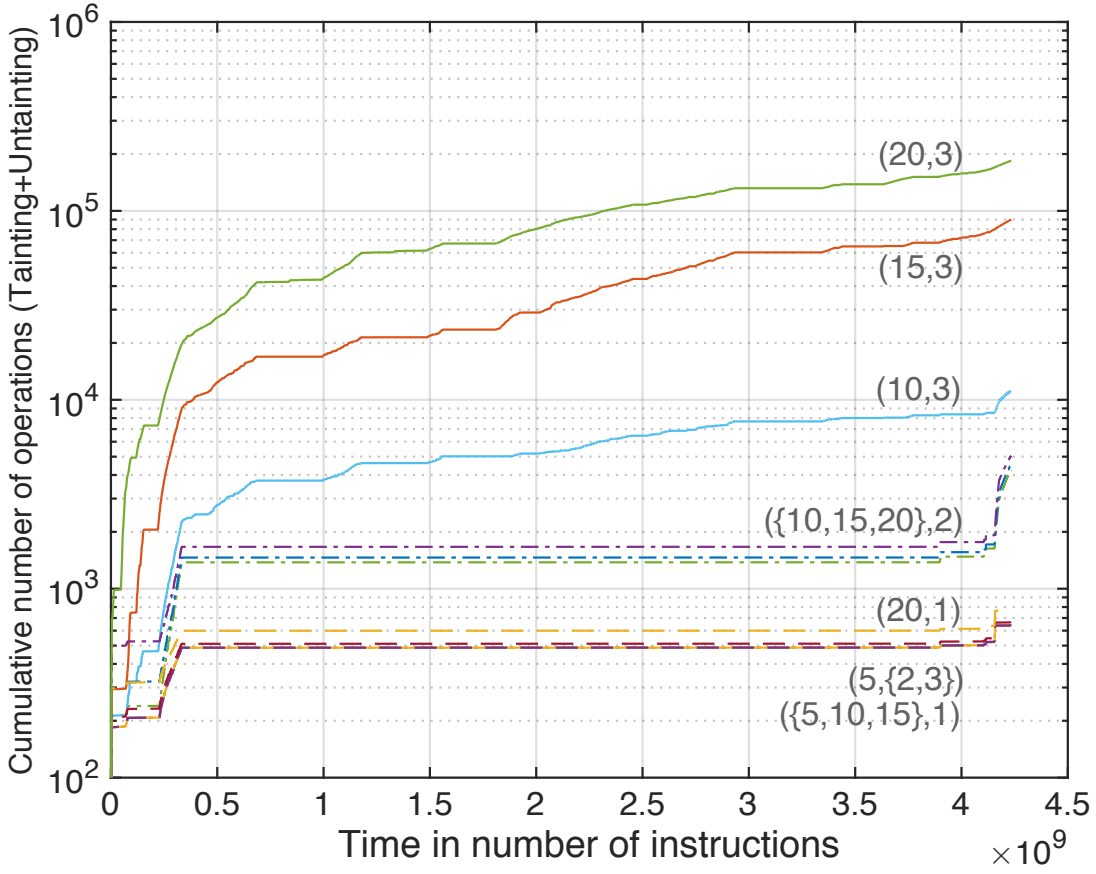


Maximum size of tainted regions



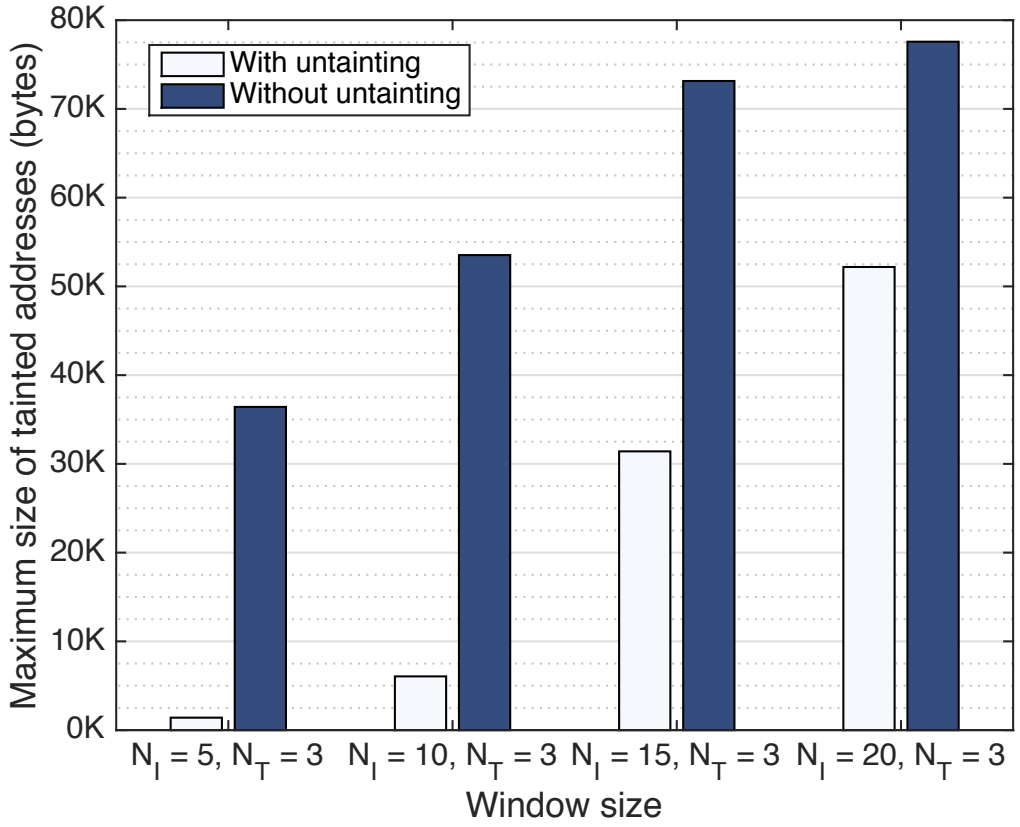
Overhead Evaluation

Number of operations (tainting + untainting) over time



Overhead Evaluation

Maximum size of tainted regions with/without untainting



Conclusion

- **PIFT**
  - Information flow tracking using only memory load-store instructions
  - Takes advantage of structural characteristics in Android apps and runtime
  - Accurate, yet lightweight

Conclusion

- **PIFT**
  - Information flow tracking using only memory load-store instructions
  - Takes advantage of structural characteristics in Android apps and runtime
  - Accurate, yet lightweight

