Virus, teoricamente.

Michael Lodi
matr. 0000607041

Approfondimento per l'esame di
Sicurezza e Crittografia

Introduzione

In biologia, i *virus* sono parassiti endocellulari capaci di riprodursi solo all'interno della cellula che infettano. Sono gli organismi più piccoli e più semplici, e la loro classificazione è ancora discussa. Essi sono caratterizzati da un involucro che contiene parte di codice genetico virale (DNA o RNA). Da soli, i virus sono incapaci di fenomeni vitali e, per attivarsi e funzionare, devono sempre entrare in una cellula vivente. Una volta entrati, i virus possono iniziare una serie di attività tra cui *moltiplicare il proprio genoma* e *autoassemblarsi*. Il tutto con danni più o meno gravi per la cellula ospite.

Con forte analogia a quanto avviene in natura, definiamo **virus** (informatico) un programma che ha l'abilità di infettare altri programmi modificandoli per includere una copia (possibilmente evoluta) di se stesso.

Molti studi sono stati fatti sui virus, ma la maggior parte si concentrano su aspetti prettamente pratici.

I lavori teorici esistono ma sono pochi e non uniformi. Solo di recente alcuni lavori di Marion e altri hanno cercato di creare un framework per la modellazione dei virus. Inoltre recentissimamente Marion ha proposto un modello di macchina a registri che ha la caratteristica chiave di poter *modificare il proprio stesso codice*.

Scopo del presente lavoro è proprio quello di fare una panoramica su questo framework e di mostrare come il modello delle macchine a registri automodificantesi può validamente modellare i virus, *mostrando alcuni esempi e applicazioni non presenti negli articoli (in particolare ILoveYou e tutti quelli legati al teorema di Smullyan).*

Verrà poi effettuata una doverosa panoramica sull'individuazione dei virus, obbiettivo guida dei lavori sulla virologia, e si accennerà a una promettente strada basata sull'interpretazione astratta.

Indice

Capitolo 1

Panoramica

Seguendo [DP07], possiamo definire **malware** un programma con intenti malevoli che ha il potenziale di danneggiare, senza il consenso dell'utente, la macchina su cui viene eseguito o la rete su cui comunica. L'aumentare della diffusione, della complessità e della varietà di strumenti informatici e della connettività tramite Internet ha portato ad un'ampia diffusione di codice maligno.

Possiamo classificare il malware a seconda delle sue azioni malevole (*payload)* e del suo metodo di propagazione in:

Virus propriamente detti Un virus è un programma auto-propagante che si lega a programmi ospiti e si propaga con l'esecuzione di questi. Tipicamente consiste in una *procedura di infezione,* che cerca un nuovo programma da infettare, e una *procedura di danneggiamento,* che esegue il payload. I virus sono progettati per danneggiare le macchine corrompendo programmi, cancellando file o riformattando l'hard disk. Altri virus, chiamati "benigni", semplicemente si replicano, corrompendo però anch'essi le macchine ospiti tramite l'occupazione di memoria o di spazio su disco.

Worm Generalmente si parla di worm per indicare programmi che utilizzano una rete per spedire copie di se stessi ad altri sistemi ad essa connessi. Non necessitano di un programma ospite e non è detto che contengano un payload. Sono comunque molto dannosi, per esempio a causa dell'aumento di traffico che generano.

Trojan horses Come i virus, si nascondono in programmi ospiti apparentemente utili o innocui per un utente ignaro. Proprio sulle azioni dell'utente, che li esegue, basano il loro attacco.

Backdoor Una backdoor è un programma progettato per bypassare le politiche di sicurezza e permettere ad entità esterne di avere un controllo (di solito remoto) della macchina o della rete. Possono essere programmi autonomi o nascondersi dentro altri software.

Spyware Uno spyware è generalmente un programma progettato per monitorare le azioni dell'utente e raccogliere informazioni private che lo riguardano, per poi inviarle a un'entità esterna. Gi scopi possono essere i più diversi: dai più maligni (es. scoprire il numero della carta di credito) ai "meno" (es. pubblicità).

Questa relazione prenderà in considerazione modelli che possono essere adatti a rappresentare molte delle forme di malware viste, a cui ci riferiremo comunque con il termine radicato di ***virus***.

1.1 Virologia Teorica

Gli studi sui virus sono iniziati con lavori teorici di Cohen [Coh87] e Adleman [Adl88] negli anni ottanta. Paradossalmente, esistono oggi pochissimi lavori nel campo della *virologia teorica*.

Per contrasto, una grande quantità di opere trattano degli aspetti pratici (per lo stato dell'arte si veda ad esempio l'ottima rassegna [Szo05]).

Tra i lavori più recenti, di grande interesse e innovazione sono alcuni articoli di Bonfante, Kaczmarek e Marion [BKM05, BKM06, BKM07, Mar12], che qui studieremo nel dettaglio.

In tali lavori si mostra, tra le altre cose, come le definizioni in essi date possono catturare (ed estendere) definizioni date in altri lavori teorici.

1.2 Definizione informale di virus

Una definizione di virus è stata originariamente data in [Coh87] e ripresa in [Szo05], e pare generalmente accettata.

Definizione. Un **virus** è un programma che ha l'abilità di infettare altri programmi modificandoli per includere una copia (possibilmente evoluta) di se stesso.

Dunque un virus è in primis un programma che si *auto-riproduce*. Centrale è quindi *una visione omogenea di programmi e dati*. Questo non stupisce, poiché, fino dalla definizione di Macchina di Turing Universale, i programmi possono essere visti come input di altri programmi, cioè come dati.

Basandoci su questo importante concetto, possiamo un'idea della struttura di un virus tramite la sua *specifica virale*.

```
viralSpec(virus y){
        esegui azioni maligne;
        propaga il codice y ad altri file;
}
```

Una soluzione a questa specifica è un programma $\mathbf{v}$ che sia un *punto fisso* per `viralSpec`, ovvero $\mathbf{v} = \texttt{viralSpec}(\mathbf{v})$. Un modo semplice e *costruttivo* di trovare tale soluzione è usare il noto *secondo teorema di ricorsione di Kleene*.

I virus prodotti in questo modo sono tutti uguali. Ma il teorema, avendo *infinite soluzioni*, ci permette di costruire copie di virus differenti (e quindi difficili da individuare).

Prima di formalizzare e dettagliare queste intuizioni, approfondiamo il tema della ***auto-riproduzione.*** Essa si compone di vari aspetti, in particolare:

- **auto-riferimento**: è la capacità di un programma di riferirsi a se stesso. Un tipico esempio è un algoritmo ricorsivo, che ha un puntatore a se stesso;
- **auto-replicazione**: è la capacità di un programma di copiarsi. Tipico esempio sono i *quine* (programmi che restituiscono in output il proprio codice, [Hof79]). Il processo di duplicazione potrebbe produrre dei discendenti diversi dal programma padre, e si parla in questo caso di *mutazione*;
- **auto-modifica**: è la capacità di un programma di modificare se stesso, nel senso di *alterare il proprio codice*. Potrebbe modificare le proprie istruzioni a runtime o generare nuovo codice a partire da alcuni dati, modificando il programma complessivo.

1.3 Un nuovo approccio

L'approccio seguito in queste note si differenzia dai lavori teorici precedenti, per diversi motivi.

1. *Un virus è un programma e non una funzione*. Dunque più che un approccio funzionale (tradizionalmente basato sulle Macchine di Turing), ci si sposta verso un approccio "orientato alla programmazione".

2. Viene tenuta in considerazione ed esplicitata la *funzione di propagazione*, ovvero, informalmente, un programma che *descrive come il virus si propaga e muta*. Ne parleremo ampiamente.

3. I modelli proposti sono abbastanza generali da tenere in considerazione moltissimi tipi di virus, in particolare:

   (a) l'utilizzo di teoremi di ricorsione di crescente potenza permette di *modellare comportamenti complessi*;

   (b) l'utilizzo di modelli in cui c'è uniformità tra programmi e dati permettono di modellare il fatto, piuttosto concreto, che *sia i programmi che i dati possano essere infettati*;

   (c) i modelli proposti sono adatti a catturare il *polimorfismo*, sempre più utilizzato dai virus per sfuggire al riconoscimento.

4. Viene adottato un punto di vista costruttivo: è possibile *scrivere concretamente* dei virus.

Capitolo 2

Strumenti

In questo capitolo individuiamo gli strumenti necessari allo studio. Alcuni sono risultati noti e classici della teoria della calcolabilità, altri invece sono di recentissima introduzione.

Il nostro studio verrà condotto in particolare tramite:

- una definizione generale e astratta di linguaggio di programmazione Turing-completo e della sua semantica;
- uno specifico modello di computazione: le Self-modifying register machines.

2.1 Un linguaggio di programmazione e alcuni teoremi

Per la definizione di virus verranno utilizzati strumenti tipici della teoria della calcolabilità. Visto l'approccio più concreto e orientato ai linguaggi di programmazione, è bene introdurre (o reintrodurre) alcuni concetti. Per uno studio approfondito, si rimanda al libro di N.D. Jones [Jon97], che si pone proprio come obbiettivo quello di presentare tali concetti visti da una prospettiva più legata ai linguaggi di programmazione.

Fissiamo un dominio di computazione $\mathbb{D}$, che contiene uniformemente programmi e dati. Sia $\mathbf{p}$ un programma scritto in un qualche linguaggio di programmazione accettabile.

Sia $[\![\cdot]\!] : \mathbb{D} \to (\mathbb{D} \hookrightarrow \mathbb{D})$ la funzione semantica di tale linguaggio, tale che, per ogni programma $\mathbf{p}$, $[\![\mathbf{p}]\!]$ sia la funzione (parziale: $\hookrightarrow$) calcolata da $\mathbf{p}$.

2.1.1 Convenzioni, notazione, semplificazioni

- Non distinguiamo programmi e dati. Nonostante questo, indicheremo in grassetto un elemento di $\mathbb{D}$ che rappresenta un programma (es. $\mathbf{p} \in \mathbb{D}$), tenendo presente che $x \in \mathbb{D}$ non rappresenta necessariamente un dato.
- Supponiamo di avere una funzione di *pairing* interna a $\mathbb{D}$ e due funzioni di *proiezione*. Per cui non faremo distinzione tra funzioni unarie ed n-arie.
  Useremo $\mathbf{p}$ per indicare un programma e $\vec{\mathbf{p}}$ per indicarne una sequenza; allo stesso modo $x \in \mathbb{D}$ e $\vec{x} = (x_1, ..., x_n) \in \mathbb{D}$.
  Ancora, per semplificare la notazione, spesso ometteremo la freccia $\vec{}$ sui vettori di programmi/dati.

- Quando non ci sarà confusione, ci riferiremo alla *funzione* $[\![\mathbf{p}]\!]$ con il nome p.
- Quando confrontiamo due funzioni (es. $f = g$ o $[\![\mathbf{p}]\!] = [\![\mathbf{q}]\!]$), l'uguaglianza indica che per ogni x, $f(x)\downarrow$ e $g(x)\downarrow$ e $f(x) = g(x)$ oppure che $f(x)\uparrow$ e $g(x)\uparrow$.

2.1.2 Teoremi di ricorsione

Vale allora il seguente risultato, dovuto a Kleene.

Teorema 1 (s-m-n)**.** *Esiste un programma* $\mathbf{spec}$ *tale che, per ogni* $\mathbf{p}, \vec{x}, \vec{y} \in \mathbb{D}$

$$[\![\mathbf{p}]\!](\vec{x}, \vec{y}) = [\![[\![\mathbf{spec}]\!](\mathbf{p}, \vec{x})]\!](\vec{y})$$

Dimostrazione. Basta pensare ad un programma in cui tutte le occorrenze del primo parametro formale siano sostituite con il primo parametro attuale. □

Veniamo ora al risultato centrale nella costruzione di virus.

