Analysis of programs

Proving properties of programs can be automated BUT we must abstract from the exact behaviour

- Correctness of the abstraction with respect to the property we are considering.
- We consider abstract memories. For example,
  - odd/even numbers
  - signs (+, -)
  - ....

  We execute the commands of the program on abstract memories.
  We define an abstract operational semantics of the language.

.

Confidentiality property

- Data leakage
- Security policy
- Information flow in programs
- Examples of illegal flow of information

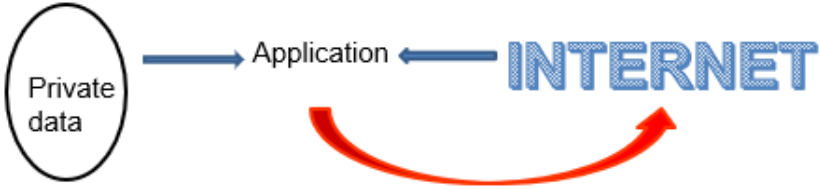
GENERAL DATA PROTECTION REGULATION(GDPR) - UE 2016/679

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

Limit of Firewall and Access control mechanisms



Application authorized to access private data
Application authorized to access internet

Control on the information sent on the internet!!!!!

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

Certification of applications 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."

Data leakage

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

Information flow in programs

Modular programming

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

a security policy that allows the classification of data and users based on a system of hierarchical security levels.

Lattice.
Let be given a set $\mathcal{S}$ and order relation $\sqsubseteq$ on $\mathcal{S}$.
($\mathcal{S}$, $\sqsubseteq$) is a lattice if every pair of elements in $\mathcal{S}$ has both a greatest lower bound (glb, $\sqcap$) and a least upper bound (lub, $\sqcup$).

$\mathcal{S} = \{l, h\}$, with $l \sqsubset h$

Public data: l Private data: h

Inputs and outputs are classified as either low sensitive (public) or high sensitive (private).

the security domain private is non-interfering with domain public if no input by private can influence subsequent outputs 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 and the activities (if any) of the high user.

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

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)

In both cases 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.

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

1. **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 = $\{h, l\}$. l:= h typing error h:= l correct

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

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

Secure information flow checking

[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, Issues 1â3, May 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)

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

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

- concrete instrumented semantics recording the information flow (collecting semantics)
- abstract semantics taking only what concerns the information flow
- correctness of the abstraction
- model checking

Standard Operational semantics

Given a program $P = \langle c, H, L \rangle$ and an initial memory $m \in \mathcal{M}^{\epsilon}_{Var(c)}$, we denote by $E(P, m)$ the transition system defined by $\longrightarrow^{\epsilon}$ starting from the initial state $\langle c, m \rangle$.

The semantics of programs is given by means of a transition system.
The semantic rules define a relation $\longrightarrow^{\epsilon} \subseteq \mathcal{Q}^{\epsilon} \times \mathcal{Q}^{\epsilon}$, where $\mathcal{Q}^{\epsilon}$ is a set of states. Each state is either a pair $\langle c, m \rangle$ of a command and a memory, or a single memory $\langle m \rangle$.

Actually, since the program is deterministic, there exists at most one final state, i.e. a state $\langle m \rangle$ for some memory m.

Concrete operational semantics

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

The concrete semantics is an instrumented semantics which:

- Handles values (k, σ) annotated with a security level $(k = 0, 1, 2 \cdots)$.
- Executes instructions under a security environment σ.
- C(P, M): concrete transition system for P

(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.

$(e)^{\sigma}$ and $(c)^{\sigma}$
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.

Concrete Operational semantics

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

Let $(\mathcal{S}, \sqsubseteq)$, with $\mathcal{S} = \{l, h\}$, be a lattice of security levels, ordered by $l \sqsubset h$, where $\sqcup$ denotes the least upper bound between levels.
A program P is a triple $\langle c, H, L \rangle$
$c \in com$
H are the high variables of P
L are the low variables of P
$H \cup L = Var(c)$ and $H \cap L = \emptyset$

Concrete Operational semantics

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

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

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

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

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

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

$$\textbf{Seq}_1 \quad \frac{\langle c_1^{\sigma}, M\rangle \longrightarrow \langle M'\rangle}{\langle c_1^{\sigma}; w, M\rangle \longrightarrow \langle w, M'\rangle} \qquad \textbf{Seq}_2 \quad \frac{\langle c_1^{\sigma}, M\rangle \longrightarrow \langle w', M'\rangle}{\langle c_1^{\sigma}; w, M\rangle \longrightarrow \langle w'; w, M'\rangle}$$

Mod finds the set of variables *mod*ified in a command i.e. those which are on the left of an assignment
Impl possibly upgrades the security level of the values of the variables to take into account an *impl*icit flow.