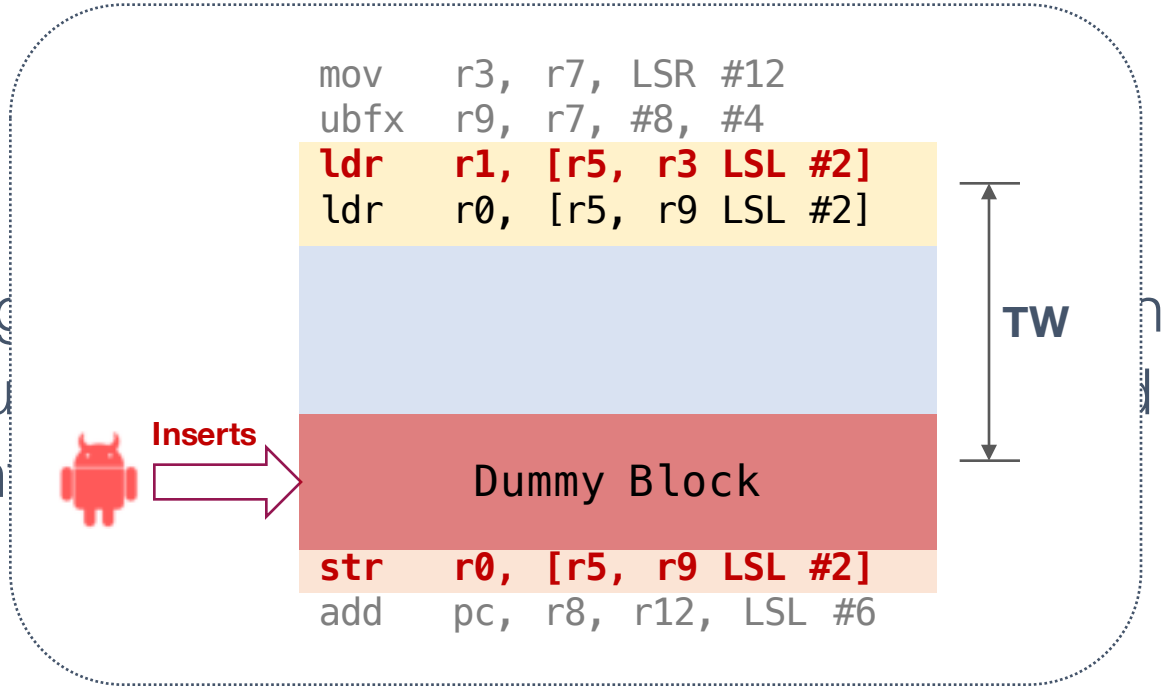
- **Future work**
  - Compiler support for preventing native code obfuscation
    - Detect/eliminate dummy code, relocate instructions, etc.

Conclusion

- **PIFT**
  - Information flow tracking
  - Takes advantage of stru
  - Accurate, yet lightweigh

ns

l runtime



- **Future work**
  - Compiler support for preventing native code obfuscation
    - Detect/eliminate dummy code, relocate instructions, etc.

Conclusion

- **PIFT**
  - Information flow tracking using only memory load-store instructions
  - Takes advantage of structural characteristics in Android apps and runtime
  - Accurate, yet lightweight

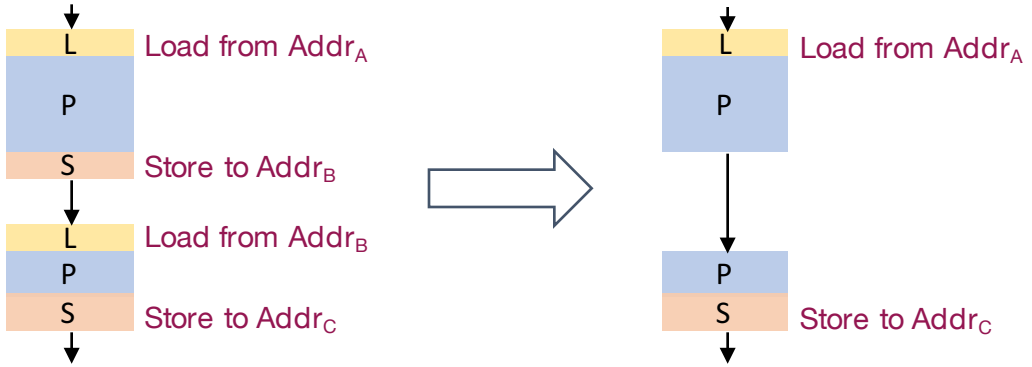
- **Future work**
  - Compiler support for preventing native code obfuscation
    - Detect/eliminate dummy code, relocate instructions, etc.
  - Large-scale experiments
    - Find optimal tainting window size for different data type

Thank you

Backups

Discussion – Impact of JIT and ART

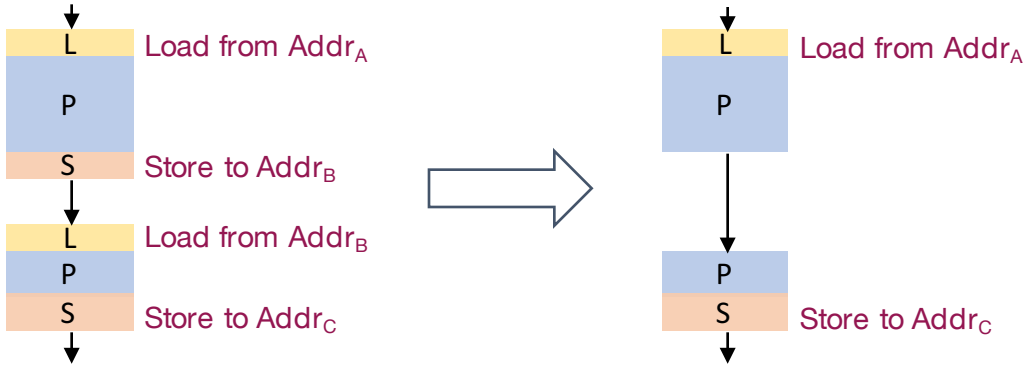
- JIT (Just-In-Time) optimization might eliminate or reorder some load/store instructions



  - Only on frequently executed parts
  - All experiments were done with JIT enabled and no accuracy drop occurred

Discussion – Impact of JIT and ART

- JIT (Just-In-Time) optimization might eliminate or reorder some load/store instructions



  - Only on frequently executed parts
  - All experiments were done with JIT enabled and no accuracy drop occurred
- AOT (Ahead-Of-Time) of ART
  - Not tested (because not supported by the simulator)
  - ART's optimizations are a limited subset of JIT optimizations

Discussion – Implicit Flows

- PIFT does not directly address implicit flows
- But, some cases can be detected
  - Example: ImplicitFlows_ImplicitFlow1 app in DroidBench

```
for(char c : imei.toCharArray()){
    switch(c){
        case '0' : result += 'a'; break;
        case '1' : result += 'b'; break;
        case '2' : result += 'c'; break;
        ...
```

