

Parliamo di politiche di sicurezza e dei meccanismi che garantiscono che queste politiche di sicurezza vengono controllate e assicurate quando il linguaggio di programmazione le incontra. Quindi ancora una volta stiamo vedendo delle cose che avete già visto, ma la visione è il linguaggio di programmazione. Quindi la motivazione che abbiamo è: abbiamo un meccanismo M che può essere immerso nella struttura del linguaggio di programmazione, quindi vuol dire che può essere programmato da chi sta programmando un applicazione in un particolare linguaggio di programmazione. Questa persona programmatore definisce una politica di sicurezza P e chiaramente la definisce in base alle caratteristiche che il linguaggio di programmazione permette. A questo punto noi vogliamo avere delle assicurazioni che ci dice che il meccanismo M permette di fare in modo che venga controllata che la politica di sicurezza P sia correttamente rispettata. Allora la prima cosa che ci domandiamo è: qual'è una buona nozione di politica di sicurezza? Possiamo avere delle garanzie che ci permettono di comprendere qual è la potenza espressiva di una politica di sicurezza? Tutte le politiche di sicurezza che abbiamo in mente possono essere espresse e verificate all'interno della struttura di un linguaggio di programmazione o di un sistema operativo o in generale di un sistema digitale programmato? La seconda domanda ha a che vedere con un problema di espressività e di definibilità delle politiche di sicurezza. La seconda cosa è: cosa vuol dire che una politica viene verificata e qual è il meccanismo che ci permette di dire che questa politica è correttamente valutata e verificata. Poi, cosa vuol dire fare questa all'interno di un linguaggio di programmazione? Ci sono dei limiti? Come sono fatte le politiche? E quali sono le strategie per verificare una politica? Quindi quello che noi vogliamo fare è avere un panorama completo di che cosa vuol dire La matematica che ci sta dietro per definire le politiche, le classi di politiche e i meccanismi di verifica delle politiche.

# SECURITY POLICIES

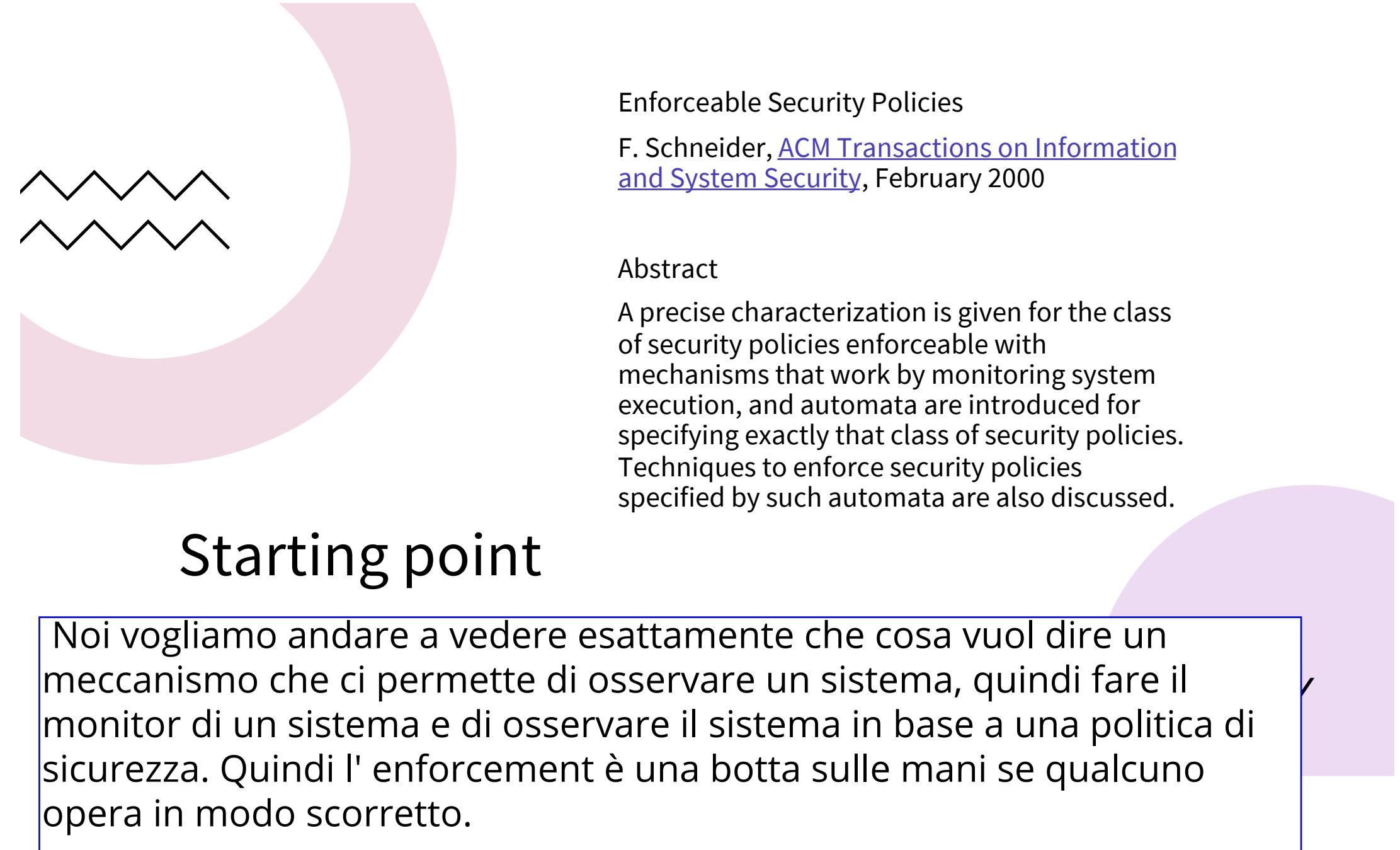
AND THEIR  
ENFORCEMENTS



## ○ Motivations

- Can we prove that mechanism **M** enforces security policy **P**?
  - What is the mathematical definition of a policy?
  - What does it mean to “enforce” a policy?
  - What does it mean to enforce a policy within a programming language?
- Are there limits to what is enforceable?
  - Which enforcement approaches are best suited to which
- Policies?
  - Are there some policies that are completely beyond any known enforcement strategy?
  - Are some enforcement approaches strictly more powerful than others?
- What is the mathematical landscape of policies, policy classes, and enforcement mechanisms?





# Starting point

Noi vogliamo andare a vedere esattamente che cosa vuol dire un meccanismo che ci permette di osservare un sistema, quindi fare il monitor di un sistema e di osservare il sistema in base a una politica di sicurezza. Quindi l' enforcement è una botta sulle mani se qualcuno opera in modo scorretto.

## Enforceable Security Policies

F. Schneider, [ACM Transactions on Information and System Security](#), February 2000

### Abstract

A precise characterization is given for the class of security policies enforceable with mechanisms that work by monitoring system execution, and automata are introduced for specifying exactly that class of security policies. Techniques to enforce security policies specified by such automata are also discussed.



# Execution Monitor

- Execution Monitors (EMs)

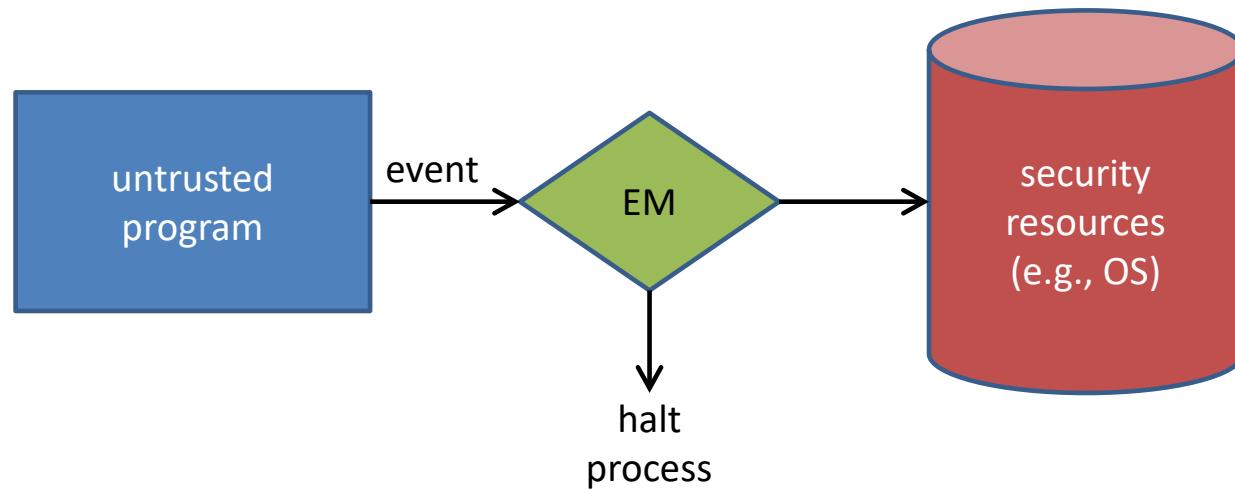
- EMs watch un-trusted programs at runtime
- Events mediated by the EM
- Violations solicit EM interventions (termination)

- Example: File system access control

- EM is inside the OS
- decides policy violations using access control lists(ACLs)

Noi vogliamo osservare il comportamento del programma e tipicamente quello che uno vuol fare, è avere un execution monitor che è un meccanismo che ci permette di osservare un comportamento di un programma che possiamo sempre assumere untrusted, quindi non stiamo ad esempio scaricando in un browser un programma dalla rete e lo mandiamo in esecuzione me se lo mandiamo in esecuzione gli diamo un grande diritto. Vogliamo andare a vedere come il monitor di esecuzione valuta i comportamenti del programma untrusted. La prima cosa che dobbiamo fare, dobbiamo capire quali sono gli eventi o le azioni riguardanti la sicurezza che sono significativi per l'esecuzione. Quindi dobbiamo comprendere che cosa andiamo ad osservare, cioè se fa delle operazioni interne che non fanno niente sul nostro sistema lo possiamo tranquillamente mandare in esecuzione, ma se cerca di leggere le nostre password magari qualche dubbio ci viene di sicurezza. L'idea dell'execution monitor è che osserva il comportamento e dà una bacchettata sulle mani, cioè fa abortire questo programma untrusted quando le azioni che vengono fatte dal programma richiedono un suo intervento. Allora questo modo di procedere è già presente nei sistemi, ad esempio gli execution monitor sono, nei sistemi operativi, le entità che vanno a controllare il controllo degli accessi. Quindi la prima cosa è che sappiamo di avere un modo di osservare il programma in esecuzione in base agli eventi e alle azioni rilevanti per la sicurezza e a questo punto l'execution monitor può bloccare il comportamento del programma.

