Data leakage

- Security policy
- Secure Information flow in programs
- Abstract Interpretation of the operational semantics
- Certification of secure flow

Data leakage

GENERAL DATA PROTECTION REGULATION(GDPR) - UE 2016/679

Regulation of the European Parliament and of the Council on the protection of natural persons with regard to the processing of personal data and on the free movement of such data

- explicit (private data made publicly available)
- interference between private and public data

Data leakage

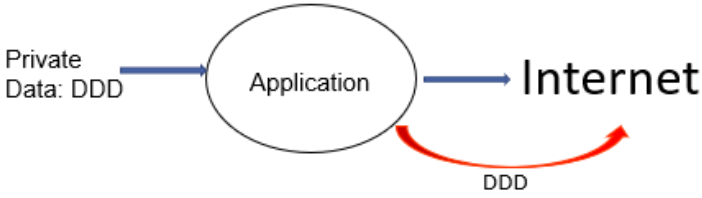
- *secret hard-coded*
  detect hard-coded private data before it causes a data leakage by text search and regular expressions (source and binary code)
- *restriction of access to resources by authorisation*
  Firewall and access control mechanism. Statically check the rights to access the resource
- *code that without malicious objectives* propagate disclose private data to unauthorised users
- *malicious code (malware)*
  programs with the objective to cause data leakage during the system use

An example of Data Leakage Tool

- NG SAST (Next Generation Static Application Security Testing) platform, ShiftLeft
https://www.shiftleft.io/blog/detecting-sensitive-data-leaks-that-matter-03-17-2021/

"..... The *source* of a sensitive data leak is usually a variable containing sensitive information or any functionality that uses the variable. And a *sink* in this context could be any function that causes information to be displayed to users, such as logging, sending automated emails, and writing to web pages. If the sensitive *source* can make its way to a *sink* function, the sensitive data could be leaked."

Data leakage



Secure information flow is violated

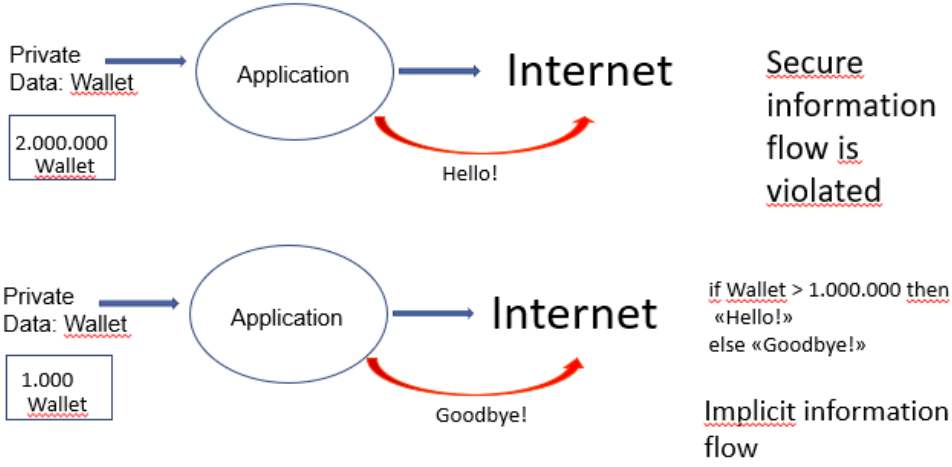
Application authorized to access private data
Application authorized to access Internet

Explicit information flow

Control on the information sent on Internet!!!!!

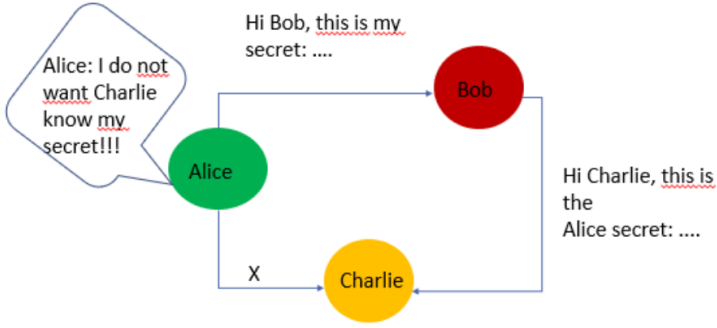
Limit of the firewall and access control mechanism !!!

Data leakage



Certificate that the application does not send data that may reveal any private information

Data leakage



Secure information flow is violated !!!

Certification of code for secure information flow

Colluding apps

The Independent (British online newspaper)

Taken from: http://www.independent.co.uk/life-style/gadgets-and-tech/news/android-app-steal-users-data-colluding-each-other-research-cartel-information-a7663976.html

The team reports that the types of app fall into two major categories / Justin Sullivan/Getty Images

The biggest security risks can come from some of the least capable apps

"Android apps are mining smartphone users data by secretly colluding with each other, according to a new study. Pairs of apps can trade information, a capability that can lead to serious consequences in terms of security."

Language-based security

Can be studied by defining a security policy and by using the theory of information flow in programs

Security policy:
a set of security-motivated constraints that must be satisfied by an organization or a computer system
e.g. policy regarding the public disclosure of company information, policy of the

physical and networked access to company computers, constraints on how information may flow within a system.

Methods for
- formally expressing and analysing security policies
- the enforcement of the policy

Multilevel security policy

Multilevel Security policy: a security policy that allows the classification of data and users based on a set of hierarchical security levels.

Example:

$\mathcal{S} = \{Public, Private\}$

Private level is higher than Public level.

Multilevel security policy

Definition (multi-level seurity policy)

A multilevel security policy $\mathcal{L}$ is a pair that consists of
(i) a set of security levels $\mathcal{S}$ and
(ii) an ordering relation $\sqsubseteq$ between the levels.
Moreover, every pair of elements in $\mathcal{S}$ has both:
- a greatest lower bound (glb, $\sqcap$) and
- a least upper bound (lub, $\sqcup$).

$$\mathcal{L} = (\mathcal{S}, \sqsubseteq)$$

The relation $\sqsubseteq$ is reflexive and transitive. Moreover, $\sqsubseteq$ is antisymmetric.

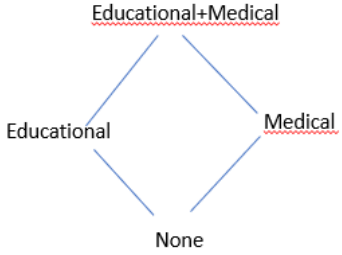
$(\mathcal{S}, \sqsubseteq)$ is a **lattice** of security levels.

Example:
$\mathcal{L} = (\mathcal{S}, \sqsubseteq)$
$\mathcal{S} = \{\textit{Public}, \textit{Private}\}$
$\sqsubseteq$ defined as follows: $\textit{Public} \sqsubseteq \textit{Private}$.