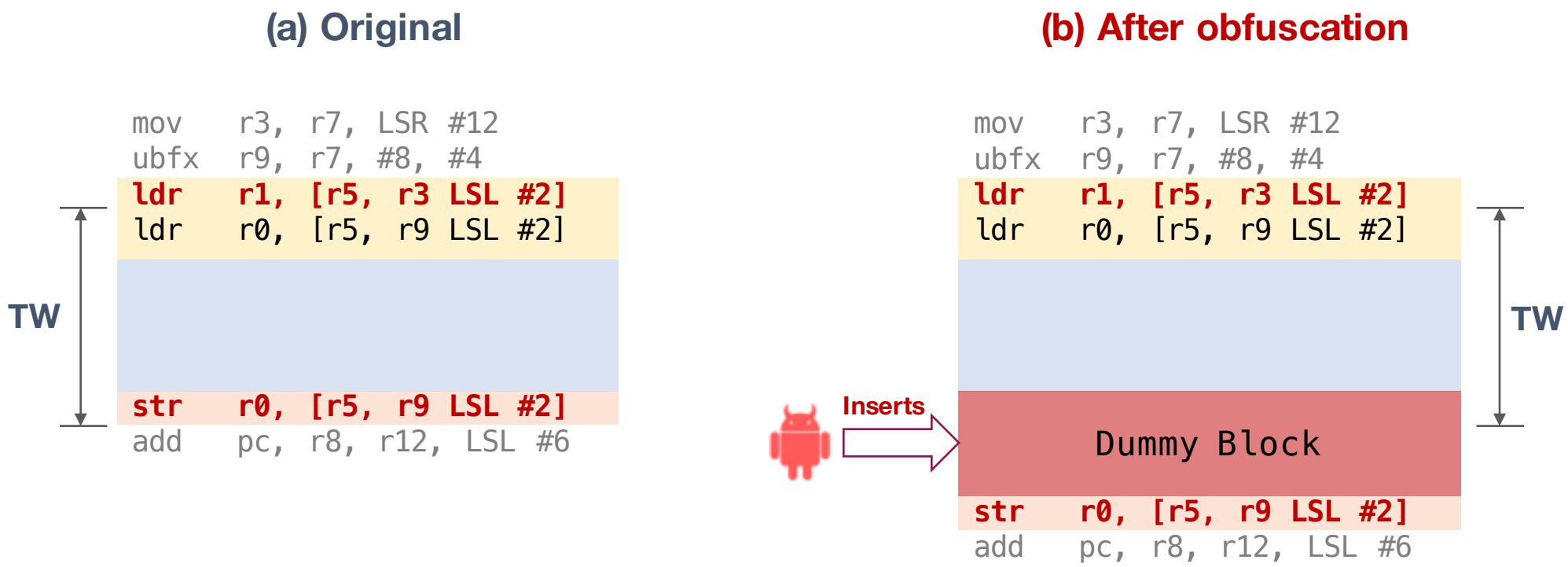
Reading `c` is a tainted load

Writing to `result` is close enough

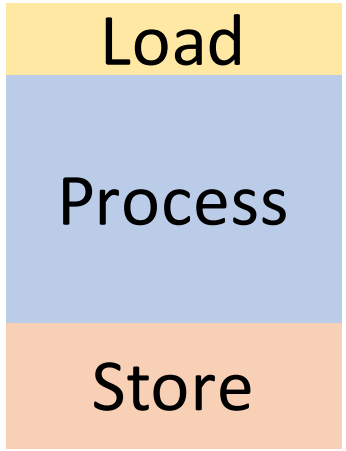
(In terms of # of native instructions)

Discussion – Native Code Obfuscation

- PIFT can be circumvented by inserting a long, dummy block of native instructions between load and stores

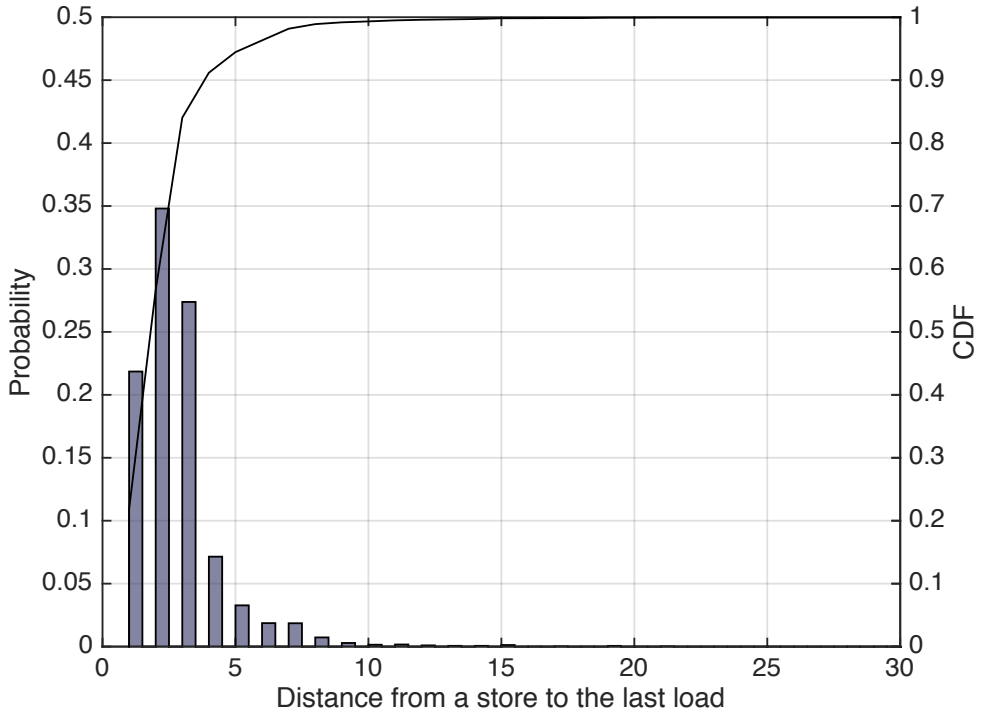


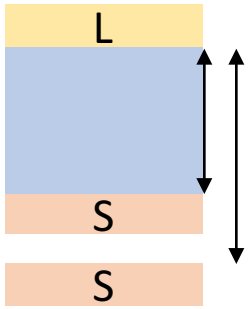
Load-Store Characteristics



Load-Store Characteristics

- Analysis of instruction stream made by a real-world malware (LGRoot) on gem5
  - 2.2 million load and 0.8 million store instructions



