Indice

Capitolo 1

Introduzione

Con la nascita di Internet, la sicurezza del codice mobile è diventata una delle sfide più grandi nell'area di ricerca della computazione. Ogni giorno diventiamo sempre più dipendenti dal mondo dell'informatica e di conseguenza vulnerabili agli attacchi *maliziosi* e a software *buggy*. Un codice malizioso è un qualsiasi codice aggiunto, modificato o rimosso da un sistema software con l'intenzione di causare danni oppure sovvertire le funzionalità del sistema [1]. Esempi tradizionali di codice malizioso includono virus, worm, cavalli di Troja ed inoltre attacchi da parte di codice esterno.
Supponiamo di voler scaricare ed eseguire un programma da una sorgente non conosciuta oppure non affidabile. Prima di eseguire il programma, è essenziale avere delle garanzie sulla sicurezza di tale programma, dove il concetto di sicurezza può essere interpretato diversamente a seconda dell'applicazione in questione. La definizione di "sicuro" utilizzata per una particolare applicazione, è chiamata *politica di sicurezza* di quell'applicazione. Ovvero, una politica di sicurezza indica l'insieme di operazioni che non sono considerate accettabili all'interno di un sistema. Queste operazioni possono riguardare, ad esempio, il controllo degli accessi. Come minimo, una qualsiasi politica

di sicurezza per programmi potenzialmente non affidabili, deve garantire le seguenti proprietà di sicurezza:

- *Sicurezza del Flusso di Controllo.* Il programma non deve mai eseguire un salto oppure una chiamata a locazioni di memoria casuali, ma solo ad indirizzi interni al suo segmento di codice che contiene istruzioni valide. Inoltre, tutte le chiamate alle funzioni e i return devono essere fatti da locazioni valide.

- *Sicurezza della Memoria.* Il programma non deve mai accedere a locazioni casuali della memoria, ma soltanto a locazioni valide nel suo segmento di dati statico, nella memoria del heap esplicitamente allocata dal sistema al programma e ai frame validi dello stack.

- *Sicurezza dello Stack.* Per le architetture che usano uno stack a tempo di esecuzione, lo stack si deve preservare durante le chiamate alle funzioni.

Queste tre forme di sicurezza risultano essere interdipendenti. Esistono tecniche basate sulla nozione di *tipo* per verificare queste (ed altre) forme di sicurezza. Naturalmente queste tecniche sono applicabili solo alla presenza di una disciplina di tipaggio.

Garantire la sicurezza all'interno di un sistema è un compito difficile e complesso. Si devono soddisfare diversi requisiti, tra i più importanti e comuni ci sono i seguenti:

- Un utente non deve poter leggere dei dati ai quali non ha accesso. (Confidenzialità)

- Un utente non deve poter modificare dei dati se non ha i diritti necessari. (Integrità)

- Un utente si deve autenticare all'interno di un sistema in modo sicuro. Ovvero, il sistema di autenticazione deve essere in grado di riconoscere un utente tramite un qualsiasi meccanismo, ad esempio password ed impedire agli intrusi di ottenere le credenziali di un altro utente. (Autenticazione)

- Un utente che possiede i diritti di accesso a dei dati, deve poter effettivamente accedere ad essi. (Disponibilità)

Per garantire la sicurezza dei computer e della rete esistono diversi meccanismi. Tra quelli più usati e comuni ci sono: kernel come monitor di referenza, crittografia e instrumentazione del codice. Questi meccanismi vengono descritti in dettaglio nel capitolo 2. È stato inoltre recentemente proposto un nuovo approccio alla sicurezza è *basato sui linguaggi*, che consiste in un insieme di tecniche basate sulla teoria e l'implementazione dei linguaggi di programmazione, che utilizzano nozioni di semantiche, tipi, ottimizzazione e verifica, applicate alla sicurezza. I capitoli 2 e 3 danno una panoramica di queste tecniche.

Un esempio tipico della sicurezza basata sui linguaggi è il *Java bytecode verification*. Questa verifica del bytecode viene fatta per evitare la violazione di sicurezza da parte di un'*applet* Java. Nelle prime versioni di Java invece, le applet potevano sfruttare errori semantici nella definizione del linguaggio per portare avanti un attacco al sistema di sicurezza. Per esempio, le discrepanze tra la semantica di Java e del suo bytecode permettevano alle applet di controllare il `ClassLoader` del sistema ed utilizzarlo in modo malizioso. Una spiegazione dettagliata di questo esempio viene data nell'appendice A.

Tra i requisiti per la sicurezza che abbiamo visto precedentemente, la confidenzialità è uno dei più complessi e delicati. La parte più difficile nel garantire la confidenzialità è quella di proteggere i dati sensibili durante la computazione degli stessi. Tra i metodi tradizionali per garantire la confidenzialità troviamo il controllo degli accessi, firewall, antivirus, crittografia e *mandatory access control*. Un altro metodo è quello del controllo del flusso di informazioni. In particolare, vedremo nel capitolo 3 come questo possa essere effettuato staticamente utilizzando un *sistema dei tipi di sicurezza*. Diversi autori [8, 10] hanno proposto sistemi dei tipi capaci di catturare forme diverse di *information flow*. Nel presente tirocinio ci siamo concentrati sul sistema proposto da Myers e Sabelfeld in [8] che cattura l'*implicit flow* (vd. capitolo 3). Notando che dei semplici programmi C non venivano riconosciuti sicuri da questo sistema pur essendolo, abbiamo proposto nel capitolo 4 una variante delle regole di tipaggio che cattura alcuni di essi. Abbiamo dimostrato che ogni programma tipabile nel sistema dei tipi di [8] è tipabile anche nel nostro sistema dei tipi, ed inoltre abbiamo dimostrato che questo nuovo sistema è corretto (*sound*).

Il resto di questa trattazione è organizzato come segue: il secondo capitolo tratta le varie tecniche di sicurezza, dividendole in tecniche tradizionali e tecniche basate sui linguaggi. Nel terzo capitolo discutiamo una particolare tecnica basata sui linguaggi, il controllo del flusso di informazioni, e presentiamo il sistema di Myers e Sabelfeld. Nel quarto capitolo estendiamo questo sistema e mostriamo esempi di derivazione per programmi sicuri non catturati dal sistema precedente.

Capitolo 2

Tecniche di Sicurezza

In questo capitolo, tratto dall'articolo *Language-Based Security* di Dexter Kozen [5], parliamo di varie tecniche di sicurezza, dividendole in tecniche tradizionali e tecniche basate sui linguaggi.

La sicurezza si basa sul concetto di *trust*. È possibile partizionare il software in due parti, *trusted* e *untrusted*. Tutto il software che è trusted forma il cosiddetto *trusted computing base*, che vogliamo sia il più piccolo possibile; questo per il principio di *minimal trusted computing base* che afferma: la garanzia che un meccanismo si comporti come deve è tanto più grande quanto più piccolo e semplice è tale meccanismo.

2.1 Tecniche Tradizionali

Le tecniche tradizionali per la sicurezza offrono un insieme fissato di politiche di sicurezza basilari con poca flessibilità. Tra questi meccanismi si includono i seguenti.

Kernel come monitor di referenza. È il più vecchio e diffuso meccanismo di sicurezza usato nei sistemi software. Questo meccanismo consiste nell'isolare le operazioni fatte sui dati e sulle componenti critiche del sistema nel *kernel del sistema.* Il kernel è un codice privilegiato che può accedere a questi dati e componenti direttamente, mentre tutti gli altri processi possono accedervi solo mediante di esso, comunicando le operazioni che vogliono effettuare tramite messaggi al kernel. Questo impedisce al codice non affidabile di manipolare il sistema ed inoltre permette al kernel di monitorare tutti gli accessi.

Crittografia. La crittografia può scorraggiare l'accesso ai dati sensibili che transitano in una rete non affidabile ed essere usata per l'autenticazione. Non può essere usata da sola come meccanismo di sicurezza in quanto non garantisce che il codice scaricato può essere eseguito in modo sicuro, ma soltanto da quale sorgente è partito e se ha subito alterazioni durante il transito.

Instrumentazione del codice. Si riferisce al processo di alterare (instrumentare) il codice macchina in modo che le operazioni critiche possano essere monitorate durante l'esecuzione.
Tale instrumentazione è fatta in un modo tale che (i) il codice instrumentato abbia lo stesso comportamento funzionale del codice originale, se quest'ultimo non ha violato la politica di sicurezza; altrimenti, (ii) nel momento della violazione, il codice instrumentato impedisce che tale violazione abbia degli effetti sul resto del sistema, oppure la rileva e passa il controllo al sistema il quale chiude il processo che ha violato la sicurezza.

Un esempio di instrumentazione del codice è il *software fault isolation* (SFI) oppure *sandboxing* dove il codice non affidabile viene caricato in un blocco continuo della memoria, chiamato "sandbox" e poi viene instrumen-

tato, riscrivendo le istruzioni che contengono accessi di memoria e indirizzi di salto al di fuori del sandbox. In questo modo un programma corretto non viene modificato, mentre un programma non corretto viene riscritto. Tale riscrittura, effetuata solo all'interno del sandbox, non influisce sul resto del sistema.

