

Language Based Security

Ivano Salvo

Dipartimento di Informatica

Univ. di Roma “La Sapienza”

Materiali didattici

Essenzialmente basato sui lavori:

“Language Based Security” Dexter Kozen

“Attacking Malicious Code”

G. McGraw, G. Morrisett

“A Language approach to Security”

F. B. Schneider, G. Morrisett, R. Harper

Ulteriori approfondimenti e materiali:

Visitare pagina del corso di TSL:

twiki.di.uniroma1.it/twiki/view/

TSL/WebHome

Il problema della sicurezza

Tecniche convenzionali

- * Crittografia***
- * Firewall***
- * monitoraggio di sistema (system call)***
- * controllo degli accessi***

Problemi

- * download/execute di codice***
- * architetture software aperte (plug-in)***
- * utenti inesperti***

Evoluzione sistemi di calcolo

Negli anni 70 sistemi di calcolo erano isolati

- * pochi aggiornamenti software fatti da **amministratori esperti*****
- * chiara classificazione tra codice eseguibile e dati***
- * attacchi (es. virus) necessitavano di **accesso fisico** (floppy :)***
- * il costo economico di guasti e malfunzionamenti relativamente basso***
- * OS kernel relativamente piccoli***

Sempre La Rete...

Oggi...

aggiornamenti software continui e a volte fatti “inconsapevolmente”

ovunque è presente codice eseguibile

attacchi (es. virus) si diffondono rapidamente nella rete

aumento applicazioni critiche

(es: e-commerce, home banking, ...)

architetture aperte di molti programmi

Attacchi

Numerosi attacchi...

... Morris Worm (1988), Melissa & Love Bug (1999), ...

Tecniche

sfruttano vulnerabilità di comuni programmi

(MS outlook, X windows rlogin)

acquisizione brute force di password

utenti pochi esperti

Conseguenze:

una rilevante percentuale di calcolatori infettati

Perché hanno successo

Scripts invocati in modo trasparente

Eseguono con i privilegi dell'utente colpito

OS kernel “non conosce” messaggi mail, database dei contatti etc.

Recipients hanno fiducia dei sender

Interazioni in sistemi complessi non previste

Soluzioni 1

Eliminare tutti i contenuti eseguibili?

ok per e-mail (solo testo)

... ma in generale contenuti eseguibili sono utili (script, client-side form, codecs, ...)

Trade-off

Software aperto e estensibile vs sistemi chiusi
flessibilità vs sicurezza (least privilege)

Eliminare estensibilità è, in un certo senso, accettare un denial of service attack

Soluzioni 2

Anti virus

*basati su controlli **sintattici del codice**
Funzionano contro virus noti (**black list**)*

Code signing

*tecniche di crittografia per garantire che un
fornitore fidato ha prodotto il codice
(garantisce autenticità), ma:*

- * chiavi possono essere rubate*
- * **autentico non implica non malizioso***
- * difficile identificare eventuali
responsabilità*

Scanning, Signing, ...

Esamina proprietà superficiali del codice

- * “facile” da progettare***
- * dipendente dalla “forma” del codice***

Non analizza proprietà semantiche

- * difficile scoprire nuove forme di attacco***
- * si affida ad assunzioni spesso malintese (fiducia nel fornitore)***

Language Based Security

Tentativo di cercare soluzioni più generali basate sulla semantica e sull'analisi comportamento del codice

Spostare in parte l'onere della verifica e monitoraggio dal sistema ai linguaggi, sfruttando tecnologie ben sviluppate: compilatori, type-checker, partial evaluation, code rewriter...

Verifiche

Statiche (prima dell'esecuzione)

- *Analisi del codice (type-checking, dimostrazioni di correttezza)
- *Riscrittura del codice (strumentazione)
- *Salvataggio dati per garantire ripristino

Dinamiche (durante l'esecuzione)

- monitoraggio dell'esecuzione
- log di comportamenti indesiderati
- recover da comportamenti indesiderati

Talk outline

Protezione type-based e tipaggio di programmi assembler (Typed Assembly Language, JVM)

Monitoraggio e proprietà di safety

Instrumentazione del codice (Software-based fault Isolation e Security Automata-based Software-fault Isolation)

Parte I

***Typed Assembly
Language***

Typed Assembly Language

Idea

Mantenere nel codice oggetto informazioni di tipo generate durante la compilazione