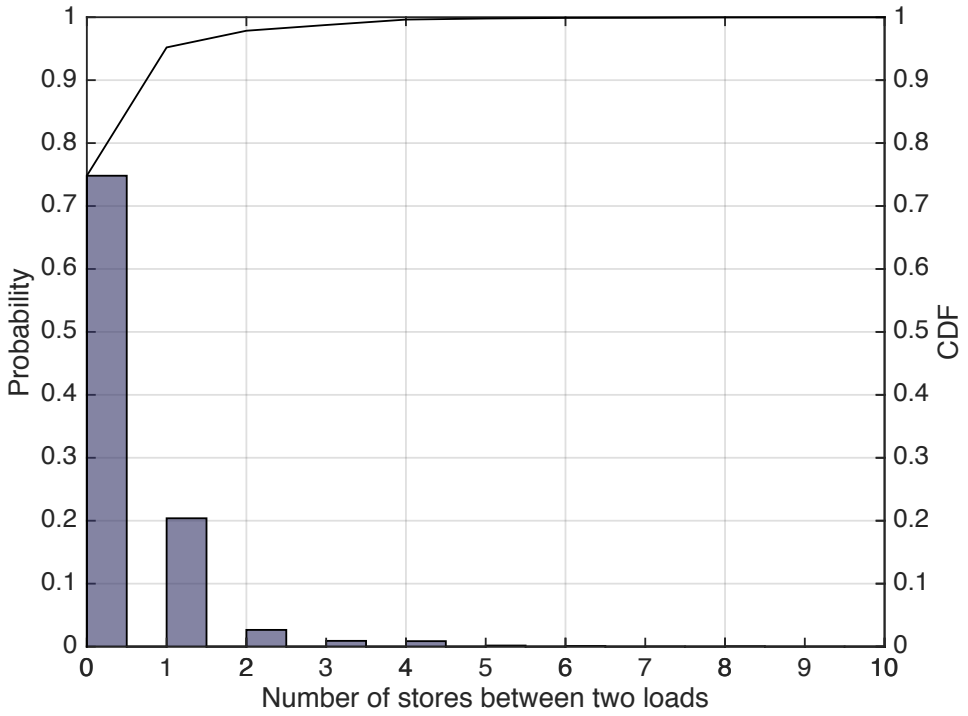


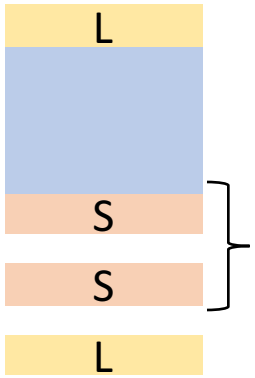
Stores are close to loads.

A limited monitoring window after each load would suffice.

Load-Store Characteristics

- Analysis of instruction stream made by a real-world malware (LGRoot) on gem5
  - 2.2 million load and 0.8 million store instructions





The number of stores after a load is limited.

Monitoring a few stores after each load would suffice.

Tainting Window Size

- Top 30 bytecodes in the number of appearances in the dex files of
  - Android stock applications
    - Browser, Contacts, KeyChain, Email, Gallery, Phone, Calendar, …
  - Android system libraries
    - Core, framework, services
- Most bytecodes have a short load-store distance

Dalvik Bytecode	%	L-S Distance
invoke-virtual	11.06%	
move-result-object	8.98%	2
iget-object	7.10%	5
const/4	5.19%	
const-string	4.85%	
invoke-static	4.45%	
move-result	4.42%	2
invoke-direct	4.31%	
return-void	3.19%	
goto	3.10%	
invoke-interface	3.04%	
const/16	2.82%	
if-eqz	2.82%	
return-object	2.79%	1
aput-object	2.50%	10
new-instance	2.36%	
iput-object	1.97%	5
move-object/from16	1.84%	2
return	1.68%	1
iget	1.46%	5
if-nez	1.40%	
check-cast	1.31%	
sget-object	1.09%	3
add-int/lit8	0.80%	5
iput	0.74%	4
move	0.68%	3
move/from16	0.65%	2
throw	0.64%	
const	0.60%	
move-object	0.53%	3

Stock apps (1.2 M lines)

Dalvik Bytecode	%	L-S Distance
invoke-virtual	12.57%	
iget-object	7.51%	5
move-result-object	7.46%	2
const/4	5.64%	
invoke-direct	4.57%	
move-result	4.16%	2
const-string	3.84%	
invoke-static	3.59%	
goto	3.30%	
if-eqz	3.26%	
move-object/from16	3.22%	2
return-void	2.83%	
iget	2.60%	5
new-instance	2.57%	
iput-object	1.76%	5
if-nez	1.61%	
invoke-interface	1.57%	
const/16	1.50%	
return-object	1.44%	1
throw	1.30%	
iput	1.27%	4
return	1.17%	1
move/from16	1.13%	2
move-exception	1.12%	
add-int/lit8	0.96%	5
check-cast	0.95%	
sget-object	0.91%	3
monitor-exit	0.82%	
invoke-virtual/range	0.74%	
move	0.74%	3

System libraries (1.3 M lines)