Teorema 2 (Secondo teorema di ricorsione di Kleene)**.** *Sia* g *una funzione parziale calcolabile. Allora esiste un programma* $\mathbf{e}$ *tale che*

$$[\![\mathbf{e}]\!](x) = g(\mathbf{e}, x)$$

Dimostrazione. Come detto, sia $spec = [\![\mathbf{spec}]\!]$. Sia $[\![\mathbf{p}]\!](y, x) = g(spec(y, y), x)$. Allora avremo

$$\begin{array}{rcl} g(spec(y,y),x) & = & [\![\mathbf{p}]\!](y,x) \\ \text{per s.m.n} & = & [\![spec(\mathbf{p},y)]\!](x) \end{array}$$

Se poniamo $\mathbf{e} \stackrel{\text{def}}{=} spec(\mathbf{p}, \mathbf{p})$ allora si ha

$$\begin{array}{rcl} g(\mathbf{e},x) & = & g(spec(\mathbf{p},\mathbf{p}),x) \\ & = & [\![spec(\mathbf{p},\mathbf{p})]\!](x) \\ & = & [\![\mathbf{e}]\!](x) \end{array}$$

□

Osservazione 1. Possiamo vedere $spec(\mathbf{p}, x)$ come un programma che contiene il dato x al suo interno. Dunque $spec(\mathbf{p}, \mathbf{p})$ corrisponde ad un programma che ha al suo interno il suo stesso codice. È l'**auto-riferimento**.
Inoltre, il teorema di Kleene permette di utilizzare questa caratteristica per l'**auto-replicazione**. Sia infatti $p_1(y, x) = y$. Per Kleene, possiamo calcolare il suo punto fisso $\mathbf{e}$, ottenendo un *quine*: $[\![\mathbf{e}]\!](x) = p_1(\mathbf{e}, x) = \mathbf{e}$.

Una generalizzazione a due funzioni del teorema precedente è dovuta a Smullyan [Smu93].

Teorema 3 (Teorema di doppia ricorsione di Smullyan)**.** *Siano* g_1 *e* g_2 *due funzioni parziali calcolabili. Allora esistono due programmi* $\mathbf{e_1}$ *ed* $\mathbf{e_2}$ *tali che*

$$[\![\mathbf{e_1}]\!](x) = g_1(\mathbf{e_1}, \mathbf{e_2}, x) \;\; e \;\; [\![\mathbf{e_2}]\!](x) = g_2(\mathbf{e_1}, \mathbf{e_2}, x)$$

Dimostrazione. Siano $[\![\mathbf{p_1}]\!](z,y,x) \stackrel{\text{def}}{=} g_1(spec(z,z,y), spec(y,y,z), x)$

e $[\![\mathbf{p_2}]\!](z,y,x) \stackrel{\text{def}}{=} g_2(spec(z,z,y), spec(y,y,z), x)$. Avremo allora

$$\begin{aligned} g_1(spec(z,z,y), spec(y,y,z), x) &= [\![\mathbf{p_1}]\!](z,y,x) \\ &= [\![spec(\mathbf{p_1}, z, y)]\!](x) \end{aligned}$$

e parallelamente

$$\begin{aligned} g_2(spec(z,z,y), spec(y,y,z), x) &= [\![\mathbf{p_2}]\!](z,y,x) \\ &= [\![spec(\mathbf{p_2}, z, y)]\!](x) \end{aligned}$$

Definiamo ora $\mathbf{e_1} \stackrel{\text{def}}{=} spec(\mathbf{p_1}, \mathbf{p_1}, \mathbf{p_2})$ ed $\mathbf{e_2} \stackrel{\text{def}}{=} spec(\mathbf{p_2}, \mathbf{p_2}, \mathbf{p_1})$. Avremo allora

$$\begin{aligned} g_1(\mathbf{e_1}, \mathbf{e_2}, x) &= g_1(spec(\mathbf{p_1}, \mathbf{p_1}, \mathbf{p_2}), spec(\mathbf{p_2}, \mathbf{p_2}, \mathbf{p_1}), x) \\ &= [\![spec(\mathbf{p_1}, \mathbf{p_1}, \mathbf{p_2})]\!](x) \\ &= [\![\mathbf{e_1}]\!](x) \end{aligned}$$

e

$$\begin{aligned} g_2(\mathbf{e_1}, \mathbf{e_2}, x) &= g_2(spec(\mathbf{p_1}, \mathbf{p_1}, \mathbf{p_2}), spec(\mathbf{p_2}, \mathbf{p_2}, \mathbf{p_1}), x) \\ &= [\![spec(\mathbf{p_2}, \mathbf{p_2}, \mathbf{p_1})]\!](x) \\ &= [\![\mathbf{e_2}]\!](x) \end{aligned}$$

□

2.2 Self-modifying Register Machines (SRM)

Come già detto, una terza importante caratteristica da modellare è la possibilità del codice di **auto-modificarsi**. Sebbene sia usata in pratica in molti ambiti - compreso l'offuscamento per sfuggire al riconoscimento - (si veda l'Appendice A), i modelli teorici tradizionali mal si prestano alla modellazione di tale fenomeno.

Per questo Marion [Mar12] propone l'utilizzo di una macchina a registri con caratteristiche interessanti, ispirata a quella proposta in [Mos06].

Una Self-modifying register machine (SRM) è una macchina costituita da:

- **un numero finito (m) di registri** $R_0, R_1 ... R_m$. Ogni registro contiene parole sull'alfabeto $\{1, \#\}$ e può essere visto come una coda (FIFO) illimitata: la lettura avviene rimuovendo il primo carattere in testa e la scrittura aggiungendo un carattere in coda. Ogni registro può contenere programmi e dati (che sono visti in modo uniforme, senza distinzione) e *può essere eseguito*.
- **un register pointer** RP, che contiene l'indice del registro corrente (R_{RP}).
- **un istruction pointer** IP, che contiene l'indice (≥ 1) dell'istruzione corrente all'interno del registro corrente.

Un **programma** è una parola che può essere decodificata come un insieme di istruzioni.

Il programma è inizialmente memorizzato nel registro R_0. Il primo step consiste nel fetch ed esecuzione dell'istruzione corrente. L'istruzione successiva dipende dal tipo di istruzione eseguita: infatti il flusso di esecuzione può muoversi in un altro registro (se viene modificato RP), oppure semplicemente viene incrementato IP e si passa all'istruzione successiva.

Due importanti caratteristiche della macchina permettono *il movimento del codice*:

- un programma ha la possibilità di *copiare o spostare dati da un registro all'altro*;
- un programma ha la possibilità di *attivare un altro registro*, cioè di eseguire un altro programma.

Dunque, un programma all'interno di un registro può scrivere un programma in un altro registro ed eseguirlo. Il *programma complessivo*, cioè l'insieme di tutte le istruzioni eseguite, è sparso tra i registri. Possiamo intuitivamente pensare ad una SRM come a una macchina astratta in cui ogni registro contiene un processo; esiste un unico processo in esecuzione in ciascun istante e un processo può attivare e passare il controllo ad un altro processo.

Le istruzioni sono riassunte nella tabella 2.1. Oltre al loro codice nell'alfabeto $\{1, \#\}$ (in cui c^n indica la ripetizione n volte del carattere c), ne diamo una versione leggibile (che useremo nei nostri esempi) e un significato intuitivo.

Istruzione	***Opcode***	***Significato***
`put n,1`	$1^{n+1}\#$	Scrive 1 alla fine del registro R_{RP+n}, lascia invariato RP, incrementa IP di 1
`put n,#`	$1^{n+1}\#\#$	Scrive $\#$ alla fine del registro R_{RP+n}, lascia invariato RP, incrementa IP di 1
`jmp +n`	$1^{n+1}\#\#\#$	Incrementa IP di n.
`jmp -n`	$1^{n+1}\#\#\#\#$	Decrementa IP di n.
`case n`	$1^{n+1}\#\#\#\#\#$	Legge (e rimuove) il primo carattere in R_{RP+n}: - se è ϵ, incrementa IP di 1 - se è 1, incrementa IP di 2 - se è $\#$, incrementa IP di 3
`exec n`	$1^{n+1}\#\#\#\#\#\#$	Il controllo passa ad R_{RP+n}, ovvero $RP := RP + n$ e $IP := 0$

Tabella 2.1: Istruzioni utilizzabili in una SRM

Notiamo come tutte le istruzioni, a parte `exec`, incrementano IP e lasciano invariato RP. Notiamo inoltre che:

- tutte le istruzioni sono *relative al registro corrente*, quindi i programmi possono essere "shiftati" in avanti senza sforzi;
- *un programma può manipolare ed eseguire solo registri alla sua destra*, non avendo accesso a quelli che lo precedono o a se stesso.

2.2.1 Semantica operazionale.

Per descrivere formalmente il comportamento di una SRM ne diamo ora una semantica operazionale.

Una **configurazione** $(\mathfrak{R}, RP, n)$ consiste di:

- uno store $\mathfrak{R}$, che è una funzione finita tale che $\mathfrak{R}(m) = R_m$ per ogni m;
- un puntatore di registro RP;
- un valore n che può essere il valore del puntatore IP oppure $Halt$ (e in questo caso sarà una **configurazione finale**).

L'aggiornamento dello store, l'aritmetica dei puntatori, la concatenazione di stringhe sono definite nel modo usuale.

Il programma corrente è quello puntato da RP, inizialmente in R_0. L'istruzione corrente è $RP[IP]$.

Definiamo

- una **transizione in un passo**: $(\mathfrak{R}, RP, n) \to (\mathfrak{R}', RP', IP')$ vale se l'esecuzione dell'istruzione $RP[IP]$ modifica la configurazione come indicato in tabella 2.2, e dunque la prossima istruzione sarà $RP'[IP']$;
- una **computazione**: la chiusura riflessiva e transitiva di $\to$.

Un *programma termina* se:

- IP "esce" dal registro corrente
  - perché supera l'ultima istruzione,
  - perché un salto viene eseguito a un'istruzione fuori dal registro (in avanti o indietro);
- IP punta ad una parola che non è un'istruzione legale.

`put n,1`	$(\mathfrak{R}, RP, n) \to (\mathfrak{R}[R_{RP+n} \leftarrow R_{RP+n} \cdot 1], RP, IP + 1)$	se $RP[IP] = 1^{n+1}\#$
`put n,#`	$(\mathfrak{R}, RP, n) \to (\mathfrak{R}[R_{RP+n} \leftarrow R_{RP+n} \cdot \#], RP, IP + 1)$	se $RP[IP] = 1^{n+1}\#^2$
`jmp +n`	$(\mathfrak{R}, RP, n) \to (\mathfrak{R}, RP, IP + n)$	se $RP[IP] = 1^{n+1}\#^3$
`jmp -n`	$(\mathfrak{R}, RP, n) \to (\mathfrak{R}, RP, IP - n)$	se $RP[IP] = 1^{n+1}\#^4$
`case n`		se $RP[IP] = 1^{n+1}\#^5$
	$(\mathfrak{R}, RP, n) \to (\mathfrak{R}, RP, IP + 1)$	se $R_{RP+n} = \epsilon$
	$(\mathfrak{R}, RP, n) \to (\mathfrak{R}[R_{RP+n} \leftarrow u], RP, IP + 2)$	se $R_{RP+n} = 1 \cdot u$
	$(\mathfrak{R}, RP, n) \to (\mathfrak{R}[R_{RP+n} \leftarrow u], RP, IP + 3)$	se $R_{RP+n} = \# \cdot u$
`exec n`	$(\mathfrak{R}, RP, n) \to (\mathfrak{R}, RP + n, 0)$	se $RP[IP] = 1^{n+1}\#^6$
altrimenti	$(\mathfrak{R}, RP, n) \to (\mathfrak{R}, RP, Halt)$	

Tabella 2.2: Semantica operazionale delle SRM, in cui $R_{RP+n} = \mathfrak{R}(RP + n)$

2.2.2 Semantica sequenziale

Introduciamo una semantica sequenziale, che permette di esprimere in modo più semplice le computazioni di SRM.

Scriviamo

$$(d_0, ..., d_i \cdot \underline{\mathbf{p}}, ..., d_n)$$

per indicare la configurazione $(\mathfrak{R}, RP, n)$ tale che $RP = i$, IP punta al primo carattere di $\mathbf{p}$,

$$\mathfrak{R}(j) = \begin{cases} d_i \cdot \mathbf{p} & se\ j = i \\ d_j & se\ 0 \leq j \leq n \\ \epsilon & se\ j > n \end{cases} .$$

La *sottolineatura* sta ad indicare che la parte di programma denotata da d_i è già stata eseguita, e iniziamo ad eseguire la sua parte restante: $\mathbf{p}$.