Obiettivi

Verificare staticamente che il codice non eseguirà operazioni “pericolose” (es: accessi scorretti in memoria)

Type Systems

Tipi

Oggetti sintattici che descrivono:

- *range di valori ammessi*
- *operazioni possibili*

Favoriscono la data abstraction

Usualmente, i controlli di tipo vengono effettuati staticamente (*no run-time overhead*)

Type Safety

I programmi ben tipati non generano errori di tipo durante l'esecuzione

Errori di tipo: *dipendono dal linguaggio*

- ***OOP: message-not-understood***
- ***mescolare interi e puntatori***

Necessari, a volte controlli dinamici:
array index out-of-bound
(viene generato codice)

Type Safety: conseguenze

Numerosi benefici:

- **no buffer overruns**
- **no codice che si automodifica**

Garantire la type safety: difficile

**Necessita di una dimostrazione formale
tediosa (**proof-assistant**): usualmente
dimostrata solo per core-language**

Trade-off: safety vs flessibilità/efficienza

Typed Assembly Language

Altri Obiettivi:

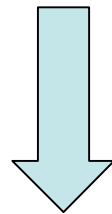
Ridurre la TCB (type-checker, no compilatore)

Non interferire con le comuni ottimizzazioni dei compilatori

Evitare costose verifiche run-time

Schema TAL

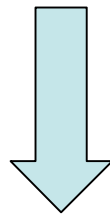
Programma alto livello
(linguaggio *type safe*)



Compilatore

untrusted

Programma Assembler
(corredato di *annotazioni*
di tipo)



Type Checker

TCB

Esecuzione "safe"

TAL: un assembler RISC idealizzato

Registri: $r \in \{\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \dots\}$

Labels: $L \in Identifier$

Interi: $n \in [-2^{k-1} .. 2^{k-1})$

Blocchi: $B ::= \mathbf{jmp} \ v \mid \iota ; B$

Istruzioni: $\iota ::= aop \ r_d, r_s, v \mid bop \ r, v \mid \mathbf{mov} \ r, v$

Operandi: $v ::= r \mid n \mid L$

Aritmetica: $aop ::= \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \dots$

Branching: $bop ::= \mathbf{beq} \mid \mathbf{bne} \mid \mathbf{bgt} \mid \dots$

Semantica Operazionale

***Attraverso transizioni di una
macchina:***

(H, R, B)

dove:

H ***associazioni label-blocco***

R ***associazioni registri-valori***

B ***blocco in esecuzione***

Semantica Operazionale

$(H, R, \text{mov } r_d, v ; B) \rightarrow (H, R[r_d := \underline{R}(v)], B)$

dove $\underline{R}(r) = R(r)$ altrimenti $\underline{R}(v) = v$

$(H, R, \text{add } r_d, r_s, v ; B) \rightarrow (H, R[r_d := n], B)$

dove $n = \underline{R}(v) + R(r_s)$

$(H, R, \text{jmp } v) \rightarrow (H, R, B)$

dove $\underline{R}(v) = L$ e $H(L) = B$

$(H, R, \text{beq } r, v ; B) \rightarrow (H, R, B)$

dove $R(r) \neq 0$

$(H, R, \text{beq } r, v ; B) \rightarrow (H, R, B')$

dove $R(r) = 0$, $\underline{R}(v) = L$, e $H(L) = B'$

Errori

Sono considerate di errore le configurazioni in cui si usano in modo incoerente interi e label.

Ad esempio:

$(H, R, \text{add } r_d, r_s, v; B)$ con r_s e v non interi

$(H, R, \text{jmp } v)$ con v non label

$(H, R, \text{beq } r, v; B)$ con v non label, r non intero

La semantica operativa è uno strumento concettuale!

