Università degli Studi di Verona

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Informatica

Tesi di laurea triennale

Compilatore di Taint Analysis

DAL FUNZIONAMENTO AL FALLIMENTO DINAMICO

Candidato:
Giovanni Bellorio
Matricola VR386665

Relatore:
Prof. Roberto Giacobazzi

Anno Accademico 2016-2017

Indice

Capitolo 1

Descrizione dei concetti

La motivazione per cui quest'area di ricerca esiste, è la seguente domanda: per i pacchetti di reti o i files pdf *"è possibile misurare il livello di influenza che hanno i dati esterni su qualche applicazione?"*.

1.1 Information Flow

Analizzando qualsiasi applicazione usando un debugger si può notare che l'informazione contenuta nei dati viene copiata e modificata molte volte. In altre parole, *l'informazione si muove sempre.*

L'analisi di taint può essere vista come una forma di analisi sul flusso delle informazioni dei dati.

Una definizione rigorosa è stata data da Dorothy Denning nella sua pubblicazione *"Certificazione di programmi per una sicurezza sul flusso di informazioni"*: *"Il* ***flusso*** *di informazioni da un oggetto X a un oggetto Y, denotato come* $X \rightarrow Y$*, ogni volta che viene salvato in X, viene trasferito anche in Y"*.

*"Un'operazione o una serie di operazioni che utilizzano il valore di un oggetto, per esempio X, per derivare un risultato da un altro oggetto, per esempio Y, genera un **flusso** da X a Y".*

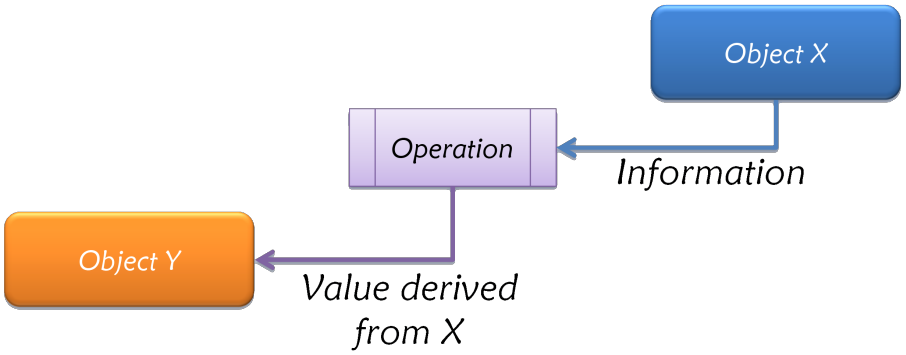

Figura 1.1: Flow

1.2 Oggetti Tainted

Se la **sorgente** del valore dell'oggetto X **non è affidabile**, diciamo che X è **tainted** (sporca). Il dato sarà **untrusted** (non veritiero).

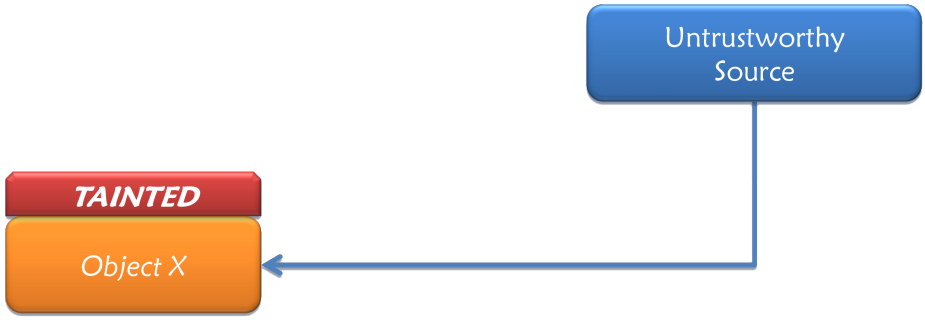

Figura 1.2: Oggetti Tainted

"*Tinteggiare*" un dato-utente significa inserire un qualche tipo di **etichetta** (taint/untaint) per ogni oggetto del dato-utente.