- La *configurazione iniziale* è del tipo $(\underline{\mathbf{p}}, d_1, ..., d_n)$
- Una *computazione* è scritta come

$$(d_0, ..., d_i \cdot \underline{\mathbf{p}}, ..., d_n) \Rightarrow (d'_0, ..., d'_j \cdot \underline{\mathbf{q}}, ..., d'_m)$$

- Una *configurazione* è *finale* (la computazione termina) se non ha nessuna parola sottolineata.

Mobilità di programmi

Come già detto, grazie al fatto che i programmi possono modificare ed eseguire solo registri alla propria destra, i registri a sinistra possono essere in un certo senso dimenticati.

Fatto 1. ***I registri a sinistra non vengono modificati da una computazione.*** *Per ogni* $\mathbf{p}$, *se*

$$c_0, ..., c_i, \underline{\mathbf{p}}, d_1, ..., d_n \Rightarrow c'_0, ..., c'_i, \mathbf{p}', d'_1, ..., d'_m$$

allora

$$i = j,\ c_k = c'_k \textit{ per ogni } 0 \leq k \leq i$$

Fatto 2. ***La computazione è indipendente dal contesto dei registri a sinistra.*** *Per ogni* $i, j, \mathbf{p}$ *se*

$$c_0, ..., c_i, \underline{\mathbf{p}}, d_1, ..., d_n \Rightarrow c_0, ..., c_i, \mathbf{p}', d'_1, ..., d'_m$$

allora

$$c'_0, ..., c'_j, \underline{\mathbf{p}}, d_1, ..., d_n \Rightarrow c'_0, ..., c'_j, \mathbf{p}', d'_1, ..., d'_m$$

Fatto 3. ***Un programma può essere rilocato traslandolo a destra.*** *Se*

$$\underline{\mathbf{p}}, d_1, ..., d_n \Rightarrow \mathbf{p}', d'_1, ..., d'_m$$

allora per ogni $i, c_0, ..., c_i$

$$c_0, ..., c_i, \underline{\mathbf{p}}, d_1, ..., d_n \Rightarrow c_0, ..., c_i, \mathbf{p}', d'_1, ..., d'_m$$

Nel seguito, useremo quindi la seguente come una *regola di semplificazione* o di *rilocazione dei programmi*:

$$c_0, ..., c_i, \underline{\mathbf{p}}, d_1, ..., d_n \Rightarrow \underline{\mathbf{p}}, d_1, ..., d_n \tag{2.1}$$

Composizione di programmi

Proprietà importante delle SRM è che la *composizione di programmi* si ottiene con la *concatenazione di stringhe*.

Fatto 4. ***Condizione di connettività sequenziale.*** *Per ogni* $\mathbf{p}, \mathbf{q}, d_1, ..., d_n$*, se*

$$\underline{\mathbf{p}}, d_1, ..., d_n \Rightarrow \mathbf{p}, d'_1, ..., d'_m$$

e se, posto che RP *sia invariato ed* IP *punti all'ultima istruzione di* $\mathbf{p}$*,*

$$\underline{\mathbf{q}}, d'_1, ..., d'_m \Rightarrow \mathbf{q}, d''_1, ..., d''_k$$

allora

$$\underline{\mathbf{p} \cdot \mathbf{q}}, d_1, ..., d_n \Rightarrow \mathbf{p} \cdot \mathbf{q}, d''_1, ..., d''_k$$

Turing completezza

Il modello delle SRM è Turing-completo.

Inoltre, è facile scrivere un auto-interprete $\mathbf{interp}$ che soddisfi

$$\underline{\mathbf{interp}}, \mathbf{p}, d_1, ..., d_n \Rightarrow \underline{\mathbf{p}}, d_1, ..., d_n$$

Infatti, possiamo implementare il programma $\mathbf{interp}$ nel seguente modo (dove, ora e nel seguito, il numero di riga indica il valore del registro IP):

```
1 exec 1
```

2.2.3 Esempi di programmi

Vediamo ora alcuni esempi di programmi in SRM. I programmi sono, per semplicità, parametrizzati da un numero n, che è il numero massimo di argomenti. I registri con indice superiore si suppongono inizializzati a ϵ (cioè vuoti) e sono utilizzati come registri di lavoro.

Sposta

Il programma $\mathbf{sposta\,i\,j}$ sposta il contenuto del registro R_i alla fine del registro R_j e cancella il contenuto di R_i. Formalmente

$$\underline{\mathbf{sposta\,i\,j}}, d_1, ..., d_i, ..., d_j, ..., d_n \Rightarrow \mathbf{sposta\,i\,j}, d_1, ..., \epsilon, ..., d_j \cdot d_i, ..., d_n$$

Una possibile implementazione per $\mathbf{sposta\,i\,j}$ è la seguente:

```
1 case i          (*Legge il prossimo carattere del registro i..*)
2 jmp 6           (*... se è vuoto salto a IP=8, cioè Halt)
3 jmp 3           (*... se inizia con 1 salto all'istruzione 6)
4 put j,#         (*... se inizia con # scrivo # nel registro j)
5 jmp -4          (* e poi ritorno all'inizio)
6 put j,1         (*scrivo 1 nel registro j)
7 jmp -6          (*e ritorno all'inizio)
```

Concatena

Il programma **concatena i j** copia (concatena) il contenuto del registro R_i alla fine del registro R_j. Formalmente

$$\underline{\textbf{concatena i j}}, d_1, ..., d_i, ..., d_j, ..., d_n \Rightarrow \textbf{concatena i j}, d_1, ..., d_i, ..., d_j \cdot d_i, ..., d_n$$

Cancella

Il programma **cancella j** elimina il contenuto del registro R_j

$$\underline{\textbf{cancella j}}, d_1, ..., d_j, ..., d_n \Rightarrow \textbf{cancella j}, d_1, ..., \epsilon, ..., d_n$$

Shift

Il programma **shift n** sposta a destra i registri $R_1...R_n$

$$\underline{\textbf{shift n}}, d_1, ..., d_n \Rightarrow \textbf{shift n}, \epsilon, d_1, ..., d_n$$

Backshift

Il programma **backshift n** sposta a sinistra i registri $R_2...R_{n+1}$ di una posizione, cancellando il contenuto di R_1

$$\underline{\textbf{backshift n}}, d_1, ..., d_{n+1} \Rightarrow \textbf{backshift n}, d_2, ..., d_{n+1}, \epsilon$$

Shift indicizzato

Il programma **shift n, i** sposta a destra di una posizione n registri a partire da R_i.

$$\underline{\textbf{shift n, i}}, d_1, ..., d_i..., d_{n+i} \Rightarrow \textbf{shift n, i}, d_1, ..., \epsilon, d_i, ..., d_{n+i}$$

Copia

Il programma **copia i j** cancella il contenuto di R_j e poi copia il contenuto di R_i anche in R_j.

$$\underline{\textbf{copia i j}}, d_1, ..., d_i, ..., d_j, ..., d_n \Rightarrow \textbf{copia i j}, d_1, ..., d_i, ..., d_i, ..., d_n$$

Scambia

Il programma $\mathbf{scambia\,i\,j}$ scambia il contenuto di R_j con quello di R_i.

$$\underline{\mathbf{scambia\,i\,j}}, d_1, ..., d_i, ..., d_j, ..., d_n \Rightarrow \mathbf{scambia\,i\,j}, d_1, ..., d_j, ..., d_i, ..., d_n$$

Esercizio. Implementare i programmi visti, anche utilizzando programmi già implementati. Per le soluzioni, si veda l'appendice B.

2.2.4 Il secondo teorema di ricorsione di Kleene

Come già visto, il secondo teorema di ricorsione di Kleene gioca un ruolo determinante nella costruzione di virus. Per questo, dimostriamo nel modello SRM due importanti risultati.

Con una tecnica del tutto analoga a quella vista in precedenza (per i dettagli si veda direttamente il lavoro [Mar12]) si dimostra l'esistenza di una funzione $fix : \{1, \#\}^* \to \{1, \#\}^*$ tale che, per ogni $\mathbf{p} \in \{1, \#\}^*$, allora $fix(\mathbf{p})$ è un programma SRM con il seguente comportamento:

$$\underline{fix(\mathbf{p})}, d_1, ..., d_n \Rightarrow \mathbf{p}, fix(\mathbf{p}), d_1, ..., d_n$$

Teorema 4 (Kleene su SRM)**.** *Per ogni naturale n e ogni programma $\mathbf{g} \in \{1, \#\}^*$, esiste un programma $\mathbf{e} \in \{1, \#\}^*$ tale che, per ogni $d_1, ..., d_n \in \{1, \#\}^*$*

$$\underline{\mathbf{e}}, d_1, ..., d_n \Rightarrow \mathbf{g}, \mathbf{e}, d_1, ..., d_n$$

Dimostrazione. Basta scegliere $\mathbf{e} \stackrel{\text{def}}{=} fix(\mathbf{g})$. □

Da quanto detto segue una versione uniforme del teorema, dovuta a Myhill.

Teorema 5 (Myhill)**.** *Esiste un programma $\mathbf{fix} \in \{1, \#\}^*$ tale che, per ogni programma $\mathbf{g} \in \{1, \#\}^*$ e per ogni $d_1, ..., d_n \in \{1, \#\}^*$ allora*

$$\begin{aligned} \underline{\mathbf{fix}}, p &\Rightarrow \mathbf{fix}, fix(\mathbf{p}) \\ \underline{fix(\mathbf{p})}, \mathbf{p}, d_1, ..., d_n &\Rightarrow \mathbf{p}, fix(\mathbf{p}), d_1, ..., d_n \end{aligned}$$

Dimostrazione. Basta fornire il programma $\mathbf{fix}$ che implementa la funzione fix. □

2.2.5 Il teorema di doppia ricorsione di Smullyan

Possiamo pensare di utilizzare questo risultato nelle SRM.

Teorema 6 (Smullyan su SRM)**.** *Per ogni naturale n e ogni programma $\mathbf{g_1}, \mathbf{g_2} \in \{1, \#\}^*$, esistono due programmi $\mathbf{e_1}, \mathbf{e_2} \in \{1, \#\}^*$ tali che, per ogni $d_1, ..., d_n \in \{1, \#\}^*$*

$$\begin{aligned} \underline{\mathbf{e_1}}, d_1, ..., d_n &\Rightarrow \mathbf{g_1}, \mathbf{e}_1, \mathbf{e_2}, d_1, ..., d_n \\ \underline{\mathbf{e_2}}, d_1, ..., d_n &\Rightarrow \mathbf{g_2}, \mathbf{e}_1, \mathbf{e_2}, d_1, ..., d_n \end{aligned}$$

Capitolo 3

Definizione di Virus

Un virus può essere pensato come un programma che si riproduce ed esegue alcune (altre) azioni. Quindi, un virus è un programma il cui meccanismo di propagazione può essere descritto tramite una funzione calcolabile $\mathcal{B}$, che chiamiamo *funzione di propagazione*. La funzione $\mathcal{B}$ cerca e seleziona uno o più programmi $\mathbf{p}$ e replica il virus $\mathbf{v}$ all'interno di essi.

Diamo dunque ora la definizione formale di **virus**, che useremo *in tutto il resto della trattazione*.

Definizione 1 (Virus)**.** Sia $\mathcal{B}$ una funzione parziale calcolabile. Un **virus** $\mathbf{v}$ con rispetto per la funzione $\mathcal{B}$ è un programma tale che, per ogni $\mathbf{p}, x \in \mathbb{D}$

$$[\![\mathbf{v}]\!](\mathbf{p}, x) = [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](x)$$

Osservazione. Non dobbiamo commettere l'errore di pensare a $\mathcal{B}$ come alla specifica virale (che in generale chiameremo f). Possiamo pensare a $\mathcal{B}$

- come ad una **trasformazione di programmi**: $\mathcal{B}$ trasforma $\mathbf{p}$ nella sua forma infettata da $\mathbf{v}$, ovvero $\mathcal{B}(\mathbf{v}, \mathbf{p})$;
- come ad una **vulnerabilità**: è una proprietà del linguaggio di programmazione che permette al virus di propagarsi.