## OS Execution Monitor



Quindi se noi lo vediamo in termini del sistema operativo, abbiamo le risorse del sistema operativo, abbiamo l'execution monitor, vede l'evento, filtra se l'evento è compliant, quindi soddisfa la politica in vigore, allora a questo punto gli diamo accesso alla risorsa nel caso di un ACL altrimenti lo facciamo bloccare.

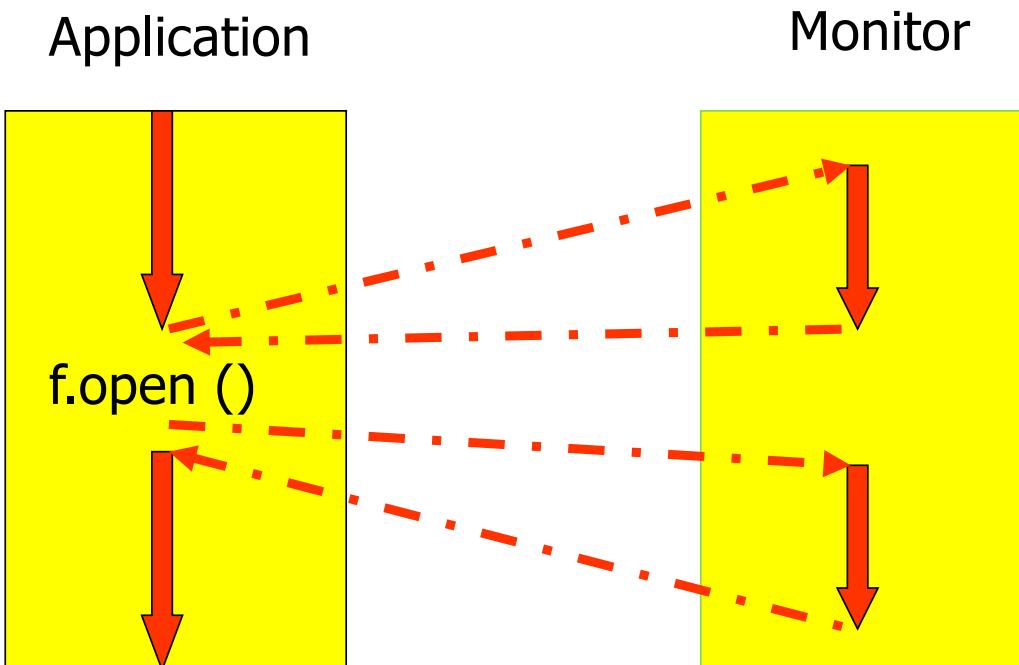


# Execution Monitor

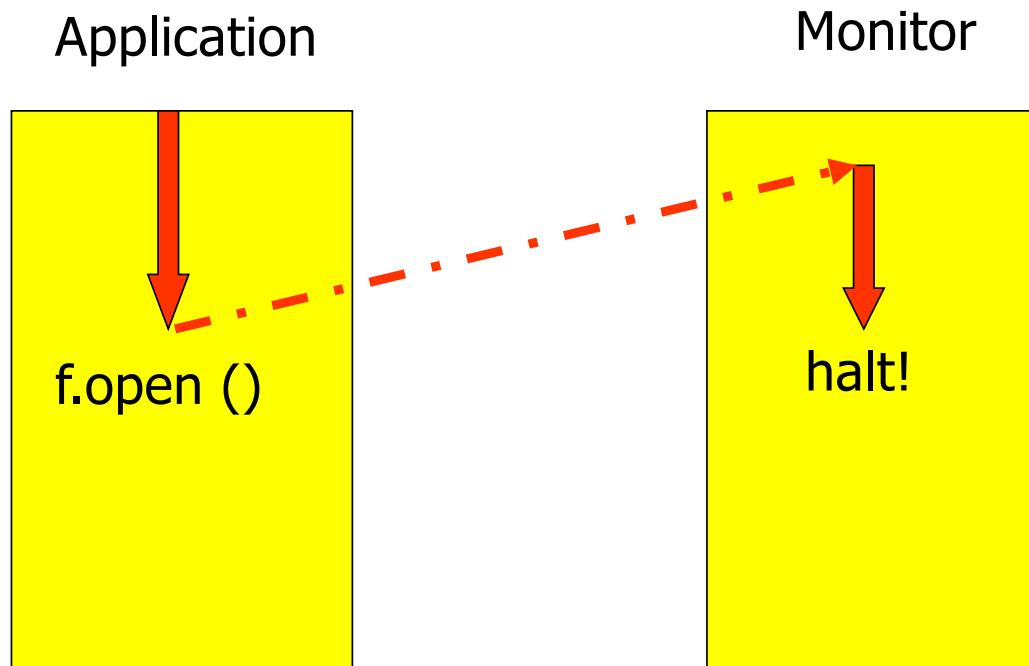
- EMs are modules that runs in parallel with an application
  - monitors may detect, prevent, and recover from application errors at run time
  - monitor decisions may be based on **execution history**

Allora da questo punto di vista, se noi lo vediamo in termini dell'implementazione del linguaggio di programmazione, che cosa è un execution monitor? In esecuzione vanta una struttura del run time, quindi sarà importante capire cosa c'è a runtime? E' una struttura del run time che quello che fa è di essere mandata in esecuzione in parallelo, quindi contemporaneamente al programma e può bloccare il comportamento del programma. Immaginiamo di scaricare nel nostro browser un'applicazione dalla rete. Se l'applicazione dalla rete legge, ad esempio, dei dati locali e poi non fa altro che operazione sui dati locali, ad esempio le applet che vengono scaricate per visualizzare dei dati di esperimenti fanno questo, cioè scaricano dalla rete un codice che permette di interpretare sul browser dei dati magari complicati. Allora leggono dalla macchina locale le istruzioni che permettono la visualizzazione, quindi operazioni di questo tipo che fa una sequenza di `read()` vanno benissimo. Il punto è se dopo aver fatto questa sequenza di `read()`, ad esempio cerca magari di fare un'operazione di `write()`, quindi di mandare informazione fuori dall'ambiente protetto, allora questo tipicamente viene vietato nei browser a meno che uno non faccia una cosa specifica e sa che dall'altra parte chi gli ha mandato il codice è trusted e quindi a questo punto può operare.

## ○ EM: Good Operations



- EM: Bad Operation



## ○ Programs and Policies

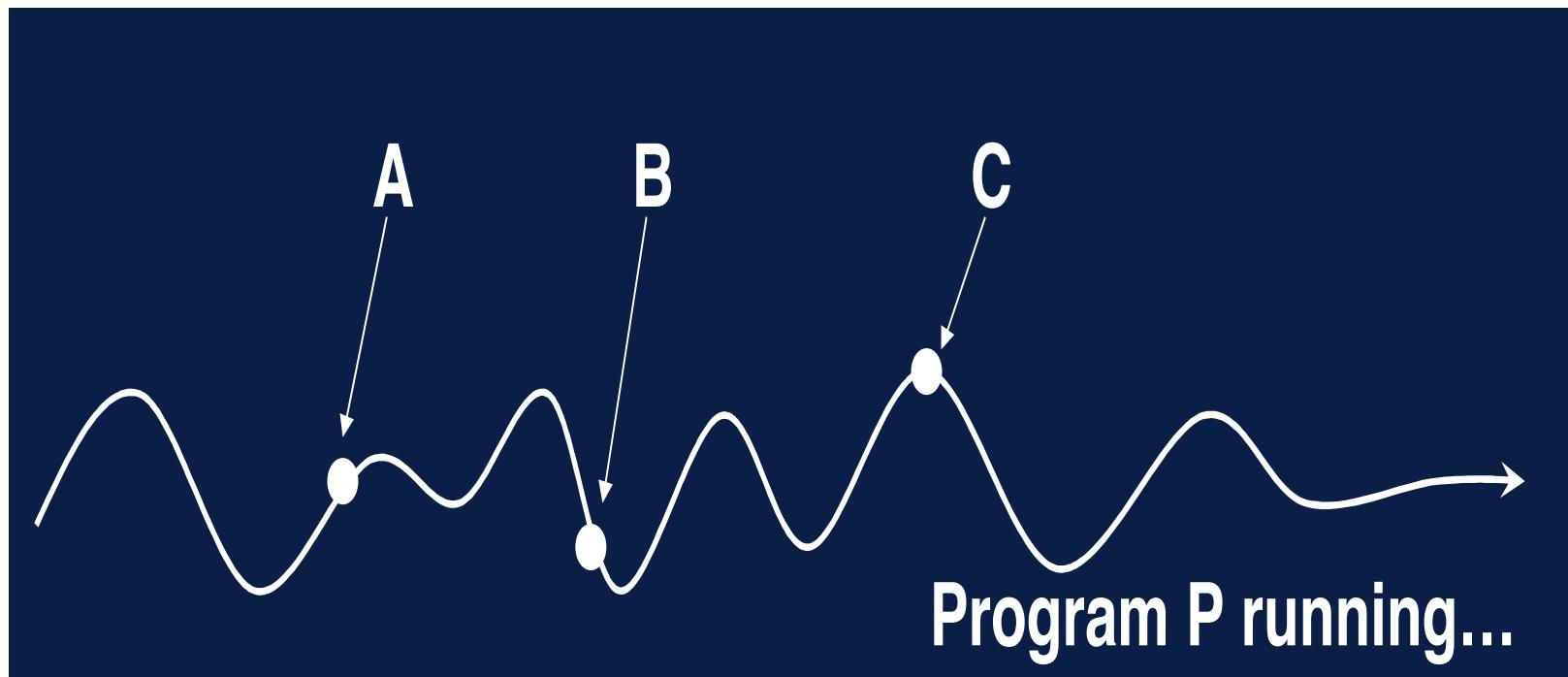
- An *execution* (or *trace*)  $s$  is a sequence of security-relevant program events  $e$  (also called actions)
- Sequence may be **finite** or **(countably) infinite**
  - $s = e_1; e_2; \dots; e_k; e_{\text{halt}}$
  - $s = e_1; e_2; \dots; e_k; \dots$
  - The empty sequence  $\varepsilon$  is an execution
  - If  $s$  is the execution  $e_1; e_2; \dots; e_i; \dots; e_l; \dots$   then  $s[i]$  is the execution  $e_1; e_2; \dots; e_i;$
- We simplify the formalism.
  - We model program termination as an infinite repetition of  $e_{\text{halt}}$  event.
  - Result: now all executions are infinite length sequences