Un'altro esempio è il *security automata*, dove ogni politica di sicurezza è descrivibile mediante un automa che accetta il linguaggio delle esecuzioni sicure. Il codice viene instrumetato in modo che, ogni istruzione sia preceduta da una chiamata all'automa. Se tale chiamata porta ad uno stato sicuro dell'automa, allora l'istruzione viene eseguita, altrimenti no.
In seguito parliamo di meccanismi di sicurezza orientati ai linguaggi, che sono dei meccanismi più recenti rispetto ai meccanismi tradizionali appena discussi.

2.2 Sicurezza Basata sui Linguaggi

L'approccio alla sicurezza basato sui linguaggi consiste in un insieme di tecniche basate sulla teoria e l'implementazione dei linguaggi di programmazione, includendo semantiche, tipi, ottimizzazione e verifica, applicate alla sicurezza.
L'idea è di spostare in parte la verifica e il monitoraggio dal sistema ai linguaggi e si basa sull' **analisi** e la **riscrittura** del programma.
I meccanismi di *enforcement* che si occupano di rilevare programmi reputati non sicuri prima della loro esecuzione, quindi staticamente, vengono definiti di **analisi statica**. Tali meccanismi devono verificare se il programma preso in analisi soddisfi o meno la politica di sicurezza in un tempo finito. I programmi accettati da tali politiche di sicurezza possono essere mandati in

esecuzione, gli altri no. Un esempio di tale approccio è il sistema dei tipi di TAL che verrà discusso in seguito.
La **riscrittura** dei programmi si riferisce a quei meccanismi di enforcement che, in un tempo finito, modificano un programma malizioso prima della sua esecuzione. Da quanto detto, la tecnica di instrumentazione del codice, precedentemente vista, è un'istanza della tecnica di sicurezza basata sui linguaggi.

L'approccio di sicurezza basata sui linguaggi non è ristretto solamente ai sistemi programmati in un linguaggio ad alto livello (come ad esempio JFlow [6]). Infatti, tanti lavori, come vedremo in seguito, applicano le politiche di sicurezza a livello di codice oggetto. Questo perché (i) il trusted computing base è più piccolo senza il compilatore e (ii) le politiche di sicurezza si possono applicare anche in assenza del codice sorgente.

I compilatori per i linguaggi di programmazione di alto livello accumulano tante informazioni durante la compilazione di un programma, come ad esempio informazioni sui tipi, restrizioni sui valori delle variabili, informazioni riguardanti la struttura o sui nomi. Dopo una compilazione con successo, il compilatore butta tutte le informazioni accumulate. Queste informazioni possono avere implicazioni riguardanti la sicurezza del codice oggetto compilato. Quindi, l'idea che sta dietro a questa versione di sicurezza basata sui linguaggi è quella di conservare nel codice oggetto alcune informazioni aggiuntive ottenute compilando un programma scritto in un linguaggio di alto livello. Queste informazioni aggiuntive, che chiameremo *certificato*, vengono impacchettate insieme al codice oggetto. Quando viene scaricato il codice oggetto, viene scaricato anche il certificato. A questo punto il consumatore del programma esegue un *verificatore* che prende in input il codice e il

certificato e ne controlla la consistenza con la politica di sicurezza. Il verificatore è trusted (vd. introduzione), mentre il codice oggetto, il certificato e il compilatore non lo sono necessariamente. Il beneficio principale di questo appriocio è che la responsabilità di verificare la conformità con la politica di sicurezza in questione è spostata dal consumatore al fornitore del codice. Infatti, è il fornitore del codice che deve fornire il certificato, mentre il compito del consumatore del codice è quello di *controllare* il certificato e non più di *provare* che il codice sia sicuro ad essere eseguito, sicuramente un compito più facile.

Figura 2.1: Sicurezza Basata sui Linguaggi

Un esempio di questo approccio è il *Java bytecode verification.*
Il linguaggio Java dispone di meccanismi di sicurezza contro le applet (vd. cap.1 e appendice) maliziose. Un programma scritto in linguaggio Java viene compilato in un codice indipendente dalla piattaforma, chiamato *bytecode*, il quale viene dato in input alla JVM, per essere poi interpretato oppure compilato da un compilatore Just-in-time (JIT). Il bytecode contiene informazioni rilevanti per quanto riguarda la sicurezza, come ad esempio informazioni sui tipi. Le applet scaricate sono in bytecode e vengono date alla JVM. In fase di caricamento del codice, questo per evitare di farlo dinamicamente in fase di esecuzione e quindi evitare l'overhead, l'applet viene verificata da un *bytecode verifier* che fa parte della VM. Il verificatore controlla che il bytecode sia ben formato e che rispecchi criteri di sicurezza e di garanzia. Le informazioni controllate dal verificatore sono: correttezza dei tipi, stack overflow e underflow, inizializzazione degli oggetti ecc. Il bytecode che passa la verifica con successo può essere eseguito velocemente facendo solo quelle verifiche dinamiche (ovvero a tempo di esecuzione) che non si possono fare staticamente,

come ad esempio il controllo sui bound degli array.
In seguito vedremo dei meccanismi di sicurezza basati sui linguaggi come il Proof Carrying Code, Linguaggio Assembler Tipato e Certificati di Codice Efficienti.

2.2.1 Proof Carrying Code (PCC)

Proof Carrying Code si riferisce alla metodologia di produrre e verificare dimostrazioni formali di politiche di sicurezza generali, prima che il codice venga eseguito. Le condizioni di sicurezza sono espresse con la logica dei predicati del primo ordine estese con predicati per la sicurezza sui tipi e sulla memoria. Il meccanismo funziona come segue.
Il fornitore di codice produce un programma che consiste di codice oggetto annotato e lo manda al consumatore. Le annotazioni sono invarianti di ciclo e pre- e postcondizioni di funzioni. Il consumatore, che ha in mente una particolare politica di sicurezza, produce una *condizione di verifica*, una formula logica che implica che il programma mandato dal fornitore soddisfa la politica di sicurezza. Il consumatore manda questa condizione di verifica al fornitore. Una dimostrazione della condizione di verifica è una dimostrazione di sicurezza, per quanto riguarda la politica di sicurezza in questione. Quindi al fornitore, per provare che il programma soddisfa la politica di sicurezza del consumatore, basta dimostrare la condizione di verifica. Per avere tale dimostrazione il fornitore fa uso di un *theorem prover*. Infine la manda al consumatore, il quale esegue un *proof checker* che controlla se la dimostrazione è valida. In tal caso il programma può essere eseguito in maniera sicura.

2.2.2 Linguaggio Assembler Tipato (TAL)

Il Linguaggio Assembler Tipato è una tecnica nella quale le informazioni di tipo, derivanti da un linguggio di programmazione fortemente tipato, sono trascinati durante il processo di compilazione, chiamato *type preserving compilation*, fino ad un linguaggio intermedio tipato indipendente dalla piattaforma e alla fine inclusi nel codice oggetto. Il risultato è un'annotazione di tipo del codice oggetto, cioè il ceritificato è rappresentato da annotazioni di tipo. Tale certificato può essere controllato semplicemente da un *typechecker* ordinario. Il typechecker fa parte del trusted computing base, mentre il compilatore no.

2.2.3 Certificati di Codice Efficienti (ECC)

L'idea che sta dietro ad ECC (*efficient code certification*)è quella di rendere la produzione e la verifica dei certificati più semplice ed efficiente possibile, sopratutto per le applicazioni non affidabili, piccole, che si eseguono una sola volta, come ad esempio le applet. Per fare questo, ECC cerca di identificare la minima informazione possibile per garantire un livello di sicurezza basilare ma non banale, includendo la sicurezza del flusso di controllo, della memoria e dello stack. Il certificato consiste quindi di annotazioni che forniscono informazioni sulla struttura e l'intenzione del codice e anche qualche informazione basilare sui tipi. Quello che rende i certificati di dimensione piccola è che ECC usa largamente le convenzioni del compilatore, eliminando certe informazioni che si dovevano includere in modo esplicito in PCC e TAL, cosa che porta alla dipendenza dalla implementazione del compilatore.
Nel seguente capitolo discuteremo del flusso di informazioni, il quale è legato alla confidenzialità, e su come controllarlo.

Capitolo 3

Flusso di Informazioni

In questo capitolo ci concentriamo sulla confidenzialità dei dati che viene cosi definita: un utente non deve poter leggere dei dati ai quali non ha accesso.
Questo principio è chiamato in letteratura anche con i termini di *information confinement*, *secrecy* oppure *privacy*. Va notato che un altro requisito importante è l'integrità che può essere considerata come il duale della confidenzialità.

La confidenzialità richiede che l'informazione non possa raggiungere destinazioni inappropriate che la potrebbero violare, mentre l'integrità richiede che l'informazione non possa provenire da sorgenti inappropriate.

La confidenzialità all'interno di un sistema può essere violata per causa di errori non intenzionali nella specifica e nell'implementazione dei programmi che possono utilizzare dati confidenziali. Inoltre, un sistema può incorporare degli host o del codice non affidabili o anche maliziosi, cosa che rende la garanzia della confidenzialità ancora più difficile. La parte più complessa nel proteggere le informazioni confidenziali è quella di prevenire la perdita

di tali informazioni durante la computazione, ovvero prevenire il cosiddetto *information leakage*. Le politiche di sicurezza per la confidenzialità devono asserire che dati segreti in ingresso non possano essere rilevati da un attaccante tramite l'osservazione dei dati in uscita. Quindi, tali politiche devono regolare il *flusso di informazioni* (*information flow*). Nel resto della nostra trattazione ci occuperemo dunque, delle *politiche di flusso di informazioni* e dei meccanismi con i quali esse vengono messe in atto: i *controlli del flusso di informazioni*. Va notato che anche l'integrità può essere garantita tramite il flusso di informazioni.