Taint Propagations

v3	v4
2	key

```
void foo() {
   ...
   int result = bar(key,456);
   ...
}
```

Java

```
int bar(int x, int y) {
   return 2*x + y;
}
```

Java

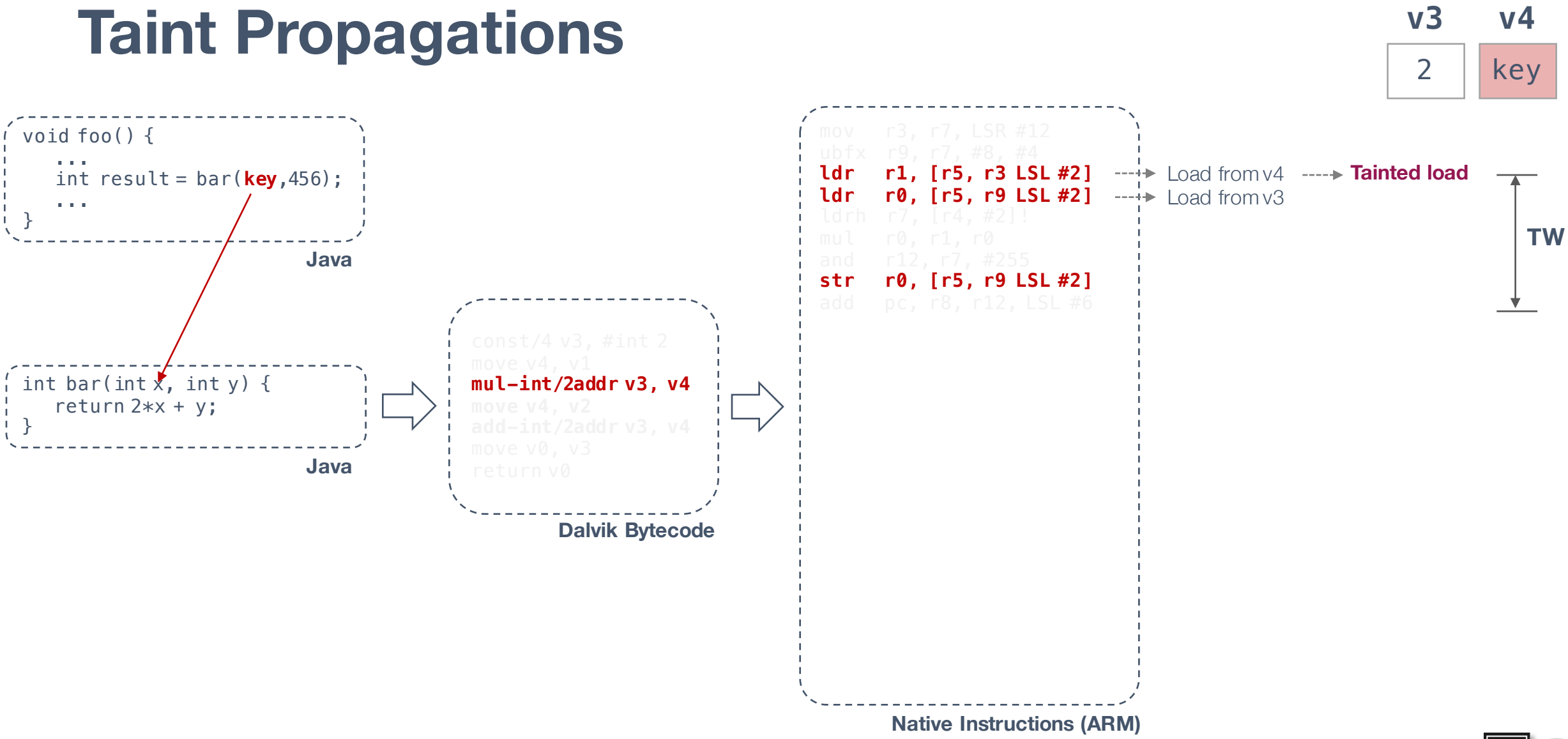
```
const/4 v3, #int 2
move v4, v1
mul-int/2addr v3, v4
move v4, v2
add-int/2addr v3, v4
move v0, v3
return v0
```

Dalvik Bytecode

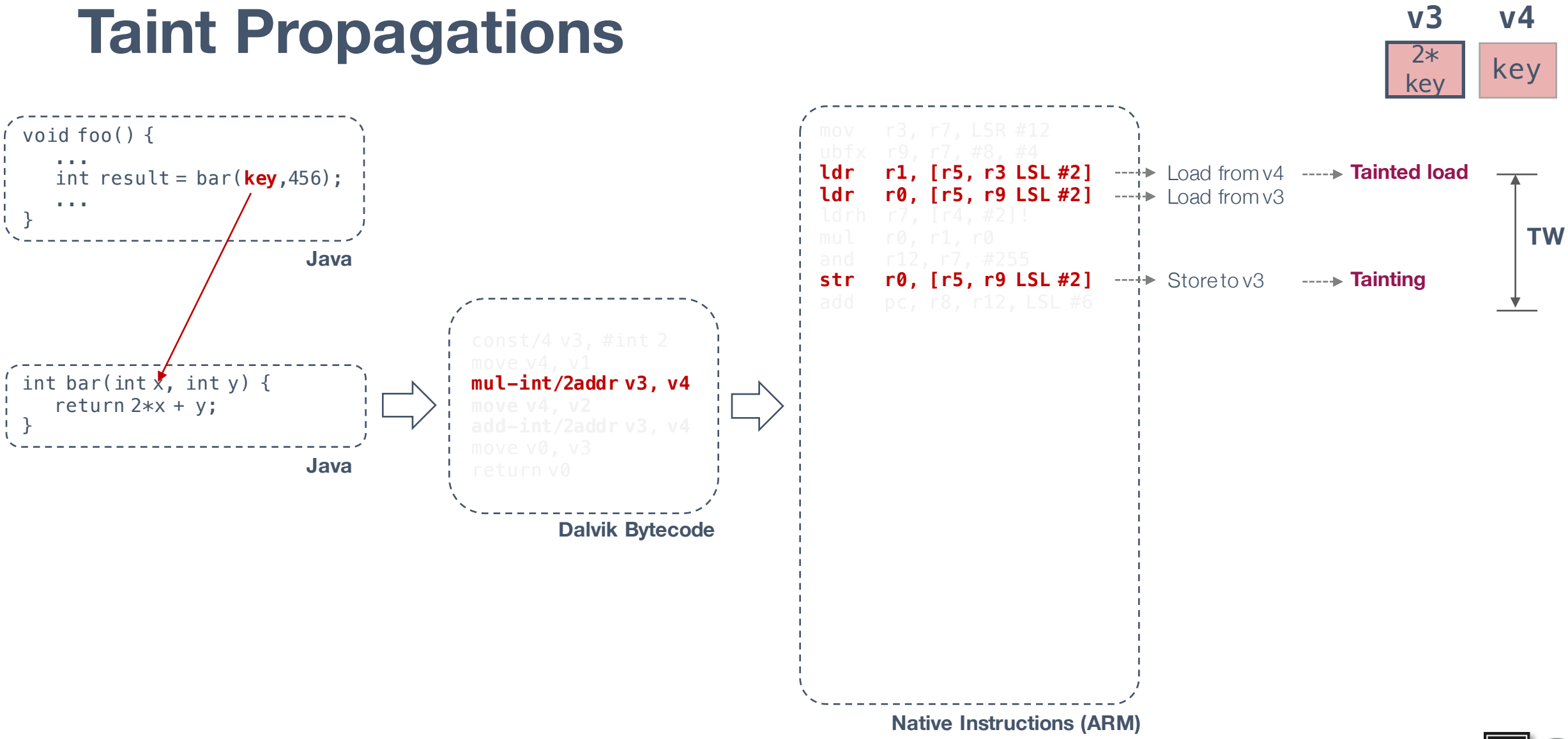
```
mov   r3, r7, LSR #12
ubfx  r9, r7, #8, #4
ldr   r1, [r5, r3 LSL #2]
ldr   r0, [r5, r9 LSL #2]
ldrh  r7, [r4, #2]!
mul   r0, r1, r0
and   r12, r7, #255
str   r0, [r5, r9 LSL #2]
add   pc, r8, r12, LSL #6
```

Native Instructions (ARM)