## Execution traces

**program's execution on a given input as a sequence of runtime events**



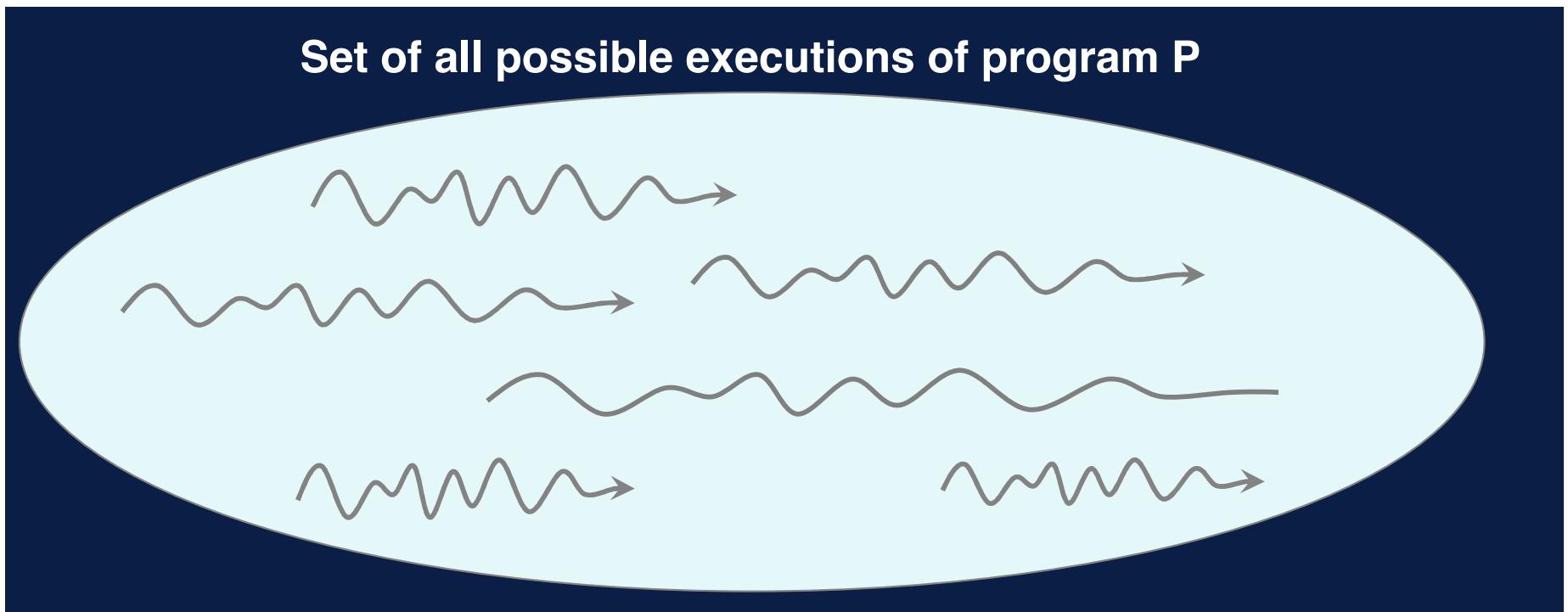
Una traccia è esattamente una sequenza di esecuzione che vede gli eventi a run time, quindi vuol dire che a runtime devo avere un modo di fare la detection degli eventi relativi alla sicurezza. Questa continua a far comprendere come deve cambiare la nozione del run time rispetto al modo standard di implementare un linguaggio di programmazione.

# ○ Programs and Policies

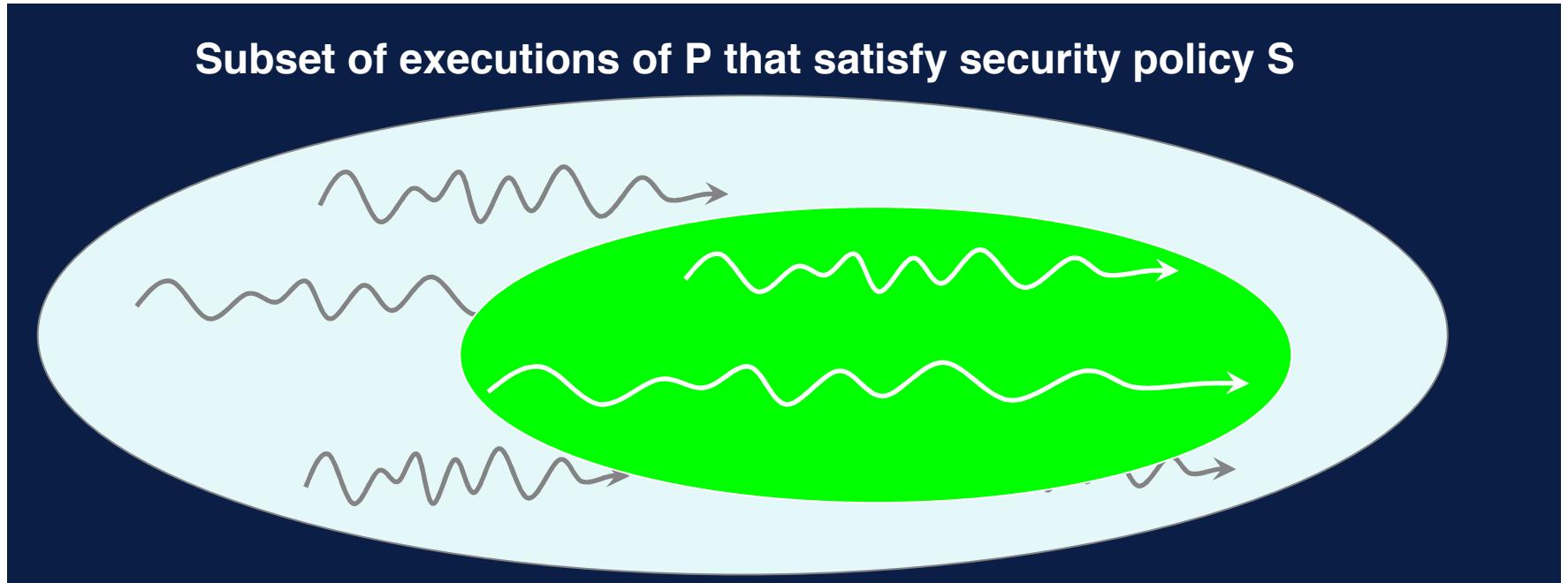
- A program **S** is a **set** of sequences (possible executions)
  - A program is modelled as the set  $S = \{s_1, s_2, \dots\}$
- A policy **P** is a **property** of programs
- A policy partitions the program space into two groups:
  - Permissible
  - Impermissible
- Impermissible programs are censored somehow(e.g.,terminated on violating runs)

A questo punto diciamo che cos'è un programma e che cos'è una politica? Allora un programma è un insieme di sequenza, quindi vedete ancora una volta stiamo vedendo il programma in termini di un modello matematico astratto. Sono le sequenze di esecuzione, possono essere, nel caso del linguaggio di programmazione che abbiamo visto prima, le sequenze dell'interprete che fanno la valutazione del programma e quindi vuol dire che un programma è interpretato come un insieme potenzialmente infinito di sequenze che sono tracce di esecuzione. Una proprietà è invece un qualcosa che mi dice che il mio programma è partizionato sostanzialmente in due parti, le sequenze che soddisfano la proprietà e quelle che non soddisfano la proprietà, quindi quelle che soddisfano la sono permessibili, sono comportamenti adeguati e permessi, quelle che non soddisfano la proprietà sono comportamenti che non vogliamo avere nel nostro programma, quindi vuol dire che sono comportamenti che ritengono di avere la necessità di avere un intervento del detector, quindi, devono avere un meccanismo che li blocca. Quindi a questo se noi vediamo un programma è un insieme di sequenze fatte in un modo standard. A questo punto vediamo un programma associato a una politica di sicurezza, vuol dire che tutte queste sequenze sono partizionate in quelle che soddisfano la proprietà, che sono quelle verdi, in quelle che non le soddisfano, quindi abbiamo partizionato il programma.

- Execution Traces



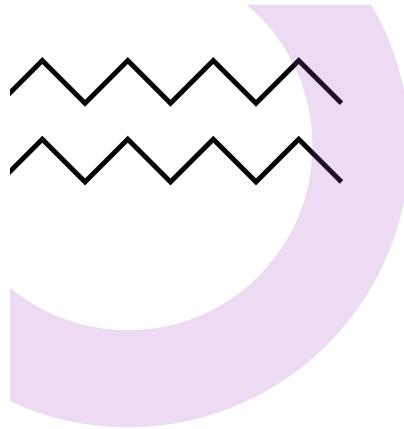
## ○ Security Policies



## ○ Security Policies: Examples

- **Access Control policies** specify that no execution may operate on certain resources such as files or sockets, or invoke certain system operations.
- **Availability policies** specify that if a program acquires a resource during an execution, then it must release that resource at some (arbitrary) later point in the execution.
- **Bounded Availability policies** specify that if a program acquires a resource during an execution, then it must release that resource by some fixed point later in the execution

Adesso andiamo a vedere come può essere una politica di sicurezza. Una politica di sicurezza, può essere una politica di controllo degli accessi, ci dice se possiamo accedere o meno a una certa risorsa che può essere un file può essere un socket oppure possiamo invocare un metodo su una determinata risorsa. Possono essere politiche di disponibilità availability. Possiamo dire che a un certo punto il programma può accedere a una certa risorsa, a un certo punto del programma dell'esecuzione e poi la può lasciare questa risorsa a un punto arbitrario, dopo cioè quindi una politica di accessibilità non dice che questo poi la deve tenere, che questa entità la deve tenere per un numero, per una quantità finita di tempo, ci dice quale deve rilasciare, ma non ci dice quando la deve rilasciare invece una politica di bounded availability ci dice, non solo. che la può accedere ma poi la deve rilasciare dopo un certo periodo di tempo fissato.