Un concetto fondamentale nello studio del flusso di informazioni è quello di *covert channel* [7].
I meccanismi utilizzati per trasferire delle informazioni all'interno di un sistema vengono chiamati *channel*. I channel che sfruttano dei meccanismi, il cui compito principale non è quello di trasferire delle informazioni, vengono chiamati *covert channel*, ovvero, i covert channel trasferiscono le informazioni in una maniera che potrebbe violare la sicurezza del sistema. I covert channel sono difficili da rilevare, perché la loro intenzione è quella di nascondere la trasmissione che si sta effettuando. Quindi, pongono una grande sfida nel prevenire la perdita di informazioni. I covert channel sono divisi in queste categorie:

Implicit flow trasmettono le informazioni tramite i costrutti di controllo di un programma.

Termination channel trasmettono le informazioni tramite la terminazione o la non terminazione di un programma.

Timing channel trasmettono le informazioni tramite il tempo nel quale si è compiuta un'azione piuttosto che dai dati associati all'azione stessa. Questa

azione può essere anche la terminazione di un programma e quindi informazioni confidenziali possono essere rilevati dal tempo totale di esecuzione di un programma.

Probabilistic channel trasmettono le informazioni cambiando di volta in volta la distribuzione della probabilità dei dati osservabili. Tali channel possono diventare pericolosi quando un attaccante esegue ripetutamente un programma e osserva le sue proprietà aleatorie.

Resource exhaustion channel trasmettono le informazioni tramite un possibile esaurimento di risorse finite e condivise come memoria e disco.

Power channel inglobano le informazioni nella energia consumata da un computer, assumendo che l'attaccante sia in grado di misurare tale consumo.

Passiamo adesso a considerare degli esempi di covert channel. Nel seguito della trattazione assumiamo, per semplicità e senza perdita di generalità, che esistono due livelli di confidenzialità: *alto* e *basso*. Useremo la metavariabile *h*, variamente indicizzata, per indicare variabili di tipo *alto*, ed *l*, anch'essa variamente indicizzata, per le variabili di tipo *basso*.
Un esempio di implicit flow è il seguente:

```
if (h)  then { l := true; }
        else { l := false;}
```

Un implicit flow è in opposizione con un *explicit flow* nel quale si ha un passaggio diretto del valore di una varibile *alta* ad una variabile *bassa*, ad esempio:

```
l:=h
```

I seguenti sono esempi di termination channel e timing channel, rispettivamente, dove per il timing channel si introduce un meccanismo di attesa (wait):

```
if (h) then { while(true); }

if (h) then { wait(1000); }
```

In seguito parleremo dei meccanismi tradizionali e daremo le ragioni perché tali meccanismi non sono in grado di garantire in pieno la confidenzialità all'interno di un sistema.

3.1 Meccanismi Tradizionali

I meccanismi tradizionali che discutiamo in questa sezione sono ormai molto diffusi e ben conosciuti come: controllo degli accessi, firewall, antivirus, crittografia e inoltre un meccanismo più complesso come il *mandatory access control*.

Controllo degli accessi. È un modo standard per proteggere dati sensibili: si richiedono dei privilegi per accedere agli oggetti (file o processi) che contengono questi dati sensibili. Il controllo degli accessi si può implementare tramite la *matrice degli accessi*, le righe rappresentano gli utenti e le colonne gli oggetti. La casella $[i, j]$ della matrice indica i permessi di accesso che l'utente i ha sull'oggetto j. Inoltre, si possono usare le *liste di controllo degli accessi*, dove ad ogni oggetto è associata una lista che contiene gli utenti e le operazioni che possono effettuare sull' oggetto, oppure si possono usare le *liste di capability*, dove ad ogni utente è associata la lista degli oggetti a cui può accedere e le operazioni che può effettuare su di esso.
Il controllo degli accessi permette di rilasciare l'informazione solo a chi ha i permessi neccessari per accedervi, ma non controlla la sua propagazione e come viene utilizzata una volta rilasciata. Il programma che ha acceduto l'informzione, può per errore o per malizia utilizzarla in maniera da violare

la confidenzialità. Concludendo, il controllo degli accessi non può sostituire il controllo del flusso di informazioni.

Firewall. Il firewall è un meccanismo, hardware oppure software, utilizzato per regolare il flusso del traffico dati attraverso reti di diversi livelli di affidabilità. Un esempio tipico è Internet (rete publica) che è una zona non affidabile e una rete interna (rete privata) che è una zona più affidabile. Il firewall viene usato per prevenire intrusioni in una rete privata, quindi crea un filtro per il traffico dati. Durante la comunicazione bidirezionale tra le due reti, si può verificare una perdita di informazione e quindi una violazione della confidenzialità, rilevarla è fuori dal compito del firewall. Inoltre, la confidenzialità può essere violata anche all'interno della rete privata, senza communicazione col mondo esterno. Il firewall non effettua controlli nella rete privata, e quindi non la può proteggere da attacchi interni.

Antivirus. La scansione effettuata dall'antivirus di un programma che utilizza dati confidenziali non garantisce che il programma controllato rispetti la confidenzialità. Inoltre, l'antivirus controlla un programma basandosi su una lista di virus conosciuta come *black list* e offre una scarsa protezione per quanto riguarda i nuovi possibili attacchi.

Crittografia. (vd. cap.2). La crittografia permette di creare un canale di informazione a cui solo il mittente ed il destinatario hanno accesso. Una volta che i dati sono arrivati a destinazione e poi decrittati, non abbiamo nessuna garanzia che la computazione effettuata da parte del destinatario rispetti la confidenzialità dei dati ricevuti.

Mandatory Access Control [8]. In questo approccio, ogni dato di un programma viene etichettato con un corrispondente *livello di sicurezza* che indica una semplice politica di confidenzialità. Il flusso di informazioni viene controllato aumentando la computazione dei dati durante l'esecuzione nor-

male di un programma con la computazione delle etichette associate ad essi in modo da controllare la diffusione dei dati in questione. Tale meccanismo introduce un overhead a tempo di esecuzione e per di più non è in grado di rilevare gli implicit flow.
Si consideri il seguente codice che indica un implicit flow, con h ed l variabili di tipo *alto* e *basso* rispettivamente:

```
l := false;
if (h) then { l := true; }
       else skip
```

L'insicurezza di questo codice deriva dall'assegnazione l := `true` che permette di rilevare il valore della variabile h. Il mandatory access control, oltre alle etichette sopra menzionate, introduce un'etichetta associata ad ogni processo che tiene traccia della sensibilità dei dati che controllano tale processo, in questo caso il dato è la variabile h. Con l'uso di questa etichetta il mandatory access control può catturare l'assegnazione precedente a tempo di esecuzione: un processo *alto* aggiorna una variabile *bassa* e quindi questa esecuzione non è sicura. Se invece dell'asseganzione, si esegue l'altro ramo dell'`if` ovvero `skip`, non si hanno assegnamenti insicuri e non si effetuano controlli a tempo di esecuzione. Anche in questo caso però, il valore della variabile h si può scoprire notando che l = `false`. In questo caso il mandatory access control fallisce.
Un modo per risolvere il problema degli implicit flow è quello di imporre una restrizione che non permetta l'utlizzo delle variabili basse a partire dal momento in cui venga acceduta una variabile *alta*. Questo, peró, risulta essere troppo restrittivo e quindi il mandatory access control difficilmente può essere utilizzato per scrivere applicazioni generali utili.

Nella seguente sezione introduciamo un approccio diverso per controllare il flusso di informazioni, quello basato sui linguaggi, in particolare sul sistema dei tipi di un linguaggio.

3.2 Controllo Statico del Flusso di Informazioni

Il problema con il mandatory access control, discusso nella sezione precedente, è quello di controllare il flusso di informazioni a tempo di esecuzione per quella particolare esecuzione del programma in questione. La confidenzialità non è una proprietà di una particolare esecuzione, ma piuttosto una proprietà dell' insieme di tutte le possibili esecuzioni di un programma. Controllare una ad una tutte le esecuzioni di questo insieme è intrattabile. Quello che si può fare invece, è sviluppare un sistema che permetta di rilevare un programma nel quale viene attivato un flusso di informazioni da una variabile *alta* ad una variabile *bassa*.
Questa è l'idea che sta dietro al *controllo statico del flusso di informazioni*.

L'analisi statica di un programma può essere usata per controllare il flusso di informazioni con più precisione e senza overhead a tempo di esecuzione. In particolare, il controllo del flusso di informazioni può essere effettuato sfruttando una *teoria dei tipi*. Questo approccio è stato implementato nel compilatore Jif [6, 9] e consiste nell'assegnare ad ogni espressione di un programma un *tipo di sicurezza* composto da un tipo normale, ad esempio `int`, ed un'etichetta che indica come si deve utilizzare il valore dell'espressione, ovvero indica una politica di confidenzialità sui dati etichettati. Queste etichette sono determinate in modo completamente statico e non vengono computa-

te a tempo di esecuzione come le etichette utilizzate dal mandatory access control. La sicurezza viene rafforzata dal controllo sui tipi: il compilatore prende in input un programma che contiene tipi etichettati e nell'effettuare il controllo sui tipi si assicura che non ci saranno a tempo di esecuzione flussi di informazione che violano la confidenzialità. Il sistema dei tipi di un linguaggio che usa questa tecnica viene chiamato *sistema dei tipi di sicurezza*.

