UNIVERSITÀ DI PISA

Secure Information Flow in Java bytecode

FMSS 2019-2020

Java bytecode: a simple instruction set

`op`	pop two operands off the stack, perform the operation, and push the result onto the stack
`pop`	discard the top value from the stack
`push k`	push the constant k onto the stack
`load x`	push the value of variable x onto the stack
`store x`	pop off the stack and store the value into x
`if j`	pop off the stack and jump to j if non-zero
`goto j`	jump to j
`jsr j`	at address p, jump to address j and push p+1 onto the operand stack
`ret x`	jump to the address stored in x
`halt`	stop

Standard operational semantics

Constants	V	k, k', ..
Addresses	A	i, j, ..
Memories	$Mem = var \rightarrow V \cup A$	m, m', ..
Stacks	$Stack = (V \cup A)^*$	s, s', ..

Transition system:

A state consists of: <program counter, memory, operand stack>

State = (A x Mem x Stack)
Transition: rule of the standard operational semantics

Standard Operational Semantics

x: 5
y: 1

state: <program counter, memory, operand stack>

$$\langle PC, [MEM(x)\ MEM(y)], STACK \rangle$$

```
0 load y
1 if 4
2 push 1
3 goto 5
4 push 0
5 store x
6 halt
```

$$\langle 0, [5, 1], \lambda \rangle$$
$$\downarrow_{\textbf{load}}$$
$$\langle 1, [5, 1], 1 \rangle$$
$$\downarrow_{\textbf{if}_{\textbf{true}}}$$
$$\langle 4, [5, 1], \lambda \rangle$$
$$\downarrow_{\textbf{push}}$$
$$\langle 5, [5, 1], 0 \rangle$$
$$\downarrow_{\textbf{store}}$$
$$\langle 6, [0, 1], \lambda \rangle$$

Basics of information flow

```
1  load x
2  store y
3  halt
```

explicit flow

x is loaded onto the stack, then it is stored into y, that is, y depends explicitly on x

```
1  load x
2  if 5
3  push 1
4  goto 6
5  push 0
6  store y
7  halt
```

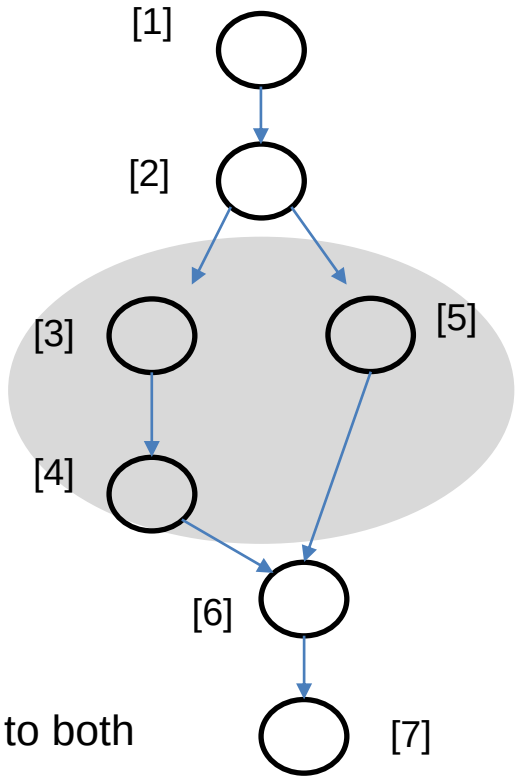
implicit flow

variable x is loaded onto the stack. Depending on the value of x, either the constant 1 or the constant 0 is pushed onto the stack, and successively stored onto y

In both cases observing the final value of y reveals information on the value of x

Implicit flow

```
1  load x
2  if 5
3  push 1
4  goto 6
5  push 0
6  store y
7  halt
```



Implicit flow starts at [2]

When implicit flow terminates?

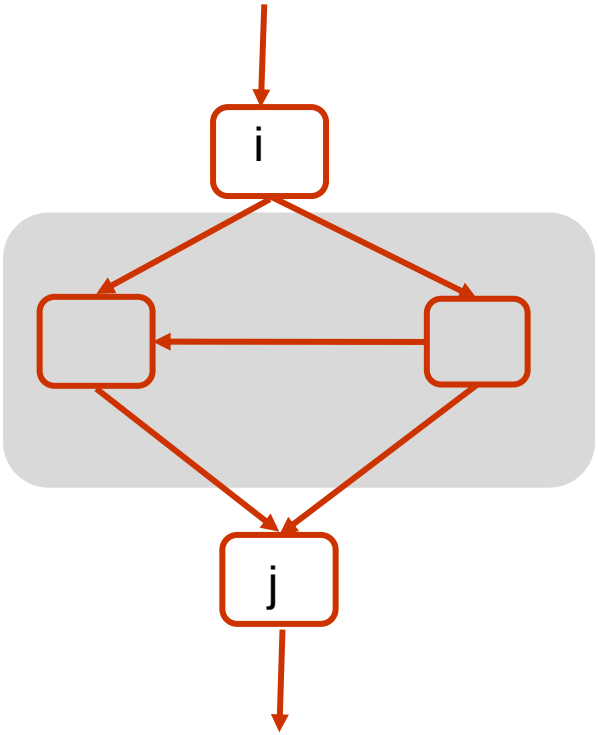
[6] is the first instruction that is common to both branches

The implicit flow terminates at [6]

[6] is the first instruction that is not under the implicit flow

Implicit flow

We use the concept of immediate postdominator on the CFG of the program to handle implicit flows