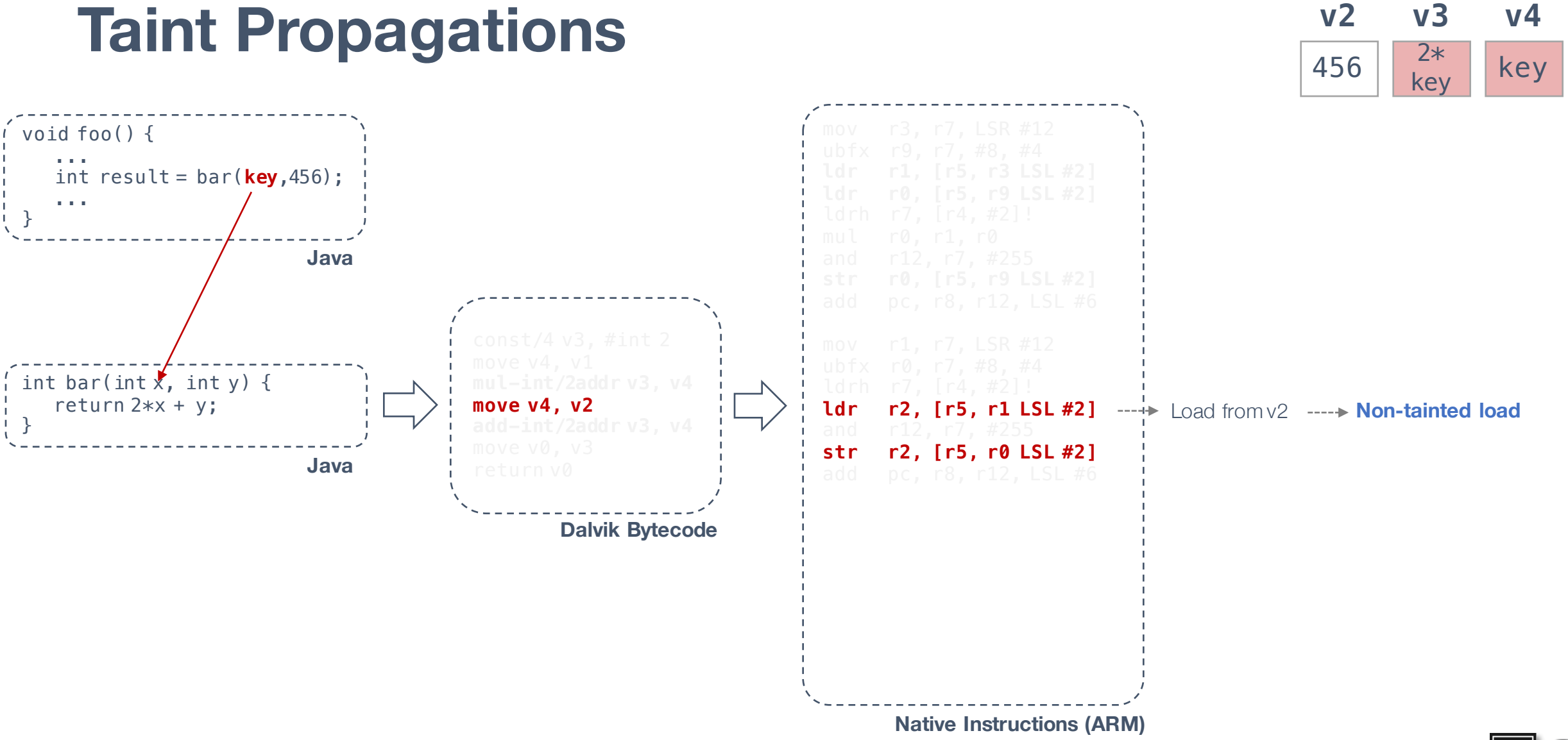
Taint Propagations



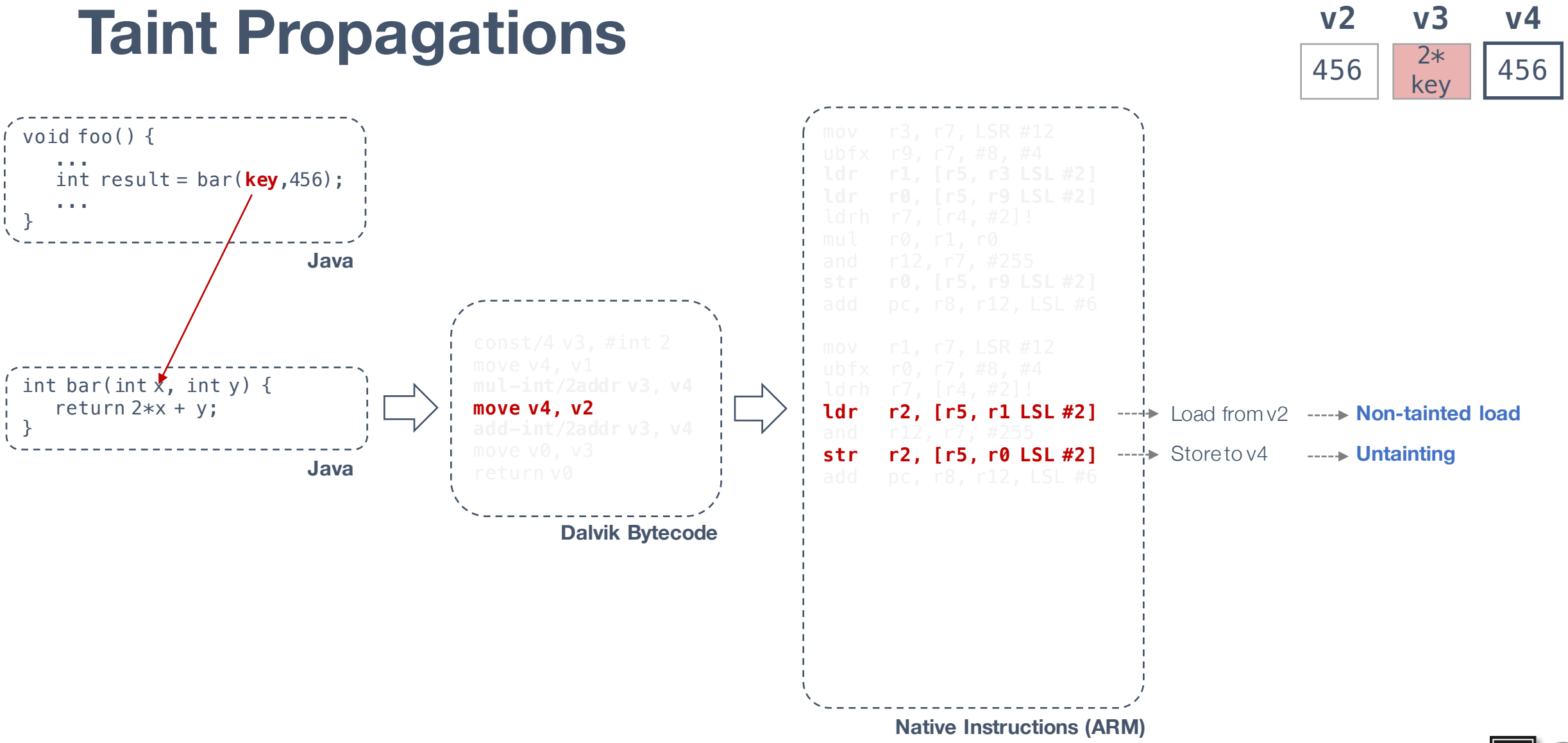
Taint Propagations



Taint Propagations



Taint Propagations



Taint Propagations

v2	v3	v4
456	2* key	456

```
void foo() {
   ...
   int result = bar(key,456);
   ...
}
```