Un vantaggio di questo approccio è che si possono rilevare gli implicit flow. Per fare questo si introduce un'etichetta *pc* (*program counter*) che si associa ad ogni istruzione ed espressione e che assumiamo possa assumere i valori *alto* e *basso*. L'etichetta pc rappresenta le informazioni che si possono imparare dall'esecuzione dell'istruzione o dalla valutazione dell'espressione. Consideriamo di nuovo il codice:

```
l := false;                          {basso}
if (h) then { l := true; }           {alto}
       else skip                     {alto}
       ...                           {basso}
```

Il valore dell'etichetta *pc* nei rami dell'`if` è *alto*, indicando quello che si può imparare dall'esecuzione di questa istruzione, ovvero la confidenzialità di *h*. Un'assegnazione ad una variabile viene detta sicura se il valore dell'etichetta della variabile in questione è restrittivo almeno quanto il *pc*. Nel nostro esempio la variabile *l* ha etichetta *bassa*, non restrittiva quanto il *pc*, quindi l'assegnazione all'interno dell'`if` non è sicura. Questo porta a reputare il programma insicuro.

Una caratteristica dell'etichetta *pc* è che il suo valore associato all'istruzione che segue l'`if` è uguale al valore che ha l'etichetta *pc* dell'istruzione che precede l'`if`, se tali istruzioni esistono.

Intuitivamente, questo ci dice che non impariamo niente dall'esecuzione dell'`if` e che l'istruzione successiva si esegue a prescindere dal valore della variabile h.

3.2.1 Una nozione Semantica di Sicurezza

Introduciamo adesso il concetto di *non interferenza*. Se un utente vuole mantenere dei dati confidenziali, deve poter stabilire una politica che permetta ai programmi di utilizzare o modificare questi dati, fintantochè i dati visibili in uscita non rivelano i dati confidenziali. Una politica di sicurezza di questo genere si chiama *politica di non interferenza*, perché afferma che i dati confidenziali non interferiscono con i dati pubblici.
Il sistema dei tipi di sicurezza che andremo a vedere in seguito rafforza la non interferenza. Prima, peró definiamo formalmente la non interferenza utilizzando il concetto di semantica di un programma. Prendiamo in considerazione un linguaggio imperativo che contiene: `skip`, assegnazione, composizione sequenziale, costrutto condizionale `if` e cicli `while`. La sintassi di questo *while language* è la seguente:

$$C ::= \texttt{skip} \mid var := exp \mid C_1 \; ; \; C_2$$
$$\mid \texttt{if } exp \texttt{ then } C_1 \texttt{ else } C_2 \mid \texttt{while } exp \texttt{ do } C$$

Con *var* abbiamo indicato l'insieme delle variabili di tipo *alto* e di tipo *basso*. Con *exp* abbiamo indicato le espressioni che si formano applicando operatori aritmetici a variabili e costanti.
Passiamo adesso a definire la non interferenza.

La non interferenza significa che *una variazione dei dati confidenziali (alti) in ingresso non causa una variazione dei dati pubblici (bassi) in uscita* [8].

Denotiamo con $S : Var \to Val$ l'insieme degli stati che mappa variabili in valori. Con $LVar \subseteq Var$ indichiamo l'insieme delle variabili di tipo *basso* mentre tutte le altre vengono considerate di tipo *alto*. Supponiamo che la computazione inizi in uno stato s, il programma termina in uno stato s' oppure non termina.
La semantica $[\![C]\!]$ di un programma C è una funzione $[\![C]\!] : S \to S_\perp$ dove $S_\perp = S \cup \{\perp\}$ e $\perp \notin S$ che mappa uno stato di ingresso $s \in S$ in uno stato di uscita $[\![C]\!]s \in S$, oppure in $\perp$ se il programma non termina.
La variazione dell'input *alto* si può descrivere come una relazione di equivalenza $=_L \subseteq S \times S$ chiamata *equivalenza bassa*; due input sono equivalenti se e solo se concordano sui valori *bassi*: $s =_L s'$ se e solo se $s_l = s'_l$.
L'osservazione da parte di un attaccante può essere descritta tramite la relazione $\approx_L \subseteq S_\perp \times S_\perp$ sui comportameti (definiti dalla semantica) del programma tale che: due comportamenti sono relazionati dall' $\approx_L$ se e solo se sono indistinguibili da parte dell'attaccante. È neccessario che questa relazione $\approx_L$, sia almeno riflessiva e simmetrica, ma normalmente è una relazione di equivalenza.
Per un dato modello di semantica, la non interferenza viene formalizzata come segue:

C è sicuro se e solo se
$\forall\, s_1,\, s_2 \in S, \quad s_1 =_L s_2 \implies [\![C]\!](s_1) \approx_L [\![C]\!](s_2)$

che può essere letto: "se due stati in input condividono gli stessi valori *bassi*, allora i comportamenti del programma eseguito su questi stati sono indistinguibili per l'attaccante".
Il concetto di indistinguibilità lo possiamo interpretare diversamente a seconda della politica di sicurezza che vogliamo rafforzare. Possiamo stabilire,

ad esempio per il nostro linguaggio: $s \approx_L s'$ se e solo se $s, s' \in S$ implica $s =_L s'$.

Passiamo adesso ad alcuni esempi: il programma in seguito è sicuro

```
if (l = 5) then { h := h + 1; }
           else { l := l + 1;}
```

perché i valori finali di l dipendono solo dai suoi valori iniziali.

Invece, i seguenti non sono dei programmi sicuri, il primo indica un explicit flow e il secondo un implicit flow.

```
l := h
```

```
if (h = 3) then { l := 5; }
           else skip
```

Ad esempio, prendiamo come valori iniziali della variabile h gli interi 2 e 3. Per il primo programma, con l che assume il valore 0, avremo:
$(h \mapsto 2, l \mapsto 0) =_L (h \mapsto 3, l \mapsto 0)$ ma $[\![l := h]\!](h \mapsto 2, l \mapsto 0) = (h \mapsto 2, l \mapsto 2) \not\approx_L (h \mapsto 3, l \mapsto 3) = [\![l := h]\!](h \mapsto 3, l \mapsto 0)$
Lo stesso ragionamento si fa anche per il secondo programma.

3.2.2 Il Sistema dei Tipi di Sicurezza

Il *sistema dei tipi di sicurezza* [8] che vediamo in questa sezione è un insieme di *regole di inferenza* che consentono di associare un *tipo di sicurezza* ad un programma oppure ad una espressione. In questo sistema ci sono due tipi di regole: $\vdash exp : \tau$ per indicare che l'espressione *exp* ha tipo τ e $[pc] \vdash C$ per indicare che il comando C è *tipabile* nel *contesto di sicurezza pc*. Per semplicità, il contesto di sicurezza è l'etichetta program counter *pc* precedentemente introdotta. Il contesto pc rappresenta il livello di confi-

denzialità, *alto* o *basso*, ovvero, indica quanto è restrittivo l'ambiente in cui stiamo tipando un comando C. Introduciamo adesso le regole di inferenza per il while language:

$$\frac{}{\vdash exp : alto} \tag{3.1}$$

$$\frac{h \notin Vars(exp)}{\vdash exp : basso} \tag{3.2}$$

$$\frac{}{[pc] \vdash skip} \tag{3.3}$$

$$\frac{}{[pc] \vdash h := exp} \tag{3.4}$$

$$\frac{\vdash exp : basso}{[basso] \vdash l := exp} \tag{3.5}$$

$$\frac{[pc] \vdash C_1 \qquad [pc] \vdash C_2}{[pc] \vdash C_1; C_2} \tag{3.6}$$

$$\frac{\vdash exp : pc \qquad [pc] \vdash C}{[pc] \vdash \texttt{while}\ exp\ \texttt{do}\ C} \tag{3.7}$$

$$\frac{\vdash exp : pc \qquad [pc] \vdash C_1 \qquad [pc] \vdash C_2}{[pc] \vdash \texttt{if}\ exp\ \texttt{then}\ C_1\ \texttt{else}\ C_2} \tag{3.8}$$

$$\frac{[alto] \vdash C}{[basso] \vdash C} \tag{3.9}$$

Secondo le regole (3.1) e (3.2) qualsiasi espressione (includendo le costanti, la variabile h e anche l) può avere un tipo *alto*, ma può avere un tipo *basso* solo se non contiene occorrenze della variabile h. Le regole (3.3) e (3.4) indicano

che i comandi `skip` e $h := exp$ sono tipabili in qualsiasi contesto. Il comando $l := exp$, nella regola (3.5), è tipabile solo se *exp* è di tipo *basso*, questo per prevenire gli explicit flow. Inoltre, questo comando è tipabile solo in un contesto di sicurezza *basso*, cosa che previene gli implicit flow. La regola (3.6) afferma che dati due programmi C_1 e C_2 tipabili in un contesto *pc*, anche la loro composizione è tipabile nello stesso contesto. Il sistema dei tipi di sicurezza risulta essere composizionale: programmi sicuri si combinano formando programmi sicuri complessi. Le regole (3.7) e (3.8) richiedono che, dato un `if` (o un `while`) con la condizione *alta*, allora i rami dell'`if` (corpo del `while`) devono essere tipati in un contesto di sicurezza *alto*. La regola (3.8) intercetta una possibile perdita di informazioni confidenziali del tipo mostrato nell'esempio della sottosezione 3.2.1. L'ultima regola garantisce che se un programma è tipabile in un contesto *alto* allora, è tipabile anche in un contesto *basso*. La premessa di questa regola assicura che non occorrono assegnazioni a variabili *basse* nel programma C. Questa regola permette di resettare il program counter a *basso* dopo che si è eseguito un `if` o un `while` con condizione *alta*.