Inoltre, è bene notare come come un virus $\mathbf{v}$ accetta un numero infinito di funzioni di propagazione. Non è detto quindi che un virus con rispetto per una certa funzione di propagazione infetti tramite quella: potrebbe usare un altro meccanismo.

Nei prossimi due capitoli daremo le definizioni e numerosi esempi di quattro tipologie di virus, ottenute rispettivamente tramite i due teoremi di ricorsione e le loro varianti polimorfe.

Capitolo 4

Virus con Kleene

4.1 Virus Blueprint

In questa categoria rientrano i virus che si riproducono lasciando fuori dal processo di duplicazione la funzione di propagazione. In altre parole, la funzione di propagazione esiste (per poter parlare di virus) ma non viene utilizzata.

L'idea alla base di questa categoria è, banalmente, la copia del codice di $\mathbf{v}$. Infatti, per mantenere l'infezione, un virus deve diventare un organismo indipendente (essendo propriamente una nuova entità oppure prendendo l'identità di un programma ospite).

Osservazione 2. Nel modello basato sul generico linguaggio di programmazione non c'è a priori nessun meccanismo che permette di svegliare il virus creato/iniettato in un ospite (se non quello di eseguire l'ospite stesso).
Ciò è invece possibile e naturale nel modello delle SRM, che permettono l'esecuzione di altri registri, e lo si noterà nelle differenti implementazioni che mostreremo.

L'esistenza di blueprint-virus è garantita dal seguente.

Teorema 7. *Data una funzione parziale f, esiste un virus $\mathbf{v}$ tale che*

$$[\![\mathbf{v}]\!](\mathbf{p}, x) = f(\mathbf{v}, \mathbf{p}, x)$$

Dimostrazione. Per il teorema del punto fisso, esiste un programma $\mathbf{v}$ tale che

$$[\![\mathbf{v}]\!](\mathbf{p}, x) = f(\mathbf{v}, \mathbf{p}, x)$$

Bisogna provare che $\mathbf{v}$ è un virus, cioè che $[\![\mathbf{v}]\!](\mathbf{p}, x) = [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](x)$ per una qualche funzione di propagazione $\mathcal{B}$.

Sia $[\![\mathbf{e}]\!] = f$, allora per s-m-n avremo

$$[\![\mathbf{e}]\!](\mathbf{v}, \mathbf{p}, x) = [\![spec(\mathbf{e}, \mathbf{v}, \mathbf{p})]\!](x)$$

e quindi $\mathcal{B}(\mathbf{v}, \mathbf{p}) \stackrel{\text{def}}{=} spec(\mathbf{e}, \mathbf{v}, \mathbf{p})$. □

Possiamo pensare a f come alla *viral specification*, cioè al prototipo (*blueprint*, appunto) che il virus deve rispettare. Dunque un blueprint virus $\mathbf{v}$ per f è un programma che calcola f utilizzando il proprio codice $\mathbf{v}$ e il proprio ambiente $\mathbf{p}, x$.

Notiamo inoltre che, come già detto, questo tipo di virus non utilizza in alcun modo la sua funzione di propagazione.

Un'intuizione è data dal fatto che la funzione $\mathcal{B}$ si appoggia alla funzione $spec$, codificabile in ogni linguaggio di programmazione accettabile, e dunque la presenza dello specializzatore può essere vista come una *falla di sicurezza* intrinseca nel linguaggio stesso.

Vediamo ora alcuni esempi di blueprint virus.

4.1.1 Overwriting virus

Un overwriting è un virus che rimpiazza uno o più programmi con una copia di se stesso:

$$[\![\mathbf{v}]\!](\mathbf{p_1}, ..., \mathbf{p_n}) = (\mathbf{v}, ..., \mathbf{v})$$

Il virus che stiamo cercando sarà dunque un punto fisso per la specifica virale

$$f(y, \mathbf{p_1}, ..., \mathbf{p_n}) = (y, ..., y)$$

Il Teorema 7 ci garantisce l'esistenza di tale virus e ci dà gli strumenti concreti per costruirlo.

Overwriting virus in SRM

Poiché nel modello delle SRM possiamo attivare un programma dopo averlo scritto, la specifica sarà lievemente diversa.

$$\underline{\mathbf{owrt}}, y, p_1, ..., p_n \Rightarrow \mathbf{owrt}, \underline{y}, y, p_2, ..., p_n$$

Un'implementazione per $\mathbf{owrt}$ potrebbe essere:

```
1 copia 1,2
2 exec 1
```

Ponendo $\mathbf{v_{owrt}} \stackrel{\text{def}}{=} fix(\mathbf{owrt})$, il teorema di Kleene (4) permette di trovare una soluzione alla specifica, infatti:

$$\begin{aligned} \underline{\mathbf{v_{owrt}}}, p_1, ..., p_n &\Rightarrow \underline{\mathbf{owrt}}, \mathbf{v_{owrt}}, p_1, p_2, ..., p_n \\ &\Rightarrow \mathbf{owrt}, \underline{\mathbf{v_{owrt}}}, \mathbf{v_{owrt}}, p_2, ..., p_n \end{aligned}$$

Ora la computazione procede, visto che il nuovo virus viene attivato.

4.1.2 Organismo

Un organismo si duplica senza modificare alcun programma:

$$[\![\mathbf{v}]\!](\mathbf{p}) = (\mathbf{v}, \mathbf{p})$$

La funzione f da soddisfare è $f(y, \mathbf{p}) = (y, \mathbf{p})$.

La pericolosità di questo virus sta ovviamente nella possibilità di occupare spazio sull'hard disk o sulla memoria, rendendo il sistema inutilizzabile.

Organismo in SRM

Una possibile specifica per un organismo nelle SRM potrebbe essere

$$\underline{\mathbf{org}}, y, p_1, ..., p_n \Rightarrow \mathbf{org}, \underline{y}, y, p_1, , ..., p_n$$

Un'implementazione per $\mathbf{org}$ sarà:

```
1 shift n
2 copia 2,1
3 exec 1
```

e ancora una volta il teorema di Kleene ci dà la costruzione necessaria.

4.1.3 Ecto-simbionte

Un ecto-simbionte vive nella superficie di un altro programma, cioè mantiene la struttura del programma inalterata e si copia all'inizio o alla fine di esso:

$$[\![\mathbf{v}]\!](\mathbf{p_1}, ..., \mathbf{p_n}) = (\delta(\mathbf{v}, \mathbf{p_1}), ..., \delta(\mathbf{v}, \mathbf{p_n}))$$

in cui δ è una funzione di duplicazione, come ad esempio $\delta(\mathbf{v}, \mathbf{q}) = \mathbf{v} \cdot \mathbf{q}$.

Il virus che stiamo cercando è dunque una soluzione in y per la specifica virale $f(y, \mathbf{p_1}, ..., \mathbf{p_n}) = (\delta(y, \mathbf{p_1}), ..., \delta(y, \mathbf{p_n}))$.

Ecto-simbionte in SRM

Realizzare un ecto-simbionte è facile nelle SRM. Infatti avremo

$$\underline{\mathbf{ecto}}, y, p_1, ..., p_n \Rightarrow \mathbf{ecto}, y, \underline{p_1 \cdot y}, p_2, ..., p_n$$

L'implementazione di $\mathbf{ecto}$ può essere

```
1 concatena 1,2
2 exec 2
```

4.1.4 Un esempio reale: `ILoveYou`

Consideriamo un tipico esempio di blueprint virus, concettualmente analogo al famoso virus `ILoveYou`. Questo programma arriva come allegato di una e-mail dall'attraente oggetto "I Love You". L'apertura dell'allegato scatena l'attacco: dapprima il virus cerca sul computer le password della vittima e le invia all'attaccante, poi il virus si duplica inviando se stesso ad ogni contatto della rubrica mail.

Per modellare questo scenario, dobbiamo definire un processo che gestisca le mail.

Una mail è una coppia $m = \langle @, y \rangle$ formata da un indirizzo mail $@$ e di un dato y. Supponiamo che l'ambiente contenga una mailbox $mb = \langle m_1, ..., m_l \rangle$, costituita da una lista di mail. Per inviare una mail, essa va aggiunta alla mailbox $mb := \mathbf{cons}(m, mb)$. Supponiamo poi l'esistenza di un processo esterno che gestisca le mail.

Supponiamo che nell'ambiente di esecuzione $fs = \langle f_1, ..., f_n \rangle$ sia il filesystem locale, e $@bk = \langle @_1, ..., @_n \rangle$ sia la rubrica degli indirizzi mail.

Infine, supponiamo l'esistenza di un programma **find** che scansiona un filesystem alla ricerca di password, e ne restituisce in output una lista.

La viral specification per `ILoveYou` è data dal seguente programma **f**:

```
f(y,mb,@bk,fs){
        pass:=find(fs);
        mb:=cons(cons("attaccante@dom.it",pass),mb);
        foreach ad in @bk {
                mb:=cons(cons(ad,y),mb);
        }
        /* Il valore di ritorno è l'ambiente modificato. */
        return cons(mb,cons(@bk,fs));
}
```

Sia $f = [\![\mathbf{f}]\!]$, il Teorema 7 ci garantisce l'esistenza di **ILoveYou** come punto fisso per f rispetto a y.

Implementazione in SRM

Implementiamo ora questo meccanismo con le SRM.

Utilizzeremo una codifica per le liste, che all'interno di un registro saranno indicate con $\langle\,\rangle$, e i seguenti programmi ausiliari:

$$\begin{aligned}
\underline{\mathbf{cons}}, el_1, l_1, d &\Rightarrow \mathbf{cons}, \langle el_1, l_1 \rangle, d \\
\underline{\mathbf{head}}, \langle el_1, l_1 \rangle, d &\Rightarrow \mathbf{head}, el_1, l_1, d \\
\underline{\mathbf{tail}}, \langle el_1, l_1 \rangle, d &\Rightarrow \mathbf{tail}, l_1, d \\
\underline{\mathbf{empty}}, \epsilon, d &\Rightarrow \mathbf{empty}, 1, \epsilon, d \\
\underline{\mathbf{empty}}, \langle el_1, l_1 \rangle, d &\Rightarrow \mathbf{empty}, \#, \langle el_1, l_1 \rangle, d
\end{aligned}$$

Supponiamo infine che esista un programma **find** tale che, dato l'indice di inizio e la lunghezza dei registri su cui cercare, restituisca le password in essi memorizzate.

$$\underline{\mathbf{find\,i\,k}}, d_1, ..., \underbrace{d_i, ..., d_n}_{k} \Rightarrow \mathbf{find\,i\,k}, pass, d_1, ..., d_n$$

Supponiamo infine che il registro s contenga la codifica della stringa "`attaccante@dom.it`".

L'implementazione di **ILoveYou** potrebbe essere la seguente.

```
1 find 4, k        (*cerco le password*)
2 shift (k+5)      (*porto avanti l'indirizzo dell'attaccante..*)
```

```
3  sposta (6+k),1
4  cons             (*creo mail*)
5  cons             (*la inserisco nella mailbox*)
6  shift (3+k)      (*porto avanti la rubrica*)
7  copia 4,1
8  empty            (*in R1 mette 1 se rubrica vuota, altr. # *)
9  case 1
10 jmp -100         (*risultato inatteso. termina con errore*)
11 jmp +10          (*se la lista è vuota devo solo riordinare*)
12 copia 4,1        (*rubrica non vuota, preparo a mandare mail*)
13 scambia 1,2
14 head             (*estraggo primo indirizzo*)
15 scambia 2,4
16 scambia 2,3
17 cons             (*creo mail*)
18 cons             (*la inserisco nella mailbox*)
19 scambia 1,2      (*riordino per tornare al test sulla rubrica*)
20 jmp -12          (*torno al test empty*)
21 backshift 4+k    (*dove riaggiusto eliminando le cose in più*)
22 backshift 3+k    (*... e poi termino*)
```

Con un po' di pazienza si può mostrare come

$$\underline{\mathbf{f}}, mb, y, @bk, f_1, ..., f_n, s \Rightarrow \mathbf{f}, \langle \langle @_1, y \rangle, ..., \langle @_n, y \rangle, \langle s, pass \rangle, mb \rangle, y, @bk, f_1, ..., f_n$$

E ottenere $\mathbf{ILoveYou} \stackrel{\text{def}}{=} fix(\mathbf{f})$.