Struttura dei tipi

$\tau ::= \mathit{int} \mid \Gamma$
con $\Gamma = \{r_1: \tau_1, r_2: \tau_2, r_3: \tau_3, \dots\}$

Un valore di tipo Γ deve essere la label di un blocco di codice ben tipato sotto le assunzioni fatte in Γ sul tipo dei registri

record subtyping: posso fare più assunzioni del necessario

Tipaggio di Istruzioni

Permettono di derivare **giudizi** della forma:

$$\Psi \triangleright \iota : \Gamma_1 \rightarrow \Gamma_2$$

L'istruzione ι è **ben tipata** se i registri hanno i tipi in Γ_1 (**precondizioni**) e produce un tipaggio dei registri Γ_2 (**postcondizioni**), date le assunzioni Ψ sulle label (**invarianti**)

Esempi: istruzioni

Istruzioni aritmetiche

$$\frac{\Psi, \Gamma \triangleright r1: int \quad \Psi, \Gamma \triangleright v: int}{\Psi \triangleright add\ r2, r1, v : \Gamma \rightarrow \Gamma [r2 := int]}$$

Salti condizionati

$$\frac{\Psi, \Gamma \triangleright r: int \quad \Psi, \Gamma \triangleright v: \Gamma}{\Psi \triangleright bop\ r, v : \Gamma \rightarrow \Gamma}$$

Trasferimento

$$\frac{\Psi, \Gamma \triangleright v: \tau}{\Psi \triangleright mov\ r, v : \Gamma \rightarrow \Gamma [r2 := \tau]}$$

Esempi: blocchi

salto incondizionati

$$\frac{\Psi, \Gamma \triangleright v : \Gamma}{\Psi \triangleright \text{jmp } v : \Gamma}$$

sequenza

$$\frac{\Psi, \Gamma \triangleright \iota : \Gamma' \rightarrow \Gamma'' \quad \Psi \triangleright B : \Gamma''}{\Psi \triangleright \iota ; B : \Gamma'}$$

Esempio: fattoriale

```
fact: {r1:int, r2:int, r31: {r1:int}}  
    ; r1 = n, r2 = accum, r31 = return address  
    sub  r3, r1, 1; {r1:int, r31: {r1:int}, r3:int}  
    bge  r1, L2  
    mul  r2, r2, r1  
    mov  r1, r3  
    jmp  fact  
L2: {r2:int, r31: {r1:int}}  
    mov  r1, r2  
    jmp  r31
```

Esempio: errore di tipo

```
fact: {r1:int, r31:{r1:int}}
```

```
; r1 = n, r2 = accum, r31 = return address
```

```
sub r3, r1, 1; {r1:int, r31:{r1:int}, r3:int}
```

```
bge r1, L2
```

```
mul r2, r2, r1 ; ERROR! r2 non ha tipo
```

```
mov r1, r3
```

```
jmp L1 ; ERROR! L1 non è label valida
```

```
L2: {r2:int, r31:{r1:int}}
```

```
mov r31, r2
```

```
jmp r31 ; ERROR! r31 non è label valida  
dopo l'assegnazione
```

Proprietà di TAL

[Progress] *Se Σ è una configurazione ben tipata, esiste Σ' tale che $\Sigma \rightarrow \Sigma'$*

+

[Preservation] *Se Σ è ben tipata, e $\Sigma \rightarrow \Sigma'$ anche Σ' è ben tipata*

=

[Type Safety] *I programmi ben tipati non generano errori di tipo durante l'esecuzione*

Corollari

Tutti i salti sono verso label valide

Le operazioni aritmetiche sono fatte con interi

Ma usando tipi più sistemi di tipo più sofisticati molto di più:

- * **corretto uso dello stack**
- * **corretta allocazione deallocazione dell'heap**
- * **controllo quantità di memoria**
- * **moduli, link checker, dynamic link**

TAL vs JVM

Il principio di TAL è simile a quello sottostante alla JVM

TAL si propone come **target-RISC** assembly language di un generico linguaggio **type-safe**

JVM è un bytecode **specificatamente studiato** per una particolare classe di linguaggi OOP

Prototipi

PopCorn/Cyclone: dialetti type-safe C

Obiettivi:

- * *memory/type safety***
- * *facilità di portare i programmi C***
- * *new features: polimorfismo, tipi di dato algebrici, garbage collector***
- * *mantenere puntatori e relativa aritmetica***
- * *coniugare safety e performance***

Ostruzioni pratiche

I tipi risultanti dalla compilazione possono essere grandi

Soluzioni:

type inference (rende più complessa la TCB)

compressione dei tipi (rappresentazione via grafi, hashing,..)