Il sistema dei tipi appena discusso rafforza la non interferenza utilizzando come modello di semantica il seguente: $s \approx_L s'$ se e solo se $s, s' \in S$ implica $s =_L s'$. Si noti come con questa definizione della relazione $\approx_L$ si possono rilevare solo gli implicit flow, non altri covert channel. Infatti, secondo questa affermazione si ha che s $\approx_L \perp$, $\perp \approx_L$ s, $\perp \approx_L \perp$. Questo ci impedisce di rilevare i termination channel in quanto è indistinguibile un programma che termina da uno che non termina. Per rilevare anche i termination channel si deve modificare la definizione di programmi indistinguibili e di conseguenza il sistema dei tipi.

Per la non interferenza *sensibile alla terminazione* il modello utilizzato è il seguente: $s \approx_L s'$ se e solo se $s, s' \in S$ implica $s =_L s'$ oppure $s = s' = \bot$. Per non avere attacchi che sfruttano la non terminazione, il sistema dei tipi non permette che il tipo della condizione del costrutto `while` sia *alto*, inoltre gli `if` con condizione *alta* non devono contenere cicli nei loro rami.

Per la non interferenza *sensibile al tempo*, quindi per rilevare anche i timing channel, il modello è il seguente: $s \approx_L s'$ se e solo se $s = s' = \bot$ oppure $s, s' \in S$ implica $s =_L s'$ e il programma eseguito su s, s' termina nello stesso numero di passi, in questo caso $\approx_L \subseteq (S \times N)_\bot$. Per evitare la perdita di informazioni attraverso i timing channel, il sistema dei tipi non permette che il costrutto `if` con condizione *alta* abbia nei sui rami dei cicli, inoltre questi `if` vengono rachiusi in *statement* che si eseguono in modo atomico.

Capitolo 4

Un nuovo Sistema dei Tipi di Sicurezza

Nel capitolo precedente abbiamo parlato della confidenzialità, che risulta essere uno dei requisiti più importanti per quanto riguarda la sicurezza della rete e dei computer e in particolare del controllo del flusso di informazioni come un modo per garantire la confidenzialità. Abbiamo introdotto un sistema dei tipi di sicurezza, tratto da [8], che è equivalente al sistema dei tipi di sicurezza in [10], per il quale gli autori hanno dimostrato che è *corretto* (*sound*): se un programma C è tipabile in un contesto di sicurezza pc, allora tale programma è sicuro secondo il concetto di non interferenza, ovvero

$$[pc] \vdash C \implies C \text{ è non interferente.}$$

Ricordiamo dal capitolo precendente che un programma C è detto non interferente se e solo se $\forall\ s_1,\ s_2\ \in S, \quad s_1\ =_L\ s_2\ \implies\ [\![C]\!](s_1)\ \approx_L\ [\![C]\!](s_2)$ e il concetto di indistinguibilità, ovvero la relazione $\approx_L$ viene formalizzata come segue: $s \approx_L s'$ se e solo se $s, s' \in S$ implica $s =_L s'$.

Il controllo statico che si effettua sui tipi di sicurezza, risulta essere più preciso e senza overhead a tempo di esecuzione rispetto al controllo dinamico per il flusso di informazioni, ma comunque risulta essere restrittivo.

Infatti, tale sistema dei tipi non è *completo* (*complete*), perché esistono dei programmi sicuri (sempre secondo il concetto di non interferenza) che non sono tipabili nel contesto di sicurezza *pc* secondo le regole di inferenza di questo sistema.

$$C \text{ è non interferente} \quad \not\Longrightarrow [pc] \vdash C$$

Nel capitolo precedente abbiamo visto degli esempi di programmi non sicuri che violano la confidenzialità, in particolare con un *explicit flow* e un *implicit flow* come i seguenti:

```
l := h
```

```
if (h = 3) then { l := 0; }
           else { l := 1; }
```

Questi programmi, ovviamente non tipano nel sistema dei tipi di sicurezza. I programmi:

```
l := h; l := 0;
```

```
if (h = 3) then { l := 0; } else { l := 1; }; l := 0;
```

sono sicuri ma comunque non tipano perché il controllo dei tipi di sicurezza, come il controllo ordinario dei tipi (*type-checking*), è composizionale, ovvero programmi sicuri si combinano formando programmi sicuri complessi.

In seguito vedremo un nuovo sistema dei tipi di sicurezza il quale estende quello presentato nel capitolo precedente, in modo che programmi sicuri come quelli mostrati sopra vengono tipati.

4.1 Regole di Inferenza

Come visto nel capitolo precendente denotiamo con $S : Var \to Val$ l'insieme degli stati che mappa variabili in valori. Con $LVar \subseteq Var$ indichiamo l'insieme delle variabili di tipo *basso* mentre tutte le altre vengono considerate di tipo *alto*. Useremo l e h variamente indicizzate per indicare variabili *basse* e variabili *alte*. La semantica di un programma C è una funzione $[\![C]\!] : S \to S_\perp$ dove $S_\perp = S \cup \perp$ e $\perp \notin S$ che mappa uno stato di ingresso $s \in S$ in uno stato di uscita $[\![C]\!]s \in S$, oppure in $\perp$ se il programma non termina. Diamo le seguenti definizioni:

Def (equivalenza-bassa a meno di X): $\forall X \subseteq LVar$, la relazione di equivalenza bassa a meno di X $\equiv_X \subseteq S \times S$ è definita come segue:

$\forall s_1, s_2 \in S,\ (s_1 \equiv_X s_2 \quad \textit{se e solo se}, \quad \forall l \in LVars - X,\ s_1(l) = s_2(l))$

Notazione : la relazione $\equiv_\emptyset$ è equivalente alla relazione $=_L$ (vd. cap. 3)

Def (non interferenza a meno di X): C è *non interferente a meno di* X se e solo se, $\forall s_1, s_2 \in S, \quad s_1 \equiv_X s_2 \implies [\![C]\!](s_1) \equiv_X [\![C]\!](s_2)$.

Il nuovo sistema dei tipi di sicurezza che vedremo in seguito è un insieme di regole di inferenza del tipo $[pc] \vdash_X C$, che si legge: il comando C è tipabile nel contesto di sicurezza pc a *meno dell'insieme* X, con $X \subseteq LVar$.

$$\frac{}{[pc] \vdash_\emptyset skip} \tag{4.1}$$

$$\frac{}{[pc] \vdash_\emptyset h := exp} \tag{4.2}$$

$$\frac{}{[basso] \vdash_X l := exp}\ (h \notin exp\ ,\ LVar(exp) \cap X = \emptyset) \tag{4.3}$$

$$\frac{}{[pc] \vdash_{X \cup l} l := exp} \tag{4.4}$$

$$\frac{[pc] \vdash_X C_1 \qquad [pc] \vdash_X C_2}{[pc] \vdash_X C_1 \,;\, C_2} \ (C_2 \neq l := exp) \tag{4.5}$$

$$\frac{[basso] \vdash_X C}{[basso] \vdash_{X-l} C \,;\, l := exp} \ (h \notin exp \,,\ LVar(exp) \cap X = \emptyset) \tag{4.6}$$

$$\frac{[basso] \vdash_X C}{[basso] \vdash_X \texttt{while}\ exp\ \texttt{do}\ C} \ (h \notin exp \,,\ LVar(exp) \cap X = \emptyset) \tag{4.7}$$

$$\frac{[alto] \vdash_X C}{[alto] \vdash_X \texttt{while}\ exp\ \texttt{do}\ C} \ (LVar(exp) \cap X = \emptyset) \tag{4.8}$$

$$\frac{[basso] \vdash_X C_1 \qquad [basso] \vdash_X C_2}{[basso] \vdash_X \texttt{if}\ exp\ \texttt{then}\ C_1\ \texttt{else}\ C_2} \ (h \notin exp \,,\ LVar(exp) \cap X) = \emptyset) \tag{4.9}$$

$$\frac{[alto] \vdash_X C_1 \qquad [alto] \vdash_X C_2}{[alto] \vdash_X \texttt{if}\ exp\ \texttt{then}\ C_1\ \texttt{else}\ C_2} \ (LVar(exp) \cap (X) = \emptyset) \tag{4.10}$$

$$\frac{[alto] \vdash_X C}{[basso] \vdash_X C} \tag{4.11}$$

Le regole (4.1) e (4.2) indicano che i programmi *skip* e $h := exp$ sono tipabili in qualsiasi contesto di sicurezza *pc* a meno dell'insieme $\emptyset$. Le regole (4.3) e (4.4) controllano le assegnazioni alle variabili *basse*. La regola (4.3)