4.2 Virus blueprint polimorfi

I virus considerati finora duplicano se stessi senza modificare il proprio codice. Consideriamo ora quelli che mutano quando si duplicano: sono i virus polimorfi.

L'insieme delle soluzioni $\{\mathbf{e_i}\}$ di un'equazione

$$[\![\mathbf{e}]\!](\vec{x}) = f(\mathbf{e}, \vec{x})$$

è infinito ma purtroppo neanche semi-calcolabile (è addirittura Π_2^0-completo). Quindi non le possiamo enumerare tutte... ma ne possiamo generarne un numero infinito, sfruttando il Padding Lemma.

Lemma 1 (Padding Lemma)**.** *Dato un linguaggio di programmazione Turing-completo, esiste una funzione* ***totale biiettiva*** Pad *tale che per ogni programma* $\mathbf{q}$ *e per ogni* y*,*

$$[\![\mathbf{q}]\!] = [\![Pad(\mathbf{q}, y)]\!]$$

Dimostrazione. Un esempio banale di padding è l'aggiunta di operazioni quali $skip$ nel codice di $\mathbf{q}$. □

Teorema 8. *Sia* f *una funzione parziale calcolabile. Allora esiste una funzione calcolabile* Gen *tale che, per ogni* i*,*

- $Gen(i)$ *è un virus;*

- *se* $i \neq j$ *allora* $Gen(i) \neq Gen(j)$ *(il codice sorgente è diverso);*

- $[\![Gen(i)]\!](\mathbf{p}, x) = f(\mathbf{r}, i, \mathbf{p}, x)$ *in cui* $[\![\mathbf{r}]\!] = Gen$.

Inoltre, esiste un insieme di funzioni calcolabili $\{\mathcal{B}_i\}$ *tale che* $\mathcal{B}_i$ *è una funzione di propagazione per* $Gen(i)$.

Dimostrazione. Sia $[\![\mathbf{q}]\!] = f$. Sia $g(y, i) \stackrel{\text{def}}{=} Pad(spec(\mathbf{q}, y, i), i)$. Per Kleene, esiste $\mathbf{r}$ tale che

$$[\![\mathbf{r}]\!](i) = g(\mathbf{r}, i)$$

Chiamiamo $Gen = [\![\mathbf{r}]\!]$. Allora avremo

$$\begin{array}{rcl} [\![Gen(i)]\!](\mathbf{p}, x) & = & [\![Pad(spec(\mathbf{q}, \mathbf{r}, i), i)]\!](\mathbf{p}, x) \\ \text{per def di pad} & = & [\![spec(\mathbf{q}, \mathbf{r}, i)]\!](\mathbf{p}, x) \\ & = & [\![\mathbf{q}]\!](\mathbf{r}, i, \mathbf{p}, x) \\ & = & f(\mathbf{r}, i, \mathbf{p}, x) \end{array}$$

Notiamo che, poiché Pad è biiettiva, anche Gen lo è.

Non ci resta che provare che $Gen(i)$ è un virus rispetto ad una funzione $\mathcal{B}_i$, cioè che

$$[\![Gen(i)]\!](\mathbf{p}, x) = [\![\mathcal{B}_i(Gen(i), \mathbf{p})]\!](x)$$

per una qualche funzione di propagazione. Da quanto visto in precedenza avremo

$$[\![\mathbf{q}]\!](\mathbf{r}, i, \mathbf{p}, x) = [\![spec(\mathbf{q}, \mathbf{r}, i, \mathbf{p})]\!](x)$$

trovando quindi $\mathcal{B}_i(Gen(i), \mathbf{p}) \stackrel{\text{def}}{=} spec(\mathbf{q}, \mathbf{r}, i, \mathbf{p})$. □

4.2.1 Un virus polimorfo

Scriviamo un virus che emette un nuovo codice ogni volta che viene eseguito. La sua specifica è del tipo

$$[\![Gen(i)]\!](\mathbf{p}, x) = Gen(i + 1)$$

Per costruirlo, ci basterà porre $f(y, i, \mathbf{p}, x) \stackrel{\text{def}}{=} [\![y]\!](i + 1)$ e usare il teorema per trovare un punto fisso su y.

Un virus polimorfo in SRM

Se da un punto di vista teorico, come dimostrato, basta una funzione di padding (o più in generale una trasformazione *semantic preserving*), in pratica i metodi utilizzati per il polimorfismo si basano su funzioni di offuscamento.

Supponiamo dunque di avere una trasformazione di programmi Obs che preserva la semantica ma, in un qualche senso, "offusca" il codice del programma. Per un'introduzione all'offuscamento e al malware detection, si veda il capitolo 6 e, per esempio, [DP07, GJM12].

Data una viral specification $\mathbf{vs}$, vogliamo trovare $\mathbf{v}$ come la soluzione di un'equazione di punto fisso del tipo

$$\underline{\mathbf{v}}, d_1, ..., d_n \Rightarrow \underline{\mathbf{vs}}, Obs(\mathbf{v}), d_1, ..., d_n$$

Poiché Obs preserva la semantica, avremo che

$$\underline{Obs(\mathbf{v})}, d_1, ..., d_n \Rightarrow \underline{\mathbf{vs}}, Obs(Obs(\mathbf{v})), d_1, ..., d_n$$

e, supponendo Obs iniettiva, ogni mutazione sarà differente. Pensiamo a $\mathbf{vs}$ come a una delle specifiche viste: essa attiverà ogni volta un virus differente.

Sia $\mathbf{obs}$ un programma che calcola la funzione Obs, ovvero

$$\underline{\mathbf{obs}}, \mathbf{q} \Rightarrow \mathbf{obs}, Obs(\mathbf{q})$$

Allora una soluzione è data dal secondo teorema di Kleene ponendo $\mathbf{v} \stackrel{\text{def}}{=} fix(\mathbf{obs} \cdot \mathbf{vs})$. Infatti

$$\begin{aligned} \underline{fix(\mathbf{obs} \cdot \mathbf{vs})}, d_1, ..., d_n &\Rightarrow \underline{\mathbf{obs} \cdot \mathbf{vs}}, fix(\mathbf{obs} \cdot \mathbf{vs}), d_1, ..., d_n \\ &\Rightarrow \mathbf{obs} \cdot \underline{\mathbf{vs}}, Obs(fix(\mathbf{obs} \cdot \mathbf{vs})), d_1, ..., d_n \end{aligned}$$

Capitolo 5

Virus con Smullyan

I virus (blueprint) che abbiamo considerato finora non fanno uso della loro funzione di propagazione.

Vediamo ora una tipologia più avanzata di virus, che includono nel proprio codice il meccanismo di propagazione.

Sono i *virus Smith* e sono costruiti a partire dal teorema (3) di doppia ricorsione di Smullyan.

5.1 Virus Smith

Definizione 2. Un virus Smith è una coppia di programmi $\mathbf{v}, \mathbf{B}$ tali che, per ogni funzione f (che descrive il comportamento del virus)

$$\begin{cases} \mathbf{v} \text{ è un virus rispetto alla funzione di propagazione } \mathcal{B} = [\![\mathbf{B}]\!] \\ \forall \mathbf{p}, x \ [\![\mathbf{v}]\!](\mathbf{p}, x) = f(\mathbf{B}, \mathbf{v}, \mathbf{p}, x) \end{cases}$$

Teorema 9. *Data una funzione parziale f, esiste un virus Smith $\mathbf{v}, \mathbf{B}$ per essa.*

Dimostrazione. Utilizziamo il teorema di Smullyan. Siano $g_1 = prop$ e $g_2 = f$. Esistono allora due programmi $\mathbf{B}$ ed $\mathbf{v}$ tali che

$$[\![\mathbf{B}]\!](\mathbf{p}, x) = prop(\mathbf{B}, \mathbf{v}, \mathbf{p}, x) \text{ e } [\![\mathbf{v}]\!](\mathbf{p}, x) = f(\mathbf{B}, \mathbf{v}, \mathbf{p}, x)$$

Da questo segue già banalmente la seconda condizione perché $\mathbf{v}, \mathbf{B}$ siano un virus Smith. Non ci resta che dimostrare il fatto che $\mathbf{v}$ sia un virus rispetto alla funzione di propagazione $[\![\mathbf{B}]\!]$, ovvero che

$$[\![\mathbf{v}]\!](\mathbf{p}, x) = [\![[\![\mathbf{B}]\!](\mathbf{v}, \mathbf{p})]\!](x)$$

Sia $prop(\mathbf{B}, \mathbf{v}, \mathbf{p}, x) \stackrel{\text{def}}{=} spec(\mathbf{f}, \mathbf{B}, \mathbf{p}, x)$. Allora avremo

$$\begin{aligned} [\![[\![\mathbf{B}]\!](\mathbf{v}, \mathbf{p})]\!](x) &= [\![prop(\mathbf{B}, \mathbf{v}, \mathbf{v}, \mathbf{p})]\!](x) \\ &= [\![spec(\mathbf{f}, \mathbf{B}, \mathbf{v}, \mathbf{p})]\!](x) \\ &= [\![\mathbf{f}]\!](\mathbf{B}, \mathbf{v}, \mathbf{p}, x) \\ &= f(\mathbf{B}, \mathbf{v}, \mathbf{p}, x) \\ &= [\![\mathbf{v}]\!](\mathbf{p}, x) \end{aligned}$$

□

Un tipico esempio di virus Smith è il *parassita*.

5.1.1 Virus Parassita

Un virus parassita si inserisce in un programma esistente ($\mathbf{p}$). Quando il programma infettato ($[\![\mathbf{B}]\!](\mathbf{v}, \mathbf{p})$) viene eseguito, per prima cosa il programma infetta un altro programma $\mathbf{q}$ (ed è qui che fa uso della sua funzione di propagazione), e poi restituisce semplicemente il controllo al programma $\mathbf{p}$. Dunque il suo comportamento può essere specificato come

$$[\![\mathbf{v}]\!](\mathbf{p}, \mathbf{q}, x) = [\![[\![\mathbf{B}]\!](\mathbf{v}, \mathbf{p})]\!](\mathbf{q}, x) = [\![\mathbf{p}]\!]([\![\mathbf{B}]\!](\mathbf{v}, \mathbf{q}), x)$$

Possiamo definire la specifica virale come $f(z, y, \mathbf{p}, \mathbf{q}, x) \stackrel{\text{def}}{=} [\![\mathbf{p}]\!]([\![z]\!](y, \mathbf{q}), x)$ e, utilizzando il Teorema 9, ottenere una soluzione $\mathbf{B}$ per z e $\mathbf{v}$ per y.

Parassita in SRM

Vista la natura sequenziale delle SRM, anche in questo caso l'implementazione sarà lievemente diversa.

Nello scrivere le specifiche seguenti, immaginiamo di avere in z il codice della funzione di propagazione e in y il codice del virus. Possiamo definire la viral specification come

$$\underline{\mathbf{par}}, z, y, p_1, ..., p_n \Rightarrow \mathbf{par}, z, y, \underline{z}, p_1, ..., p_n$$

e la funzione di propagazione come

$$\underline{\mathbf{prop}}, z, y, p_1, ..., p_n \Rightarrow \mathbf{prop}, z, y, \underline{p_1}', y, p_2, ..., p_n$$

L'idea generale è che all'attivazione del virus, $\mathbf{par}$ chiama semplicemente la sua funzione di propagazione contenuta in z. Essa a questo punto "infetta il registro successivo", ovvero mette al posto di p_2 il codice del virus (y) e shiftando quindi in avanti gli altri registri.

Supponiamo che nel registro p_2 ci fosse un programma eseguibile. Ora, il programma p_1 potrebbe decidere di eseguire quel registro con un'istruzione `exec 1`. In questo caso, eseguirà y, che infetterà il programma successivo e poi eseguirà p_2, come desiderato.

Un unico accorgimento è dovuto al fatto che, shiftando in avanti tutti i registri $p_2, ..., p_n$ dobbiamo aggiornare, all'interno del codice p_1, i riferimenti relativi a tali registri (ovviamente ad esclusione degli `exec 1`, che devono rimanere immutati).