Multilevel security policy

Example:
Educational and *Medical* are sensitive classes of information of a user.

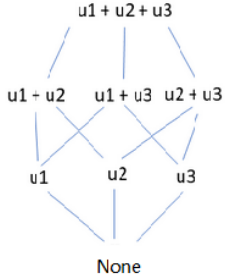
$\mathcal{S} = \{None, Educational, Medical, Educational + Medical\}$, with $\sqsubseteq$ defined as:
$None \sqsubseteq Educational$;
$None \sqsubseteq Medical$;
$Medical \sqsubseteq Educational + Medical$;
$Educational \sqsubseteq Educational + Medical$
least upper bound ($\sqcup$): $Educational \sqcup Medical = Educational + Medical$



Multilevel security policy

Example:
Assume u_i represents sensitive information of user i.
$\mathcal{S} = \{None, u1, u2, u3, u1 + u2, u1 + u3, u2 + u3, u1 + u2 + u3\}$ with

$None \sqsubseteq ui$;
$u_i \sqsubseteq ui + u_j, j \neq i$;
$u_i + u_j, j \neq i \sqsubseteq u1 + u2 + u3$
least upper bound ($\sqcup$): e.g., $u1 \sqcup u2 = u1 + u2$



Secure Information Flow in programs

We assume a security policy $\mathcal{L} = (\mathcal{S}, \sqsubseteq)$ such that:

- $\mathcal{S} = \{l, h\}$
- $\sqsubseteq$ defined as: $l \sqsubseteq h$

Input and output of a program are assigned either low level of security (l) or high level of security (h)

Secure Information Flow property:
the low output do not reveal information on the high level input. Low output are not assigned high level data.

Non-interference property

Non-interference property:
the security domain private is non-interfering with domain public if no input by private can influence subsequent outputs that can be seen by public.

A program has the non-interference property if and only if any sequence of low inputs will produce the same low outputs, regardless of what the high level inputs are.

The program responds in exactly the same manner on low outputs whether or not high sensitive data are changed.

The low user will not be able to acquire any information about data of the high user.

Information flow in programs

Information flow occurs through

- simple variables, input/output files
- array, structures, objects
- pointers, references
- objects allocated in dynamic memory
- global variables
- function calls, parameters by value/ parameters by reference, return

High-level language.

Let x, y be variables

```
y := x;
```
explicit flow

variable y is assigned the value of x, there is an explicit flow from x to y

```
if   (x = 0)
     then y:=1;
     else y:=0;
```
implicit flow

there is an implicit flow from variable x to y, since y is assigned different values depending on the value of the condition of the control instruction (variable x)

Basics of information flow

In both cases
$\rightarrow$ observing the final value of y reveals information on the value of x.

A conditional instruction in a program causes the beginning of an implicit flow.

The implicit flow begins when the conditional instruction starts (we say that we have an opened implicit flow); all the instructions in the scope of the if depend on the condition of the if.

Basics of information flow

If a function call is executed in the scope of a conditional instruction, the function is executed under the implicit flow.

```
if   (y ≤ 0)
     then f();
```

Function f() is invoked depending on the value of variable y.

Instructions of f() are executed under the implicit flow of the condition of the if statement.

Secure information flow checking

Approaches:

- Typing approach [1].
  the security information of a variable belongs to its type, and secure Information flow is checked by means of a type system.

  Hierarchy between types.
  Types = $\{high, low\}$, with $low \sqsubseteq high$.
  $low := high$ typing error
  $high := low$ correct

- Semantic-based approach [2]
  execute the program (high complexity)

- Abstract interpretation of the operational semantics approach [3]
  execute the program on abstract domains

Secure information flow checking

An advantage of abstract interpretation approach with respect to those based on typing is that it is semantics based and thus keeps information on the dynamic behaviour of programs, allowing to check more precisely the desired properties.

y:=x; y:=0;
- rejected by the typing approach
- accepted by the abstract interpretation approach

if (0) then y:=x; else skip;
- rejected by the typing approach;
- rejected by abstract interpretation approach;
- accepted by semantics approach

Abstract interpretation of the operational semantics for secure information flow in programs

Abstract interpretation of the operational semantics for secure flow

- instrumented semantics that add the security level to data and traces the information flow (enhanced semantics)
- abstract semantics that abstracts from real value and executes the program on security level (consider only the security level of data)
- correctness of the abstraction

A simple high level language

$exp ::= const \mid var \mid exp\ op\ exp$
$com ::= var := exp \mid \texttt{if}\ exp\ \texttt{then}\ com\ \texttt{else}\ com \mid$
$\texttt{while}\ exp\ \texttt{do}\ com \mid com\ ;\ com \mid \texttt{skip}$

Let $(\mathcal{S}, \sqsubseteq)$, with $\mathcal{S} = \{l, h\}$, be a lattice of security levels, ordered by $l \sqsubseteq h$, where $\sqcup$ denotes the least upper bound between levels.

A program P is a triple $\langle c, H, L \rangle$ with
- $c \in com$
- H are the high variables of P
- L are the low variables of P
- $H \cup L = Var(c)$ and $H \cap L = \emptyset$

Enhanced Operational semantics

We enrich the standard operational semantics, in such a way that a violation of security can be discovered

The enhanced semantics is an instrumented semantics which:

- Handles values (k, σ) annotated with a security level ($k = 0, 1, 2 \cdots$ and $\sigma \in \mathcal{S}$).
- Executes instructions under a security environment $\sigma \in \mathcal{S}$.
- C(P, M): enhanced transition system for $P = \langle c, H, L \rangle$, with $M(x) = (k, \sigma)$, for variable x.

Enhanced operational semantics

annotated value (k, σ)
during the execution, σ indicates the least upper bound of the security levels of the information flows, both explicit and implicit, on which k depends.

execution environment σ
during the execution, σ represents the least upper bound of the security levels of the open implicit flows.
σ is (possibly) upgraded when a branching instruction begins and is (possibly) downgraded when all branches join.

$(e)^{\sigma}$ and $(c)^{\sigma}$
expressions and commands are executed under the security environment σ

Enhanced Operational semantics

$\mathcal{Q}$ set of states state: $\langle \sigma, c, M \rangle$ $\rightarrow \subseteq \mathcal{Q} \times \mathcal{Q}$ transition system

$$\textbf{Expr}_{const} \quad \frac{}{\langle k^{\sigma}, M \rangle \longrightarrow_{expr} (k, \sigma)}$$

$$\textbf{Expr}_{var} \quad \frac{M(x) = (k, \tau)}{\langle x^{\sigma}, M \rangle \longrightarrow_{expr} (k, \sigma \sqcup \tau)}$$

$$\textbf{Expr}_{op} \quad \frac{\langle e_1^{\sigma}, M \rangle \longrightarrow_{expr} (k_1, \tau_1) \;\; \langle e_2^{\sigma}, M \rangle \longrightarrow_{expr} (k_2, \tau_2)}{\langle (e_1 \;\; op \;\; e_2)^{\sigma}, M \rangle \longrightarrow_{expr} (k_1 \;\; op \;\; k_2, \tau_1 \sqcup \tau_2)}$$

$$\textbf{Ass} \quad \frac{\langle e^{\sigma}, M \rangle \longrightarrow_{expr} v}{\langle (x := e)^{\sigma}, M \rangle \longrightarrow M[v/x]}$$

Rules description

- The rules compute the security level of the value of an expression dynamically using both the security level of the operands and the security level of the environment.