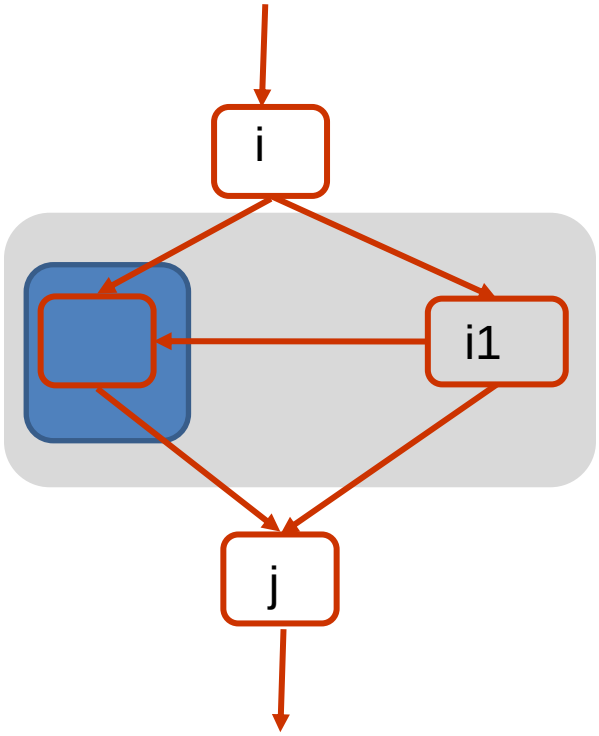
immediate postdominator of i: the first node belonging to all paths from i

ipd(i) = j

The implicit flow of an if instruction at address i terminates at the instruction with address ipd(i)

Implicit flow

What about nested control instructions?



Nested implicit flows

The innest implicit flow if the implicit flow that terminate first

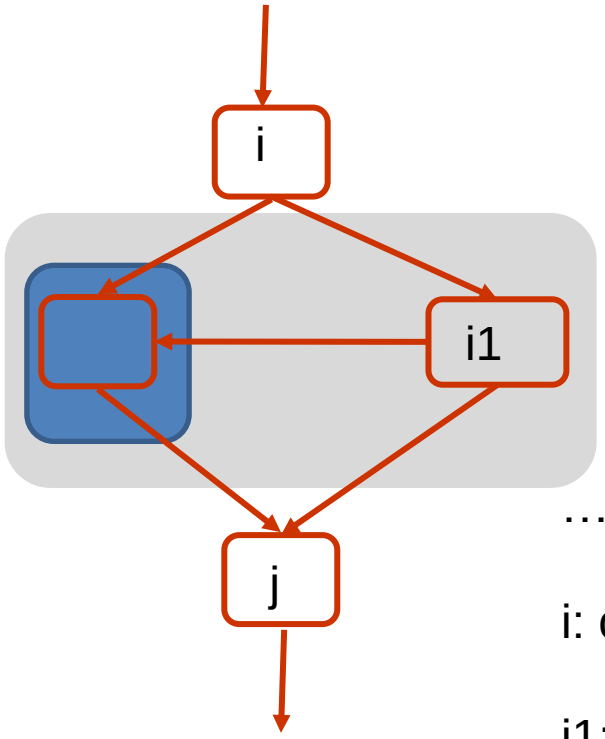
immediate postdominator of i1: the first node belonging to all paths from i

ipd(i1) = j

IPD stack:

when executing an instruction, the ipd stack mantains information on the open implict flows
IPD stack is updated any time a control instruction is enetered and any time a control instruction terminates

Implicit flow



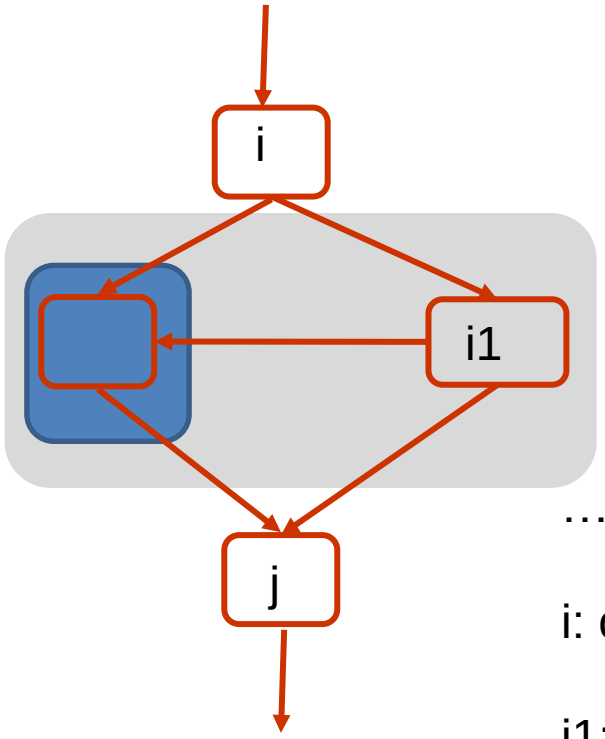
CONTROL REGION
of a branching
instruction

Execution of instructions

when an instruction j is executed: if the instruction j is the top of the ipd stack, the stack is updated by executing pop (j is removed from the stack)

…before i …….	Stack of ipd:	λ
i: control instruction	Stack of ipd:	ipd(i)
i1: control instruction	Stack of ipd:	ipd(i1) ipd(i)
j: top of the ipd stack	Stack of ipd:	ipd(i)
j: top of the ipd stack	Stack of ipd:	λ