L'etichetta ci permette di analizzare l'influenza che l'oggetto, taint/untaint, ha sull'esecuzione del programma.

Se per esempio abbiamo un'operazione che usa il valore di un oggetto **taint** (sporco), denominato X, per derivare il valore di un altro oggetto, denominato Y, allora possiamo concludere che l'oggetto denominato Y sarà **taint**.

- l'operatore di taint è definito come segue: $\mathbf{X \rightarrow t(Y)}$.
- l'operatore di taint è transitivo: X → (Y) e Y → t(Z) allora x → t(Z).

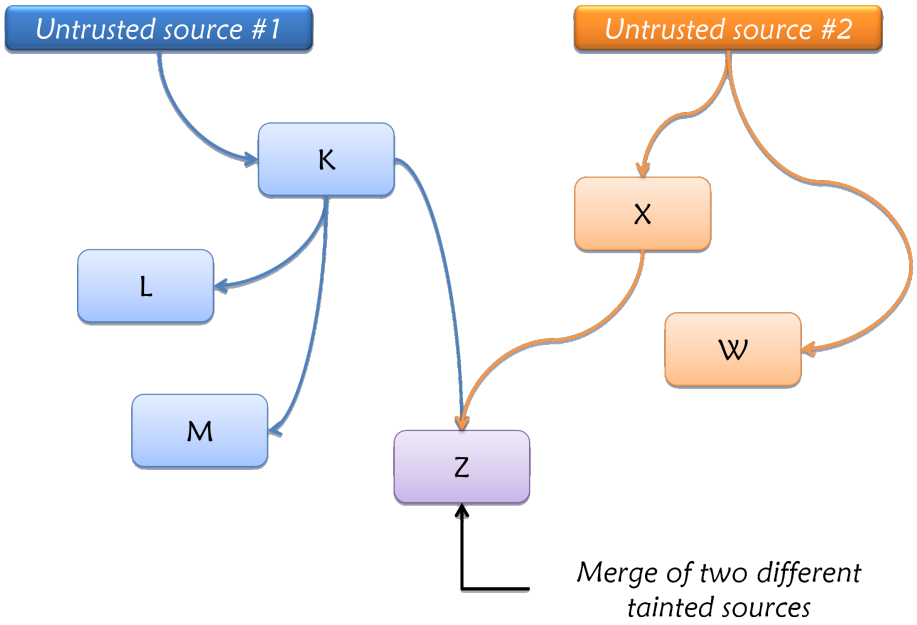

Figura 1.3: Propagazione Taint

1.3 Applicazioni

Possiamo applicare questa analisi nei seguenti campi:

- Rilevamento di exploit(sfruttamenti):
  - se possiamo rilevare il dato-utente, possiamo rilevare se un dato non affidabile raggiunge una locazione privilegiata.
  - sql/code injection per applicazioni web, buffer overflow.
  - perl tainted mode.
  - rilevamento di attacchi sconosciuti.

- Prima dell'esecuzione di qualunque commando(statement), il modulo di taint analysis cerca se il commando o lo statement è taint o meno. Se è taint avvisa che siamo in presenza di un possibile attacco.

- Analisi del ciclo di vita dei dati:

  – Jin Chow - *"Capire il ciclo di vita di un dato attraverso l'emulazione di tutto il sistema"*.

  – Creazione di un emulatore di Boch modificato (taintBochs) per tinteggiare dati sensibili.

  – Tenere traccia del ciclo di vita dei dati sensibili (password, pin, numero carta di credito) salvati nella memoria di una macchina virtuale.

  – Rilvamento di dati anche nella modalità kernel.

  – Conclusione che afferma che molte applicazioni non hanno una misura per minimizzare il ciclo di vita dei dati sensibili nella memoria.

Capitolo 2

Taint Analysis

2.1 Oggetti Taint

Nell'architettura x86 abbiamo due possibili oggetti da tinteggiare (taint/untaint):

- Locazioni di memoria.
- Registri del processore.

Per gli oggetti di memoria:

- Bisogna tenere traccia dell'indirizzo iniziale dell'area di memoria.
- Bisogna tenere traccia della dimensione dell'area di memoria.