  For example, an integer constant k results in the value (k, σ), where σ is the security level of the environment under which k is evaluated.

Enhanced Operational semantics

$$\textbf{If}_{true} \quad \frac{\langle e^{\sigma}, M\rangle \longrightarrow_{expr} (true, \tau)}{\langle (\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2)^{\sigma}, M\rangle \longrightarrow \langle c_1^{\tau}, Impl(M, Mod(c_1; c_2), \tau)\rangle}$$

$$\textbf{If}_{false} \quad \frac{\langle e^{\sigma}, M\rangle \longrightarrow_{expr} (false, \tau)}{\langle (\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2)^{\sigma}, M\rangle \longrightarrow \langle c_2^{\tau}, Impl(M, Mod(c_1; c_2), \tau)\rangle}$$

$Mod(c)$ finds the set of variables *mod*ified in the sequence of instructions c i.e. those which are on the left of an assignment

Impl upgrades the security level of the values of the previous variables to take into account an *impl*icit flow.

Rules description

- Assume that the condition of an `if` command results in a value (k, τ).

  The branch c_1 or c_2, selected according to k (*true* or *false*), is executed in the memory $Impl(M, Mod(c_1; c_2), \tau)$ under the environment τ

  In particular, if $\tau = h$, the value of every variable assigned in at least one of the two branches is upgraded to h and the selected branch is executed in a high environment.

  When the conditional command terminates, the security environment is reset to the one holding before the execution of the command

$$\textbf{While}_{true} \quad \frac{\langle e^{\sigma}, M\rangle \longrightarrow_{expr} (true, \tau)}{\langle (\texttt{while}\ e\ \texttt{do}\ c)^{\sigma}, M\rangle \longrightarrow \langle (c;\texttt{while}\ e\ \texttt{do}\ c)^{\tau}, Impl(M, Mod(c), \tau)\rangle}$$

$$\textbf{While}_{false} \quad \frac{\langle e^{\sigma}, M\rangle \longrightarrow_{expr} (false, \tau)}{\langle (\texttt{while}\ e\ \texttt{do}\ c)^{\tau}, M\rangle \longrightarrow \langle Impl(M, Mod(c), \tau)\rangle}$$

- The `while` command is handled similarly to the conditional command.

$$\textbf{skip}\quad \frac{}{\langle \texttt{skip}^{\sigma}, M\rangle \longrightarrow \langle M\rangle}$$

$$\textbf{Seq}_1\quad \frac{\langle c_1^{\sigma}, M\rangle \longrightarrow \langle M'\rangle}{\langle c_1^{\sigma}; w, M\rangle \longrightarrow \langle w, M'\rangle}$$

$$\textbf{Seq}_2\quad \frac{\langle c_1^{\sigma}, M\rangle \longrightarrow \langle w', M'\rangle}{\langle c_1^{\sigma}; w, M\rangle \longrightarrow \langle w'; w, M'\rangle}$$

- The **w** command is the continuation of the program.

Enhanced Operational semantics

Given a program $P = \langle c, H, L \rangle$ and an initial enhanced memory $M \in \mathcal{M}_{Var(c)}$, the rules define a transition system $C(P, M)$, which is the enhanced semantics of the program.

We assume that the program starts with a low security environment.

The initial state of $C(P, M)$ is: $\langle c^l, M \rangle$.

An example

Let be given the program $P1\langle c, H, L\rangle$, with $H = \{y\}$ and $L = \{x\}$. Assume the initial enhanced memory $M(x) = (1, l)$ and $M(y) = (2, h)$.

$$P1 = \langle \texttt{if y = 0 then x := 0 else x := 1; skip}, \{\texttt{y}\}, \{\texttt{x}\}\rangle$$

The memory in the final state of the transition system is not safe for $P1$, because the security level of x is h.

The transition system is the following ($H = \{y\}$ and $L = \{x\}$):

$$\langle(\texttt{if y = 0 then x := 0 else x := 1; skip})^l, [x : (1, l), y : (2, h)]\rangle$$
$$\downarrow_{if_{false}}$$
$$\langle(\texttt{x := 1})^h; (\texttt{skip})^l, [x : (1, h), y : (2, h)]\rangle$$
$$\downarrow_{Ass}$$
$$\langle(\texttt{skip})^l [x : (1, h), y : (2, h)]\rangle$$
$$\downarrow_{Skip}$$
$$\langle[x : (1, h), y : (2, h)]\rangle$$

An example

Let be given the program $P2\langle c, H, L\rangle$ with $H = \{y, z\}$ and $L = \{x\}$.

What happen if the value of $\mathtt{x}$ is equal to 1 in the initial memory?

Consider the execution starting from the initial memory:
$M(\mathtt{x}) = (1, l)$, $M(\mathtt{y}) = (2, h)$ and $M(\mathtt{z}) = (0, h)$.

$P2 = \langle \mathtt{if\ x = 1\ then\ y := x\ else\ z := 1;\ \ x := y}, \{\mathtt{y}, \mathtt{z}\}, \{\mathtt{x}\}\rangle$

The assignment $\mathtt{y:=x}$ assigns a low value to $\mathtt{y}$.
Thus the assignment $\mathtt{x:=y}$ assigns a low value to $\mathtt{x}$.
The memory in the final state is safe for $P2$.