## A bit of terminology

- Events record runtime behavior: snapshots of state or actions performed
- A sequence of events is a trace  $s$
- A property  $P$  denotes a language  $L(P)$  (a set of traces)
- $s$  satisfies  $P$  iff  $\tau \in L(P)$





## ○ EM-enforceable policies

1. EM policies are universally quantified predicates over executions
  - $\forall s. P(s)$
  - **Policy P is called the detector.**
2. The detector must be prefix-closed
  - **P( $\varepsilon$ ) holds**
  - **P(s;e) holds then P(s) holds**
3. If the detector is not satisfied by a sequence (the detector rejects the sequence) then it must do so in finite time
  - $\neg P(s) \Rightarrow \exists i \neg Ps[i]$





## ○ EM-enforceable policies

1. EM policies are universally quantified predicates over executions
2. The detector must be prefix-closed
3. If the detector is not satisfied by a sequence (the detector rejects the sequence) then it must do so in finite time

### 4. Fact

- A policy satisfies (1), (2), and (3) if and only if it is a *safety policy*
- Lamport 1977: Safety policies say that some “bad thing” never happens
- EMs enforce safety policies!



Allora andiamo a vedere quindi a questo punto quello che vuol dire in termini di implementazione. In termini di implementazione vuol dire che l'execution monitor vede solo le tracce, quindi vuol dire che è un qualcosa che è in parallelo col mio programma e vede la traccia di esecuzione del programma. Se valuta che è una traccia, la verifica in tempo finito e la cosa importante allora a questo punto è un bel meccanismo di verifica dello strumento di run time perché?

## ○ Safety properties: Nothing bad ever happens

- Safety property can be enforced **using only traces** of program
  - If  $P(t)$  does not hold, then all extensions of  $t$  are also bad
- Amenable to **run-time enforcement**: don't need to know future
- **Examples:**
  - **access control** (e.g. checking file permissions on file open)
  - **memory safety** (process does not read/write outside its own memory space)
  - **type safety** (data accessed in accordance with type)

Perché per poter decidere se l'esecuzione viola la politica, ha bisogno di un tempo finito. Quindi tale meccanismo, deve avere un run time dei linguaggi di programmazione, perché non ha bisogno di fare delle predizioni di quello che succederà dopo, ha soltanto bisogno di conoscere l'esecuzione attuale. Allora a questo punto l'access control è sicuramente un execution monitor e può essere verificata in un execution monitor. La memory safety può essere sicuramente verificata da un execution monitor, dove quest'ultima ti dice che un processo, un programma, una macchina virtuale, qualunque cosa voi avete in esecuzione controllata, non può leggere e scrivere al di fuori dello spazio di memoria che gli viene dato. Il controllo a run time del sistema di tipi sono esattamente degli execution monitors. Ovvero che i dati vengono acceduti dal linguaggio di programmazione in accordo col sistema dei tipi che viene definito.

## ○ Liveness properties: Something good eventually happens

- **Nontermination:** **The email server will not stop running**
  - Violated by **denial of service attacks**
- Liveness properties: Cannot be enforced purely at run time
- Interesting properties often involve both safety and liveness
  - Every property is the intersection of a safety property and a liveness property [Alpern & Schneider]

Se noi andiamo a vedere quello che succede nella teoria delle proprietà e quindi nel model checking che abbiamo brevemente introdotto quando abbiamo parlato di verifica e in modo particolare di verifica dei sistemi di protocolli ci sono anche delle altre politiche dette di "liveness". Esse ci dicono che in effetti poi qualche cosa di buono accadrà nel futuro. Ad esempio, un sistema safe è un sistema che non fa niente. Un sistema che non fa niente sicuramente verifica tutte le politiche, ma noi vorremmo avere dei sistemi che fanno qualcosa, ad esempio una politica di liveness, è una politica in grado di evitare che ci sia un denial of service perché la politica di liveness vuol dire che il server continuerà sempre a funzionare, non smetterà mai, in un tempo finito io vado soltanto a controllare che in quel tempo non ha smesso di funzionare, ma non posso predire che tutti i possibili comportamenti anche nel futuro non lo porteranno a bloccarsi. Quindi questo vuol dire che le politiche di sicurezza non possono essere verificate da considerazioni soltanto del run time.



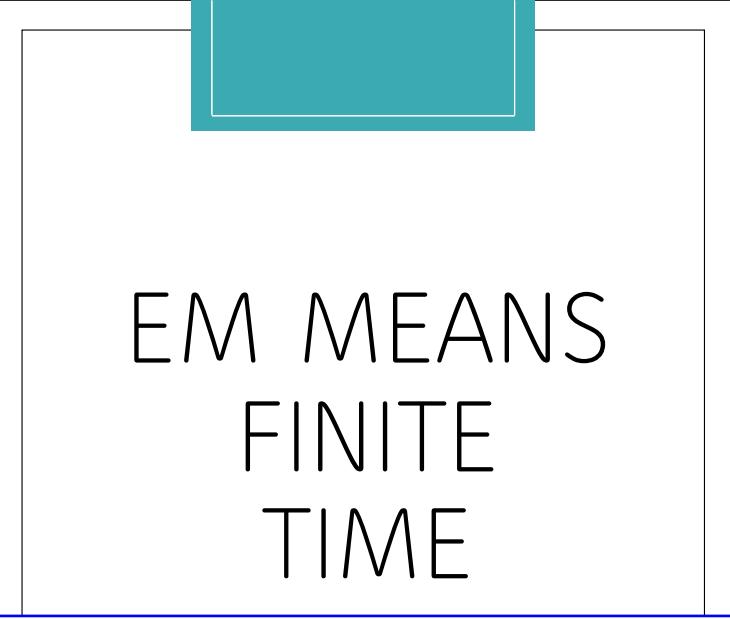
## EM ENFORCEMENT

Allora l'execution monitor enforcement è un meccanismo che deve bloccare l'esecuzione non appena abbiamo una violazione, quindi vuol dire non appena abbiamo un prefisso del comportamento che viola la proprietà allora a questo punto deve bloccare l'esecuzione. Le esecuzioni sono viste come tracce, sequenze di comportamento che quindi che sono quindi prefix closed.

Must truncate execution as soon as prefix violates policy:

$$\neg p(\tau) \Rightarrow (\forall \sigma : \neg p(\tau \sigma))$$

where  $\tau$  is a finite trace,  $\sigma$  a trace, and the juxtaposition operator extends one trace with another



EM MEANS  
FINITE  
TIME

Il secondo aspetto che è l'enforcement deve essere in grado di determinare se la politica è violata in un tempo finito, quindi permette intanto di fare l'analisi a run time e comprendere quando in un tempo finito la politica viene valutata.

Must detect violations after a finite time.

$$\neg p(\sigma) \Rightarrow (\exists i : \neg p(\sigma[..i]))$$

where the  $[..i]$  postfix operator denotes a prefix of a given trace that is  $i$  steps long

Il sommario di quello che abbiamo visto la volta scorsa quando abbiamo analizzato gli execution monitor e abbiamo capito come da un punto di vista della loro modelizzazione, gli execution monitor permettono di definire politiche di sicurezza. La prima cosa che abbiamo visto era che il meccanismo analizza una singola esecuzione, quindi abbiamo una visione del modello dei comportamenti del programma ed è una visione in cui i programmi sono caratterizzati da il linguaggio delle loro esecuzioni. Una proprietà vale se vale sull'esecuzione per cui le politiche di sicurezza partizionano i comportamenti del programma in due classi, quelle che soddisfano la proprietà di sicurezza e quelli che non la soddisfano. La proprietà che noi ci aspettiamo è che se la politica non vale, allora deve essere determinata in un tempo finito e quindi questa vuol dire che deve esistere un indice  $i$  che caratterizza la dimensione della traccia, che falsifica la politica e se una politica vale per una certa traccia, vale per tutte quelle che sono prefissi della traccia che stiamo considerando. Tutti queste tre aspetti ci dicono che le politiche di sicurezza sono delle politiche di safety. Le politiche di safety sono state introdotte negli anni 70 da Lamport e lo slogan è che non accade niente di cattivo, di sbagliato, durante l'esecuzione. Chiaramente un sistema che non fa niente è safe e quello che bisogna garantire è che il sistema faccia qualcosa.

## EM Summary

- Analyzes the single (current) execution.

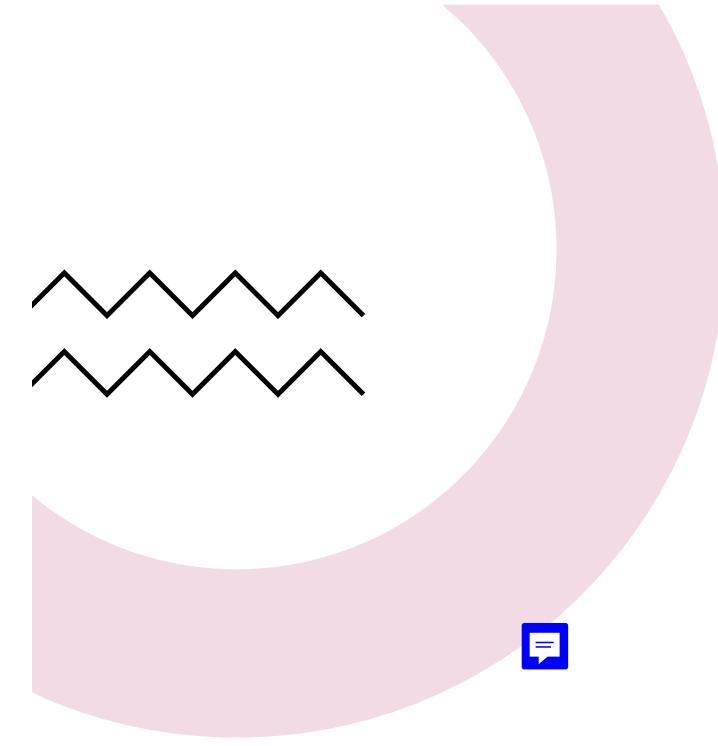
$$P(\Pi) : (\forall \sigma \in \Pi : p(\sigma))$$

- Must truncate execution as soon as prefix violates policy:  $\neg p(\tau) \Rightarrow (\forall \sigma : \neg p(\tau \sigma))$
- Must detect violations after a finite time:

$$\neg p(\sigma) \Rightarrow (\exists i : \neg p(\sigma[..i]))$$

Enforceable policy implies safety property

Allora, una volta fatto questo, la seconda cosa è: come possiamo avere dei meccanismi che ci permettono di definire esattamente quali sono le classi di politiche che sono scrivibili, in che modo sono scrivibili e come possono essere verificate da execution monitor che hanno le tre caratteristiche.