Riassumendo TAL

Trusted Computing Base

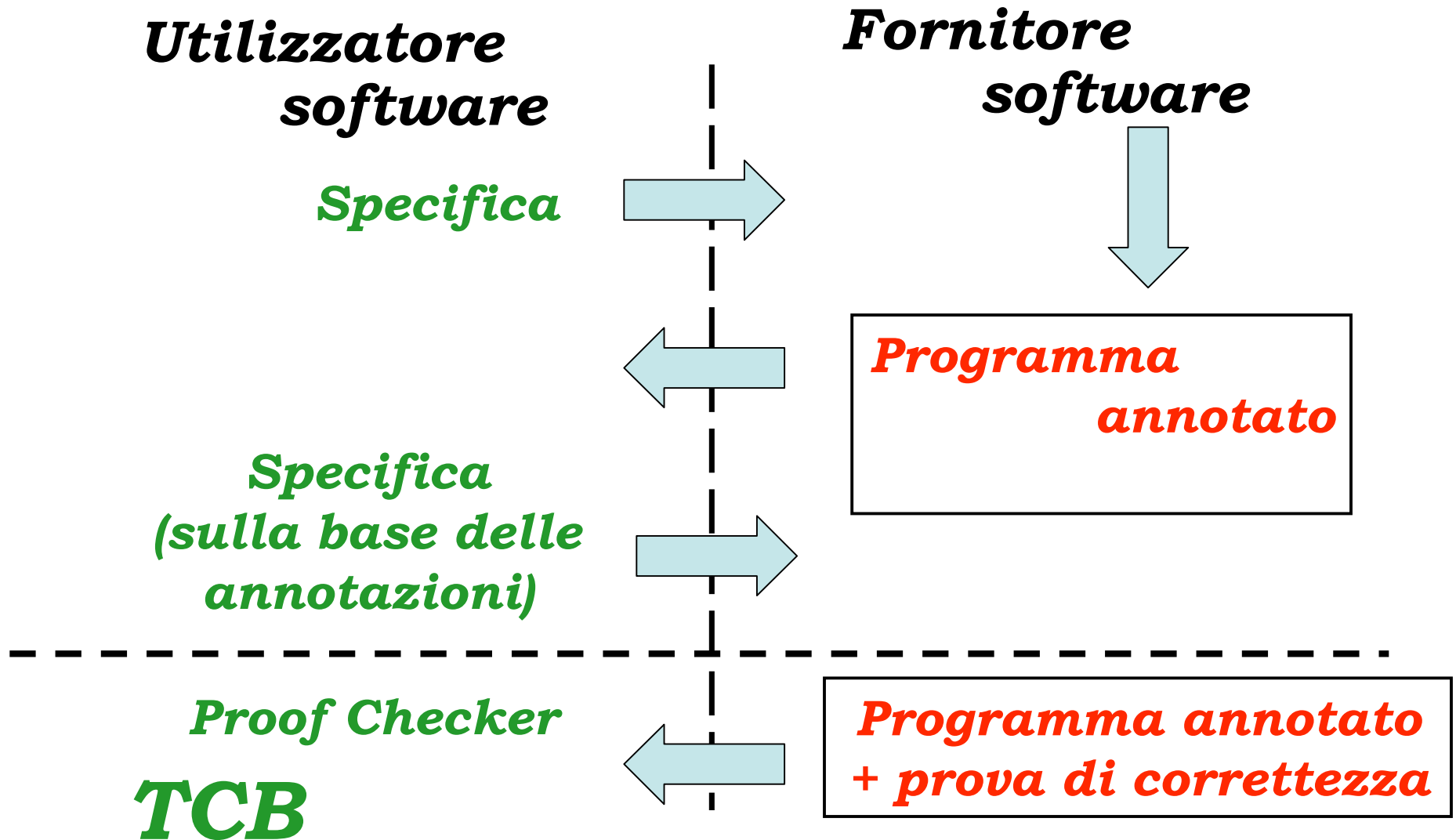
Solamente il **type-checker** del codice oggetto

Necessità:

***Un **linguaggio sorgente type safe**
(Java, ML, frammenti/dialetti del C)***

***Una **typed-preserving compilation**,
che generi le annotazioni di tipo***

Detour: Proof Carrying Code



PCC: valutazione

Campo di applicazione

- * ***requirement di sicurezza/correttezza molto importanti (interessi economici, controllo di impianti etc.)***
- * ***assolutamente generale***

Problemi

- * ***richiede programmatori/progettisti esperti e familiari con logica***
- * ***richiede ampio uso di tool come proof assistant e il giusto framework logico in cui esprimere le proprietà***