An example

The transition system is the following ($H = \{y, z\}$ and $L = \{x\}$):

$$\langle (\texttt{if x = 1 then y := x else z := 1})^l; (\texttt{x := y})^l, [x : (1, l), y : (2, h), z : (0, h)] \rangle$$
$$\downarrow_{if_{true}}$$
$$\langle (\texttt{y := x})^l; (\texttt{x := y})^l, [x : (1, l), y : (2, h), z : (0, h)] \rangle$$
$$\downarrow_{Ass}$$
$$\langle (\texttt{x := y})^l, [x : (1, l), y : (1, l), z : (0, h)] \rangle$$
$$\downarrow_{Ass}$$
$$\langle [x : (1, l), y : (1, l), z : (0, h)] \rangle$$

An example

What happen if the value of $\mathtt{x}$ is equal to 0 in the initial memory?

$$\langle (\mathtt{if\ x = 1\ then\ y := x\ else\ z := 1})^{l};\ (\mathtt{x := y})^{l}, [x : (0, l), y : (2, h), z : (0, h)]\rangle$$
$$\downarrow_{If_{false}}$$
$$\langle (\mathtt{z := 1})^{l}; (\mathtt{x := y})^{l}, [x : (0, l), y : (2, h), z : (0, h)]\rangle$$
$$\downarrow_{Ass}$$
$$\langle (\mathtt{x := y})^{l}, [x : (0, l), y : (2, h), z : (1, l)]\rangle$$
$$\downarrow_{Ass}$$
$$\langle [x : (2, h), y : (2, h), z : (0, h)]\rangle$$

the memory in the final state is not safe for $P2$ (in the final state $\mathtt{x}$ holds a high value)

Correctness

I-security. Given an assignment I of value to the low variables, a program P is I-secure if for any possible value of the high variables, when the program terminates, the value of the low variables does not change.
There is no interference between the high and the low variables

Safe memory. We introduce the definition of a memory safe for a program.

Definition (safe memory)

Given a program $P = \langle c, H, L \rangle$ and an assignment I of value to the low variables, an enhanced memory $M \in \mathcal{M}_{Var(c)}$ is *safe* for P if and only if each low variable of P holds a low value in M.

The enhanced semantics guarantees **I-security**.

Theorem.
If the final memory obtained by executing a program starting from a enhanced initial memory M is safe, then the program is I-secure, where I is the assignment of value to the low variables in M.

Why Impl?

$P = \langle(\texttt{if y} = 1 \texttt{ then x} := 2 \texttt{ else z} := 1), \{y, z\}\{x\}\rangle$.
$H = \{y, z\}$ and $L = \{x\}$. Initial assignment to the low variable: $x = 0$.

Without Impl

What happen if $y = 2$?

$$\langle(\texttt{if y} = 1 \texttt{ then x} := 2 \texttt{ else z} := 1)^{l}, [x : (0, l), y : (2, h), z : (0, h)]\rangle$$
$$\downarrow_{If_{true}}$$
$$\langle(\texttt{z} := 1)^{h}, [x : (0, l), y : (2, h), z : (0, h)]\rangle$$
$$\downarrow_{Ass}$$
$$\langle[x : (0, l), y : (2, h), z : (1, h)]\rangle$$

Why Impl?

What happen if $y = 1$?

$$\langle (\texttt{if y = 1 then x := 2 else z := 1})^{l}, [x : (0, l), y : (2, h), z : (0, h)] \rangle$$
$$\downarrow_{If_{false}}$$
$$\langle (\texttt{x := 2})^{h}, [x : (0, l), y : (2, h), z : (0, h)] \rangle$$
$$\downarrow_{Ass}$$
$$\langle [x : (2, l), y : (2, h), z : (1, h)] \rangle$$

Data leakage occurs only if y is 1.

Why Impl?

$H = \{y, z\}$ and $L = \{x\}$. Initial assignment I to the low variable $x = 0$.

With Impl

The security level of all variables assigned in the implicit flow is upgraded!!!

What happen if $y = 2$?

$$\langle (\texttt{if y = 1 then x := 2 else z := 1})^{I}, [x : (0, l), y : (2, h), z : (0, h)] \rangle$$
$$\downarrow_{if_{true}}$$
$$\langle (\texttt{z := 1})^{h}, [x : (0, l), y : (2, h), z : (0, h)] \rangle$$
$$\downarrow_{Ass}$$
$$\langle [x : (0, h), y : (2, h), z : (1, h)] \rangle$$

Impl. Guarantees that if a concrete execution starting from an assignment I of value to low variables results in a safe memory, there are no possible leakages, independently of any assignment to the high variables.

Abstract Operational semantics

The enhanced transition system could be infinite, because there are infinitely many memories.

The enhanced operational semantics cannot be used as a static analysis tool.

The purpose of abstract interpretation (or abstract semantics) is to correctly approximate the enhanced semantics of all executions in a finite way.

Abstract Operational semantics

- The first step in the construction of the abstract semantics is the definition of the abstract domains.
- The nodes of the abstract transition system contain abstractions of states.
- In particular, in our abstract semantics each enhanced value, composed of a pair of a value and a security level, is approximated by considering only its security level.
- As a consequence, when dealing with conditional or iterative commands, the abstract transition system has multiple execution paths due to the loss of precision of abstract data.

Abstract Operational semantics

Let α the abstraction function. The abstract semantics:

- abstracts enhanced values into their security level: $\alpha(k, \sigma) = \sigma$
- uses the same rules of the enhanced semantics on the abstract domains. The transition relation of the abstract semantics is denoted by $\longrightarrow^{\natural}$.
- Both rules for *if* are always applied, since *true* and *false* are both abstracted to "$\cdot$"
- Both rules for *while* are always applied, since *true* and *false* are both abstracted to "$\cdot$"

Abstract Operational semantics

The abstract semantics:

- let A(P, $M^\sharp$) be abstract transition system for P
  - finite
  - multiple paths
  - each path of C(P, M) is correctly abstracted onto a path of A(P, $M^\sharp$)

Abstract transition system

$P2 = \langle \texttt{if x = 1 then y := x else z := 1;x := y}, \{\texttt{y}, \texttt{z}\}, \{\texttt{x}\}\rangle$
$M^\sharp(\texttt{x}) = (l)$, $M^\sharp(\texttt{y}) = (h)$ and $M^\sharp(\texttt{z}) = (h)$

$$\langle (\texttt{if x = 1 then y := x else z := 1})^l; (\texttt{x := y})^l, [x : (l), y : (h), z : (h)]\rangle$$

$$\downarrow^\sharp \qquad\qquad \downarrow^\sharp$$

$$\langle (\texttt{y := x})^l; (\texttt{x := y})^l, [x : (l), y : (h), z : (h)]\rangle \qquad \langle (\texttt{z := 1})^l; (\texttt{x := y})^l, [x : (l), y : (h), z : (h)]\rangle$$

$$\downarrow^\sharp \qquad\qquad \downarrow^\sharp$$

$$\langle (\texttt{x := y})^l, [x : (l), y : (l), z : (h)]\rangle \qquad \langle (\texttt{x := y})^l, [x : (l), y : (h), z : (l)]\rangle$$

$$\downarrow^\sharp \qquad\qquad \downarrow^\sharp$$

$$\langle [x : (l), y : (l), z : (h)]\rangle \qquad \langle [x : (h), y : (l), z : (l)]\rangle$$

Abstract Operational semantics

Let $P = \langle c, H, L \rangle$ and $M^{\natural} \in \mathcal{M}^{\natural}_{Var(c)}$ with
$M^{\natural}(x) = l, \forall x \in L$ and $M^{\natural}(x) = h, \forall x \in H$.