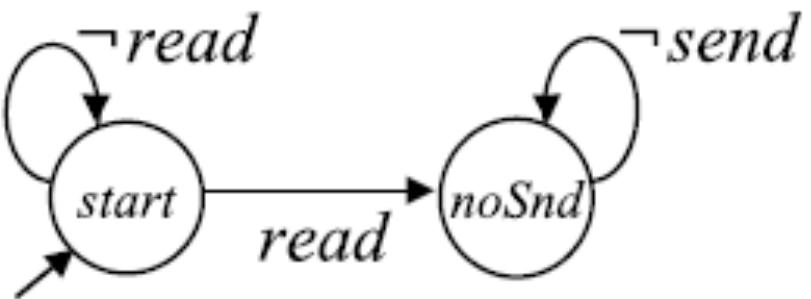


# Security Automata

- Security Automata (Erlingsson & Schneider 1999)
- Formalization of safety policies
  - finite state automaton
  - accepts language of permissible executions
  - alphabet = set of events
  - edge labels = event predicates
  - all states accepting (language is prefix-closed)

Allora quello che noi andremo a vedere da mercoledì è una formulazione di queste politiche di sicurezza in termini di una particolare classe di automi che sono stati introdotti sempre da Schneider e da Erlingsson alla fine del secolo scorso, nel 1999, chiamati security automata, automi a stati finiti che sostanzialmente accettano, quindi sono dei riconoscitore di linguaggi che accettano soltanto le esecuzioni accettabili, hanno come alfabeto gli eventi, gli archi sono sostanzialmente le proprietà che caratterizzano un passaggio di stato ad un altro e sono prefixes closed, quindi vale la proprietà che dicevamo prima sui prefissi, la proprietà due. Andremo a vedere che tipo di politiche permettano di fare, poi andremo a vedere come possono essere utilizzate nella compilazione per fare in modo che si compili definendo le politiche di sicurezza dell' execution monitor e compilare all'interno del linguaggio la politica di sicurezza descritta da questi particolari autonomi.

## ○ Security Automata (Example)



**NO SENDS AFTER READS**

Un esempio: Questo è un esempio che definisce una politica di sicurezza dove non posso mandare fuori dalla mia applicazione, questa espresso da il fatto di avere un azione send() che manda fuori delle informazioni, dopo aver letto, quindi questo cosa vuol dire? Vuol dire che io posso, ad esempio, continuare a mandare fuori delle informazioni, quindi immaginate che io scarico dalla rete una applet e voglio sottoporre questa applet all'interno di un monitor di sicurezza nell'applicazione in cui ho scaricato questo codice che assumo untrusted. Allora a questo punto, l'idea della politica è la seguente: io scarico la applet, la applet può mandare fuori dell'informazione, quindi vuol dire che manda fuori al sito o al code base che ha scritto questa applet le informazioni ma non gli permetto di mandare fuori delle informazioni dal momento in cui la mando in esecuzione, se questa ha fatto una un'operazione di read().

Se aveva delle informazioni locali che potevo utilizzare e rimandare indietro mi va bene, tanto ce l'aveva comunque, ma appena ha letto delle informazioni sul mio sistema dove gli ho dato il permesso di esecuzione, allora non può più mandare fuori niente. Come formalizziamo questa proprietà? Questa proprietà non la formalizziamo e la descriviamo da questo semplice automa: 2 stati, uno stato iniziale "start" (e i due stati sono tutti e due accettanti) e abbiamo detto che le azioni sono read e send. Quindi le azioni di sicurezza rilevanti per la nostra politica sono funzioni di lettura, operazione di invio di dati. Gli archi sono dei predici su queste azioni, il fatto che io scrivo "non-read()" vuol dire che nello stato start rimango se faccio una qualunque operazione diversa dalla lettura e quindi vuol dire che posso continuare ad esempio ad inviare fuori dei dati non appena faccio l'operazione di lettura passo dallo stato start allo stato "noSnd" in cui posso continuare a fare delle operazioni di lettura però non posso più mandare fuori delle informazioni, quindi questa è esattamente l'automa che mi descrive quella politica scritta in rosso.

# ○ Security Automata (formally)

$$(\sigma, q) \xrightarrow{\tau} A (\sigma', q')$$



$$(\sigma, q) \xrightarrow{a} A (\sigma', q') \quad (\text{A-STEP})$$

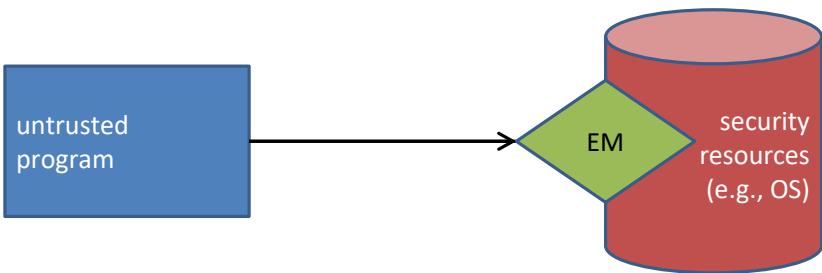
if  $\sigma = a; \sigma'$   
and  $\delta(a, q) = q'$

$$(\sigma, q) \xrightarrow{\cdot} A (\cdot, q) \quad (\text{A-STOP})$$

otherwise



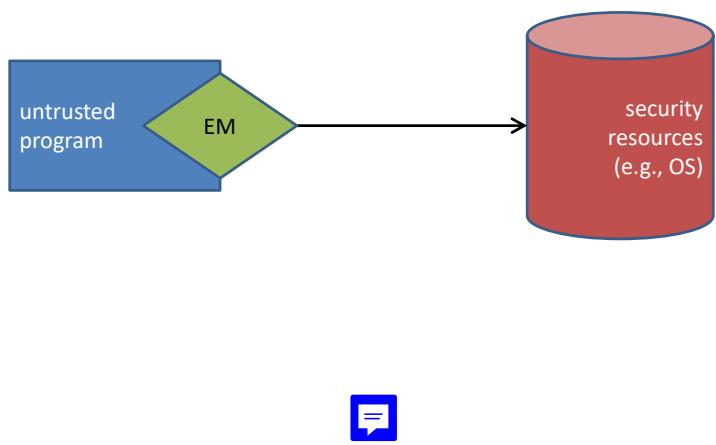
## Discussion



- Disadvantages of traditional Em's
  - inefficient: context-switch on every event
- Large TCB: EM extends the OS
- weak: EM can't easily see internal program actions
- non-modular: changing policy requires changing OS

Andiamo a vedere come posso implementare un execution monitor con il suo bel security automata. L' approccio che ho nei sistemi operativi, è che io metto l' execution monitor all'interno del sistema operativo. Vuol dire che, dato che l'execution monitor opera in parallelo con il resto delle applicazioni untrusted, l' approccio è quello di avere un overhead a tempo di esecuzione, perché bisogna avere ad ogni istante che la black box che monitora il programma untrusted che vuol fare un'azione di sicurezza di quelle monitorate, bisogna avere un context switching, cioè si va dall'esecuzione del programma all'esecuzione del monitor e si va a controllare e poi eventualmente si va avanti o si blocca il programma. L'altra cosa complicata è che la trusted computed base che è il kernel del sistema operativo viene estesa dall'execution monitor. Questo vuol dire che la piattaforma e la dimensione per lo spazio degli attacchi viene aumentata dato che prende esattamente il kernel del sistema operativo. Se immergiamo completamente l'execution monitor nel sistema operativo, se vogliamo cambiare la politica bisogna fermare e fare il patch del sistema operativo, ricambiarlo e reinstallare il sistema operativo, che ovviamente non è una cosa auspicabile.

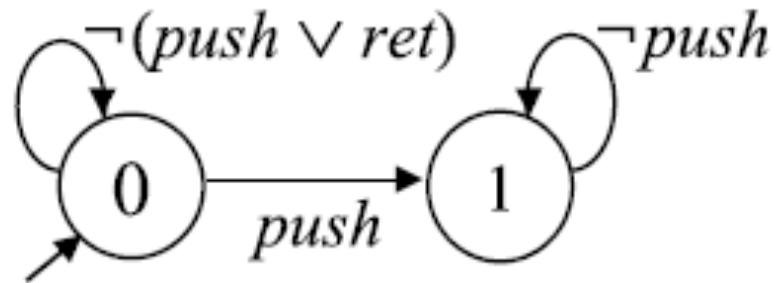
## ○ In-lined reference monitor



- Main idea: implement a execution monitor by *in-lining* its logic into the untrusted code:
  - In-lining procedure should be automated
- Challenges:
  - How to automatically generate EM code?
  - How to preserve (non-violating) program logic?
  - How to prevent (malicious) programs from corrupting the EM?



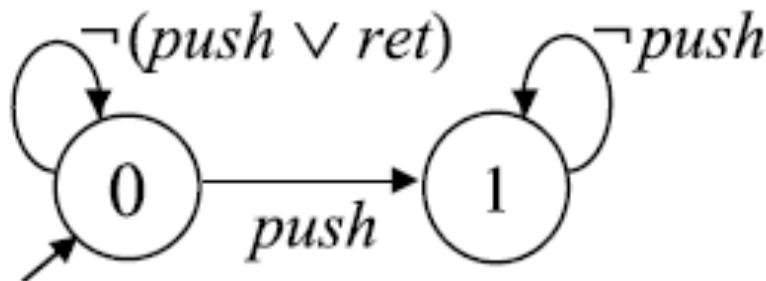
## ○ In-lining a security automaton



**Policy: push exactly once before returning**

Facciamo vedere un esempio di instrumentazione dove quindi facciamo le operazioni di inline. E' un automa che contiene 2 stati: nello stato iniziale per lo stato zero, la cosa che posso fare è: non posso fare un'operazione di push o non posso fare un'operazione dire return, quindi questo vuol dire che non posso spingere sullo stack delle informazioni oppure non posso restituire il controllo al chiamante. Mi muovo dallo stato 0 allo stato 1 quando faccio un'operazione di push sullo stack e a quel punto non posso più fare operazioni sulla stack. Vuol dire che io posso fare soltanto un'operazione di push su stack prima di restituire il controllo al chiamante.