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.

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, is executed in the memory $Impl(M, Mod(c_1) \cup Mod(c_2), \tau)$ 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

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

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

We assume that the program starts with a low security environment: the initial state of $C(P, M)$ is $\langle c^l, M \rangle$.

Correctness

We introduce the definition of a memory *safe* for a program: given a program $P = \langle c, H, L \rangle$, a concrete 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.

An example

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

and the concrete memory M with $M(\texttt{x}) = (1, l)$ and $M(\texttt{y}) = (2, h)$.

The memory in the final state of the concrete transition system is not safe for $P1$, since the security level of $\texttt{x}$ is h:

$$\langle (\texttt{if y} = 0 \texttt{ then x} := 0 \texttt{ else x} := 1)^{l}, [x : (1, l), y : (2, h)] \rangle$$
$$\downarrow$$
$$\langle (\texttt{x} := 1)^{h}, [x : (1, h), y : (2, h)] \rangle \ \texttt{Impl() sets x to h}$$
$$\langle [x : (1, h), y : (2, h)] \rangle$$

An example

$P2 = \langle \texttt{if x = 1 then y := x else skip;x := y}, \{\texttt{y}\}, \{\texttt{x}\}\rangle$ with $M(\texttt{x}) = (1, l)$ and $M(\texttt{y}) = (2, h)$
The concrete transition system is the following:

$$\langle (\texttt{if x = 1 then y := x else skip})^l; (\texttt{x := y})^l, [x : (1, l), y : (2, h)]\rangle$$
$$\downarrow$$
$$\langle (\texttt{y := x})^l; (\texttt{x := y})^l, [x : (1, l), y : (2, h)]\rangle$$
$$\downarrow$$
$$\langle (\texttt{x := y})^l, [x : (1, l), y : (1, l)]\rangle$$
$$\downarrow$$
$$\langle [x : (1, l), y : (1, l)]\rangle$$

The assignment $\texttt{y:=x}$ assigns a low value to $\texttt{y}$. Thus the assignment $\texttt{x:=y}$ assigns $\texttt{x}$ a low value. The final state is safe for $P2$. $P2$ is secure, with $I(x) = 1$.

An example

Note that $P2$ is not secure for $x = 0$ (in the final state $\mathtt{x}$ holds a high value):

$$\langle (\mathtt{if\ x} = 1\ \mathtt{then\ y} := \mathtt{x\ else\ skip})^{l};\ (\mathtt{x} := \mathtt{y})^{l}, [x : (0, l), y : (2, h)] \rangle$$
$$\downarrow$$
$$\langle \mathtt{skip}^{l}; (\mathtt{x} := \mathtt{y})^{l}, [x : (0, l), y : (2, h)] \rangle$$
$$\downarrow$$
$$\langle (\mathtt{x} := \mathtt{y})^{l}, [x : (0, l), y : (2, h)] \rangle$$
$$\downarrow$$
$$\langle [x : (2, h), y : (2, h)] \rangle$$

Abstract Operational semantics

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

In fact the concrete transition system could be infinite, because there are infinitely many memories.

The purpose of abstract interpretation (or abstract semantics) is to correctly approximate the concrete 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 abstract domains adequately describe sets of values of the concrete ones.
- The abstract semantics is a transition system whose paths represent executions and whose nodes display the program's states.
- The nodes of the abstract transition system contain abstractions of states.