Un'implementazione per $\mathbf{par}$:

```
1 shift n,4
2 copia 1,3
3 exec 3
```

e una per $\mathbf{prop}$:

```
1 shift n,3
2 copia 2,4
3 <aggiorna i riferimenti dentro a P1>
4 exec 3
```

Dal secondo teorema di Smullyan per le SRM (6) otteniamo che, dati i programmi $\mathbf{par}$ e $\mathbf{prop}$ esistono due programmi $\mathbf{B_{par}}, \mathbf{v_{par}} \in \{1, \#\}^*$ tal che, per ogni $p_1, ..., p_n \in \{1, \#\}^*$

$$\begin{aligned} \underline{\mathbf{B_{par}}}, p_1, ..., p_n &\Rightarrow \mathbf{prop}, \mathbf{B_{par}}, \mathbf{v_{par}}, p_1, ..., p_n \\ \underline{\mathbf{v_{par}}}, p_1, ..., p_n &\Rightarrow \mathbf{par}, \mathbf{B_{par}}, \mathbf{v_{par}}, p_1, ..., p_n \end{aligned}$$

Avendo trovato così il nostro virus Smith formato dai programmi $\mathbf{B_{par}}, \mathbf{v_{par}}$.

Per convincerci della correttezza, analizziamo passo passo un attacco.

$$\begin{aligned} \underline{\mathbf{v_{par}}}, p_1, ..., p_n &\Rightarrow \underline{\mathbf{par}}, \mathbf{B_{par}}, \mathbf{v_{par}}, p_1, ..., p_n \\ &\Rightarrow \mathbf{par}, \mathbf{B_{par}}, \mathbf{v_{par}}, \underline{\mathbf{B_{par}}}, p_1, ..., p_n \\ &\Rightarrow \mathbf{par}, \mathbf{B_{par}}, \mathbf{v_{par}}, \underline{\mathbf{prop}}, \mathbf{B_{par}}, \mathbf{v_{par}}, p_1, ..., p_n \\ &\Rightarrow \mathbf{par}, \mathbf{B_{par}}, \mathbf{v_{par}}, \mathbf{prop}, \mathbf{B_{par}}, \mathbf{v_{par}}, \underline{p_1}', \mathbf{v_{par}}, p_2, ..., p_n \\ &\quad ... \end{aligned}$$

5.2 Virus Smith Polimorfi

Possiamo generalizzare i virus Smith come abbiamo già fatto con i blueprint, avendo ora anche a disposizione il codice di *infiniti programmi per le funzioni di propagazione* $\mathcal{B}_i$.

Possiamo per esempio pensare di *aggiungere al virus parassita abilità polimorfe*: ogni virus della generazione i infetta un nuovo programma $\mathbf{q}$ con un virus di generazione $i + 1$, utilizzando una funzione di propagazione di generazione i.

Capitolo 6

Individuazione di Virus

In questo capitolo daremo uno sguardo generale all'individuazione di virus: le tecniche concrete, le limitazioni teoriche, una possibile direzione legata all'offuscamento e all'interpretazione astratta.

6.1 Offuscamento vs. Individuazione

Un **offuscamento** è una trasformazione atta a "confondere" un programma per renderlo difficile da comprendere, ma preservando tutte le sua funzionalità. Le principali applicazioni di questa tecnica sono la *protezione del software* e la *scrittura di virus*.

Gli sviluppatori usano l'offuscamento per *difendere i programmi contro attacchi alla proprietà intellettuale*.

Coloro che scrivono virus offuscano il loro codice per *evitarne l'individuazione*. Molte delle tecniche di individuazione del malware si basano esclusivamente su aspetti sintattici, e quindi sono molto sensibili a (anche piccole) modifiche del codice sorgente di un virus. Per questo l'offuscamento è molto efficace nell'aggirare tali tecniche.

Si nota come questi due aspetti siano intrinsecamente collegati e contrapposti: da un lato si cercano robuste tecniche di offuscamento per il codice "proprietario" e dall'altro si cerca di scrivere software che può individuare più malware offuscato possibile.

Un approccio teorico formale è necessario per progettare e *verificare* l'efficienza di algoritmi di offuscamento e individuazione.

Gli strumenti individuati negli ultimi anni da alcuni ricercatori sono in particolare legati all'interpretazione astratta.

6.2 Individuazione

La crescente complessità dei sistemi di calcolatori e delle loro interconnessioni tramite Internet rende difficile, se non impossibile, evitare i bug nel software. Questo aumenta la possibilità di attacchi da parte di codice maligno, che solitamente sfrutta i bug come vulnerabilità. Le principali tecniche per l'individuazione del malware sono l'*individuazione di anomalie di comportamento*, l'*individuazione basata su segnatura* e l'*individuazione basata su specifica*.

Le tecniche di **individuazione di anomalie di comportamento** si basano su una nozione di "comportamento normale" di un programma, e segnalano la presenza di virus quando individuano un comportamento "anomalo" rispetto a quello che considerano "normale". Il software di individuazione osserva e profila il comportamento dei programmi di un sistema tramite metodi statistici.
Gli *svantaggi* di questo sistema sono molteplici: in particolare l'elevato numero di *falsi positivi*. Infatti un sistema spesso mostra dei comportamenti leciti ma mai visti prima. Un attacco inoltre potrebbe prevedere di indurre il sistema, durante la fase di training, a considerare normale un comportamento virale.
Un ovvio *vantaggio* è dato dalla possibilità di individuare attacchi mai visti prima, senza cioè una precedente conoscenza del codice virale.

L'**individuazione basata su segnatura** classifica un programma $\mathbf{p}$ come infetto quando la segnatura di un virus - cioè una sequenza di istruzioni che lo caratterizza - compare nel programma $\mathbf{p}$. Dunque il software di individuazione monitora il sistema alla ricerca di segnature conosciute, che saranno contenute in un database di segnature virali, da tenere costantemente aggiornato. Se le segnature sono troppo specifiche, piccole varianti di un virus non vengono riconosciute; se sono troppo generiche invece si rischia un alto numero di falsi positivi.
Il principale *svantaggio* di questa tecnica è dato dal fatto che essa non è in grado di riconoscere attacchi nuovi, per i quali non ha nessuna segnatura. Nonostante ciò, è la tecnica più diffusa proprio grazie al basso numero di falsi positivi e alla relativa facilità di uso e basso impiego di risorse.

Una terza tecnica è l'**individuazione basata su specifica**. Si basa sulla presenza di una specifica formale del comportamento "normale" di un programma. Se il programma si comporta in modo diverso dalla specifica, un'anomalia viene segnalata. Il principale *svantaggio* e freno a questa tecnica è il fatto che le specifiche vanno costruite "manualmente" caso per caso.

Ci concentreremo nel seguito sull'individuazione basata su *segnatura*.

Per *evitare l'individuazione della segnatura* dei propri virus, gli autori ricorrono a tecniche per nascondere il codice. In particolare

- *crittazione* del codice maligno e sua *decrittazione* durante l'esecuzione.
- *polimorfismo* (o metamorfismo): ogni successiva generazione di malware modifica la sua sintassi lasciando la semantica inalterata. Non stupisce che l'offuscamento sia una tecnica molto utilizzata, ed è anche supportata dalla teoria: in ogni linguaggio di programmazione c'è qualche ridondanza, ed è quindi possibile applicare il Padding lemma.

6.3 Limitazioni teoriche

I lavori seminali di Cohen e Adleman ([Coh87, Adl88]) e quelli che ne sono seguiti contengono alcuni risultati teorici negativi per quanto riguarda l'individuazione di virus:

- in generale, l'individuazione di virus è indecidibile;
- l'individuazione di evoluzioni a partire da una forma base è indecidibile;
- esistono virus che non possono essere individuati [CW00];

- un problema più semplice, come quello di individuare una mutazione di un virus conosciuto di lunghezza finita, è risultato comunque NP-completo.

Analizziamo alcuni di questi risultati nel nostra formalizzazione.

6.3.1 Individuazione di codice virale

Vogliano tentare di riconoscere un programma $\mathbf{v}$ che sia un virus.

Definizione 3. Data una funzione di propagazione $\mathcal{B}$, chiamiamo $V_{\mathcal{B}}$ l'insieme dei programmi che sono virus rispetto a tale funzione:

$$V_{\mathcal{B}} = \{\mathbf{v} | (\forall \mathbf{p}, x \in \mathbb{D})([\![\mathbf{v}]\!](\mathbf{p}, x) = [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](x))\}$$

Immediatamente vale il seguente.

Teorema 10. *Esistono funzioni $\mathcal{B}$ tali che $V_{\mathcal{B}}$ è Π_2-completo.*

Dimostrazione. Sia q una funzione parziale calcolabile. Sappiamo che l'insieme $T = \{\mathbf{i} | [\![\mathbf{i}]\!] = q\}$ è Π_2-completo. Sia ora $[\![\mathbf{q}]\!] = q$ e sia $\mathcal{B}(y, \mathbf{p}) = spec(\mathbf{q}, \mathbf{p})$. Mostriamo che $V_{\mathcal{B}} = T$.

Sia $\mathbf{v}$ un virus rispetto a $\mathcal{B}$, allora $[\![\mathbf{v}]\!](\mathbf{p}, x) = [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](x) = [\![spec(\mathbf{q}, \mathbf{p})]\!](x) = [\![\mathbf{q}]\!](\mathbf{p}, x) = q(\mathbf{p}, x)$ e dunque $\mathbf{v}$ è un programma per q ($[\![\mathbf{v}]\!] = q$). Di conseguenza, $V_{\mathcal{B}} \subseteq T$.

D'altro canto, consideriamo $\mathbf{i} \in T$. Avremo

$$\begin{aligned} [\![\mathbf{i}]\!](\mathbf{p}, x) &= t(\mathbf{p}, x) \\ &= [\![\mathbf{q}]\!](\mathbf{p}, x) \\ &= [\![spec(\mathbf{q}, \mathbf{p})]\!](x) \\ &= [\![\mathcal{B}(\mathbf{i}, \mathbf{p})]\!](x) \end{aligned}$$

e di conseguenza $T \subseteq V_{\mathcal{B}}$.

Abbiamo quindi che $V_{\mathcal{B}} = T$. □

I prossimi due teoremi mostrano che in alcuni casi l'identificazione è decidibile.

Teorema 11. *Esistono funzioni $\mathcal{B}$ tali che $V_{\mathcal{B}}$ è calcolabile, cioè si può decidere se un programma è un virus o no.*

Dimostrazione. Sia la funzione parziale $f(y, \mathbf{p}, x) = [\![y]\!](\mathbf{p}, x)$. Sia $[\![\mathbf{e}]\!] = f$ e sia $\mathcal{B}(\mathbf{v}, \mathbf{p}) \stackrel{\text{def}}{=} spec(\mathbf{e}, \mathbf{v}, \mathbf{p})$. Allora, per ogni $\mathbf{v} \in \mathbb{D}$ avremo che

$$\begin{aligned} [\![\mathbf{v}]\!](\mathbf{p}, x) &= f(\mathbf{v}, \mathbf{p}, x) \\ &= [\![\mathbf{e}]\!](\mathbf{v}, \mathbf{p}, x) \\ &= [\![spec(\mathbf{e}, \mathbf{v}, \mathbf{p})]\!](x) \\ &= [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](x) \end{aligned}$$

Per cui in questo caso (degenere) $V_{\mathcal{B}} = \mathbb{D}$ ed è dunque calcolabile. □

Teorema 12. *Per ogni insieme calcolabile C che contiene tutti programmi per la funzione ovunque divergente, esiste una funzione di propagazione $\mathcal{B}$ tale che $V_{\mathcal{B}} = C$*

Dimostrazione. Sia $\mathbf{comp}$ un programma per la funzione *calcolabile* $comp(\mathbf{a}, \mathbf{b}, x) = [\![\mathbf{a}]\!]([\![\mathbf{b}]\!](x))$ e quindi $[\![\mathbf{comp}]\!] = comp$. Sia $f(x) \stackrel{\text{def}}{=} x + 1$ e sia $\mathbf{f}$ il programma che la calcola. Sia poi

$$\mathcal{B}(y, \mathbf{p}) \stackrel{\text{def}}{=} \begin{cases} spec(y, \mathbf{p}) & se\ y \in C \\ \mathbf{comp}(\mathbf{f}, spec(y, \mathbf{p})) & se\ y \notin C \end{cases}$$

Se $\mathbf{v} \in C$, allora

$$\begin{array}{rcl} & & [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](x) \\ \text{dato che } \mathbf{v} \in C & = & [\![spec(\mathbf{v}, \mathbf{p})]\!](x) \\ \text{per s-m-n} & = & [\![\mathbf{v}]\!](\mathbf{p}, x) \end{array}$$

e dunque $\mathbf{v}$ è un virus rispetto a $\mathcal{B}$.