## ○ In-lining a security automaton



**In-lining the policy in this binary code!!**

```
mul r1,r0,r0  
push r1  
ret
```

Adesso immaginiamo che il codice binario che vogliamo instrumentare è questo codice binario. Questa è una black box che esegue l'istruzione mul push e ret e andiamo ad eseguire la sequenza mul push e ret vista come separato sull'automa di sicurezza che li abbiamo descritto, quindi facciamo una mul, a questo punto possiamo rimanere nello stato 0, appena facciamo l'operazione di push ci muoviamo nello stato 1, a questo punto eseguiamo la ret, che continua ad essere accettante, quindi questo cosa vuol dire? Vuol dire che sicuramente quella sequenza vista da execution monitor separato dal programma è accettante, quindi rispetta la politica di sicurezza. Quello che noi vogliamo fare invece è immergerla nel codice binario li scritto.

## ○ The in-lining algorithm

1. Conceptually in-line the automaton just before **EVERY** event
2. Partially evaluate (i.e., specialize) the automaton edges to the event it guards
  - some edges disappear entirely
3. Generate guard code for the remaining automaton logic

Allora concettualmente dobbiamo mettere l'automa di sicurezza difronte ad ogni istruzione, perché dobbiamo verificare che ogni azione sensibile alla sicurezza verifica la proprietà. Poi quello che dobbiamo fare è specializzare, quindi fare una interpretazione e una semplificazione dell'automa di sicurezza in base allo stato corrente dell'esecuzione. Vuol dire che in base alle informazioni attuali che abbiamo del comportamento del programma, possiamo semplificare la struttura dell'automa, in modo particolare le informazioni che abbiamo quando facciamo questa operazione potrebbero non essere più presenti degli stati, sicuramente degli archi potrebbero più non aver senso. A questo punto, dopo aver fatto questa semplificazione, usiamo l'automa semplificato per poter generare il codice di instrumentazione che prenderà la forma di guardie che vanno a verificare che l'azione che il programma intende fare è conforme alla politica di sicurezza.

# ○ In-lining: the actual steps

## **Insert security automata.**

Inserts a copy of the security automaton before each target instruction.

## **Evaluate transitions.**

Evaluates any transition predicates that can be given the target instruction.

## **Simplify automata.**

Deletes transitions labelled by transition predicate that evaluated to false.

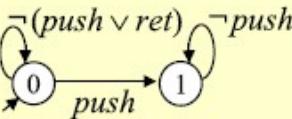
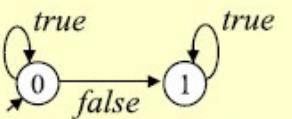
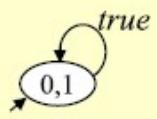
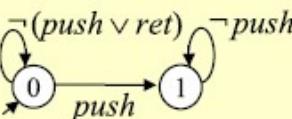
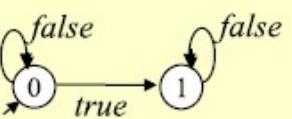
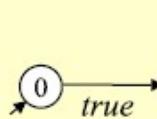
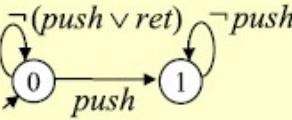
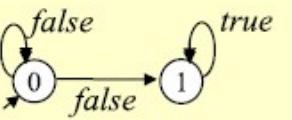
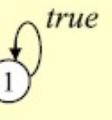
## **Compile automata.**

Translates the remaining security automata into code.

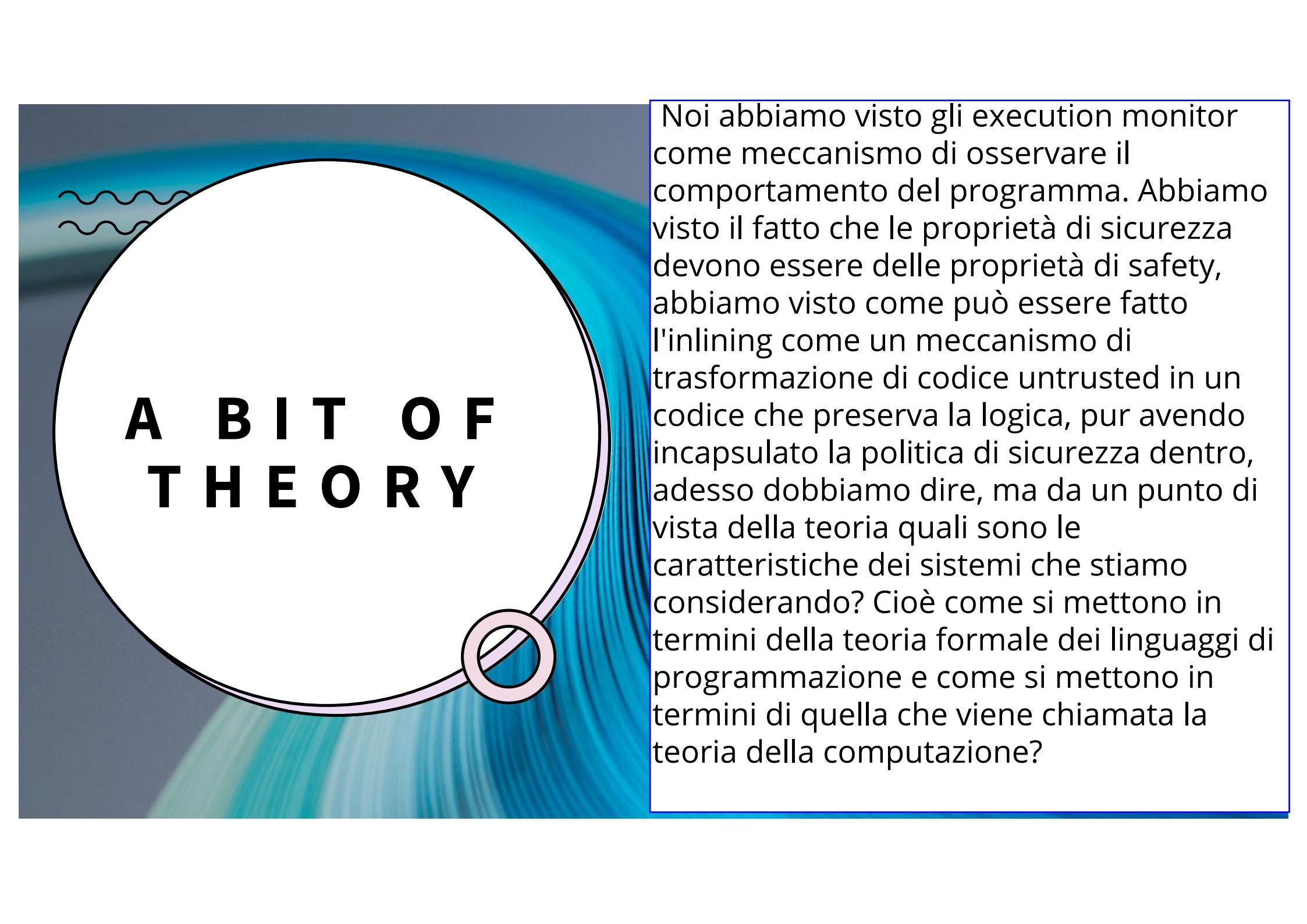
**FAIL** is invoked by the added code if the automaton rejects its input.

Inseriamo il security automata per ogni istruzione sensibile di sicurezza. Una volta fatto questo facciamo la valutazione delle transazioni in base allo stato corrente. Semplifichiamo l'automa, ad esempio cancelliamo delle transazioni di stati che non sono più significativi e in modo particolare cancelliamo tutte quelle transazioni di stato che sono etichettata col predicato falso perché vuol dire che a questo punto sono dei predici che non sono verificati nello stato attuale, quindi non sono significativi nello stato attuale. Una volta che abbiamo fatto questa operazione di trasferimento e di analisi dell'automa rispetto allo stato corrente dell'esecuzione possiamo generare poi il codice instrumentato con delle guardie. Ovviamente metteremo il codice in quelle situazioni quando la guardia che tiriamo fuori falsifica la proprietà e quindi vuol dire siamo esattamente in una situazione che dobbiamo bloccare l'esecuzione del programma. Ora diamo un'idea intuitiva con l'esempio che abbiamo visto prima col codice che abbiamo visto prima con l'automa che abbiamo visto prima.

# Example

| Insert security automata   | Evaluate transitions  | Simplify automata  | Compile automata   |
|--|---|--|--|
|  <p><b>mul r1, r0, r0</b></p> |  <p><b>mul r1, r0, r0</b></p> |  <p><b>mul r1, r0, r0</b></p> | <p><b>mul r1, r0, r0</b></p>   |
|  <p><b>push r1</b></p>        |  <p><b>push r1</b></p>        |  <p><b>push r1</b></p>        | <p><b>if state==0</b></p> <p><b>then state:=1</b></p> <p><b>else ABORT</b></p> |
|  <p><b>ret</b></p>          |  <p><b>ret</b></p>          |  <p><b>ret</b></p>          | <p><b>if state==0</b></p> <p><b>then ABORT</b></p> <p><b>ret</b></p>           |



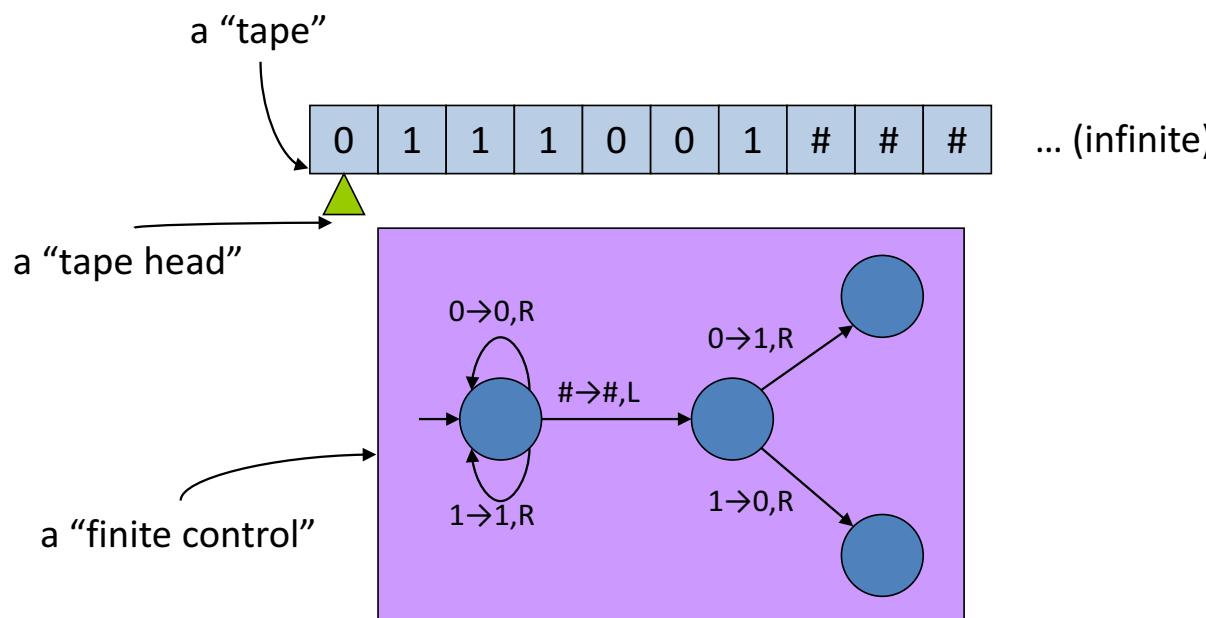


## A BIT OF THEORY

Noi abbiamo visto gli execution monitor come meccanismo di osservare il comportamento del programma. Abbiamo visto il fatto che le proprietà di sicurezza devono essere delle proprietà di safety, abbiamo visto come può essere fatto l'inlining come un meccanismo di trasformazione di codice untrusted in un codice che preserva la logica, pur avendo incapsulato la politica di sicurezza dentro, adesso dobbiamo dire, ma da un punto di vista della teoria quali sono le caratteristiche dei sistemi che stiamo considerando? Cioè come si mettono in termini della teoria formale dei linguaggi di programmazione e come si mettono in termini di quella che viene chiamata la teoria della computazione?