If for each final state $\langle M_f^{\natural} \rangle$ of $A(P, M^{\natural})$, it holds that $M_f^{\natural}(x) = l$, then Secure Information Flow property is satisfied.

The memory in each final state of the abstract transition system is safe for P.

Secure Information Flow

For each program $P = \langle c, H, L \rangle$ and abstract memory $M^\natural \in \mathcal{M}^\natural_{Var(c)}$, $A(P, M^\natural)$ is finite.

To check if a program is secure, we build the abstract transition system and examine all final states.

For example, the abstract transition system of the program $P2$ has two final states: $\langle [\mathbf{x} : \mathbf{l}, y : l, z : h] \rangle$ and $\langle [\mathbf{x} : \mathbf{h}, y : h, z : l] \rangle$. Secure Information Flow property is not satisfied because $x \in L$ and $x = h$ in the second state.

Secure Information Flow

Secure Information Flow (SIF). A program P has secure information flow if in each final state of A(P), each $x : \sigma$ holds a value $\tau \sqsubseteq \sigma$.

This approach is based on a finite-state transition system and thus has the advantage of being fully automatic.

Exercise

Apply the standard operational semantics, the enhanced and the abstract operational semantics to the following program

$P = \langle c1; c2; \cdots ; c5, \{x\}, \{y, z\}\rangle$
with $m(x) = 2$, $m(y) = 7$, $m(z) = 3$

```
c1:   z := 0;
c2:   while    (x > 0)
c3:            y:=y*10;
c4:            x:=x-1;
c5:   z:=y;
```

Does P satisfy SIF? Why?

Example: PIN cloner

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

$P = \langle Pincloner, H, L \rangle$ $H = \{PIN\}$ $L = \{clone\}$

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

PIN cloner

- 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 program has gained the access to the user private data, the access control mechanism is not able to reveal the illicit flow

Other properties

- **Secure Information Flow**. Variables are analysed at the termination of the program. Other possibilities:
  - check the variables at given program points
  - check properties of execution paths

- **Termination Agreement**. A program P satisfies Termination Agreement if it is not possible to leak high information by observing the termination of the program
  while (h $>$ 0) do skip

- **Timing Agreement**. A program P satisfies Timing Agreement if it is not possible to leak high information by observing the number of instructions executed

Correctness of the analysis

In the definition of the abstract semantics, we have applied *Abstract Interpretation* [4].

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

Java bytecode.

Instruction set (JVML0)

`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 the variable x onto the stack
`store` x	pop off the stack and store the value into variable 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 return address $p+1$ onto the operand stack
`ret` x	jump to the address stored in x
`halt`	stop

Basics of information flow

The language has an operand *stack*, a *memory* containing the local variables, simple arithmetic instructions and conditional/unconditional jumps.

In the instructions, x ranges over a set var of local variables and op over a set of binary arithmetic operations (`add`, `sub`, ..). Note that the language supports subroutine calls via the `jsr` and `ret` instructions.

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

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

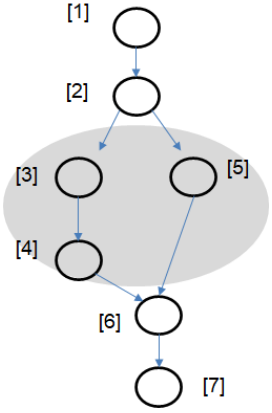
implicit flow

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 stored into 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?
The implicit flow terminates at [6], which is the first instruction that is common to both branches. It is the first instruction that is not executed under the implicit flow.

Basics of information flow

Definition (control flow graph)

Given a program c, the *control flow graph* of the program is the directed graph (V, E), where $V = \{1, .., \sharp c + 1\}$ is the set of nodes; and $E \subseteq V \times V$ contains the edge (i, j) if and only if (a) the instruction at address j can be immediately executed after that at address i; or (b) $c[i] = \texttt{halt}$ and $j = \sharp c + 1$.

The node $\sharp c + 1$ is the final node of the graph and does not correspond to any instruction.

Definition (postdomination)

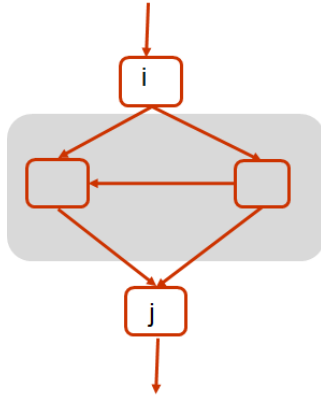
Let i and j be nodes of the control flow graph of a program. We say that node j *postdominates* i, denoted by j *pd* i, if $j \neq i$ and j is on every path from i to the final node.
We say that node j *immediately postdominates* i, denoted by $j = ipd(i)$, if j *pd* i and there is no node r such that j *pd* r *pd* i.

Implicit flow

What about nested control instructions?

immediate postdominator of i:
the first node belonging to
all paths from i: ipd(i) = j

Nested implicit flows:
the innest implicit flow is the
implicit flow that terminate first



IPD stack

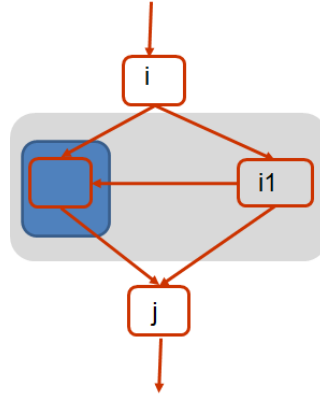
when executing an instruction, the ipd stack maintains information on the open implict flows. IPD stack is updated any time a control instruction is entered and any time a control instruction terminates

Execution of instructions

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

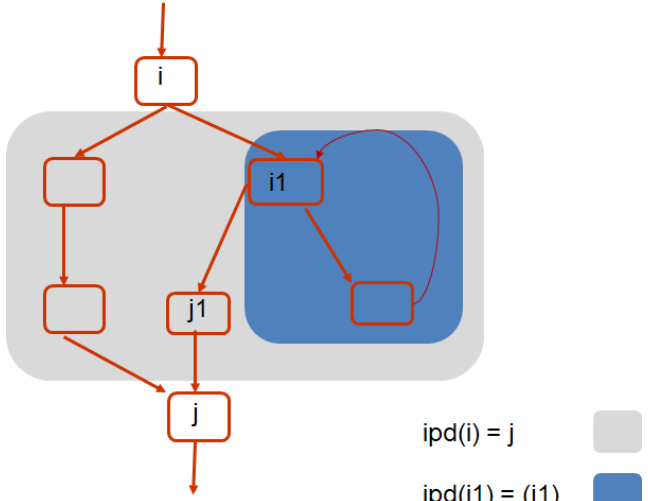
pc:	c[pc]	Stack of ipd
...	...	λ
i:	control instr.	ipd(i)
i1:	control instr.	ipd(i1) ipd(i)
j:	top of ipd stack	ipd(i)
j:	top of ipd stack	λ

grey: control region of i
blue: control region of i1



Implicit flow

Nested control instructions



Implicit 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

Secure Information flow

$H = \{x\}$ $L = \{y\}$

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

A program $P = \langle c, H, L \rangle$ 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

A program is a sequence c of instructions, numbered starting from address 1;
$\forall i \in \{1, \cdots, \sharp c\}$, $c[i]$ is the instruction at address i.

