



EXECUTION MONITOR AS PROGRAMMING LANGUAGE PRIMITIVE

L'execution monitor, invece di vederlo come un aspetto del sistema operativo, lo vediamo come se fosse una primitiva linguistica del linguaggio di programmazione. Vuol dire che andiamo esattamente a vedere quello che può essere fatto facendo le operazioni di portare all'interno del linguaggio di programmazione i controlli sicurezza quindi definire nuove classi di linguaggi di programmazione che hanno le proprietà aspettate.

Execution monitor

- Observes the execution of a program and halts the program if it's going to violate the security policy.
 - Common Examples:
 - memory protection
 - access control checks
 - routers
 - firewalls
- Most current enforcement mechanisms are execution monitors (aka reference monitors)

Implementation requirements

- Must have (reliable) **access to run-time** data about security-relevant actions of the program
 - what instruction is it about to execute?
-  ◦ Must have the ability to “**stop**” the program ... or program transition to a “good” state.
- Must protect the **monitor’s state and code from tampering**
 - key reason why a kernel’s data structures and code aren’t accessible by user code
 - **Low overhead** in practice

Allora a questo punto ci sono altri aspetti, questi aspetti che nell'esercizio che vediamo oggi sono abbastanza solo detti qui e non vengono utilizzati nell'implementazione. Il punto è che un execution monitor deve vedere tutto il passato. Vuol dire che deve essere in grado di poter memorizzare nel suo stato un astrazione dell'esecuzione che viene fatta e quindi deve essere in grado di memorizzare questa astrazione ancora una volta in una trusted computed base.

EM Behaviour

- Execution monitors can only **see the past**
 - They can enforce safety properties but not liveness properties
- **Assumptions:**
 - monitor can have access to entire state of computation.
 - monitor can have arbitrarily large state
 - safety properties enforced are modulo computational power of monitor
- Monitors **can't guess the future**
 - The predicate the monitor uses to determine whether to halt a program must be **computable**.

Questo oggi non lo vediamo ma tra un paio di settimane vedremo un esempio completo di un linguaggio dove abbiamo un execution monitor che deve memorizzare per poter funzionare una parte non banale del comportamento dei programmi. L'esempio che andremo a vedere è un esempio di function as a service, quindi applicazioni che poi sono gestite nella rete che va dal cloud al fog. Il fatto che l'execution monitor, come abbiamo visto nella teoria, caratterizza politiche di safety vuol dire che a livello del costrutto del linguaggio di programmazione guarda solo il passato e non può predire il futuro. Tutto questo è dovuto alla teoria che abbiamo visto.

An example

- Design and implementation of a functional programming language equipped with a simple execution monitor.
- Approach: Exploit OCML to represent
 - The intermediate representation (Abstract Syntax Tree)
 - The interpreter of the language

Implementiamo un linguaggio di programmazione funzionale che è equipaggiato con execution monitor. Ocaml verrà inoltre usato per l'strumentazione del security automata.

An example

- Design and implementation of a functional programming language equipped with a simple execution monitor.

A simple functional language

OCAML type for arithmetic expressions with variables

```
type ide = string  
type exp = Eint of int  
| Den of ide  
| Sum of exp*exp  
| Times of exp * exp  
| Minus of exp * exp;;
```

Allora partiamo subito dalla sintassi astratta del linguaggio. È un linguaggio di calcoli matematici molto semplice. Sostanzialmente ho un tipo che sono gli identificatori che sono delle stringhe, l'espressione possono essere delle espressioni di valori interi, la denotazione di un identificatore, quindi voglio andare a vedere qual è il valore di ambiente legato un identificatore, poi la somma, la moltiplicazione, la differenza etc...

Environment: mapping names to values

```
let emptyenv = []
(* the empty environment *)

let rec lookup env x =
  match env with
  | []          -> failwith ("not found")
  | (y, v)::r -> if x = y then v else lookup r x

let bind env (x:string) (v:int) = (x,v)::env;;
```

Per dare il significato agli identificatori dobbiamo usare il solito ambiente come visto. L'ambiente visto come un insieme di coppie, nome dell'identificatore, valore dell'identificatore. Abbiamo la possibilità di creare l'ambiente vuoto con una lista, quindi l'ambiente è una lista di coppia. Abbiamo un'operazione di lookup per vedere nell'ambiente cosa è associato a un particolare identificatore. Poi abbiamo un'operazione di bind che mi estende l'ambiente introducendo un nuovo legame tra il nome dell'identificatore, che è il parametro della bind e il valore che è un altro parametro della bind.