# ○ Turing Machines and Computability

- Turing Machine
  - Alan Turing (1936)
  - simple mathematical model of a computer
  - consists of:



Allora il punto della teoria e delle macchine di turing è la teoria della calcolabilità. Questi teoremi vengono fatti su modelli di calcolo e i modelli di calcolo sono le macchine di Turing. Concettualmente è data da un nastro potenzialmente infinito, dove posso scrivere dei simboli, ad esempio codificare con 0 e 1 e poi dei simboli particolari che sono degli imitatori, che sono le cartelline che vedete in questo esempio. Quindi c'è un nastro, una testina di lettura che legge il simbolo dal nastro e poi c'è un controllo finito. Quindi è una specie di automa, una specie di automa a stati finiti. Legge il simbolo e a seconda del valore si sposta a destra o a sinistra a seconda ovviamente dello stato. Nell'esempio che vediamo qui, se siamo nello stato corrente e legge uno 0 vedete si scrive 0 e poi va a destra, se legge un 1, scrive un 1 e poi va a destra. Invece, se è arrivato alla destra del nastro e trova un simbolo di lettura, sostituisce questo simbolo con zero e poi si sposta a sinistra e poi fa altre operazioni. Il nastro è la memoria e il controllo finito è sostanzialmente il processore, cioè di fatto questo è già un modello della macchina di Von Neumann, dove il programma è memorizzato in memoria e il processore, il sistema di elaborazione, è in grado di leggere, decodificare il programma, esattamente quello che poi diventeranno l'architettura dei moderni calcolatori.

## ○ TM Expressive power

- Can do simple arithmetic
- TMs don't necessarily terminate
- Can evaluate a C program encoded in binary
- Can simulate arbitrary TMs (given as input) on arbitrary inputs (given as input): the “**universal TM**”
- Fact: **Can do anything a real computer can do (but very, very slowly)**
- TMs can't solve **undecidable problems** (e.g., **halting problem**)

Allora cosa può fare una macchina di turing? Beh, può fare tutta la matematica ne non necessariamente termina cioè nel senso può scrivere dei programmi che vengono eseguiti per sempre. Possono, ad esempio, eseguire programmi codificati in binario programmi in C, può simulare una qualunque macchina di Turing perché possiamo mettere nel nastro potenzialmente infinito, la codifica numerica, si chiama Godelizzazione, stesso principio che è alla base del teorema di incompletezza di Godel dell'aritmetica e poi ci sono delle relazioni molto simili tra le problematiche studiate da Godel della aritmetica e le macchine di Touring e a questo punto io gli do questo programma e la macchina può simulare quel programma. A questo punto la cosa importante è quello scritto in rosso, cioè una macchina in Turing può fare ogni cosa che può fare un computer attuale, lo fa molto lentamente, ci vuole parecchio più tempo, ma lo esegue. Uno dei problemi principali che la teoria sviluppata da Alan Turing mette in evidenza è che però ci sono dei problemi che non sono risolubili. Un problema tipico è quello del halting problem, che dice: possiamo prendere una macchina di Turing, gli diamo in pasto un programma e un dato e dire se termina, cioè se dice SI o NO? E questo lo dobbiamo fare per ogni programma e per ogni dato? Allora questo problema si chiama il problema della fermata.

## ○ The halting problem

- The **halting problem** is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever.
- Turing proved a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.
  - **the halting problem is undecidable over Turing machines**
- Turing result is significant to practical computing efforts, **defining a class of applications which no programming invention can possibly perform perfectly.**

Il problema della fermata è: dato la descrizione di un arbitrario programma e di un input su cui opera quel particolare programma, si può dire se quel programma termina o continuerà a essere eseguito su quel particolare dato bene? Turing ha dimostrato che il problema della fermata non è decidibile, cioè non esiste una macchina di Turing, a cui do un arbitraria macchina in pasto, un dato e il risultato è sì il programma su quel dato termina oppur no, non termina. Quindi questo problema non è decidibile. Quindi a questo punto ha dato un'idea molto chiara di che cosa vuol dire potenza di calcolo, che cosa vuol dire algoritmo e di che cosa vuol dire un modello universale per il calcolo.

# ○ Computation theory



- RE (recursively enumerable) is the class of decision problems for which a 'yes' answer can be verified by a Turing machine in a finite amount of time.
  - if the answer to a problem instance is 'yes', then there is some procedure which takes finite time to determine this, and this procedure never falsely reports 'yes' when the true answer is 'no'. However, when the true answer is 'no', the procedure is not required to halt
- co-RE is the set of all languages that are complements of a language in RE.
  - co-RE contains languages of which membership can be disproved in a finite amount of time, but proving membership might take forever.



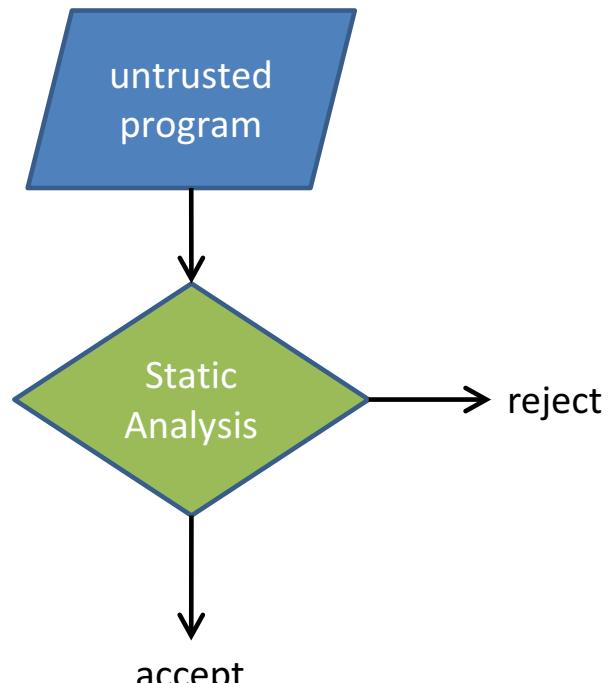
Perché tutta questa discussione sulla teoria ha un effetto sulla pratica, sulla pratica di quello che stiamo esaminando oggi e che abbiamo visto un po la volta scorsa su analisi statiche e i metodi di enforcement delle politiche di sicurezza.

## ENFORCEMENT STRATEGIES



Allora il primo tentativo di analisi che facciamo sono analisi statiche. Come è fatta l'analisi statica? Prende un programma untrusted, analizza il codice prima di mandare in esecuzione e dice: ACCETTO se soddisfa la politica di sicurezza, RIFIUTO se non soddisfa la politica e lo deve fare in un tempo finito. Questa è la risoluzione dei nostri problemi di sicurezza: abbiamo una risposta a tempo statico e quindi il codice quando poi lo mandiamo in esecuzione viaggia al massimo, perché non ha overhead di esecuzione. L'unica contro è un overhead in loading perché prima di mandarlo in esecuzione dobbiamo passare per un programma, che ci impiegherà un po' di tempo. Quindi l'analisi statica ha questa caratteristica.

## ○ Take 1: static analysis

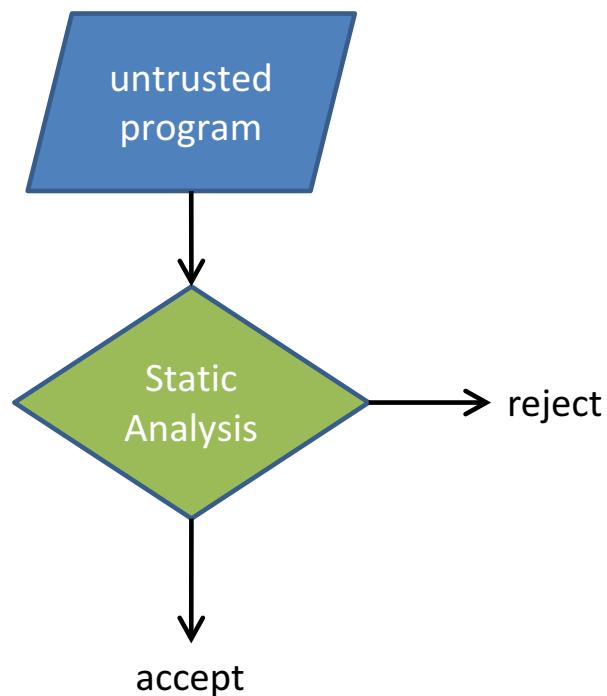


- Approach:
  - analyze untrusted code BEFORE it runs
  - return “accept” or “reject” in finite time
- Pros:
  - immediate answer
  - code runs at full speed
- Cons:
  - high load overhead
  - weak in power...?