$Var(c)$ the variable names occurring in c
We assume that:

- a program is always executed starting from the instruction $c[1]$ and with an empty operand stack.
- programs respect the following static constraints checked the Java bytecode Verifier: (i) no stack overflow and underflow occur, and (ii) executions will not jump to undefined addresses.

Concrete semantics

$\mathcal{V} = (\mathcal{V}^{\epsilon} \times \mathcal{L})$ is the domain of concrete values.
Concrete values are pairs (v, σ), where $v \in \mathcal{V}^{\epsilon}$ and $\sigma \in \mathcal{L}$.

$\mathcal{A} = (\mathcal{A}^{\epsilon} \times \mathcal{L})$ concrete domain of addresses. Note that also addresses need to be annotated, since the decision on the address to jump to, can be made depending on high information.

For each $x \in \mathit{var}$, $\mathcal{M}_X = X \to (\mathcal{V} \cup \mathcal{A})$ is the domain of concrete memories, ranged over by $M, M', \cdots$ and $\mathcal{S} = (\mathcal{V} \cup \mathcal{A})^{\star}$ are the concrete operand stacks, ranged over by $S, S', \cdots$.

Concrete semantics

The domain of concrete states is $\mathcal{Q} = \mathcal{L} \times \mathcal{A}^{\epsilon} \times \mathcal{M} \times \mathcal{S} \times ((\mathcal{A}^{\epsilon})^{\star} \cup \{0\})$.

Each state is a configuration the state variables

$\langle ENV, PC, MEM, STACK, IPD \rangle$

where *ENV* is the environment and contains a security level, *PC*, *MEM* and *STACK* are the program counter, the memory and the operand stack, respectively, and *IPD* is the IPD stack used to handle high implicit flow, as explained below.

Concrete semantics

$\mathcal{Q} = \mathcal{L} \times \mathcal{A}^{\epsilon} \times \mathcal{M} \times \mathcal{S} \times ((\mathcal{A}^{\epsilon})^{\star} \cup \{0\})$
$\langle \sigma, PC, M, S, \rho \rangle$

σ	security environment
PC	program counter
M	memory
S	operand stack $(k, \tau) \cdots (k', \tau')$
ρ	ipd stack $(j, \sigma) \cdots (j', \sigma')$

IPD Stack ρ:

- if $\rho = (j1, \sigma_1) \cdots (jn, \sigma_n)$
there are n open implicit flows
$j1$ holds the address where first implicit flow terminates
σ_1 holds the level of the environment that must be restored

- if $\rho = \lambda$
there are no open implicit flow

Concrete semantics

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

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

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

Concrete semantics

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

i is the ipd of a control instruction

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

the execution continues at instruction j

Concrete semantics

$$\textbf{if}_{true} \quad \frac{c[i] = \mathit{if}\ j, k \neq 0, \mathit{not\_top}(i, \rho)}{\langle \sigma, i, M, (k, \tau) \cdot S, \rho \rangle \longrightarrow \langle \sigma \cup \tau, j, \mathit{up}(M), \mathit{up}(S), (\mathit{ipd}(i), \sigma) \cdot \rho \rangle}$$

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 ($\sigma \cup \tau$)
($\mathit{ipd}(\mathit{pc}), \sigma$) is pushed on the ipd stack r

$\mathit{up}(M)$ upgrades the value of the variables assigned in the scope of the implicit flow beginning at i

$\mathit{up}(S)$ upgrades all elements in the stack

Concrete semantics

$$\textbf{if}_{false} \quad \frac{c[i] = \mathit{if}\ j, \mathit{not\_top}(i, \rho)}{\langle \sigma, i, M, (0, \tau) \cdot S, \rho \rangle \longrightarrow \langle \sigma \cup \tau, i + 1, \mathit{up}(M), \mathit{up}(S), (\mathit{ipd}(i), \sigma) \cdot \rho \rangle}$$

if : assume condition non-zero

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 ($\sigma \cup \tau$)
($\mathit{ipd}(\mathit{pc}), \sigma$) is pushed on the ipd stack r

$\mathit{up}(M)$ upgrades the value of the variables assigned in the scope of the implicit flow beginning at i

$\mathit{up}(S)$ upgrades all elements in the stack

Concrete semantics

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

the address (i+1) is added to the operand stack with security level the level of the environment

Concrete semantics

$$\textbf{ret}_{\tau \sqsubseteq \sigma} \quad \frac{c[i] = \mathit{ret}\ x, M(x) = (j, \tau), \tau \sqsubseteq \sigma, \mathit{not\_top}(i, \rho)}{\langle \sigma, i, M, S, \rho \rangle \longrightarrow \langle \sigma, j, M, S, \rho \rangle}$$

$$\textbf{ret}_{\tau \not\sqsubseteq \sigma} \quad \frac{c[i] = \mathit{ret}\ x, M(x) = (j, \tau), \tau \not\sqsubseteq \sigma, \mathit{not\_top}(i, \rho)}{\langle \sigma, i, M, S, \rho \rangle \longrightarrow \langle (\sigma \cup \tau), j, \mathit{up}(M), \mathit{up}(S), (\mathit{ipd}(i), (\sigma \cup \tau)) \cdot \rho \rangle}$$

an implicit flow begins if the level of the address is higher than the level of the environment

Abstract interpretation

`ret` *x* instruction.
take into account the security level of the address stored in *x*.

Note that having only two security levels simplifies the semantics. In fact, if we consider whatever number of levels, *IPD* would be a stack of addresses, instead of a single address. In our case, a high `if` that depends on another high `if` is

already in a high region and the region terminates at the ipd of the outermost `if`.

For the same reason, the upgrading of environment, memory and stack, and the modification of *IPD* is performed only when an `if` (`ret` *x*) instruction is executed in the low environment, and with a high value on the top of the stack (resp. a high address stored in *x*).

Abstract 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 (if_{true} and if_{false})

Moreover, since addresses maintain their identity, also all possible return points are explored.

A(P) : abstract transition system for P finite multiple path each path of C(P) is correctly abstracted onto a path of A(P)

Abstract interpretation

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

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

Abstract interpretation

If we ignore information on security, then the concrete semantics is isomorphic to the standard semantics of the language. The concrete semantics has an only extra case (case $i = i'$ concerning the handling of *IPD*).