Implicit flow



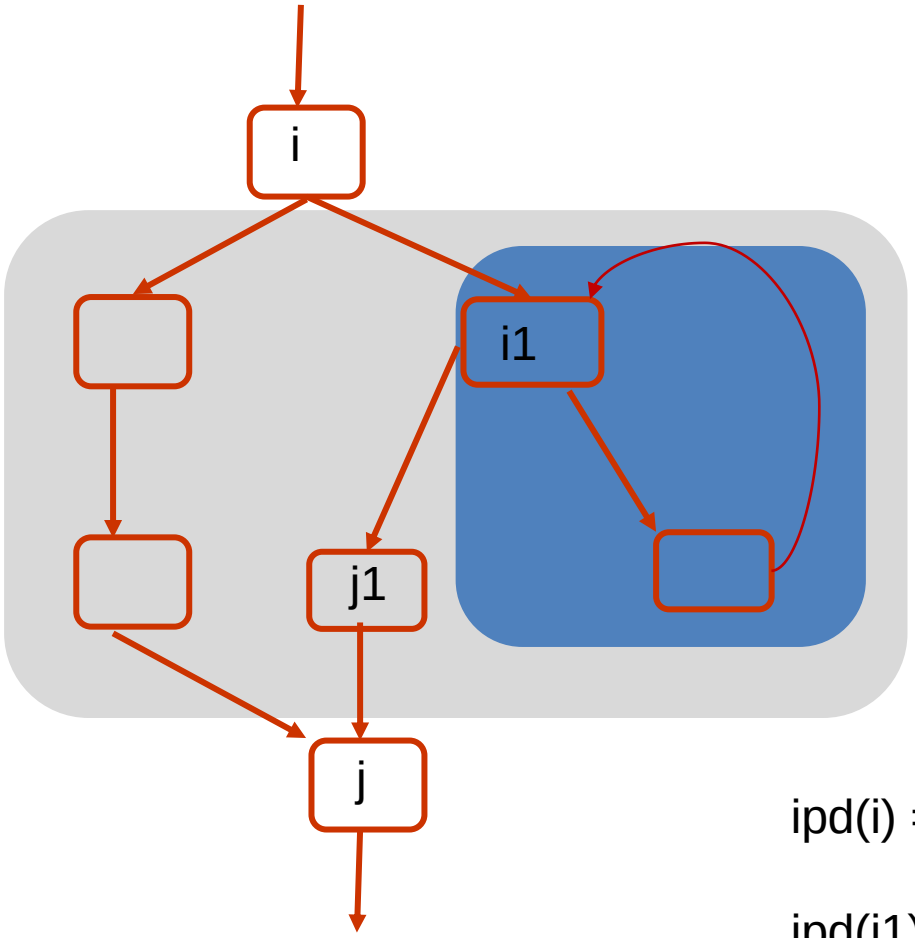
CONTROL REGION
of a branching
instruction

Execution of instructions

when an instruction j is executed: if the instruction j is the top of the ipd stack, the stack is updated by executing pop (j is removed from the stack)

…before i …….	Stack of ipd:	λ
i: control instruction	Stack of ipd:	ipd(i)
i1: control instruction	Stack of ipd:	ipd(i1) ipd(i)
j: top of the ipd stack	Stack of ipd:	ipd(i)
j: top of the ipd stack	Stack of ipd:	λ

Implicit flow



ipd(i) = j

ipd(i1) = (j1)

Basics of information flow

Influence of the implicit flow onto the operand stack

the stack may be manipulated in different ways by the branches of a branching instruction: they can perform a different number of pop and push operations, and with a different order.

The length and the content of the operand stack may be a means by which security leakages can occur

```
1  push 1
2  load x
3  if 5
4  pop
5  halt
```

The stack is empty or not, depending on the value of x

Basics of information flow

Secure Information flow

```
1  load x
2  if 5
3  push 1
4  goto 6
5  push 0
6  store y
7  halt
```

H={x} L={y}

A program P = <c, H, L > satisfies secure information flow if the final value of each low variable does not depend on the initial value of the high variables.

Basics of information flow

Termination Agreement

H={x} L={}>

```
1  load x
2  if 1
3  halt
```

it is not possible to leak high information by observing the termination of the program

Timing Agreement

H={x} L={y}>

```
1  load x
2  if 5
3  push 1
4  goto 6
5  push 0
6  store y
7  halt
```

the number of instructions executed in a computation may reveal information on the value of the high variables

Domains of the concrete semantics

Security levels	$\mathcal{L} = \{L < H\}$	$\sigma, \tau, ..$
Constants	V	k, k', ..
Addresses	A	i, j, ..
Concrete Values	$\mathcal{V} = V \times \mathcal{L}$	(k, σ)
Concrete Addresses	$\mathcal{A} = A \times \mathcal{L}$	(i, σ)
Concrete Memories	$\mathcal{M} = var \rightarrow (\mathcal{V} \cup \mathcal{A})$	M, M', ..
Concrete Stacks	$\mathcal{S} = (\mathcal{V} \cup \mathcal{A})^*$	S, S', ..
Environments	$\mathcal{E} = \mathcal{L}$	$\sigma, \tau, ..$

Concrete Semantics

STATES $\mathcal{L} \times A \times \mathcal{M} \times \mathcal{S} \times \mathcal{A}^*$

< σ, PC, M, S, ρ >

σ	environment
PC	program counter
M	memory
S	operand stack (**σ1 …. σn**)
ρ	ipd stack (j, **σ**)…..(j', **σ**')

IPD Stack ρ

if ρ = (j1, **σ1**)…..(jn, σn)
there are n open implicit flows
j1 holds the address where first implicit flow terminates
σ1 holds the level of the environment that must be restored

if ρ = λ
there are no open implicit flow

Transition relation rules

$c[i] = \textbf{load } x$, $M[x] = (k, \tau)$, $not\_top(i, \rho)$

load ———————————————————————

$< \sigma , i, M, S, \rho > \rightarrow$

$< \sigma , i+1, M, (k, \sigma \cup \tau) \cdot S, \rho >$

Transition relation rules

$$\textbf{store}\ \frac{\texttt{c[i]= } \textbf{store x}\ ,\ \text{not\_top}(i, \rho)}{\langle \sigma, i, M, (k, \tau) \cdot S, \rho \rangle \rightarrow \langle \sigma, i, M[\ (k, \sigma \cup \tau)/\ \textbf{x}\ ], S, \rho \rangle}$$

Transition relation rules

$$\rho = (i, \tau) \,.\, \rho'$$

ipd ———————————————————————————

$$< \sigma \,, i, M, S, (i, \tau) \,.\, \rho' > \rightarrow < \tau \,, i \,, M, S, \rho' >$$

i is the ipd of a control instruction

Transition relation rules

$$\textbf{goto}\ \frac{c[i] = \textbf{goto j}\ ,\ \text{not\_top}(i, \rho)}{\langle \sigma, i, M, S, \rho \rangle \rightarrow \langle \sigma, j, M, S, \rho \rangle}$$

i is the ipd of a control instruction

Transition relation rules

$$\frac{c[i] = \textbf{if } j \textbf{ ,} \quad \text{not\_top}(i, \rho)}{\text{if-false}}$$

$$\langle \sigma, PC, M, (0, \tau) \cdot S, \rho \rangle \rightarrow \langle \sigma \cup \tau, PC+1, up(M), up(S), (\sigma, ipd(i)) \rho \rangle$$