Quindi noi sappiamo già qual è la risposta dal teorema di Rice, questo vuol dire che per poter operare bene con l'analisi statica devo avere delle politiche di sicurezza decidibili, ovvero sono in grado di fare staticamente sì o no, a questo punto ci mettiamo assieme il teorema di Rice per avere dell'astrazione delle politiche di sicurezza decidibili. Però c'è questo problema, le politiche di sicurezza devono essere ricorsive. Adesso prendiamo gli execution monitor, l'approccio delle execution monitor è quello di avere il programma, non lo vedo, vedo gli eventi relativi alla sicurezza che genera il programma.

L'execution monitor interviene e mi blocca l'esecuzione quando trova un evento che sta per violare la politica di sicurezza. Cosa vuol dire? Vuol dire che è implementato in parallelo al programma. Allora, a questo punto quali sono le caratteristiche? Ovviamente ho bisogno di andare in esecuzione, non posso avere come prima, nel caso dell'analisi statica una risposta senza eseguire. Ovviamente abbiamo quell'overhead dovuto al context switching cioè dato che abbiamo il programma e il monitor in esecuzione, ogni volta abbiamo un'operazione di sicurezza dobbiamo fermare il programma, fare una operazione di switching del contesto, analizzare se viene violata e poi abortire se viene violata e far continuare il programma o no. Dal punto di vista delle caratteristiche positive, vabbè, non abbiamo il problema di loading time, cioè prendiamo il programma, lo sbattiamo, mandiamo in esecuzione lui e il monitor, adesso andiamo a vedere, ma questa cosa qui? dal punto di vista delle politiche di sicurezza in termini di espressività e quindi di teoria della computazione, che cosa vuol dire?

## ○ Take 1: static analysis

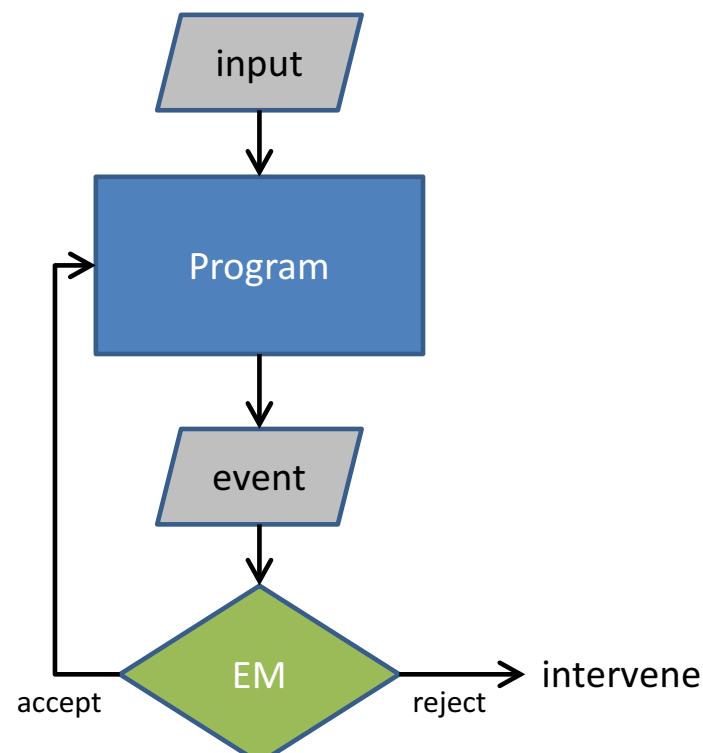


- Approach:
  - analyze untrusted code BEFORE it runs
  - return “accept” or “reject” in finite time
- Pros:
  - immediate answer
  - code runs at full speed
- Cons:
  - high load overhead
  - weak in power...?

Recursively Decidable Policies



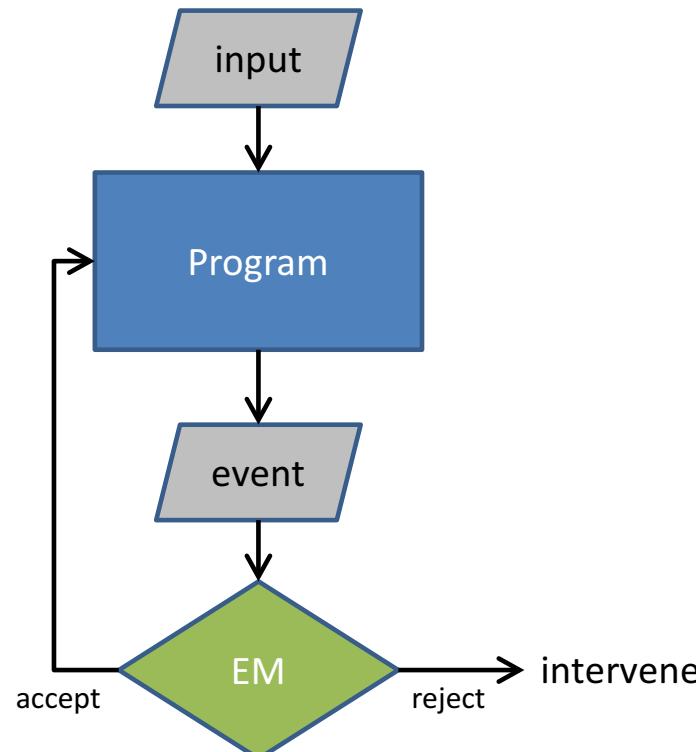
## ○ Take 2: EMs



- Approach:
  - EM monitors events
  - intervenes to prevent violations
  - implemented outside program
- Cons:
  - no answer until execution
  - runtime slow-down (context-switches)
- Pros:
  - lower load-time overhead than static analysis
  - more powerful...?



Vuol dire che stiamo considerando politiche che sono le duali di quelle ricorsivamente numerabili, proprio perché li automi di sicurezza sono accettanti quindi noi blocchiamo il programma in esecuzione quando troviamo un evento che viola la politica e quindi trovando un evento che viola la politica stiamo sostanzialmente decidendo l'accettazione non del linguaggio della politica, ma stiamo decidendo l'accettazione di un elemento che non appartiene al linguaggio, quindi sono esattamente la duale delle politiche ricorsivamente enumerabili e quindi questo ci dice una caratteristica in termini di esprevissità del modello di execution monitor.



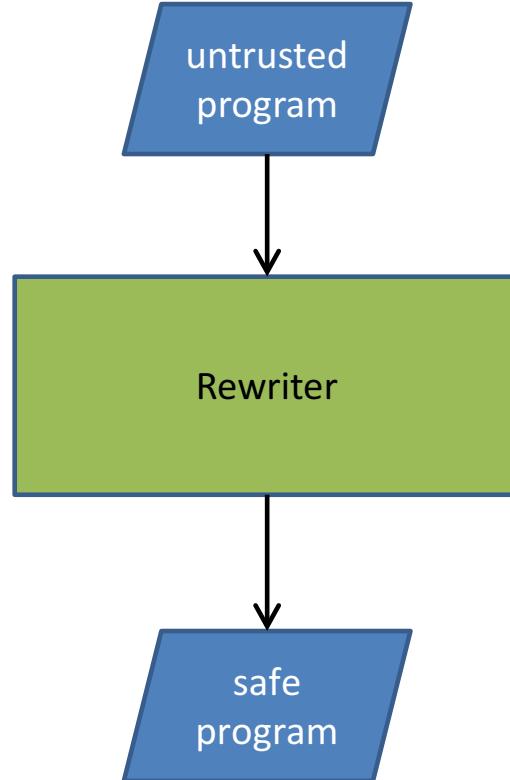
- Approach:
  - EM monitors events
  - intervenes to prevent violations
  - implemented outside program
- Cons:
  - no answer until execution
  - runtime slow-down (context-switches)
- Pros:
  - lower load-time overhead than static analysis
  - more powerful...?

co-Recursively Enumerable Policies



Adesso andiamo a vedere la strategia del inline reference monitor. Qui non ci sarebbe da fare, e questo è il motivo di questa parte bianca nella slide successiva, ci sarebbe da fare tutto un discorso sulle classi e sulle gerarchie di classi aritmetiche della teoria della calcolabilità che non ho voglia di farvi e che vi annoio. Quindi da questo punto di vista li possiamo vedere esattamente come i reference monitor, andiamo però a vedere l' approccio, analizzare le caratteristiche i pro e i contro.

## IRM Strategy



- Approach:
  - transform untrusted code
  - must return new program in finite time
  - transformed code must satisfy policy
  - behavior of safe code must be preserved
- Pros:
  - lowest runtime overhead
  - load-time overhead is once-only
  - sometimes no answer until execution



Allora l'approccio della inline reference monitor, quindi della instrumentazione è quello di trasformare del codice untrusted in un codice dove la politica viene messa all'interno del programma. Il punto è che dobbiamo farlo in modo automatico, quindi vuol dire che quella politica di instrumentazione deve essere fatta da un programma in un tempo finito. Chiaramente la trasformazione permette di codificare all'interno del programma untrusted la politica di sicurezza e poi bisogna garantire che il meccanismo di instrumentazione non intervenga sulla logica del programma, quindi questo è l'approccio, però è abbastanza ragionevole. Allora a questo punto andiamo ad esaminare le caratteristiche positive. Ovviamente l'overhead a runtime è limitato, perché l'analisi che viene fatta del codice prima della instrumentazione semplifica l'automa, quindi sostanzialmente vengono eseguite delle guardie nei punti critici e questo è sicuramente una cosa che non ci da un grande overhead a tempo di esecuzione. Il tempo di loading è immediato sto caricando il programma e prima faccio la trasformazione con instrumentazione ovviamente, e questo è quello che dicevo, è che anche qui siamo in una situazione simile al caso degli execution monitor, ho il risultato soltanto quando mando il programma in esecuzione, quindi non sono in grado di farlo prima di mandare il programma esecuzione. Questo era per concludere la parte sulla definizione delle politiche, quindi sappiamo che le politiche possono essere ricorsivamente, numerabile o co-ricorsivamente numerabili, le strategie di inlining e le strategie di execution monitor. Adesso facciamo un piccolo esercizio, ancora una volta ci mettiamo in un ottica a linguaggio di programmazione e facciamo vedere come possiamo scrivere e vedere non tanto l'inlining che lo faremo venerdì, ma faremo vedere come possiamo avere un linguaggio di programmazione in cui mettiamo una primitiva linguistica che è esattamente un security monitor, quindi questa è la del corso, cioè progettiamo un piccolo linguaggio in modo particolare, vedrete, è il linguaggio che andremo a progettare è un linguaggio banale di operazione aritmetiche, dove immersiamo un costrutto specifico che è il security monitor e quindi lo immersiamo a livello linguistico, che è la strategia che vogliamo vedere nel corso.