The abstract semantics is finite. In fact, since security levels, environments and abstract values are finite, then abstract memories are finite too. Abstract operand stacks are finite because we assume stack boundedness.

Example: CP

x:(0,H) y:(1,L) ipd(2) = 5, ipd(6)=10 $\langle ENV, PC, [M(x), M(y)], Stack, IPDstack \rangle$

```
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**

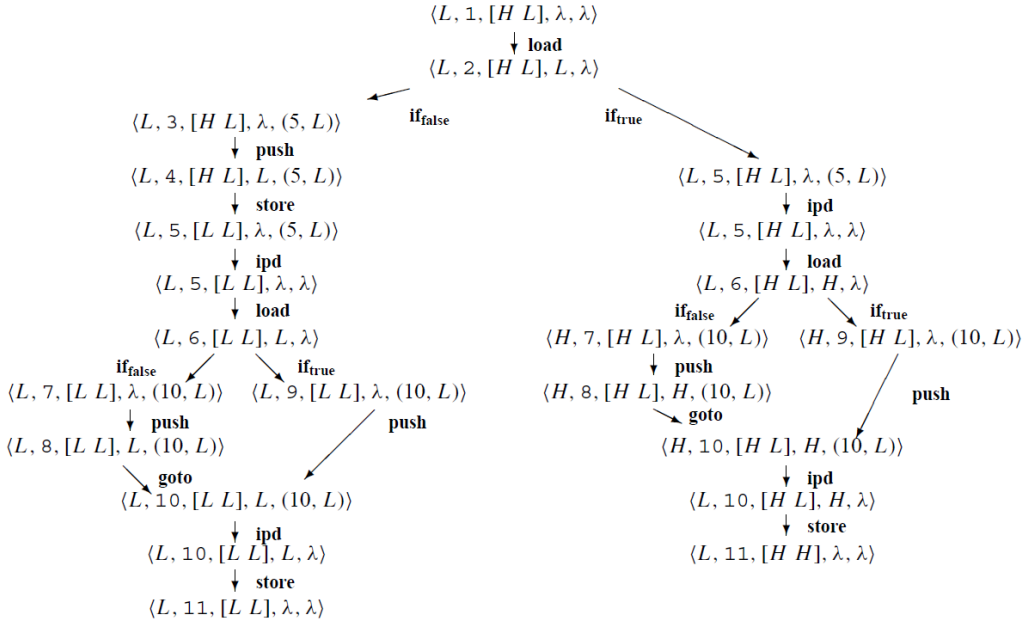
$\langle L, 10, [(0, H)(1, L)], (1, H), \lambda \rangle$

↓ **store**

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

Example: AP

$M^\sharp(x) = H$ $M^\sharp(y) = L$



Abstract semantics and properties

The following theorems relate the abstract semantics with the above properties.

Theorem

P satisfies SIF if for each state of $A(P)$ such that $c[PC] = \texttt{halt}$, then $\forall x \in L, MEM[x] = l$ or $MEM[x] = (i, l)$ for some i.

Theorem

P satisfies TERM if every state of $A(P)$ such that $ENV = h$ does not belong to a cycle.

Theorem

P satisfies TIME if:

- all paths in $A(P)$ starting from a state satisfying $STACK[1] = h$ and $PC = i$ where $c[i] = \texttt{if}$ and ending with a state satisfying $PC = ipd(i)$ have the same length.

- all paths in $A(P)$ starting from a state satisfying $PC = i$ and $MEM[x] = (j, h)$ where $c[i] = \texttt{ret}\ x$ and ending with a state satisfying $PC = ipd(i)$ have the same length.

Abstract semantics and properties

To check *SIF* it suffices to examine the final states of the abstract semantics, and, in particular, to check that in these states the low variables hold low values and the stack contains only low values.

TERM can be controlled by checking that no instruction is executed more than once under a high environment.

To ensure *TIME*, the branches starting from an `if` instruction at address i with an high condition (the value on top of the stack is h) must have the same length until $ipd(i)$ is reached. A similar condition is stated for `ret` instructions with high return address.

Examples

Consider the following code with $L = \{\mathtt{x}\}$ and $H = \{\mathtt{y}\}$. The code has a non-secure implicit flow. It corresponds to the program:

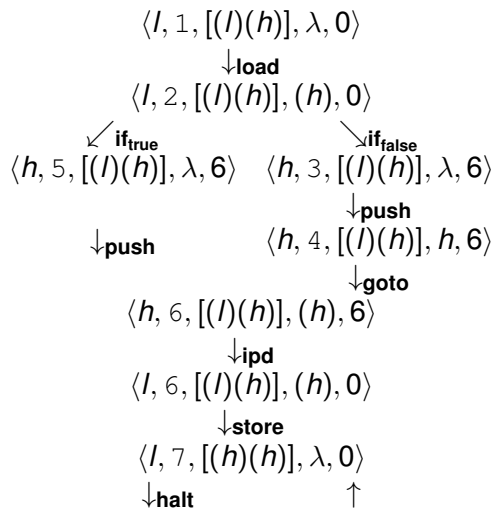
```
if  y=0  then  x:=1  else  x:=0.
```

$A(P)$ does not satisfy *SIF* nor *TIME*, while *TERM* is satisfied.

A program not satisfying SIF

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

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



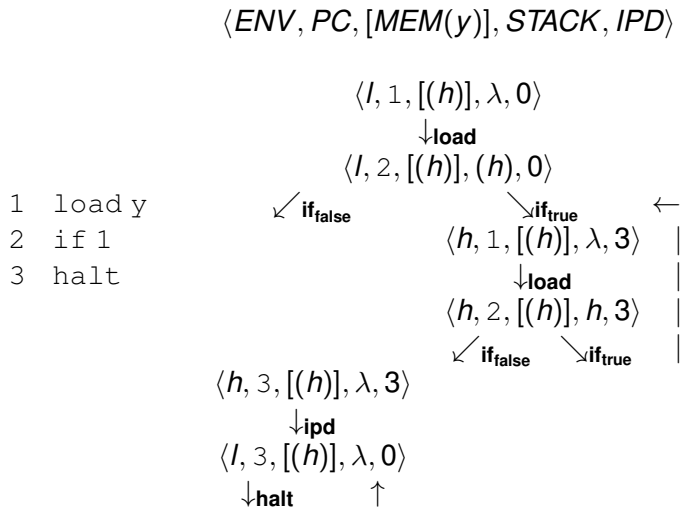
Consider the following code with $L = \{\mathtt{x}\}$ and $H = \{\mathtt{y}\}$. The code is an example of violation of termination agreement. This program terminates depending on the value non-zero or zero of the high security level variable $\mathtt{y}$.

It corresponds to the high level program:

```
while (y) do skip.
```

Note that it satisfies *SIF*, but not *TERM*: there is a cycle including states with $ENV = h$.

A program not satisfying *TERM*



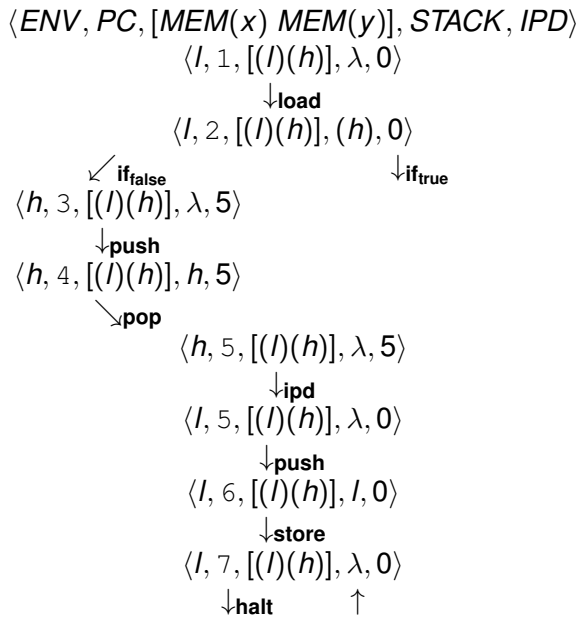
The following code is an example of not secure program due to to a timing channel. The number of steps of the program depends on the value of the high security level variable `y`.

When the program terminates the low variable `x` always holds `1`.

AP satisfies *SIF* and *TERM*, but it does not satisfy *TIME*.

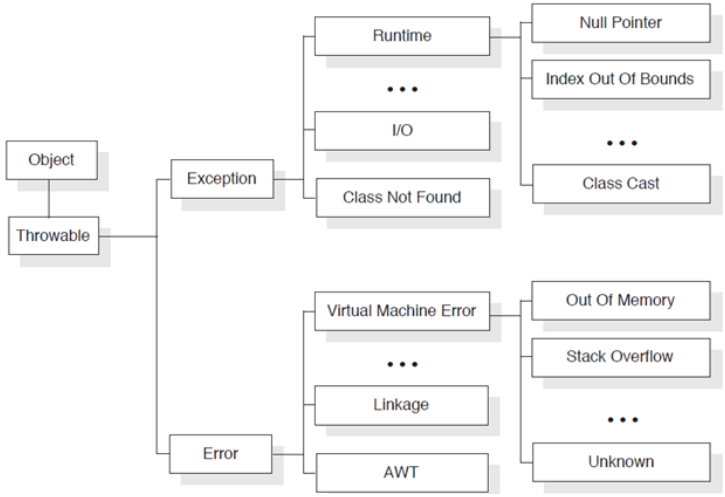
A program not satisfying *TIME*

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



Information flow through exceptions

Java Exception Hierarchy (incomplete)



Java code

```
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 }
```

The bytecode

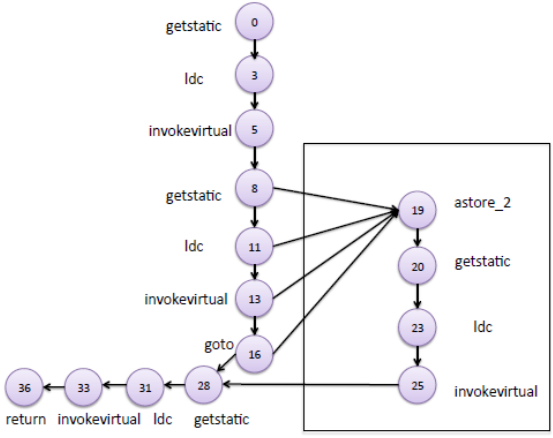
```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1 //Method java/lang/Object."<init>":()V
       4: return

  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
}
```

Extended control flow graph



Extended control flow graph of the bytecode

PinCloner applet

Pin_file and Clone_file are the input and the output files
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. After every character has read, it will be written in a the public file by the handler of the exception.
PINCloner application clones the characters of the Pin_file by throwing different

kind of exceptions depending on the value read (NullPointerException and ArithmeticException).
The exceptions are thrown directly with a throw statement

PinCloner Java code

```
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 Java bytecode

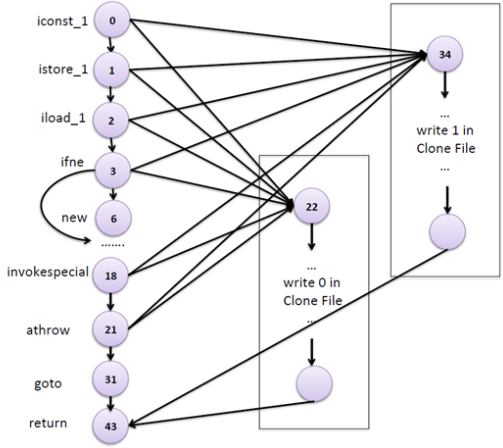
```
Compiled from "PinCloner.java"
public class PinCloner {
  public PinCloner();
    Code:
       0: aload_0
       1: invokespecial #1 // Method java/lang/Object."<init>":()V
       4: return

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

  public void PinCloner();
    Code:
       0: iconst_1
       1: istore_1
       2: iload_1
       3: ifne          14
       6: new           #2 // class java/lang/ArithmeticException
       9: dup
      10: invokespecial #3 // Method java/lang/ArithmeticException."<init>":()V
      13: athrow
      14: new           #4 // class java/lang/NullPointerException
      17: dup
      18: invokespecial #5 // Method java/lang/NullPointerException."<init>":()V
      21: athrow
      22: ......          // write 1 in Clone_File
      31: goto          43
      34: ...             // write 1 in Clone_File
      43: return

    Exception table:
       from    to  target type
           0    22    22   Class java/lang/ArithmeticException
           0    22    34   Class java/lang/NullPointerException
}
```

Extended control flow graph



control region of instruction i: set of instructions under the implicit flow of instruction i: (set of instruction on any path from i to ipd(i), ipd(i) excluded)

PinCloner app

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.

The application violates the secure information flow because high security data are written on a public file and our methodology successfully detects this data leakage.

References

[1] D. Volpano, G. Smith, C. Irvine. A sound type system for secure flow analysis. Journal of Computer Security, 4(3), 1996, pp. 167-187.

[2] Rajeev Joshi, K.Rustan M.Leino. A semantic approach to secure information flow Science of Computer Programming, Volume 37, 2000.
High compexity in space and time

[3] Roberto Barbuti, Cinzia Bernardeschi, Nicoletta De Francesco Abstract interpretation of operational semantics for secure information flow. Inf. Process. Lett. 83(2): 101-108 (2002)

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