Abstract Operational semantics

In particular, in our abstract semantics each concrete 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 concrete values into their security level: $\alpha(k, \sigma) = \sigma$
- uses the same rules of the concrete 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 "$\cdot$"

Abstract Operational semantics

The abstract semantics:

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

Abstract transition system

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

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

Abstract Operational semantics

Let $P = \langle c, H, L \rangle$ and $M^{\natural} \in \mathcal{M}^{\natural}_{Var(c)}$ with $Low(M^{\natural}) = L$. If for every final state $\langle M'^{\natural} \rangle$ of $A(P, M^{\natural})$, it holds that $M'^{\natural}$ is safe for P, then P is secure.

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.

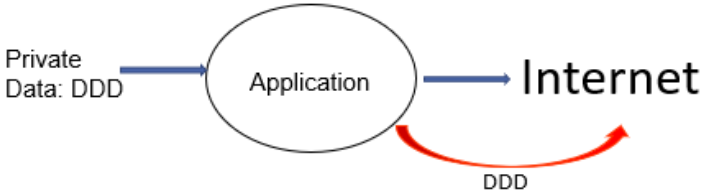
To check if a program is secure, we build the abstract transition system and examine all final states.

For example, the abstract transition system of the program $P2$ of the previous section has two final states: $\langle [x : l, y : l] \rangle$ and $\langle [x : h, y : h] \rangle$, where the second one is not safe for $P2$. Therefore $P2$ is not secure.

Secure Information Flow. 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 can be put at an intermediate level between a syntactic approach and a fully semantic one.
On one side, it is dynamic and thus allows us to be more permissive than a syntactic approach. On the other side, it is based on a finite-state transition system and thus has the advantage of being fully automatic.

Data leakage



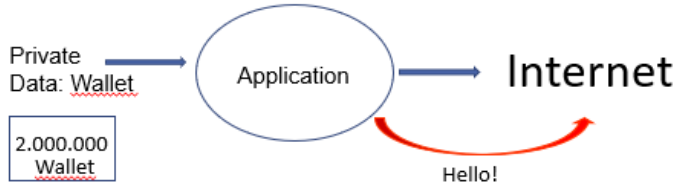
Secure information flow is violated

Application authorized to access private data
Application authorized to access Internet

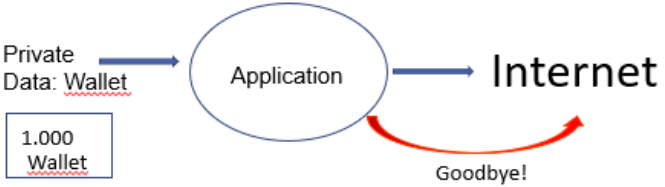
Explicit information flow

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

Data leakage



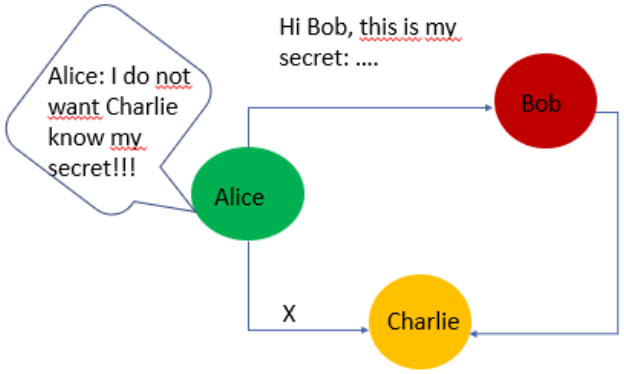
Secure information flow is violated



if Wallet > 1.000.000 then
«Hello!»
else «Goodbye!»

Implicit information flow

Data leakage



Secure information flow is violated

Multilevel security policy

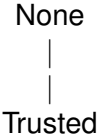
Security lattice $(\mathcal{S}, \sqsubseteq)$
every pair of elements in $\mathcal{S}$ has both a greatest lower bound (glb, $\sqcap$) and a least upper bound (lub, $\sqcup$). Moreover, $\sqsubseteq$ is reflexive and transitive. $\sqsubseteq$ is antisymmetric

Confidentiality: $\mathcal{S} = \{\textit{Public}, \textit{Private}\}$, with $\textit{Public} \sqsubset \textit{Private}$

SIF guarantees that:
information in public variables not depend on information in private variables