Se $\mathbf{v} \notin C$, allora $[\![\mathbf{v}]\!]$ non è sempre divergente, cioè esistono $\mathbf{p}$ e x tali che $[\![\mathbf{v}]\!](\mathbf{p}, x)$ è definita. Avremo

$$\begin{array}{rcl} & & [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](x) \\ \text{dato che } \mathbf{v} \notin C & = & [\![\mathbf{comp}(\mathbf{f}, spec(\mathbf{v}, \mathbf{p}))]\!](x) \\ \text{per s-m-n} & = & [\![\mathbf{v}]\!](\mathbf{p}, x) + 1 \end{array}$$

Poiché $[\![\mathbf{v}]\!](\mathbf{p}, x)$ è definita, $[\![\mathbf{v}]\!](\mathbf{p}, x) \neq ([\![\mathbf{v}]\!](\mathbf{p}, x) + 1 = [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](x))$ e quindi $\mathbf{v}$ non è un virus rispetto a $\mathcal{B}$.

Concludiamo quindi $V_{\mathcal{B}} = C$. □

Si nota però come questi siano chiaramente casi patologici.

6.3.2 Individuazione di programmi infettati

Proviamo a concentrarci ora sull'individuazione dei programmi infettati.

Assunzione 1. *L'attaccante può fare in modo che vengano eseguite forme infette di programmi (cioè nella forma $\mathcal{B}(\mathbf{v}, \mathbf{p})$), ma non il virus in sé.*

Definizione 4. Dato un virus $\mathbf{v}$ rispetto a $\mathcal{B}$, definiamo *insieme di infezioni*

$$I_{\mathbf{v},\mathcal{B}} \stackrel{\text{def}}{=} \{\mathcal{B}(\mathbf{v}, \mathbf{p}) | \mathbf{p} \in \mathbb{D}\}$$

Teorema 13. *Esiste un virus $\mathbf{v}$ rispetto a $\mathcal{B}$ tale che $I_{\mathbf{v},\mathcal{B}}$ è ricorsivamente enumerabile ma non ricorsivo.*

Dimostrazione. Sia $K = \{\mathbf{p} | [\![\mathbf{p}]\!](\mathbf{p}) \downarrow\}$, insieme r.e. per antonomasia. In altre parole, esiste una funzione totale calcolabile f tale che $K = Img(f) = \{f(x) | x \in \mathbb{D}\}$.

Definiamo ora $\mathcal{B}(y, \mathbf{p}) = f(\mathbf{p})$. Notiamo immediatamente che $K = \{\mathcal{B}(y, x) | x \in \mathbb{D}\}$. Ci basta quindi dimostrare che $\mathcal{B}$ è la funzione di propagazione per un qualche virus. Consideriamo $\mathbf{v}$ tale che $[\![\mathbf{v}]\!](\mathbf{p}, x) = [\![f(\mathbf{p})]\!](x)$. Avremo

$$\begin{array}{rcl} [\![\mathbf{v}]\!](\mathbf{p}, x) & = & [\![f(\mathbf{p})]\!](x) \\ & = & [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](x) \end{array}$$

E dunque $I_{\mathbf{v},\mathcal{B}} \stackrel{\text{def}}{=} \{\mathcal{B}(\mathbf{v},x)|x \in \mathbb{D}\} = K$ è ricorsivamente enumerabile ma non ricorsivo, cioè esiste un virus tale per cui non possiamo decidere il suo insieme di infezioni. □

Proviamo ora a rilassare ulteriormente la nozione di *individuazione*, cercando di riconoscere, più in generale, i programmi *che si comportano come* programmi infettati.

Definizione 5. Dato un $\mathbf{v}$ rispetto a $\mathcal{B}$, definiamo l'insieme dei suoi *germi* come

$$G_{\mathbf{v},\mathcal{B}} \stackrel{\text{def}}{=} \{\mathbf{q}|\exists \mathbf{p}(\llbracket \mathbf{q} \rrbracket = \llbracket \mathcal{B}(\mathbf{v},\mathbf{p}) \rrbracket)\}$$

Poiché $G_{\mathbf{v},\mathcal{B}}$ è un insieme *estensionale* (usa $\llbracket \mathbf{q} \rrbracket$) *non banale*, sappiamo che non è decidibile dal Teorema di Rice. Nonostante questo, possiamo trovare una strategia di individuazione.

Definizione 6. Un virus $\mathbf{v}$ è detto **isolabile all'interno del suo insieme di germi** se esiste un insieme decidibile R tale che $I_{\mathbf{v},\mathcal{B}} \subset R \subset G_{\mathbf{v},\mathcal{B}}$

In questa situazione, prima di eseguire un programma $\mathbf{q}$ l'utente controlla se $\mathbf{q} \in R$. Se così è, sa che $\mathbf{q}$ si comporta come virus e va fermato. Se invece $\mathbf{q} \notin R$ allora $\mathbf{q}$ potrebbe comunque appartenere a $G_{\mathbf{v},\mathcal{B}} \setminus R$, ma questo in realtà non è possibile perché $\mathbf{q} \notin I_{\mathbf{v},\mathcal{B}}$, ovvero non è nella forma $\mathcal{B}(\mathbf{v},\mathbf{p})$, che per l'assunzione 1 è l'unica cosa che l'attaccante può far eseguire. Per cui, $\mathbf{q}$ è un programma sano.

Sfortunatamente vale il seguente.

Teorema 14. *Esiste un virus* $\mathbf{v}$ *che non è isolabile all'interno del suo insieme di germi.*

Dimostrazione. Sia di nuovo $\mathbf{v}$ il virus della dimostrazione precedente ($\llbracket \mathbf{v} \rrbracket(\mathbf{p},x) = \llbracket f(\mathbf{p}) \rrbracket(x)$ con f tale che $K = Img(f)$ e quindi $\mathcal{B}(y,\mathbf{p}) = f(\mathbf{p})$).

Ipotesi assurda: $\mathbf{v}$ è isolabile all'interno del suo insieme di germi. Dunque esiste un insieme decidibile R tale che $I_{\mathbf{v},\mathcal{B}} \subset R \subset G_{\mathbf{v},\mathcal{B}}$. Sia $\bar{R}$ il complementare (decidibile) di R.

Sia $\mathbf{r}$ un programma tale che $\llbracket \mathbf{r} \rrbracket$ ha dominio $\bar{R}$, ovvero $\llbracket \mathbf{r} \rrbracket(x) \downarrow$ sse $x \in \bar{R}$. Ricordando che $I_{\mathbf{v},\mathcal{B}} \stackrel{\text{def}}{=} \{\mathcal{B}(\mathbf{v},x)|x \in \mathbb{D}\} = K$, abbiamo che $K \subset R$ e dunque K ed $\bar{R}$ sono *disgiunti*. Si danno due casi.

Se $\llbracket \mathbf{r} \rrbracket(\mathbf{r}) \downarrow$ allora $\mathbf{r} \in K$. Ma sappiamo che $\llbracket \mathbf{r} \rrbracket(\mathbf{r}) \downarrow$ sse $\mathbf{r} \in \bar{R}$. Ma questo è *assurdo* perché K ed $\bar{R}$ sono *disgiunti*.

Se $\llbracket \mathbf{r} \rrbracket(\mathbf{r}) \uparrow$ allora $\mathbf{r} \notin \bar{R}$ e quindi $\mathbf{r} \in R$ e di conseguenza $\mathbf{r} \in G_{\mathbf{v},\mathcal{B}}$, cioè esiste $\mathbf{p}$ tale che

$$\llbracket \mathbf{r} \rrbracket = \llbracket \mathcal{B}(\mathbf{v},\mathbf{p}) \rrbracket$$

Poiché $\mathcal{B}(\mathbf{v},\mathbf{p}) \in K = I_{\mathbf{v},\mathcal{B}}$ avremo che $\llbracket \mathcal{B}(\mathbf{v},\mathbf{p}) \rrbracket(\mathcal{B}(\mathbf{v},\mathbf{p})) \downarrow$. Ma per quanto visto $\llbracket \mathbf{r} \rrbracket(\mathcal{B}(\mathbf{v},\mathbf{p})) \downarrow$ e $\mathcal{B}(\mathbf{v},\mathbf{p}) \in \bar{R}$. Ma questo è *assurdo* perché K ed $\bar{R}$ sono *disgiunti*.

Abbiamo quindi provato che $\mathbf{v}$ non è isolabile. □

6.4 Un possibile approccio formale da [DP07]

Non scoraggiati da queste limitazioni, molti sono stati gli approcci formali all'individuazione di malware, e si basano su semantica, analisi statica, model checking, proram slicing, data mining.

Qui presentiamo l'approccio di Mila Dalla Preda [DP07]. Secondo questo lavoro (e molti altri che ne sono seguiti) un approccio basato sulla semantica e l'interpretazione astratta può essere la chiave per *migliorare gli algoritmi di individuazione basati su segnatura*.

Come già detto, versioni diverse (offuscate) di un virus devono condividere almeno il loro "comportamento malevolo", ovvero avere la stessa *semantica* rispetto a tale comportamento, anche se espresso in forme sintattiche diverse.

L'idea è quella di *modellare il virus e la sua versione offuscata con una semantica* (es. semantica a tracce di esecuzione) e utilizzare l'*interpretazione astratta* per nascondere i dettagli (*astrarre*, appunto) cambiati dall'offuscamento. La semantica di versioni diverse di malware deve essere equivalente rispetto a una qualche astrazione.

Dunque, dato un offuscamento $\mathcal{O}$, va individuata una proprietà semantica astratta $\mathcal{A}$ condivisa sia dal malware $\mathcal{M}$ sia dalla sua versione offuscata $\mathcal{O}(\mathcal{M})$.

Un punto cruciale è dunque la definizione di $\mathcal{A}$. Infatti, se $\mathcal{A}$ è troppo astratta, molti programmi sarebbero (erroneamente) classificati come infetti, mentre se $\mathcal{A}$ è troppo concreta, avremo esattamente l'effetto opposto: il processo sarebbe troppo sensibile all'offuscamento e quindi versioni modificate del malware sfuggirebbero al riconoscimento.

Conclusioni e sviluppi futuri

La rapidità di diffusione e la pervasività dei computer nella vita di tutti i giorni porta inevitabilmente a una rapidissima crescita dei virus, in numero e in complessità. Se lo studio di tecniche pratiche è fondamentale per individuare e risolvere i problemi, un framework teorico è fondamentale per uno studio organico e ad ampio raggio.

La crescente complessità delle tecniche di riconoscimento richiede che, per essere riutilizzate, debbano fondarsi su solide basi teoriche che astraggano dai crescenti dettagli.

Dal lato della definizione, sembra che i teoremi di ricorsione siano lo strumento ideale per modellare molte tipologie di virus. L'utilizzo di teoremi più complessi può portare alla cattura di tipologie sempre più raffinate di virus.

Vale la pena soffermarsi in particolare su due punti che possono essere sviluppati immediatamente a partire da questo lavoro.

1. La definizione delle caratteristiche della funzione Obs (§ 4.2.1) perché essa sia un "buon" offuscamento, magari contro un individuatore basato su interpretazione astratta (§ 6.4). Una promettentissima strada è l'articolo di Giacobazzi e altri: "Obfuscation by partial evaluation of distorted interpreters.", [GJM12], che mostra proprio come costruire un offuscamento resistente ad un'interprete astratto tramite la modifica dell'interprete di un linguaggio.

2. L'utilizzo del modello di computazione delle SRM per modellare, virus a parte, gli altri aspetti (si veda l'appendice A) legati al codice automodificantesi.

Bibliografia

[Adl88] L.M. Adleman. The theory of computer viruses. In *Proceedings Crypto*, volume 88, pages 443–450, 1988.

[BGI+01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Advances in Cryptology—CRYPTO 2001*, pages 1–18. Springer, 2001.