Supponiamo di fare un controllo degli accessi, quindi avere nel run time del linguaggio la possibilità di fare delle operazioni matematiche in base a una lista di controllo degli accessi. Ancora una volta stiamo facendo dell'astrazione quindi, qui ci abbiamo messo somma, moltiplicazione e differenza, ma avremmo potuto mettere una valanga di operazione anche molto complicate.

Access Control Lists

```
type acl = Empty  
        | AC of string * acl  
(* Access control lists *)
```

```
let rec emCheck alist op =  
  match alist with  
    | Empty    -> false  
    | AC(aop, als)-> if op = aop then true else emCheck als op;
```

Andiamo a vedere come possiamo codificare la lista dei controlli degli accessi nel nostro run time e nel nostro linguaggio di programmazione. Quindi introduciamo un tipo ACL che è quindi il tipo algebrico della ACL che è fatto empty vuol dire che non permette di eseguire niente oppure ho una capability di accesso per una determinata operazione, quella che è il costruttore AC, che si prende la capability di accesso per quell'operazione e poi la continuazione della lista di controllo degli accessi. Devo andare a vedere se una determinata operazione è ammessa in base alla lista di controllo degli accessi che ho quindi definisco questa funzione ricorsiva che opera per pattern matching sulla lista di controllo degli accessi.

Se la lista è vuota, io voglio eseguire un'operazione e la lista di controllo degli accessi non mi dà un criterio per stabilire se questa operazione è eseguita bene allora do false Se invece la lista non è vuota e la lista ha una capability che è quell'operazione "aop" e poi "als" qualunque cosa sia il resto, vado prima a controllare se la stringa "aop" è esattamente uguale all'operazione che voglio eseguire, se uguale all'operazione che voglio eseguire ho la possibilità di eseguirlo, quindi dò come risultato True altrimenti, non posso ancora dire che mi viene vietato di eseguire, devo andare a vedere sul resto della lista di accesso se posso eseguire quella operazione. Allora ovviamente abbiamo il solito discorso: dove deve essere poi messa a run time la lista di controllo degli accessi? deve essere in una memoria non eseguibile della trusted computed base del computer, in modo tale che l'attaccante non è in grado di operare su questa.

Interpreter: from Operational semantics to OCAML code (eval)

$$\frac{env(x) = v}{env \triangleright Den\ x \Rightarrow v}$$

$$\frac{env \triangleright e_1 \Rightarrow v_1, env \triangleright e_2 \Rightarrow v_2}{env \triangleright Sum(e_1, e_2) \Rightarrow v_1 + v_2}$$



eval Den x, env -> lookup x env

eval Sum(e1, e2) env ->
eval e1 env + eval e2 env

Wrapped Interpreter: eval env acl

$$\frac{env(x) = v}{env\ acl \triangleright Den\ x \Rightarrow v}$$

$$\frac{Sum \in al\ env \triangleright e_1 \Rightarrow v_1, env \triangleright e_2 \Rightarrow v_2}{env\ acl \triangleright Sum(e_1, e_2) \Rightarrow v_1 + v_2}$$

eval Den x, env, al -> lookup x env



```
eval Sum(e1, e2) env al ->
  if (emCheck alist "add")
    then (eval e1 env al) + (eval e2 env al)
  else failwith("Sum not allowed")
```

The OCAML code

```
let rec eval exp env (alist:acl) = match exp with
| Eint(n) -> n
| Den x -> lookup env x
| Sum(e1,e2) -> if (emCheck alist "add")
    then (eval e1 env alist) + (eval e2 env alist)
    else failwith("Sum not allowed")
| Times(e1,e2) -> if (emCheck alist "prod")
    then (eval e1 env alist) * (eval e2 env alist)
    else failwith("times not allowed")
| Minus(e1,e2) -> if (emCheck alist "sub")
    then (eval e1 env alist) - (eval e2 env alist)
    else failwith("Minus not allowed");;
```

La struttura a run time dell'interprete è una struttura che prende del codice eseguibile, prende il run time stack e la lista di controllo degli accessi e quindi vi dice chiaramente quali sono le informazioni che poi dovranno essere presenti a run time.