Parte II

In-line reference Monitor

In-line reference monitor

Idea

instrumentare opportunamente il codice prima dell'esecuzione, proteggendo operazioni pericolose

Vantaggi rispetto a un monitor di sistema:

Evita l'overhead dei context-switching

Flessibilità (è possibile definire le proprietà di sicurezza che si intende garantire)

Reference monitor

***Osserva l'esecuzione di un programma
e lo ferma se viola una politica di
sicurezza***

Esempi

sistema operativo (hardware-based)

interpreti (software-based)

firewall

Proprietà di un monitor

- * Deve avere **accesso all'informazione** su cosa il programma sta per fare*
- * Deve poter essere in grado di **fermare il programma***
- * Deve poter **proteggere se stesso** dal programma monitorato*
- * Deve avere un **basso overhead***

Quali politiche di sicurezza?

Un monitor vede solo **sequenze di esecuzione** di un programma

Si possono solo garantire politiche \mathcal{P} nella forma:

$$\mathcal{P}(S) = \forall \sigma \in S. P(\sigma)$$

con P **predicato sulla singola esecuzione** σ .

Un insieme di esecuzioni è detto **proprietà** se **l'appartenenza è determinata dalle singole esecuzioni**

Quali politiche di sicurezza?

- * Un monitor non vede il futuro
- * Deve prendere decisioni in tempo finito
- * Se $P(\sigma)$ è vero, ma $P(\sigma')$ falso con σ' prefisso di σ il monitor elimina σ' e tutti i suoi suffissi.

Formalmente, P è prefix-closed:

$\forall \sigma \in S. P(\sigma)$ se e solo se $P(\sigma')$, $\sigma' \leq \sigma$

Una proprietà prefix-closed è una safety property

Safety Properties

Le safety properties non caratterizzano tutte le proprietà di sicurezza

Information flow è una proprietà di insiemi di esecuzioni e non di esecuzioni singole

Le safety property garantiscono che qualcosa di indesiderato non accada, ma non:

- * terminazione del programma***
- * rilascio di lock***
- * liveness properties***

Safety Property: vantaggi

****Hanno buone proprietà **composizionali*****

Se P e R sono safety, lo è anche $P \& R$ (intersezione delle tracce)

****Le safety property **possono*****

*****approssimare** proprietà di **liveness*****

(esempio: un programma termina entro k passi)

Possono approssimare altre proprietà di sicurezza

Approccio Tradizionale

Modello Semplificato:

Un sistema è una collezione di **processi** (soggetti) e **file** (oggetti)

I processi possono fare delle operazioni sulla base dei **diritti di un utente**

I file hanno una **lista di controllo degli accessi** che determina chi può fare cosa

Obiettivi e Tecniche

Integrity: protezione tra utenti (e OS)

- * **esecuzioni in isolamento**
- * **operazioni privilegiate (OS, su)**

Availability: processi non possono mantenere indefinitamente il controllo di risorse condivise

- * **time-out**
- * **memory quota**

Secrecy? Confidentiality? Access control?

Software-based enforcement: motivazioni

***Idealmente i sistemi possono essere
decomposti in sottoprocessi separati
(according **least-privilege principle**)***

***Costi computazionali di comunicazione
spesso inducono programmatori a non
farlo***

 ***molte applicazioni hanno una
struttura **monolitica*****

Software Fault Isolation (SFI)

Wahbe et al. [1993]

Idea:

***mantenere diversi componenti software
nello stesso spazio di indirizzamento***

***proteggere ogni jump, read, write con
un controllo preventivo***

***Trade-off: check overhead vs
communication***

SFI: usare interpreti

Vantaggi:

- * facile da implementare (piccola TCB)***
- * funziona per qualsiasi tipo di assembler***
- * facile implementare diverse politiche di sicurezza***

Svantaggi:

Overhead del ciclo decodifica/esequi istruzione (fino a 70 volte più lento)

Interpreti: partial evaluation

Si specializza l'interprete per il codice da eseguire, eseguendo tutto ciò che è staticamente eseguibile

Si valuta tutto ciò che è staticamente valutabile (semplice in principio)

Si cancella tutto il codice, che a seguito della specializzazione risulta inutile

Esempio di partial evaluation

Codice binario:

```
0: add r1,r2,r3
1: ld r4,r3(12)
2: jmp r4
...
```

Interprete:

```
while (true) {
    if (pc >= codesz) exit(1);
    int inst = code[pc];
    ...
}
```