Java

```
int bar(int x, int y) {
   return 2*x + y;
}
```

Java

```
const/4 v3, #int 2
move v4, v1
mul-int/2addr v3, v4
move v4, v2
add-int/2addr v3, v4
move v0, v3
return v0
```

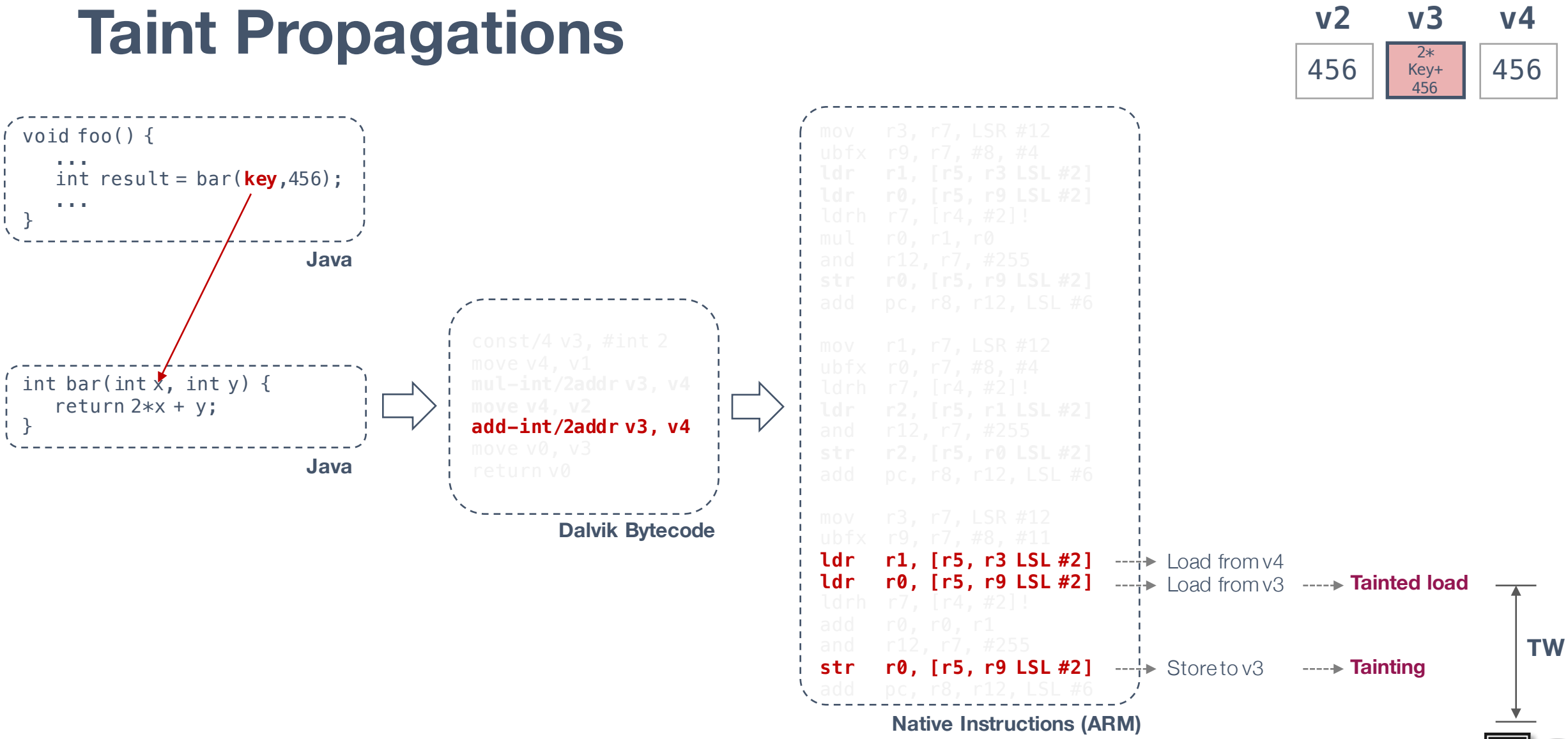
Dalvik Bytecode

```
mov   r3, r7, LSR #12
ubfx  r9, r7, #8, #4
ldr   r1, [r5, r3 LSL #2]
ldr   r0, [r5, r9 LSL #2]
ldrh  r7, [r4, #2]!
mul   r0, r1, r0
and   r12, r7, #255
str   r0, [r5, r9 LSL #2]
add   pc, r8, r12, LSL #6

mov   r1, r7, LSR #12
ubfx  r0, r7, #8, #4
ldrh  r7, [r4, #2]!
ldr   r2, [r5, r1 LSL #2]
and   r12, r7, #255
str   r2, [r5, r0 LSL #2]
add   pc, r8, r12, LSL #6

mov   r3, r7, LSR #12
ubfx  r9, r7, #8, #11
ldr   r1, [r5, r3 LSL #2]   ----> Load from v4
ldr   r0, [r5, r9 LSL #2]   ----> Load from v3   ----> Tainted load
ldrh  r7, [r4, #2]!
add   r0, r0, r1
and   r12, r7, #255
str   r0, [r5, r9 LSL #2]
add   pc, r8, r12, LSL #6
```

Native Instructions (ARM)