An implicit flow begins, whose level is the least upper bound between the security environment (σ) and the security level of the condition of the **`if`** **(τ). The new security environment is** **($\sigma \cup \tau$)**
(ipd(pc), σ) is pushed on the ipd stack ρ

up(M) upgrades the value of the variables assigned in the scope of the implicit flow beginning at PC

up(S) upgrades all elements in the stack

Transition relation rules

$$\text{if-true}\ \frac{\texttt{c[i]= if j , k!=0,}\ \text{not\_top}(i, \rho)}{\langle \sigma, i, M, (k, \tau) \cdot S, \rho \rangle \rightarrow \langle \sigma \cup \tau, j, \text{up}(M), \text{up}(S), (\text{ipd}(i), \sigma) \cdot \rho \rangle}$$

An implicit flow begins, whose level is the least upper bound between the security environment (σ) and the security level of the condition of the `if (`τ**). The new security environment is** **($\sigma \cup \tau$)**
(ipd(pc), σ) is pushed on the ipd stack ρ

up(M) upgrades the value of the variables assigned in the scope of the implicit flow beginning at PC

up(S) upgrades all elements in the stack

Concrete rules

$$\textbf{ipd}\quad \frac{\rho = (i, \tau) \cdot \rho'}{\langle \sigma, i, M, S, \rho \rangle \longrightarrow \langle \tau, i, M, S, \rho' \rangle}$$

$$\textbf{op}\quad \frac{c[i] = \mathtt{op} \quad not\_top(i, \rho)}{\langle \sigma, i, M, (k_1, \tau_1) \cdot (k_2, \tau_2) \cdot S, \rho \rangle \longrightarrow \langle \sigma, i + 1, M, (k_1\ op\ k_2, \tau_1 \sqcup \tau_2) \cdot S, \rho \rangle}$$

$$\textbf{pop}\quad \frac{c[i] = \mathtt{pop} \quad not\_top(i, \rho)}{\langle \sigma, i, M, (k, \tau) \cdot S, \rho \rangle \longrightarrow \langle \sigma, i + 1, M, S, \rho \rangle}$$

$$\textbf{push}\quad \frac{c[i] = \mathtt{push}\ k \quad not\_top(i, \rho)}{\langle \sigma, i, M, S, \rho \rangle \longrightarrow \langle \sigma, i + 1, M, (k, \sigma) \cdot S, \rho \rangle}$$

$$\textbf{load}\quad \frac{c[i] = \mathtt{load}\ x \quad M(x) = (k, \tau) \quad not\_top(i, \rho)}{\langle \sigma, i, M, S, \rho \rangle \longrightarrow \langle \sigma, i + 1, M, (k, \tau \sqcup \sigma) \cdot S, \rho \rangle}$$

$$\textbf{store}\quad \frac{c[i] = \mathtt{store}\ x \quad not\_top(i, \rho)}{\langle \sigma, i, M, (k, \tau) \cdot S, \rho \rangle \longrightarrow \langle \sigma, i + 1, M[(k, \tau)/x], S, \rho \rangle}$$

$$\textbf{goto}\quad \frac{c[i] = \mathtt{goto}\ j \quad not\_top(i, \rho)}{\langle \sigma, i, M, S, \rho \rangle \longrightarrow \langle \sigma, j, M, S, \rho \rangle}$$

$$\textbf{if}_{false}\quad \frac{c[i] = \mathtt{if}\ j \quad not\_top(i, \rho)}{\langle \sigma, i, M, (0, \tau) \cdot S, \rho \rangle \longrightarrow \langle \tau, i + 1, up_M(M, mod^P(F^P(i)), \tau), up_S(S, \tau), (ipd(i), \sigma) \cdot \rho \rangle}$$

$$\textbf{if}_{true}\quad \frac{c[i] = \mathtt{if}\ j \quad not\_top(i, \rho)}{\langle \sigma, i, M, (k \neq 0, \tau) \cdot S, \rho \rangle \longrightarrow \langle \tau, j, up_M(M, mod^P(F^P(i)), \tau), up_S(S, \tau), (ipd(i), \sigma) \cdot \rho \rangle}$$

Abstract semantics

Abstract constants	$V^{\#} = \{ \cdot \}$
Abstract security levels	$\mathcal{L}^{\#} = \mathcal{L}$
Abstract Values	$\mathcal{V}^{\#} : V^{\#} \times \mathcal{L}^{\#} \equiv \mathcal{L}$
Abstract Addresses	$\mathcal{A}^{\#} = \mathcal{A}$
Abstract Memories	$\mathcal{M}^{\#} : \text{var} \rightarrow (\mathcal{L} \cup \mathcal{A})$
Abstract Stacks	$\mathcal{S}^{\#} : (\mathcal{L} \cup \mathcal{A})^{*}$
Abstract Environments	$\mathcal{E}^{\#} = \mathcal{E} = \mathcal{L}$
Abstract States:	$\mathcal{L} \times A \times \mathcal{M}^{\#} \times \mathcal{S}^{\#} \times (A \cup \{ 0 \})$

Abstract operational semantics

the abstract semantics:

• abstracts concrete values into their security level:

$$\alpha(k,\sigma)=\sigma$$

• uses the same rules of the concrete semantics on the abstract domains

Both rules for if are always applied - $\mathbf{if}_{\mathbf{true}}$ $\mathbf{if}_{\mathbf{false}}$

A(P) : abstract transition system for P

• finite
• multiple path
• each path of C(P) is correctly abstracted onto a path of A(P)

Results

Theorem 1

A program P satisfies SIF if for each state of A(P) such that c[i] = halt , then for each x : L it is:

M[x] = L (value)
or
M[x]=(i, L) for some i (address)

Results

Theorem 2

A program P satisfies TERM if each state of A(P)

$< \sigma , i, M, S, \rho >$ such that $\sigma = H$

does not belong to a cycle.