indica che l'assegnazione $l := exp$ è tipabile nel contesto *basso* a meno di X con la condizione che *exp* non contiene occorrenze di variabili *alte*, ed inoltre l'intersezione dell'insieme $LVar(exp)$ con l'insieme X è $\emptyset$. La regola (4.4) afferma che l'assegnazione $l := exp$ è tipabile in qualsiasi contesto *pc* a meno dell'insieme $X \cup l$. La regola (4.5) indica che dati due programmi C_1 e C_2 con C_2 diverso dall'assegnazione ad l, tipabili in un contesto *pc* a meno di X, la loro composizione $C_1 ; C_2$ è tipabile nello stesso contesto a meno di X. La regola (4.6) tratta la composizione con un'assegnazione $l := exp$. Dato un programma C tipabile a meno di X in un contesto *basso* la composizione con $l := exp$ è tipabile a meno dell'insieme $X - l$, rispettando la condizione laterale $h \notin exp, LVars(exp) \cap X = \emptyset$. Le regole (4.7) e (4.8) controllano il costrutto `while` nel contesto di sicurezza *basso* e *alto*, rispettivamente. La prima indica che dato un programma C tipabile a meno di X, allora anche il programma `while` *exp* `do` C è tipabile a meno di X, con la condizione che $h \notin exp$ ed inoltre $LVars(exp) \cap X = \emptyset$. L'altra regola è analoga, senza assunzioni sulle occorrenze di h in *exp*, ma rimane comunque la condizione laterale $LVars(exp) \cap X = \emptyset$. La regola (4.9) controlla il costrutto `if-then-else` nell'ambiente *basso*: dati due programmi C_1 e C_2 tipabili in *basso* a meno di X, il programma `if` *exp* `then` C_1 `else` C_2 è tipabile in *basso* a meno di X con la condizione che *exp* non contiene occorrenze di variabili *alte*. La regola (4.10) è analoga a quella precedente per il programma `if-then-else` tipabile nell'ambiente *alto*, senza restrizioni sul contenuto di *exp*. Tutte e due le regole del costrutto `if-then-else` hanno come condizione laterale $LVars(exp) \cap X) = \emptyset$. Per chiarimenti sulla necessità di avere la condizione laterale $LVars(exp) \cap X = \emptyset$ nell'assegnazione ad l in un contesto *basso*, nella regola della composizione con $l := exp$, nel costrutto `while` e nel costrutto `if-then-else` si riamanda alla sezione seguente. La regola (4.11) estende

la regola (3.9) del capitolo precedente. Questa regola afferma che dato un programma C tipabile in un contesto di sicurezza *alto* a meno dell'insieme X, tale programma è tipabile anche nel contesto *basso* a meno di X.
Vogliamo dimostrare adesso che ogni programma C tipabile nel sistema dei tipi di sicurezza di [8] è tipabile anche nel nostro sistema dei tipi. Ovvero, vogliamo dimostrare il seguente:

Teorema 4.1.1. $[pc] \vdash C \implies [pc] \vdash_{\emptyset} C$

Dimostrazione. Osserviamo in primo luogo come per ogni assioma nel sistema dei tipi di [8], esiste un'assioma equivalente nel nostro sistema dei tipi:

$$[pc] \vdash skip \qquad \Longrightarrow \qquad [pc] \vdash_{\emptyset} skip$$

$$[pc] \vdash h := exp \qquad \Longrightarrow \qquad [pc] \vdash_{\emptyset} h := exp$$

inoltre la regola

$$\frac{\vdash exp : basso}{[basso] \vdash l := exp}$$

è equivalente alla regola (4.3) nel nostro sistema dei tipi:

$$\frac{}{[basso] \vdash_X l := exp} \ (h \notin exp \ , \ LVar(exp) \cap X = \emptyset)$$

considerando $X = \emptyset$.
Passiamo adesso a considerare le regole della composizione, del costrutto `while` e del costrutto `if`. Per queste regole esistono delle regole equivalenti nel nostro sistema dei tipi. In particolare, per la regola della composizione abbiamo due regole, a seconda che C_2 sia uguale o diverso da $l := exp$, e basta avere $X = \emptyset$ per ottenere delle regole equivalenti alla regola della composizione in [8]. Per i costrutti `while` e `if` abbiamo due regole nel nostro

sistema dei tipi a seconda che i programmi `while` e `if` vengono tipati in un ambiente *basso* o *alto*. Queste regole risultano equivalenti a quelle in [8] se consideriamo $X = \emptyset$. Infine, la regola (3.9) ha un equivalente nel nostro sistema dei tipi: la regola (4.11) nel caso $X = \emptyset$. Da quanto detto, possiamo dedurre che per ogni programma che ha un albero di inferenza nel sistema dei tipi di [8], possiamo costruire un albero di inferenza utilizzando le regole del nostro sistema dei tipi, ovvero se un programma C è tipabile nel sistema dei tipi di [8], allora è tipabile anche nel nostro sistema dei tipi a meno dell'insieme $\emptyset$. □

4.2 La correttezza del Sistema dei Tipi di Sicurezza

In questa sezione vogliamo dimostrare la correttezza (*soundness*) del sistema dei tipi che abbiamo introdotto. Per la dimostrazione faremo uso delle seguenti proposizioni:

Proposizione 4.2.1. $s_1 \equiv_X s_2 \implies s_1 \equiv_{X \cup Y} s_2$

Proposizione 4.2.2. $[alto] \vdash_X C, \implies \forall s \in S, \quad s \equiv_X [\![\text{C}]\!](\text{s})$

Lemma 4.2.3. $[pc] \vdash_X C,$ per pc qualunque $\implies$ C è non interferente a meno di X.

Dimostrazione. Dimostriamo il lemma per gli assiomi del sistema dei tipi di sicurezza:
l'assioma (4.1):

$$\frac{}{[pc] \vdash_\emptyset skip}$$

è banalmente vero per la semantica di *skip*: $\forall s_1, s_2 \in S, \quad s_1 \equiv_\emptyset s_2 \implies [\![skip]\!](s_1) = s_1 \equiv_\emptyset s_2 = [\![skip]\!](s_2)$.
l'assioma (4.2):

$$\frac{}{[pc] \vdash_\emptyset h := exp}$$

$\forall s_1, s_2 \in S, \quad s_1 \equiv_\emptyset s_2 \implies s_1[h \mapsto exp] \equiv_\emptyset s_2[h \mapsto exp] \implies [\![h := exp]\!](s_1) = s_1[h \mapsto exp] \equiv_\emptyset s_2[h \mapsto exp] = [\![h := exp]\!](s_2)$.
l'assioma (4.3):

$$\frac{}{[basso] \vdash_X l := exp} \ (h \notin exp \ , \ LVar(exp) \cap X = \emptyset)$$

$\forall s_1, s_2 \in S, \quad s_1 \equiv_X s_2 \implies s_1[l \mapsto exp] \equiv_X s_2[l \mapsto exp] \implies [\![l := exp]\!](s_1) = s_1[l \mapsto exp] \equiv_X s_2[l \mapsto exp] = [\![l := exp]\!](s_2)$, dove *exp* non contiene occorrenze di variabili *alte* e vale $LVar(exp) \cap X = \emptyset$.
l'assioma (4.4):

$$\frac{}{[pc] \vdash_{X \cup l} l := exp}$$

$\forall s_1, s_2 \in S, \quad s_1 \equiv_{(X \cup l)} s_2 \implies [\![l := exp]\!](s_1) \equiv_{(X \cup l)} [\![l := exp]\!](s_2)$, che è ovviamente vera.
Per dimostrare la correttezza delle altre regole del nostro sistema dei tipi di sicurezza, supponiamo per ipotesi che le premesse di queste regole sono non interferenti a meno di X, e dimostriamo che lo è anche il risultato dell'applicazione della regola in questione.
Cominciamo con la regola (4.5) della composizione:

$$\frac{[pc] \vdash_X C_1 \qquad [pc] \vdash_X C_2}{[pc] \vdash_X C_1 \, ; \, C_2} \ (C_2 \neq l := exp)$$

supponiamo per ipotesi che C_1 e C_2 sono non interferenti a meno di X, ovvero:

$\forall s_1, s_2 \in S, \quad s_1 \equiv_X s_2 \implies [\![C_1]\!](s_1) \equiv_X [\![C_1]\!](s_2)$ per il programma C_1,

$\forall s_1, s_2 \in S, \quad s_1 \equiv_X s_2 \implies [\![C_2]\!](s_1) \equiv_X [\![C_2]\!](s_2))$ per il programma C_2,

dobbiamo dimostrare che: $\forall s_1, s_2 \in S, \quad s_1 \equiv_X s_2 \implies [\![C_1 ; C_2]\!](s_1) = [\![C_2]\!]([\![C_1]\!](s_1)) \equiv_X [\![C_2]\!]([\![C_1]\!](s_2) = [\![C_1 ; C_2]\!](s_2)$. Infatti:

$s_1 \equiv_X s_2 \implies s = [\![C_1]\!](s_1) \equiv_X [\![C_1]\!](s_2) = s'$ per ipotesi di C_1