“srotolo” il ciclo decodifica/esegui




Interprete specializzato:

```
reg[1] = reg[2] + reg[3];
addr = reg[3] + 12;
if (addr >= memsz) exit(1);
reg[4] = mem[addr];
pc = reg[4]
```

Codice compilato:

```
0: add r1,r2,r3
1: addi r5,r3,12
2: subi r6,r5,memsz
3: jab _exit
4: ld r4,r5(0)
...
```



SFI: code rewriter & problemi

Ogni accesso viene protetto da 4 istruzioni, richiede 5 registri dedicati,
**** funziona male su architetture con piccoli insiemi di registri***

**** Interagisce allocazione registri dei compilatori!***

Rilocazione dei jump (difficile per i jump calcolati)

**** Deve assicurare che le istruzioni di controllo non siano aggirate***

SASI

***S*ecurity**

***A*utomata-based**

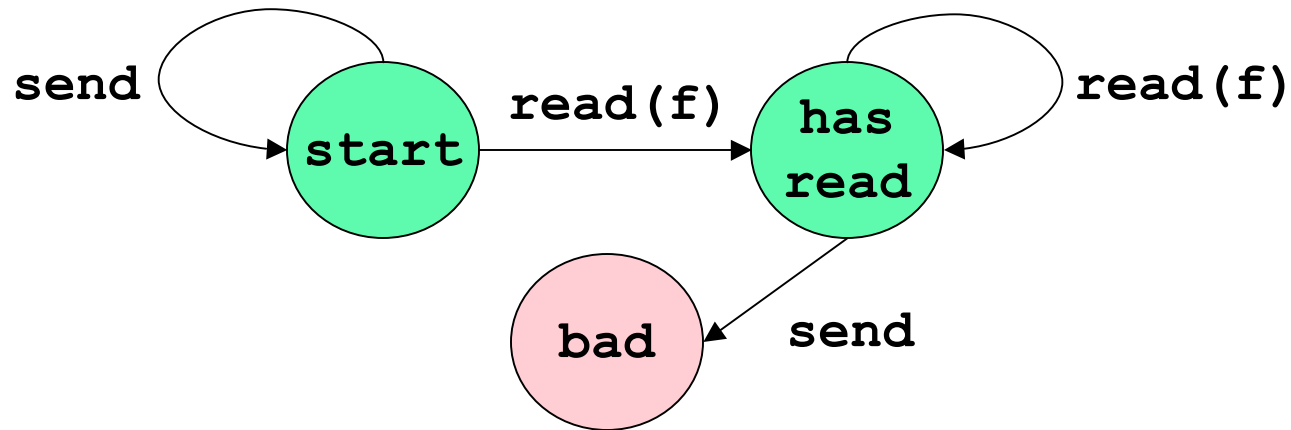
***S*oftware fault**

***I*solation**

[Schneider & Erlingsson, 1999]

***Ogni safety è descrivibile mediante
un automa che accetta il
linguaggio delle esecuzioni safe***

SASI: Esempio



*Automa descritto con un linguaggio stile i
guarded-Command di Dijkstra
(condizione → transizione):*

$B \rightarrow S$

SASI: funzionamento

Specifica automa, quindi:

- 1. Inserire il Security automata **prima** di ogni istruzione***
- 2. Valutare le transizioni (ove possibile: **partial evaluation**)***
- 3. Cancellare transizioni che valutano a false***
- 4. Tradurre l'automata in codice***

Si può fare di più: data-flow analysis

SASI: integrity

Garantire che le *variabili* che definiscono l'automata *non* siano *modificate dal codice da monitorare*

Garantire che i *jump* *non* aggirino le *istruzioni dell'automata*

Evitare che il codice modifichi sé stesso (e l'automata)

In generale: platform dependent!

SASI: prototipo x86

***Suppone l'assembler output di gcc
(register convention, no self-
modifying code, etc.)***

***Salti ristretti a un insieme di label
valide***

***Si può usare un security automa per
bloccare codice che non rispetta
queste regole!***

Problemi:

- * gcc parte della TCB***
- * poco generale***

SASI: prototipo JVMML

Type safety di Java garantisce:

- * separazione variabili
automa/variabili di programma
- * le invocazioni di metodo sono salti
a label valide

Il bytecode JAVA conserva le astrazioni
su cui è costruito il programma
(classi, metodi) e permette su queste
di definire le politiche di sicurezza!