Execution Monitor take 1

letEM x = e1 with al in e2

Add a local access policy for the evaluation of e1  2

Adesso, una volta fatto questo, facciamo un passo ulteriore. Introduciamo l'execution monitor, che va a monitorare non tutto il programma, ma un particolare codice, quindi un particolare porzione del programma, in modo tale che mettiamo quando introduciamo l'execution monitor in quel particolare, linguaggio, mettiamo una politica che non è la politica globale della ACL, ma una politica locale solo per quel particolare frammento di codice che stiamo introducendo. Quindi vedete qui la potenza di avere il linguaggio di programmazione con le primitive perché ci permette di definire da programma non solo qual è la politica di controllo, ma quali sono le porzioni di codice in modo compositivo che sono soggette a politiche di controllo differenti rispetto a quelle globali. Questo è il motivo per cui è utile avere nei linguaggi di programmazione delle primitive di sicurezza e delle primitive che affrontano direttamente la sicurezza. Vuol dire che se usiamo questa notazione astratta alla ML, introduciamo un execution monitor che va valutare a nel body di e2 una politica definita dalla lista di controllo degli accessi al e permette di introdurre una variabile x in cui gli dà il valore e1. Quindi ha un'estensione della nozione di let con la politica di sicurezza. Stiamo aggiungendo una politica locale per la valutazione di e2 nel body del let. Vuol dire che dobbiamo estendere il linguaggio.

FUN+EM

OCAML type for arithmetic expressions with variables and execution monitor

```
type ide = string

type iexp =
  | Eint of int
  | Den of ide
  | Sum of iexp * iexp
  | Times of iexp * iexp
  | Minus of iexp * iexp
  | LetEM of ide * iexp * acl * iexp;;
```

Extending access control lists

```
let rec extend al1 al2 =  
  match al2 with  
  | Empty -> al1  
  | AC(aop, als) -> AC(aop, (extend al1 als));;
```

Nel run time abbiamo bisogno dato che introduciamo una nuova politica, abbiamo bisogno di estendere la politica corrente, quindi vuol dire che dobbiamo introdurre nel run time una nuova funzione che nel run time ha esattamente questo scopo. La funzione che introduciamo è la funzione extend. Abbiamo due politiche di controllo degli accessi, ACL1 e ACL2, quello che noi dobbiamo fare, dobbiamo estendere la prima politica ACL1 con le operazioni e i vincoli che sono introdotti da ACL. Vado a mettere in testa il controllo degli accessi descritto da ACL2 e poi continuo a mantenere al1. Cosa vuol dire? Quando questa operazione, le prime cose che vado a valutare sono quelle nuove che ho introdotto al2, quello che potrebbe accadere potrebbe quindi accadere che al2 mi va a modificare alcune accessi che invece hanno disabilitati da al1 però mi mantengo quelli comuni. Questa funzione è fatta per pattern matching su al2: se siamo arrivati in fondo benissimo, il risultato è al1, se non siamo arrivati in fondo, vuol dire che al1 contiene un item e poi una als. Mettiamo in testa la descrizione dell'operazione che ammettiamo, quindi aop, dove però il resto otteniamo chiamando la funzione ricorsiva extend con al1 che rimane inalterata con la funzione rimanente dell'ACL als.

Quello che dobbiamo fare, invece, è l'operazione che ci modifica il codice dell'interprete per il nuovo costrutto approccio al solito partiamo dalle regole e andiamo a definire, dalla struttura delle regole andiamo a definire il codice relativo, quindi siamo in un ambiente con ACL e stiamo introducendo una nuova di politica di sicurezza locale.