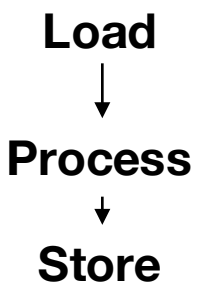
TW

Taint Propagations

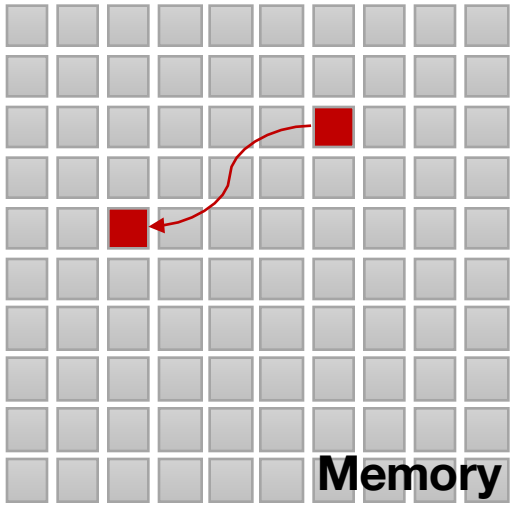


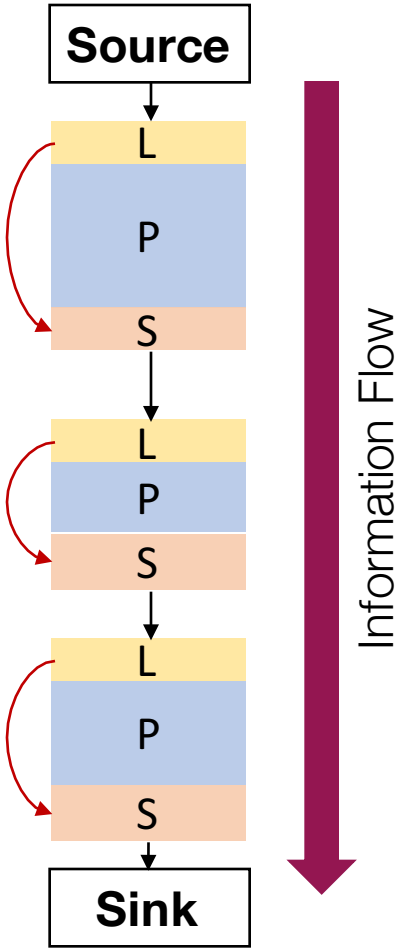
PIFT: Predictive Information Flow Tracking

- Why does it work for Android-on-ARM?
  - ARM: Load-store architecture
  - Android: fixed bytecode-to-native translation rules
    - That is, fixed “Load-Process-Store” structures

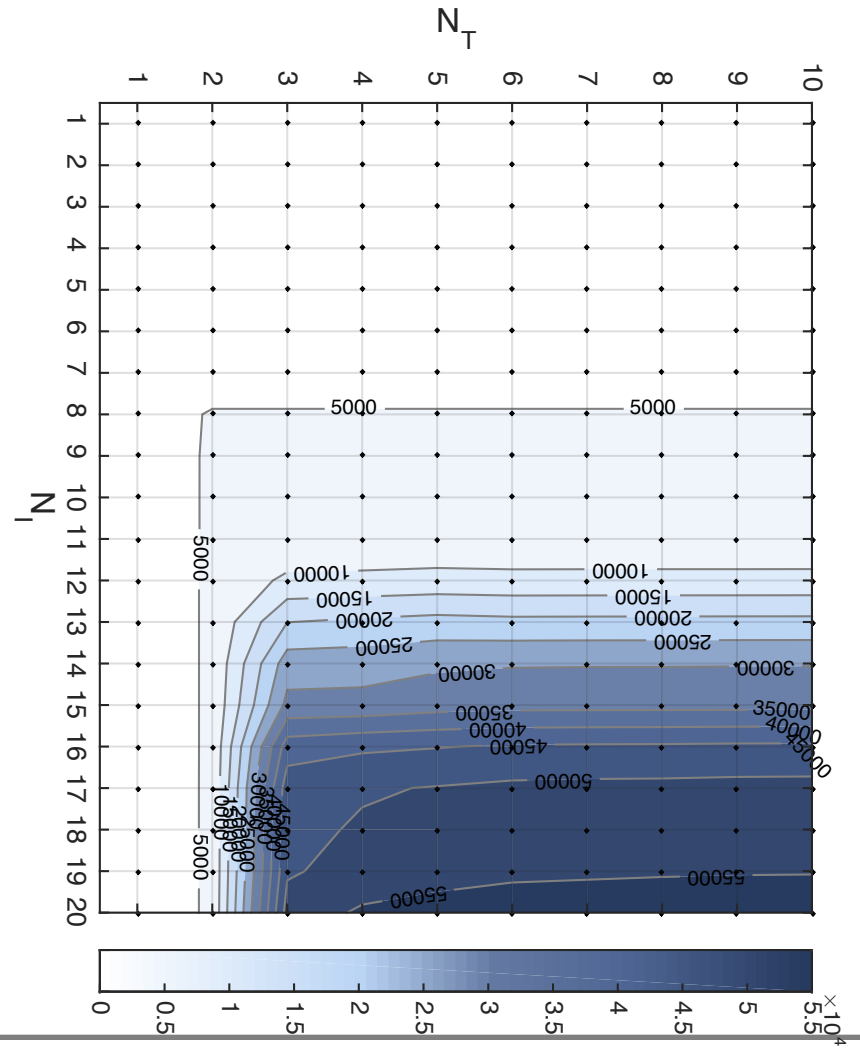



```
mov   r3, r7, LSR #12
ubfx  r9, r7, #8, #4
ldr   r1, [r5, r3 LSL #2]
ldr   r0, [r5, r9 LSL #2]
str   r0, [r5, r9 LSL #2]
add   pc, r8, r12, LSL #6
```





Maximum Size of Tainted Regions



N_I \ N_T	1	2	3	4	5	6	7	8	9	10
1	28									
2	28	28								
3	28	28	28							
4	1012	1012	1012	1012						
5	1404	1404	1404	1404	1404					
6	1404	1418	1418	1418	1418	1418				
7	1404	1418	1418	1418	1418	1418	1418			
8	1404	5552	5552	5552	5552	5552	5552	5552		
9	1404	5736	5975	5975	5975	5975	5975	5975	5975	
10	1404	5738	6035	6035	6035	6035	6035	6035	6035	6035
11	1404	5738	6049	6049	6049	6049	6049	6049	6049	6049
12	1404	5774	10999	11249	11700	11446	11467	11475	11475	11491
13	1404	5774	19924	20881	21701	21237	21283	21307	21335	21355
14	1404	5774	27561	27825	29139	29610	29674	29690	29726	29754
15	1404	5774	31411	31632	32951	33441	33495	33523	33555	33595
16	1404	5774	42624	44128	44789	45503	45658	45730	45915	45971
17	1404	5782	47705	49519	50252	51011	51212	51283	51504	51564
18	1404	5782	48633	50558	51409	52329	52545	52626	52848	52916
19	1404	5782	49393	52000	52936	53794	54065	54260	54478	54570
20	1404	5782	52140	55722	57095	58199	58578	59503	59821	59925