Soluzione generale: assembler tipato?
Disassemblare? Source code?
Abstraction preserving compilation?

SASI: valutazione

Diverse implementazioni (Naccio, Pslang)

Test condotti su diversi benchmark, con ottimi risultati (es: rappresentare con un automa le politiche di sicurezza di JVM e confrontare il risultato con la JVM stessa)

Difficile specificare complesse politiche di sicurezza su codice di basso livello

Parte III

Information Flow e prospettive

Prospettive

Short term: assicurare programmi sono *type-safe, memory safe, fine-grained control policies*

Long term: assicurare che complessi sistemi (distribuiti) soddisfino garanzie globali di sicurezza (***system-wide***)

Confidentiality (flussi ***verso destinazioni ostili***) vs **integrity** (flussi ***da sorgenti ostili***)

Confidentiality

I sistemi tradizionali basati su controllo degli accessi (MAC, DAC, ruoli) garantiscono che un utente acceda a informazioni a lui riservate, ma **non controllano l'uso fatto con queste informazioni!**

Anche software non malizioso potrebbe rilasciare informazione (**flussi impliciti, covert-channel**); errori o inaccuratazze progettuali

Information flow: esempio

```
h=h%2;
```

```
l=0;
```

```
if (h==1) then l=1; else {}
```

Il valore finale di l dipende in modo indiretto dal valore di h

Il fatto che da l si possa inferire informazione sul valore di h non è una proprietà di un'esecuzione

Non è una safety, non basta un monitor

Information flow: definizione

Non-interference [Goguen, Mesequer, 82]

Il “comportamento basso” di un programma non dipende dagli input alti (alto = confidenziale, basso = pubblico)

Facile a dirsi...

**** cosa è osservabile? (tempo di esecuzione? Distribuzione di probabilità degli output? Terminazione? struttura del programma...)***

Formalizzazione

La **semantica** di un programma (comando) è una **trasformazione di stati** (memoria)

$$|C|:S \rightarrow S_{\perp}$$

Uno stato $S=(S_L, S_H)$ distingue componenti basse e alte, su cui è definita un'**equivalenza bassa**

$$S =_L S' \text{ sse } S_H = S'_H$$

C è non interferente se la sua **semantica non è sensibile ai valori alti**

$$\forall s, s'. s =_L s' \Rightarrow |C|(s) =_L |C|(s')$$

Idea: usare i tipi

```
int {H} x; // intero privato
int {L} y; // intero pubblico
x = y;
    /* OK, informazione "sale */
y = x;
    /* BAD, informazione "scende */
if (x==y) y=1 else y=0;
    /* BAD, la confidenzialità di
    un'espressione è il sup */
```

I programmi ben tipati sono non-interferenti!

Regole di Tipo

espressioni

$$\triangleright \text{exp} : H \quad \frac{h \notin \text{Var}(\text{exp})}{[pc] \triangleright \text{skip}}$$

comandi

$$[pc] \triangleright \text{skip} \quad [pc] \triangleright h = \text{exp}$$

$$\frac{\triangleright \text{exp} : \text{low}}{[low] \triangleright l = \text{exp}} \quad \frac{[high] \triangleright c}{[low] \triangleright c}$$

$$\frac{\triangleright \text{exp} : pc \quad [pc] \triangleright C' \quad [pc] \triangleright C''}{[pc] \triangleright \text{if} (\text{exp}) C' \text{ else } C''}$$

Esempi

h=1+4;

sicuro, perché il low output non cambia

if (l==3) h=h+1; else l=l+1;

sicuro, perché il low output dipende solo dai valori di *l*

l=h+1;

if (h==3) l=5; else skip;

insicuri, perché il low output cambia al variare dei valori di *h*

Osservazioni

I programmi ben tipati sono non-interferenti (si dimostra in modo analogo a Type Soundness)

*Ci sono programmi **non-interferenti** che comunque vengono scartati dal type-system*

*I tipi che garantiscono non-interference sono **molto restrittivi!***

IF: Conclusioni

Difficile da determinare: dipende in gran parte dalle assunzioni fatte sulle **capacità dell'attaccante.**

Obiettivi:

- * Tecniche di verifica efficaci ma sicure***
- * Analisi statica di protocolli***
- * Concorrenti e distribuiti***
- * covert channel***