Results

Theorem 3

A program P satisfies TIME if:

• all paths in A(P) starting from a state satisfying top(S)=H and c[i] = if and ending with a state satisfying PC=ipd(i) have the same length.

Another example: concrete semantics

x:(0,H) y:(1,L)
ipd(2) = 5, ipd(6)=10

<ENV, PC, [M(x), M(y)], Stack, IPDstack>

```
1    load y
2    if 5
3    push 3
4    store x
5    load x
6    if 9
7    push 1
8    goto 10
9    push 0
10   store y
11   halt
```

$\langle L, 1, [(0, H)(1, L)], \lambda, \lambda \rangle$

↓ **load**

$\langle L, 2, [(0, H)(1, L)], (1, L), \lambda \rangle$

↓ **if$_{\text{true}}$**

$\langle L, 5, [(0, H)(1, L)], \lambda, (5, L) \rangle$

↓ **ipd**

$\langle L, 5, [(0, H)(1, L)], \lambda, \lambda \rangle$

↓ **load**

$\langle L, 6, [(0, H)(1, L)], (0, H), \lambda \rangle$

↓ **if$_{\text{false}}$**

$\langle H, 7, [(0, H)(1, L)], \lambda, (10, L) \rangle$

↓ **push**

$\langle H, 8, [(0, H)(1, L)], (1, H), (10, L) \rangle$

↓ **goto**

$\langle H, 10, [(0, H)(1, L)], (1, H), (10, L) \rangle$

↓ **ipd**

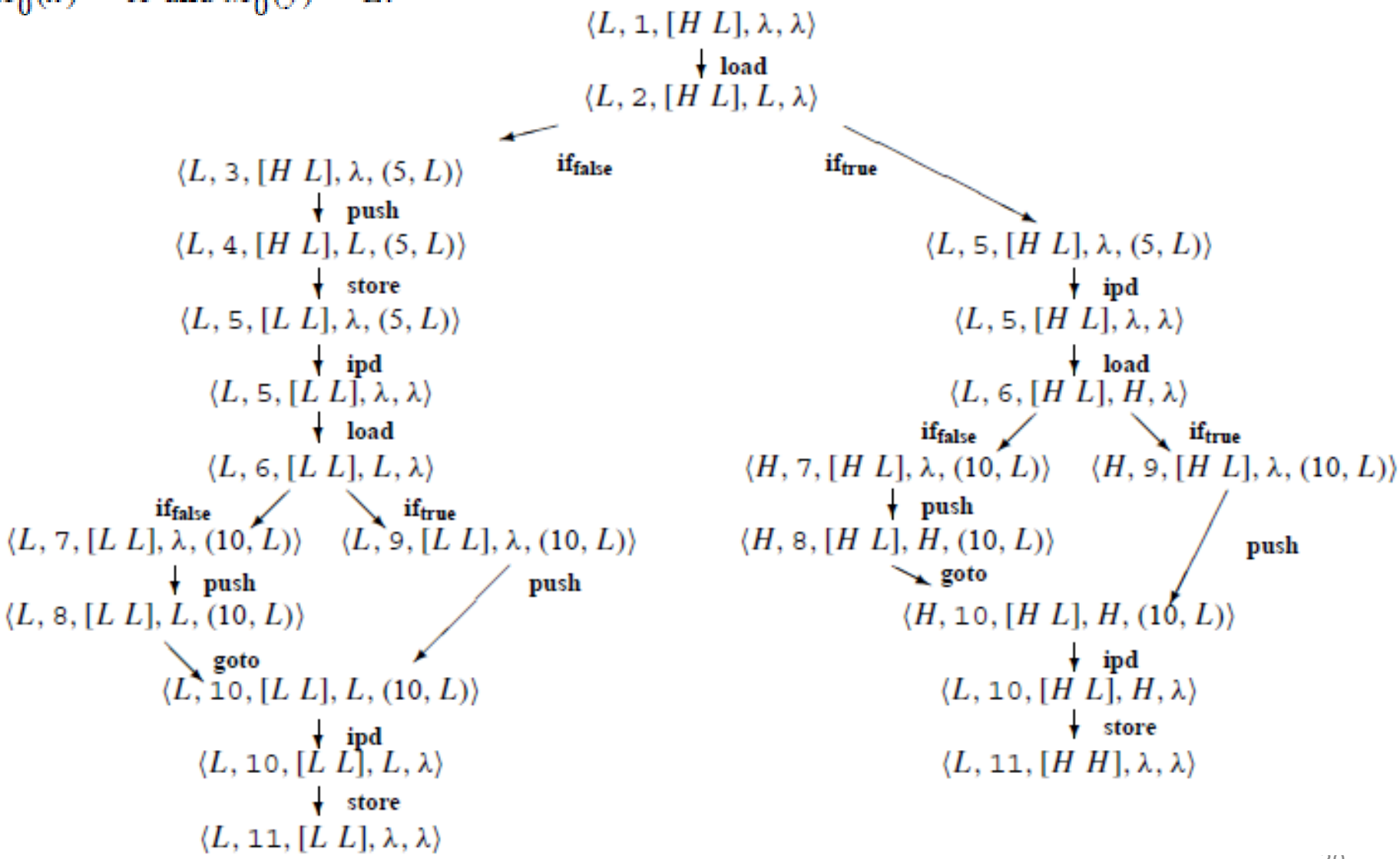
$\langle L, 10, [(0, H)(1, L)], (1, H), \lambda \rangle$

↓ **store**

$\langle L, 11, [(0, H)(1, H)], \lambda, \lambda \rangle$

An example: abstract semantics

$M_0^{\natural}(x) = H$ and $M_0^{\natural}(y) = L$.



Correctness of the analysis

In the definition of the abstract semantics, we have applied Abstract Interpretation.

Abstract interpretation is a widely applied method for designing approximate semantics of programs.