$s \equiv_X s' \implies [\![C_2]\!](s) \equiv_X [\![C_2]\!](s')$ per ipotesi di C_2

per la regola (4.6) della composizione con $l := exp$:

$$\frac{[basso] \vdash_X C}{[basso] \vdash_{X-l} C ; l := exp} \ (h \notin exp \ , \ LVar(exp) \cap X = \emptyset)$$

supponiamo per ipotesi che C è non interferente a meno di X e utilizzando la regola (4.3), $l := exp$ è non interferente a meno di X; dobbiamo dimostrare che la composizone è non interferente a meno di $X - l$. La dimostrazione procede per casi:

se $l \notin X$ allora la regola (4.6) è la regola (4.5).

altrimenti, se $l \in X$ dobbiamo dimostrare che $\forall s_1, s_2 \in S, \quad s_1 \equiv_{X-l} s_2 \implies [\![C ; l := exp]\!](s_1) \equiv_{X-l} [\![C ; l := exp]\!](s_2)$

$\forall s_1, s_2 \in S, \quad s_1 \equiv_{X-l} s_2 \implies s_1 \equiv_X s_2 \implies [\![C]\!](s_1) \equiv_X [\![C]\!](s_2) \implies [\![l := exp]\!]([\![C]\!](s_1)) \equiv_X [\![l := exp]\!]([\![C]\!](s_2))$, la prima implicazione deriva dalla propsizione 4.2.1, mentre la seconda e la terza per ipotesi su C_1 e C_2; vediamo adesso cosa succede in l:

$([\![C ; l := exp]\!](s_1))(l) = ([\![l := exp]\!]([\![C]\!](s_1)))(l) = ([\![C]\!](s_1)[l \mapsto exp])(l) = exp = ([\![C]\!](s_2)[l \mapsto exp])(l) = ([\![l := exp]\!]([\![C]\!](s_2)))(l) = ([\![C ; l := exp]\!](s_2))(l)$

quindi vale $[\![C ; l := exp]\!](s_1) \equiv_{X-l} [\![C ; l := exp]\!](s_2)$

Passiamo adesso a considerare il costrutto `while` tipabile in *basso* e in *alto*. regola (4.7):

$$\frac{[basso] \vdash_X C}{[basso] \vdash_X \texttt{while}\ exp\ \texttt{do}\ C}\ (h \notin exp\ ,\ LVar(exp) \cap X = \emptyset)$$

per ipotesi il programma C è non interferente a meno di X, dobbiamo dimostrare che il programma $while\ exp\ do\ C$ è non interferente a meno di X, ovvero $\forall s_1, s_2 \in S, \quad s_1 \equiv_X s_2 \implies [\![while\ exp\ do\ C]\!](s_1) \equiv_{(X)} [\![while\ exp\ do\ C]\!](s_2)$

Possiamo fare la seguente interpretazione:

$[\![while\ exp\ do\ C]\!](s) =$

$\perp$ oppure s per i quali la regola è banalmente vera,

$[\![C]\!](s)$ oppure

$[\![C\,;\,C]\!](s)$ oppure

$\vdots$

$[\![C\,;\cdots;\,C]\!](s)\cdots$

Partendo dalle regole per la composizione che abbiamo dimostrato precedentemente, possiamo concludere che la regola (4.7) del costrutto `while` è corretta.

Per la regola (4.8) del costrutto `while` tipabile in un ambiente *alto* a meno di X, la dimostrazione è del tutto analoga.

Passiamo alla regola (4.9) del `if` :

$$\frac{[basso] \vdash_X C_1 \qquad [basso] \vdash_X C_2}{[basso] \vdash_X \texttt{if}\ exp\ \texttt{then}\ C_1\ \texttt{else}\ C_2}\ (h \notin exp\ ,\ LVar(exp) \cap X = \emptyset)$$

Supponiamo per ipotesi che i programmi C_1 e C_2 sono non interferenti a meno di X. Dobbiamo dimostrare il seguente: $\forall s_1, s_2 \in S, \quad s_1 \equiv_X s_2 \implies [\![if\ exp\ then\ C_1\ else\ C_2]\!](s_1) \equiv_X [\![if\ exp\ then\ C_1\ else\ C_2]\!](s_2)$

Procediamo per casi:

se la valutazione di *exp* risulta `true`, allora si esegue il ramo del `then`, ovvero C_1 che per ipotesi è non interferente a meno di X.

se la valutazione di *exp* risulta `false`, allora si esegue il ramo dell'`else`, ovvero C_2 che per ipotesi è non interferente a meno di X.

Quanto detto implica che il programma $if\ exp\ then\ C_1\ else\ C_2$ è non interferente a meno di X.

Per dimostrare la correttezza della regola (4.10)

$$\frac{[alto] \vdash_X C_1 \qquad [alto] \vdash_X C_2}{[alto] \vdash_X \texttt{if}\ exp\ \texttt{then}\ C_1\ \texttt{else}\ C_2}\ (LVar(exp) \cap X = \emptyset)$$

usiamo la seguente proposizione:

Per ipotesi i programmi C_1 e C_2 sono non interferenti a meno di X. Per dimostrare che il programma $if\ exp\ then\ C_1\ else\ C_2$ è non interferente a meno di X notiamo in primo luogo che *exp* può contenere occorrenze di variabili *alte* quindi partendo da due stati s_1, s_2 tale che $s_1 \equiv_X s_2$ ed eseguendo il programma `if-then-else` si possono eseguire rami diversi a seconda che il programma viene applicato allo stato s_1 o s_2. Basandoci sulla proposizione 4.2.2 l'esecuzione di uno o dell'altro ramo dell'`if` risulta indiferente. Questo implica che il programma $if\ exp\ then\ C_1\ else\ C_2$ è non interferente a meno di X.

L'ultima regola

$$\frac{[alto] \vdash_X C}{[basso] \vdash_X C}$$

risulta banalmente vera in quanto per ipotesi il programma C è non interferente a meno di X e dobbiamo dimostrare che il programma C è non interferente a meno di X. Questo deriva dal fatto che non facciamo assunzioni sul valore del contesto di sicurezza *pc*. □

Corollario 4.2.4. $[pc] \vdash_{\emptyset} C$, per pc qualunque $\implies$ C è non interferente.

Dimostrazione. ovvia conseguenza del lemma e della notazione. □

Utilizzando la definizione di equivalenza-bassa a meno di X e il lemma che abbiamo appena dimostrato, possiamo mostrare la necessità di avere nella regola di assegnazione ad l in un contesto *basso*, e di conseguenza nella regola di composizione con il programma $l := exp$, nelle regole del costrutto `while` e del costrutto `if`, la condizione laterale $LVars(exp) \cap X = \emptyset$. Supponiamo che queste regole non abbiano tali condizioni laterali.
Dati gli stati $s_1 = (l \mapsto 0\,, l_1 \mapsto 7)$ e $s_2 = (l \mapsto 0\,, l_1 \mapsto 8)$ per i quali vale $s_1 \equiv_{l_1} s_2$, per il lemma sopra dimostrato deve valere $s_1 \equiv_{l_1} s_2 \implies [\![C]\!](s_1) \equiv_{l_1} [\![C]\!](s_2)$ per un dato programma C.
Consideriamo la regola dell'assegnazione (4.3), sia $C = (l := l_1)$ dove $LVars(exp) \cap X \neq \emptyset$. Osserviamo che $s_1 \equiv_{l_1} s_2 \not\implies [\![l := l_1]\!](s_1) \equiv_{l_1} [\![l := l_1]\!](s_2)$ perché vale $[\![l := l_1]\!](s_1) \equiv_{(l_1\,, l)} [\![l := l_1]\!](s_2)$ in quanto le variazioni di l_1 portano a variare anche l.
Consideriamo il costrutto `while` e prendiamo $C = (\ \texttt{while}\ l_1 = 7\ \texttt{do}\ \ l_1 = l_1 + 1\,;\, l = l + 1)$. Di nuovo partendo da $s_1 \equiv_{l_1} s_2$ arriviamo a $[\![C]\!](s_1) \not\equiv_{l_1} [\![C]\!](s_2)$.
Infine, per il costrutto `if-then-else` possiamo fare lo stesso ragionamento: $C = (\ \texttt{if}\ l_1 = 7\ \texttt{then}\ \ l = l + 1\ \texttt{else}\ l = l - 1)$ avremmo che $s_1 \equiv_{l_1} s_2 \implies [\![if\ \ l_1 = 7\ \ then\ \ l = l + 1\ \ else\ \ l = l - 1]\!](s_1) \equiv_{(l_1\,, l)} [\![if\ \ l_1 = 7\ then\ l = l + 1\ else\ l = l - 1]\!](s_2)$.