From Operational semantics to OCAML Code

$$\frac{\text{env } \textit{acl}[\textit{al}] \triangleright e1 \Rightarrow v1 \quad \textit{anv}[x = v1], \cancel{\textit{acl}} \triangleright e2 \Rightarrow v}{\text{env}, \textit{acl} \triangleright \textit{LetEM } x = e1 \text{ with } \textit{al} \text{ in } e2 \Rightarrow v}$$

`LetEM(i,e1,al,e2) ->`

```
let newal = (extend alist al) in  
  let v = (eval e1 env newal) in  
    (eval e2 (bind env i v) alist)
```

Quello che dobbiamo fare è estendere con la nuova politica di sicurezza che abbiamo introdotto, la politica che avevamo prima, andare a valutare e1 con questa nuova politica, quindi, non solo la politica va a vedere, e questa è una scelta, la valutazione del body, ma va anche a vedere la valutazione dell'espressione e1 che stiamo introducendo. Avremmo potuto fare una scelta opposta, andiamo a valutare l'espressione e1 con la vecchia politica, quindi far vedere quali sono gli spazi di scelta nella progettazione del linguaggio all'interno del linguaggio. Allora se la politica mi permette di valutare e1 e mi dà come risultato 1, quindi vuol dire è ammissibile rispetto alla politica estesa e ammissibile nell'ambiente corrente, estendo l'ambiente corrente con il legame tra x e v1 e a questo punto, vado a valutare e2 nella nuova politica estesa (errore nella regola). Questa scelta è molto più limitante, vado a valutare soltanto con la nuova Politica e1, non vado a valutare il corpo e2 però anche questa era una scelta possibile.

The OCAML code

Execution Monitor take 2

letEM x = e1 with al in e2

Add a local access policy for the evaluation of e2

The OCAML code

Execution Monitor take 3

letEM x = e1 with al in e2

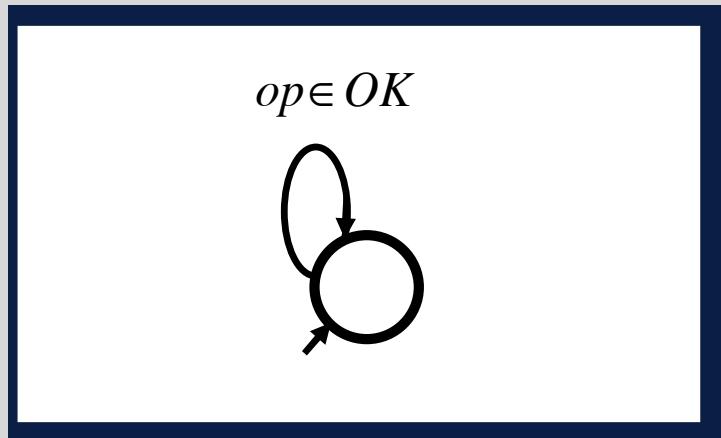
Add a local access policy for the evaluation of e1 and e2

The OCAML code

```
let rec eval iexp env (alist: acl) = match iexp with
| Eint(n) -> n
| Den x -> lookup env x
| Sum(e1,e2) -> if (emCheck alist "add")
    then (eval e1 env alist) + (eval e2 env alist)
    else failwith("Sum not allowed")
| Times(e1,e2) -> if (emCheck alist "prod")
    then (eval e1 env alist) * (eval e2 env alist)
    else failwith("times not allowed")
| Minus(e1,e2) -> if (emCheck alist "sub")
    then (eval e1 env alist) - (eval e2 env alist)
    else failwith("Minus not allowed")
| LetEM(i,e1,al,e2) -> let newal = (extend alist al) in
    let v = (eval e1 env newal)
    in (eval e2 (bind env i v) newal)
```

Discussion

Simple Acces Control Policies are just Automata



Abbiamo esaminato un automa di sicurezza che ha questa struttura. Ovvero c'è un unico stato e la proprietà "è l'operazione appartiene alla lista di controllo degli accessi". Quindi la ACL è esattamente un esempio di automa di sicurezza con unico stato e con quella proprietà.

... but EM

- Performs arbitrary computation to decide whether to allow event or halt
 - Can have side effects
 - Can change program flow

Gli execution monitor sono molto più complicati, cioè nel senso posso anche avere effetti collaterali, possono anche modificare il flusso di esecuzione perché dipendono dalla struttura del programma, quindi sono molto più complicati.

Exercize

1. Extend the simple language with execution monitor to include functions and function calls
2. Extend the language to include a variety of security policies rather than ACL.