P. Cousot, R. Cousot. Abstract interpretation frameworks. Journal of Logic and Computation, 2, 1992

Example: PINcloner malicious app

The app catches the private information of the user's Personal Identification Number (PIN) without directly assigning of it to the public variable clone. It achieves this by using a mask that reveals the value of each bit of the PIN.

The PINcloner (pseudocode)

```
input : PIN
output : clone
clone := 0x0000;
mask := 0x0001;
while(mask > 0)
        b :=PIN & mask;
        if (b ! = 0)
        clone:=clone || mask;
        mask := mask << 1;
```

PINcloner malicious app

Initially the mask has all the bits set to 0 except for the least significant bit, which is set to 1: this bit shifts one step to the left after each loop cycle and clones the value of one bit of the PIN during each cycle.

In particular, the direct assignment of the PIN bits to the clone variable is avoided by using a variable b.
This last variable is different from 0 if and only if the i-th bit of the PIN and of the mask are both equal to 1: the value of b can be used to set (or not set) the i-th bit of the clone variable.

Once the app has gained the access to the user's private data, the access control mechanism is not able to reveal the illicit flow

J_PINCloner: pseudocode

PIN read from a file and copied into another file

Files:
- PIN: H
- Clone: L

```
public class J_PINcloner{
  final int pinSize = 6;
  FileReader pin_file;
  FileWriter cloned_pin_file;

  public J_PINcloner(){
    pin_file = new FileReader("PIN");
    cloned_pin_file = new FileWriter("clone");
  }

  public static void main(String[] args){
    J_PINcloner p = new J_PINcloner();
    for(int i = 0; i < p.pinSize; i++){
      int PIN = p.pin_file.read();
      int clone = _clone(PIN);
      p.cloned_pin_file.write(clone);
      p.cloned_pin_file.flush();
    }
  }

  private static int _clone(int PIN){
    int mask = 0x0001;
    int clone = 0x0000;
    while(mask > 0){
      int b = (PIN & mask);
      if(b != 0)
        clone = clone | mask;
      mask = mask << 1;
    }
    return clone;
  }
}
```

J_PINCloner: SIF analysis

```
SIF violated in method public static void main(String[] arg

DETAILS :
  SIF Violated on Instruction :
  41 : invokejava.io.FileWriter.write(I)V

DEPENDENCES :
  DEP(41) = 0, 18;
  SL[0] = L; SL[18] = L;
  env(41) = L

BEFORE STATE OF 41
  Registers : [L, L, L, H, H]
  Operand Stack : [H · FL]

SECURITY POLICY :
  PIN = FH
  clone = FL

SECURITY CONTEXT :
  J_PINcloner.pinSize = L
  J_PINcloner.pin_file = FH
  J_PINcloner.cloned_pin_file = FL
  J_PINcloner. < init > ()V.arg0 = L
  J_PINcloner. < init > ()V.CALLER = L
  J_PINcloner._clone(I)I.arg1 = H
  J_PINcloner._clone(I)I.return = H
  J_PINcloner._clone(I)I.CALLER = L
  J_PINcloner.main(...)V.arg1 = L
  J_PINcloner.main(...)V.CALLER = L
```

J_PINCloner: SIF analysis

```
- - - - LIBRARY FUNCTIONS - - - - -
  java.lang.Object. < init > ()V.arg0 = L
  java.lang.Object. < init > ()V.CALLER = L

EXECUTION PATH :
  0 : new J_PINcloner
  3 : dup
  4 : invoke J_PINcloner. < init > ()V
  7 : astore 1
  ...
  21 : aload 1
  22 : getfield J_PINcloner.pin_file
  25 : invoke java.io.FileReader.read()I
  28 : istore 3
  29 : iload 3
  30 : invoke J_PINcloner._clone(I)I
  33 : istore 4
  35 : aload 1
  36 : getfield J_PINcloner.cloned_pin_file
  39 : iload 4
  41 : invoke java.io.FileWriter.write(I)V

FAILED, SIF VIOLATED in J_PINcloner
```

Data Leakage with Exception mechanism

Exceptions are special events used for signaling errors during the execution of a program.

The rising of an exception is referred as throwing

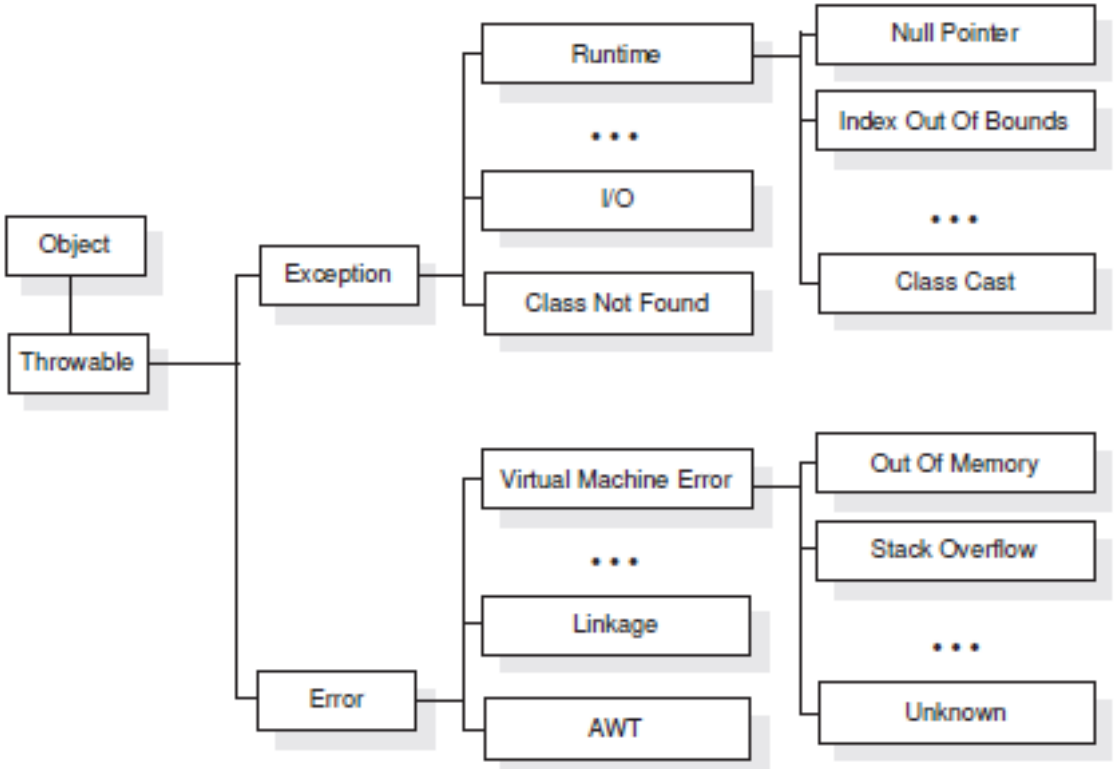
Every time an exception is thrown, the Java runtime system breaks the standard execution flow of the program and calls the handler that catches and manages the exception