4.3 Esempi di programmi sicuri

I programmi visti in precedenza, che venivano reputati insicuri dal sistema dei tipi di sicurezza di [8], sono tipabili nel sistema dei tipi di sicurezza che abbiamo appena discusso. Vediamo alcuni esempi di alberi di inferenza per quei programmi.

```
l := h; l := 0;
```

La prima assegnazione ad l è ovviamente non sicura perché indica un *explicit flow*, ma in composizione con la seconda assegnazione l := 0 il programma diventa sicuro. Questo è derivabile mediante la regola (4.6).

$$\frac{[basso] \vdash_l l := h}{[basso] \vdash_\emptyset l := h \ ; \ l := 0}$$

Vediamo adesso un secondo programma:

```
if (h = 3) then { l := 0; } else { l := 1; }; l := 0;
```

Il costrutto `if-then-else` indica un *implicit flow* ed è ovviamente insicuro. In composizione con l'assegnazione l := 0 il programma diventa sicuro. Ecco la derivazione:

$$\frac{\dfrac{\dfrac{[alto] \vdash_l l := 0 \qquad [alto] \vdash_l l := 1}{[alto] \vdash_l if\ (h = 3)\ then\ \{l := 0;\}\ else\ \{l := 1;\}}}{[basso] \vdash_l if\ (h = 3)\ then \{l := 0;\}\ else\ \{l := 1;\}}}{[basso] \vdash_l if\ (h = 3)\ then\ \{l := 0;\}\ else\ \{l := 1;\}\ ;\ l := 0}$$

Conclusioni

In questa relazione abbiamo discusso il problema della sicurezza dei programmi. Abbiamo inizialmente visto dei meccanismi tradizionali per garantirla e abbiamo visto le loro principali debolezze. Successivamente ci siamo concentrati su un nuovo approccio alla sicurezza, quello basato sui linguaggi di programmazione. Ci siamo concentrati, in particolare, sulla confidenzialità dei dati, che risulta uno dei requisiti più complessi e delicati per la sicurezza dei computer e della rete. Un metodo per garantire la confidenzialità è tramite il controllo del flusso di informazioni. Abbiamo visto il controllo statico del flusso di informazioni utilizzando un sistema dei tipi di sicurezza introdotto in [8], per il while language. Tale sistema dei tipi è corretto come visto in [10]. Inoltre abbiamo introdotto un nuovo sistema dei tipi di sicurezza che estende quello precedente e abbiamo dimostrato la sua correttezza. Per concludere abbiamo visto degli esempi di programmi sicuri, non tipabili nel sistema dei tipi di [8] ma tipabili nel nostro sistema dei tipi e abbiamo costruito gli alberi di inferenza per tali programmi. Infine notiamo che il nostro sistema dei tipi non è completo in quanto estende quello di Myers e Sabelfeld in modo da tipare solo programmi che sono in composizione con un assegnazione sicura ad una variabile *bassa* ma non comprendono composizioni con altri programmi più complessi, ad esempio con i costrutti `if` e `while`.

Appendice A

La Sicurezza nel linguaggio Java

Il linguaggio Java è nato nel 1995 ed è diventato uno dei linguaggi di programmazione piú usati, questo sopratutto grazie all'indipendenza dalla piattaforma hardware/software(basta avere la virtual machine per la combinazione hardware/software scelta). Tale indipendenza permette di scrivere codice Java che può essere eseguito praticamente su qualsiasi piattaforma. In particolare, sono nate le applet Java che sono programmi Java che risiedono su una macchina server, si possono scaricare ed essere eseguite su una macchina client. Il server ed il client sono collegati tra loro dal Web. Una volta scaricate, le applet Java potrebbero effettuare delle operazioni proibite e maliziose, per questo si impongono delle restrizioni sulle operazioni che possono effettuare all'interno del client.

La Java virtual machine fornisce un sistema runtime in grado di garantire l'accesso alla macchina virtuale stessa e al mondo esterno (per esempio, il flusso di output `System.out`). Il Java runtime environment utilizza degli oggetti devoluti al caricamento ed al controllo delle applet, come il `ClassLoader` e il `SecurityManager`, che risultano essere dei moduli importanti per la sicurezza.

La responsabilità di caricare e verificare codice Java è del `ClassLoader`. Il `ClassLoader` deve determinare le classi da caricare per completare l'esecuzione di un'applicazione ed assicurarsi che parti importanti del Java runtime environment non siano rimpiazzati da codice alterato. Per quanto detto, è ovvio che un'applet non può creare il proprio `ClassLoader` e nemmeno invocare metodi del `ClassLoader` di sistema.

Il compito del `SecurityManager` è di tener traccia degli oggetti ai quali è permesso compiere azioni pericolose all'interno del sistema. Quindi, il suo compito è di evitare l'esecuzione di molte operazioni a codice non affidabile e di permetterlo invece a codice affidabile. Se un'applet cerca di compiere delle azioni a lei proibite, il `SecurityManager` solleva una `SecurityException`. Perciò un'applet non può creare il suo `SecurityManager`.
Si noti che nel sistema runtime esiste un unico `SecurityManager`, mentre i `ClassLoader` sono di diversi tipi: il `ClassLoader` *di sistema* è il class loader utilizzato dalla macchina virtuale per poter caricare la classe iniziale dell'applicazione. Tutte le altre classi di cui necessita l'applicazione in questione sono caricate dal `ClassLoader` *di contesto*. Inoltre, esiste anche il `ClassLoader` *di bootstrap* il quale viene utilizzato per caricare le classi che appartengono alla macchina virtuale, chiamate classi di sistema come ad esempio, `Object`, `String`, `List` e cosi via [2].

Le prime versioni di Java contenevano degli errori per quanto riguarda la sicurezza [3]. Tanti di questi problemi derivavano dalla mancanza di una semantica adeguata per il linguaggio Java e dalla discrepanza tra la semantica di Java e quella del suo bytecode. Uno di questi problemi è il seguente.

Superclass Constructor [3, sec.3.7]. Il linguggio Java richiede che un costruttore, come prima azione, chiami un altro costruttore della stessa classe oppure un costruttore di una superclasse. Le classi del sistema, come il `ClassLoader` e il `SecurityManager` precedentemente menzionate, si basano su questo comportamento per garantire la sicurezza. Queste classi hanno dei costruttori per controllare se sono chiamate da applet Java e in caso affermativo sollevano una `SecurityException`. Mentre il linguaggio Java lo proibisce, il verificatore del bytecode accetta il codice equivalente alla seguente definizione di classe:

```
class CL extends ClassLoader {
    CL {
        try { super (); }
        catch (Exception e) {}
        }
}
```

Tale codice ci permette di costruire dei `ClassLoader` e `SecurityManager` parzialmente non inizializzati.

I `ClassLoader`, in particolare, sono delle classi molto interessanti da istanziare. Il `ClassLoader` non ha variabili di istanza e il codice del suo costruttore viene eseguito una sola volta, prima che l'applet venga caricata. Il codice e le considerazioni fatte portano ad avere un `ClassLoader` propriamente inizializzato che sta sotto il controllo di un'applet, il che può trasformarsi con facilità in un attacco. A questo punto l'applet può utilizzare il `ClassLoader` per caricare, ad esempio un `SecurityManager` malizioso.

Inoltre, come vediamo nel seguente esempio, creare tale `ClassLoader` dà la possibilità ad un attaccante di sconfiggere il sistema dei tipi di Java.

Attacco del `ClassLoader` ostile [4, ch.5, sec.8]. Un'altra funzionalità fondamentale del `ClassLoader` è quella di definire gli spazi dei nomi accessibili da diverse classi e le regole che mettono in relazione tali spazi dei nomi. Un `ClassLoader` ostile può creare uno spazio dei nomi intrecciato dove classi differenti vedono l'ambiente Java in maniera differente. Questa inconsistenza può creare confusione nel sistema dei tipi di Java. Assumiamo che due classi A e B facciano entrambe riferimento ad una terza classe chiamata C. Ma le due classi A e B hanno idee diverse del significato del nome C. La classe A punta ad una classe C1 mentre la classe B punta ad una classe C2 questo perché quando viene chiamato il `ClassLoader` per sapere quale classe corrisponde al nome C, questo risponde alla classe A con C1 e alla classe B con C2. Se un oggetto di tipo C è allocato in A e poi è passato come argomento ad un metodo di B, il metodo tratterà l'oggetto come se avesse un tipo C2, e non C1. Se dunque i campi di C1 hanno modificatori di visibilità (public, private, protected) diversi da quelli di C2 ecco che si è sconfitto il sistema dei tipi di Java.

Bibliografia

[1] Gary McGraw and Greg Morissett. *Attacking Malicious Code: A report to the Infosec Research Council*. IEEE Software 17(5), May 2000.

[2] Ken Arnold, James Gosling and David Holmes. *Il linguaggio Java: Manuale ufficiale* Quarta edizione. Pearson Education Italia S.r.l, 2006.

[3] Drew Dean, Ed Felten and Dan Wallach. *JAVA Security:From HotJava to Netscape and beyond*. In Proc. Symp. Security and Privacy. IEEE, May 1996.

[4] Gary McGraw and Ed Felten. *Securing JAVA: Getting Down to Business with Mobile Code*. John Wiley & Sons, Inc. January 1999. `http://www.securingjava.com/`

[5] Dexter Kozen. *Language-Based Security*. In Proc MFCS'99, Springer LNCS 1672, pp. 284-298, 1999.

[6] Andrew C. Mayers. *JFlow: Practical static information flow control*. In Proc. 26th Symp. Principles of Programming Languages. ACM, January 1999.

[7] B. W. Lampson. *A note on the confinement problem.* Comm. of the ACM, vol. 16, no. 10, pp. 613-615, October 1973.

[8] Andrei Sabelfeld and Andrew C. Myers. *Language-Based Information-Flow Security.* IEEE Journal on Selected Areas in Communications, vol. 21, no. 1, January 2003.

[9] A. C. Myers, N. Nystrom, L. Zheng and S. Zdancewic. *Jif: Java information flow*, Software release. `http://www.cs.cornell.edu/jif/`, July 2001.

[10] D. Volpano, G. Smith and C. Irvine. *A sound type system for secure flow analysis.* J.Computer Security, vol. 4, no. 3, pp. 167-187, 1996