[BKM05] G. Bonfante, M. Kaczmarek, and JY Marion. Toward an abstract computer virology. *Theoretical Aspects of Computing–ICTAC 2005*, pages 579–593, 2005.

[BKM06] G. Bonfante, M. Kaczmarek, and J.Y. Marion. On abstract computer virology from a recursion theoretic perspective. *Journal in computer virology*, 1(3):45–54, 2006.

[BKM07] G. Bonfante, M. Kaczmarek, and J.Y. Marion. A classification of viruses through recursion theorems. *Computation and Logic in the Real World*, pages 73–82, 2007.

[Coh87] Fred Cohen. Computer viruses: theory and experiments. *Computers & security*, 6(1):22–35, 1987.

[CW00] David M Chess and Steve R White. An undetectable computer virus. In *Proceedings of Virus Bulletin Conference*, volume 5, 2000.

[DP07] Mila Dalla Preda. *Code Obfuscation and Malware Detection by Abstract Interpretation*. PhD thesis, PhD thesis, Università degli Studi di Verona Dipartimento di Informatica, 2007. 6, 23, 2007.

[GJM12] Roberto Giacobazzi, Neil D. Jones, and Isabella Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 63–72, New York, NY, USA, 2012. ACM.

[Hof79] Douglas R. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc., New York, NY, USA, 1979.

[Jon97] N.D. Jones. *Computability and complexity: from a programming perspective*, volume 21. MIT press, 1997.

[Mar12] J.Y. Marion. From turing machines to computer viruses. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 370(1971):3319–3339, 2012.

[MKP11] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[Mos06] L Moss. Recursion theorems and self-replication via text register machine programs. *EATCS bulletin*, 13, 2006.

[Smu93] Raymond M Smullyan. *Recursion theory for metamathematics*. Oxford University Press, USA, 1993.

[Szo05] Peter Szor. *The art of computer virus research and defense*. Addison-Wesley Professional, 2005.

Appendice A

Codice automodificantesi

L'auto-modifica del codice è una realtà nella pratica informatica. Molti sono gli utilizzi, in particolare nell'offuscamento. Senza pretendere di entrare nel dettaglio, proviamo a farne un quadro generale, guidati dal lavoro [MKP11], in cui si trovano tutti i dettagli del caso.

La natura del codice automodificantesi risiede nell'architettura di calcolatore usata in pratica. Di solito, un programma in memoria è organizzato in diverse sezioni: codice, dati, heap, stack. Teoricamente solo la parte di codice è dunque eseguibile, ma, di fatto, questa organizzazione è puramente logica e non coincide con l'organizzazione fisica. Il codice inserito in altre sezioni può quindi comunque essere eseguito dal processore[1]. Questo permette ad un programma di salvare nella propria memoria del codice per una futura esecuzione: di fatto si tratta di una automodifica.

A.1 Utilizzi

Gli usi principali del codice automodificantesi sono vari.

Ottimizzazione Una tecnica sempre più presente nei linguaggi di programmazione è quella di *compilazione just-in-time*. Un compilatore just-in-time è parte di un interprete e genera il bytecode del linguaggio macchina oggetto direttamente a runtime.

Debugging e antidebugging Un debugger modifica il codice dell'applicazione che esegue. Tecniche di anti-debugging sfruttano questo meccanismo per impedire il debugging.

Code injection La mancanza di distinzione tra programmi e dati porta ad attacchi di iniezione di codice, in cui il codice maligno viene scritto nella memoria del programma tramite un input, ed è di seguito eseguito.

Sistemi di autoapprendimneto Nella programmazione genetica, sono diffusi sistemi di machine learning che possono modificare il proprio algoritmo di apprendimento.

Offuscamento La nozione di offuscamento è stata formalizzata da Barak e altri in [BGI+01]. Questa nozione soddisfa la proprietà di "virtual black box": un avversario che può accedere ad un programma offuscato non può scoprire più cose di quelle che avrebbe avendo

[1] È bene tuttavia sottolineare come in moderni processori a 64 bit vi è la possibilità di distinguere tra pagine di memoria contenenti programmi e contenenti dati

accesso solo all'input e all'output (black box) del programma non offuscato. Questa nozione è dimostrata essere impossibile.
Nonostante ciò, l'offuscamento è utilizzato in pratica per aumentare il costo computazionale di reverse engineering necessario per il deoffuscamento.
Nel seguito di questa appendice, analizzeremo più nel dettaglio il codice automodificantesi per l'offuscamento.

A.2 Codice automodificantesi per l'offuscamento

L'**avversario** considerato per la realizzazione di un offuscamento può essere di crescente capacità. In particolare, possiamo pensare di avere avversari:

1. **disassemblatori**: ottengono una descrizione in Assembly tramite analisi statica;
2. **debugger limitati**: si basano solo sull'analisi statica e non sono in grado di riconoscere codice sovrascritto o modificato;
3. **debugger ideali**: trattano senza problemi codice che si automodifica;
4. **tool specializzati**: l'avversario ha la possibilità di ottenere tool specificamente progettati per combattere le tecniche di protezione dello specifico offuscatore.

Gli aspetti determinanti da individuare in un offuscatore sono:

- il modo con cui viene **scelta la parte di codice da offuscare**: il programma può essere trattato "*a strati*", ciascuno dei quali viene (potenzialmente) offuscato separatamente o differentemente, oppure "*come una singola entità*";
- la **codifica del codice sorgente**: può essere una semplice *sostituzione*, una *crittazione*, una *compressione*;
- la **visibilità del codice**: può essere *totale* (il codice non è offuscato), *parziale* o *nulla*;
- l'**esposizione del codice**: può essere *totale*, se il codice viene deoffuscato e poi eseguito, ma non può offuscato, o *temporanea*, se il codice è esposto solo temporaneamente, e poi viene rioffuscato.

Nell'articolo [MKP11] sono molti gli esempi di uso di codice automodificantesi per l'offuscamento, dei quali si intende creare una tassonomia rispetto ai fattori elencati in precedenza.

Alcuni significativi esempi di uso dell'auto-modifica:

- offuscare il codice e rendere difficile la comprensione di un software sovrascrivendo le istruzioni originali con istruzioni "banali", avendo metodi per ricostruire il codice originale, quando deve essere eseguito;
- modificare le funzioni sostituendo alcune istruzioni con dati generati random, avendo modo di ripristinarle quando necessarie;
- bypassare tecniche di validazione di codice, riuscendo a scrivere, a partire da un limitato numero di istruzioni macchina, altre istruzioni senza che queste vengano individuate;
- comprimere il codice anteponendogli il decompressore;
- crittografare il codice e decrittarlo su richiesta, di modo che poche istruzioni siano contemporaneamente esposte.

Appendice B

Dimostrazioni e programmi in SRM

B.1 Implementazione di programmi di esempio

Sposta

Il programma $\mathbf{spostaij}$ sposta il contenuto del registro R_i alla fine del registro R_j e cancella il contenuto di R_i. Formalmente

$$\underline{\mathbf{spostaij}}, d_1, ..., d_i, ..., d_j, ..., d_n \Rightarrow \mathbf{spostaij}, d_1, ..., \epsilon, ..., d_j \cdot d_i, ..., d_n$$

Una possibile implementazione per $\mathbf{spostaij}$ è la seguente:

```
1 case i          (*Legge il prossimo carattere del registro i..*)
2 jmp 6           (*... se è vuoto salto a IP=8, cioè Halt)
3 jmp 3           (*... se inizia con 1 salto all'istruzione 6)
4 put j,#         (*... se inizia con # scrivo # nel registro j)
5 jmp -4          (* e poi ritorno all'inizio)
6 put j,1         (*scrivo 1 nel registro j)
7 jmp -6          (*e ritorno all'inizio)
```

Cancella

Il programma $\mathbf{cancellaj}$ elimina il contenuto del registro R_j

$$\underline{\mathbf{cancellaj}}, d_1, ..., d_j, ..., d_n \Rightarrow \mathbf{cancellaj}, d_1, ..., \epsilon, ..., d_n$$

```
1 case j          (*Rimuove il prossimo carattere del registro j..*)
2 jmp 3           (*E' vuoto. Termino*)
3 jmp -2          (*Torno all'inizio*)
4 jmp -3          (*Torno all'inizio*)
```

Shift

Il programma **shift n** sposta a destra i registri $R_1 ... R_n$

$$\underline{\textbf{shift n}}, d_1, ..., d_n \Rightarrow \textbf{shift n}, \epsilon, d_1, ..., d_n$$

```
1 sposta n,n+1
2 sposta n-1,n
3 ...
4 sposta 1,2
```

Backshift

Il programma **backshift n** sposta a sinistra i registri $R_2 ... R_{n+1}$ di una posizione, cancellando il contenuto di R_1

$$\underline{\textbf{backshift n}}, d_1, ..., d_{n+1} \Rightarrow \textbf{backshift n}, d_2, ..., d_{n+1}, \epsilon$$

```
1 sposta 2,1
2 sposta 3,2
3 ...
4 sposta n+1,n
```

Shift indicizzato

Il programma **shift n, i** sposta a destra di una posizione n registri a partire da R_i.

$$\underline{\textbf{shift n, i}}, d_1, ..., d_i ..., d_{n+i} \Rightarrow \textbf{shift n, i}, d_1, ..., \epsilon, d_i, ..., d_{n+i}$$

```
1 sposta n+1,n+2
2 sposta n,n+1
3 sposta n-1,n
4 ...
5 sposta i,i+1
```

Copia

Il programma **copia i j** cancella il contenuto di R_j e poi copia il contenuto di R_i anche in R_j.

$$\underline{\textbf{copia i j}}, d_1, ..., d_i, ..., d_j, ..., d_n \Rightarrow \textbf{copia i j}, d_1, ..., d_i, ..., d_i, ..., d_n$$

```
1 erase j
2 case i
3 jmp +9       (*se Ri è vuoto termina*)
4 jmp +4       (*se in Ri c'è i *)
5 put j,#      (*se in Ri c'è #, lo scrivo in j...*)
```

```
 6 punt n+1,#   (*... e in Rn+1*)
 7 jmp -5       (*ritorno al case*)
 8 put j,1      (*scrivo 1 in Rj*)
 9 put n+1,1    (*scrivo 1 in Rn+1*)
10 jmp -8       (*ritorno al case*)
11 move n+1,i
```

Scambia

Il programma **scambia i j** scambia il contenuto di R_j con quello di R_i.

$$\underline{\mathbf{scambia\,i\,j}}, d_1, ..., d_i, ..., d_j, ..., d_n \Rightarrow \mathbf{scambia\,i\,j}, d_1, ..., d_j, ..., d_i, ..., d_n$$

```
1 sposta i,n+1
2 sposta j,i
3 sposta n+1,j
```

Concatena

Il programma **concatena i j** copia (concatena) il contenuto del registro R_i alla fine del registro R_j. Formalmente

$$\underline{\mathbf{concatena\,i\,j}}, d_1, ..., d_i, ..., d_j, ..., d_n \Rightarrow \mathbf{concatena\,i\,j}, d_1, ..., d_i, ..., d_j \cdot d_i, ..., d_n$$

```
1 copia i,n+1
2 sposta i,j
3 sposta n+1,i
```

B.2 Interi e liste

Avendo a disposizione un alfabeto binario, è facile codificare nelle SRM interi e coppie, e quindi le liste.

Possiamo quindi fare uso di programmi per la costruzione e la distruzione di liste.

$$\begin{aligned}
\underline{\mathbf{cons}}, el_1, l_1, d &\Rightarrow \mathbf{cons}, \langle el_1, l_1 \rangle, d \\
\\
\underline{\mathbf{head}}, \langle el_1, l_1 \rangle, d &\Rightarrow \mathbf{head}, el_1, l_1, d \\
\underline{\mathbf{tail}}, \langle el_1, l_1 \rangle, d &\Rightarrow \mathbf{tail}, l_1, d \\
\\
\underline{\mathbf{empty}}, \epsilon, d &\Rightarrow \mathbf{empty}, 1, \epsilon, d \\
\underline{\mathbf{empty}}, \langle el_1, l_1 \rangle, d &\Rightarrow \mathbf{empty}, \#, \langle el_1, l_1 \rangle, d
\end{aligned}$$