The correct handler for a given exception type can be found searching backwards through the call stack of the method

If no appropriate handler is found the program terminates

Data Leakage with Exception mechanism

Java Exception Hierarchy (incomplete)



Data Leakage with Exception mechanism

A simple example with exceptions

```
1
2  import java.io.IOException;
3  public class Hello {
4    public static void main(String[] args) {
5    }
6
7    public void stampa(String s){
8           System.out.println("Outside");
9
10     try { System.out.println("Inside try");}
11     catch (ArithmeticException e) {System.out.println("Print catch");}
12
13    System.out.println("Print ...");
14    }
15 }
```

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1 //Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: return

  public void stampa(java.lang.String);
    Code:
       0: getstatic #2 //Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc       #3 //String Outside
       5: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: getstatic     #2 //Field java/lang/System.out:Ljava/io/PrintStream;
      11: ldc           #5 //String Inside try
      13: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
      16: goto          28
      19: astore_2
      20: getstatic     #2 //Field java/lang/System.out:Ljava/io/PrintStream;
      23: ldc           #7 //String Print catch
      25: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
      28: getstatic     #2 //Field java/lang/System.out:Ljava/io/PrintStream;
      31: ldc           #8 //String Print ...
      33: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
      36: return

    Exception table:
       from    to  target type
           8    16    19   Class java/lang/ArithmeticException
}
```

Data Leakage with Exception mechanism

- From the exception table: instructions from 8 to 16 are executed in a protected way. Moreover, 19 is the first instruction of the exception handler. Instruction 8 is protected and it may throw an exception, that can be captured by an exception handler (executing the code at 19).

Assume that a protected instruction is a control instruction that throws an exception depending on a high condition.

- The handler of the exception must be executed under a security environment that is high
- The execution of the exception handler that captures the exception may reveal information on the value of the high condition, thus causing a leakage of information

Data Leakage with Exception mechanism

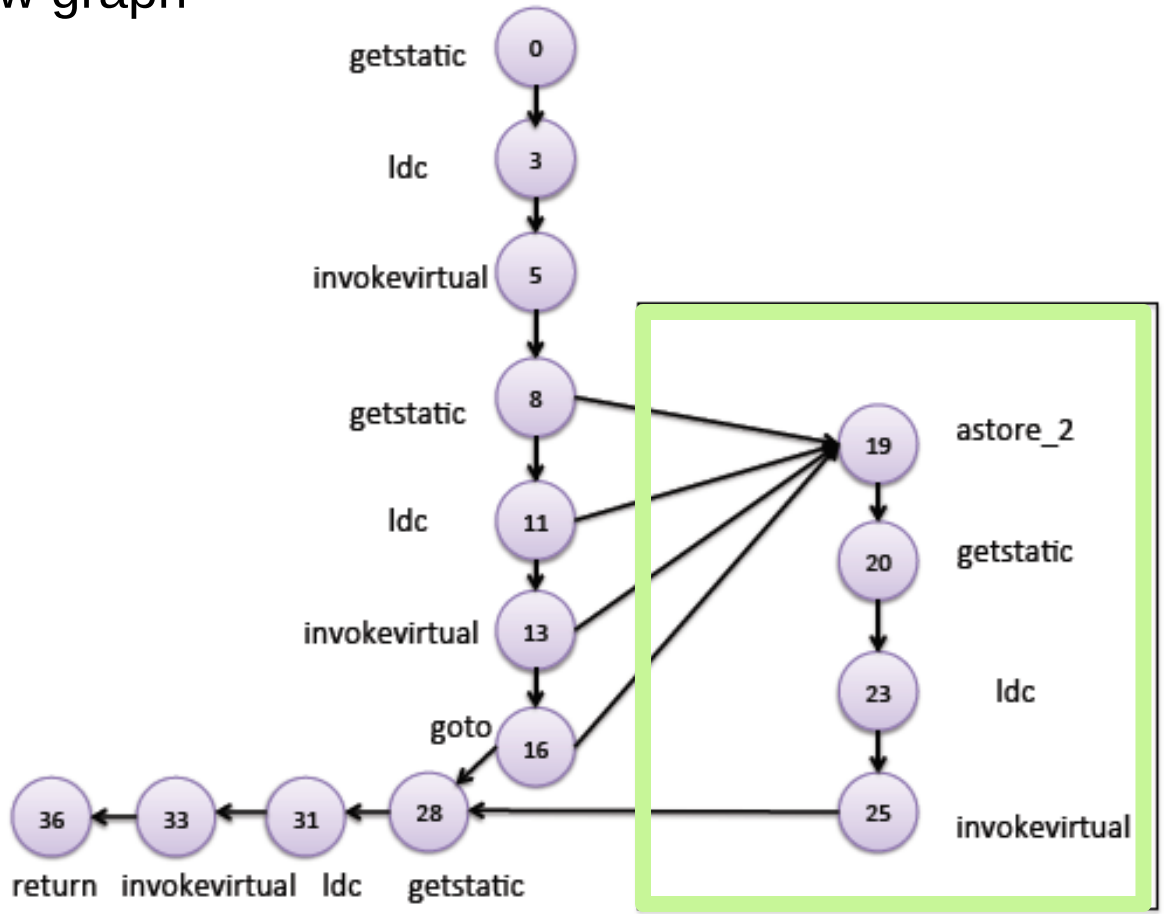
SOLUTION:

1. The body of the exception handlers can be thought as particular extensions of the methods code

2. An extended control flow graph is built