Multilevel security policy

Integrity: $\mathcal{S} = \{None, Trusted\}$ with $Trusted \sqsubset None$



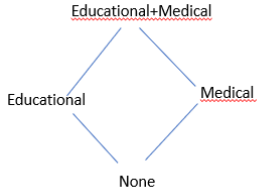
SIF guarantees that:
information in trusted variables not depend on information in not trusted variables

Multilevel security policy

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

$\mathcal{S} = \{None, Educational, Medical, Educational + Medical\}$, with
$None \sqsubset Educational$; $None \sqsubset Medical$;
$Medical \sqsubset Educational + Medical$; $Educational \sqsubset Educational + Medical$

least upper bound ($\sqcup$): $Educational \sqcup Medical = Educational + Medical$



Multilevel security policy

Let u_i represents sensitive information of user i.

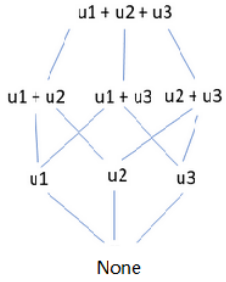
$\mathcal{S} = \{None, u1, u2, u3, u1 + u2, u1 + u3, u2 + u3, u1 + u2 + u3\}$ with

$None \sqsubset ui$;

$u_i \sqsubset ui + u_j, j \neq i$;

$u_i + u_j, j \neq i \sqsubset u1 + u2 + u3$

least upper bound ($\sqcup$): $u1 \sqcup u2 = u1 + u2$



Exercise

Apply the standard operational semantics, the concrete 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?

A variable can represent

- input/output file
- a port for network connections
- ....

Other properties

SIF analyses the variables when the program terminates.

Other possibilities:

- check the variables at given program points
- check properties of execution paths

Other properties

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

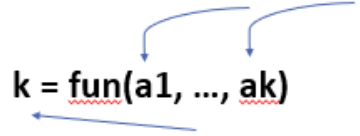
Data propagation

Propagation caused by global variables

type x;
type f(· · ·);
type g(· · ·);

Propagation caused by actual parameter/ return of a function

```
type fun (type x1, ...,  type xn) {
        ........
    return expr;
}
```



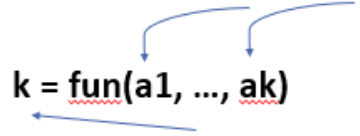
Data propagation

Propagation caused by global variables

type x;
type f(· · ·);
type g(· · ·);

Propagation caused by actual parameter/ return of a function

```
type fun (type x1, ...,  type xn) {
        ........
    return expr;
}
```



The security context

- For each global variable: the highest level of data stored

$$var\_name : \sigma$$

- For each function: the highest level of input/output parameters, return and the security environment of each invocation

$$fun\_name(param_1, \cdots, param_n) : return, calling\_evironment$$

$$fun\_name(\sigma_1, \cdots, \sigma_n) : \sigma, \sigma'$$

The security context

Secure information flow studied by using a **security context**

- For each global variable: the highest level of data stored

$$var\_name : \sigma$$

- For each function: the highest level of input/output parameters, return and the security environment of each invocation

$$fun\_name(param_1, \cdots, param_n) : return, calling\_evironment$$

$$fun\_name(\sigma_1, \cdots, \sigma_n) : \sigma, \sigma'$$

Iterative analysis

Iterative analysis until **fixpoint** is reached

A: security context
R: set of all functions
EXEC: the abstract execution interpreter of a function

The algorithm

A:= A^0
T:= R
while ($T \neq \emptyset$)
    select f $\in T$
    T := T - {f}
    $A' := EXEC(f, A)$
    if ($A' \neq A$)
        A := A';
        T:=R

Iterative analysis

Each function is executed starting from the abstract memory and the context file, and applying the abstract rules

The analysis terminates, since security levels in the context file can only be upgraded, and the number of security levels is finite

Context file

During the analysis,

- for each variable, the context file maintains the maximum security level of data stored in the variable
- for each function, the context file maintains how the function is called in terms of the maximum level of the calling environment, the actual parameter and the return

Iterative analysis

During the analysis,

- functions are analysed one at a time;
- the context file is update accordingly;
- at the end of the analysis of a function, if the context file is changed, all functions must be re-analysed starting from the new context file.
- the analysis terminates when, stating from a context file, all functions are analysed and the context file is unchanged (a fixpoint is reached).