   the extended control flow graph is defined as the graph obtained from the method graph augmented with edges starting from protected instructions to the first instruction of the protecting handlers.

Data Leakage with Exception mechanism

Extended control flow graph of the bytecode



PINCloner malicious applet

Let us consider the PINCloner applet, where PIN_FILE file and Clone_File file are the input and the output files, respectively.

PIN_FILE is a private file containing a secret PIN (a sequence of 0/1 characters, for simplicity).

Let us suppose that the PINCloner application can read from the private file. The applet clones the user secret PIN with the exception mechanism.

After every character has read, it will be written in a public file by the handler of the exception.

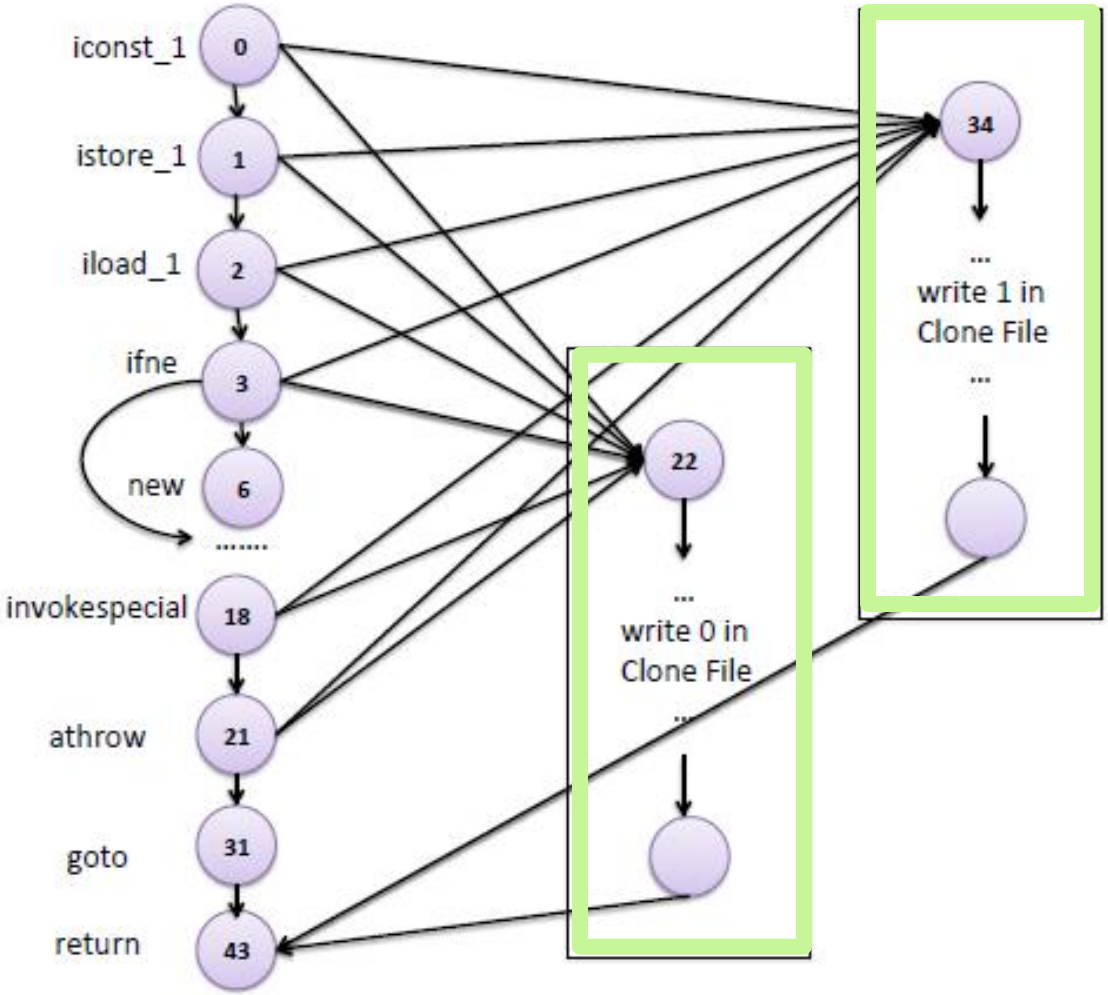
The PINCloner clones the characters of the PIN_FILE by throwing different kind of exceptions depending on the value read

- NullPointerException
- ArithmeticException

PINCloner bytecode (an excerpt)

```
1  import java.io.IOException;
2
3  public class PinCloner {
4    public static void main(String[] args) {
5    }
6
7    public void PinCloner(){
8          try {
9              int p;
10             for (...) { // reads a char from PIN_FILE and store it in p;
11             .....
12             if (p == 0) { throw  new ArithmeticException(); }
13             else { throw  new NullPointerException();}
14             }
15       }
16     catch (ArithmeticException e) { // write 0 in Clone_File
17                                     }
18     catch (NullPointerException e) {// write 1 in Clone_File
19                                     }
20         .....
21     }
22 }
```

PINCloner bytecode extended CFG



PINCloner bytecode extended CFG

- instructions from 0 to 22 are protected by two exception handles starting at instruction 22 and instruction 34, respectively. Instruction 3 is an if with four successors:
  - the natural successors,
  - plus the two entry points of the exception handlers

- The control region of 3 includes the instructions of the exception handlers, and consequently these instructions are executed in a security environment given by the condition of the ifne.

- Since the condition depends on the 0/1 value of PIN character read from a high security file, the implicit flow is high. The handler of the exception, write such value into the low security Clone_File file.

PINCloner malicious applet

- Security policy:

PIN_FILE	H
Clone_file	L

- The analysis starts with L assigned to all the other resources
- The application violates the secure information flow because high security data are written on a public file
- The leakage is detected by the static analysis