Mentre, per i registri:

- Bisogna tenere traccia dell'identificatore del registro.
- Bisogna tenere traccia di ciascun bit.

Gli oggetti taint/untaint rappresentati tengono traccia di ciascun **bit**. Alcuni tool usano un meccanismo che tiene la mantiene a livello di **byte** (Valgrind TaintChecker).

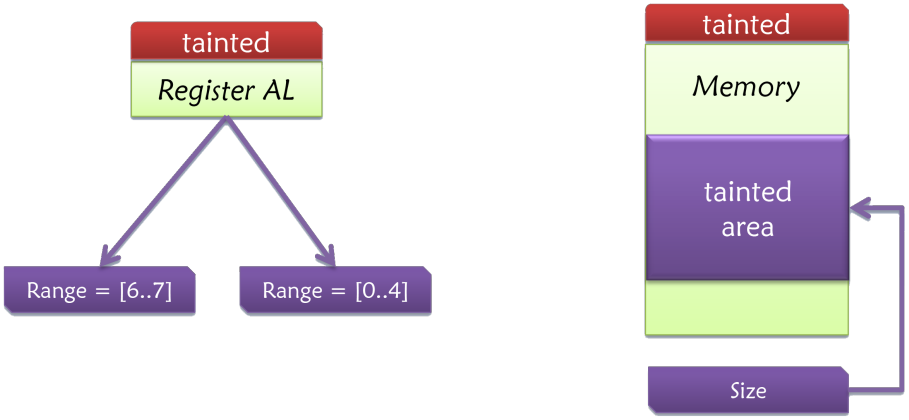

Figura 2.1: Oggetti Taint

2.2 Analisi delle istruzioni

La ISA (Instruction Set Architecture) di qualunque piattaforma può essere divisa in molte categorie:

- Assegnamento di istruzioni (load/store → mov).
- Istruzioni booleane.
- Istruzioni aritmetiche, su stringhe e di branch (condizioni).

2.2.1 Assegnamenti

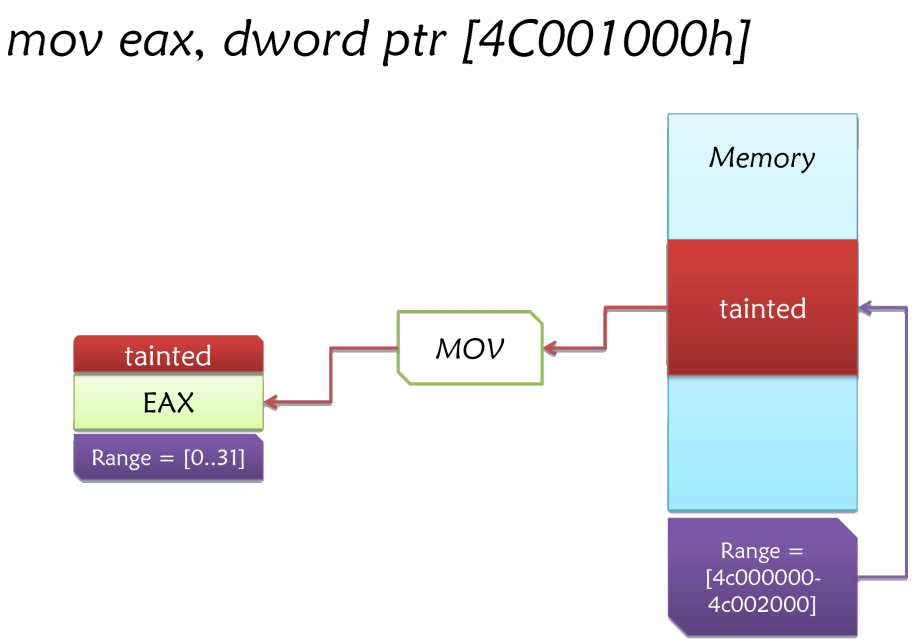

Figura 2.2: Assegnamento

2.2.2 Operatori Booleani

Possiamo applicare l'analisi di taint sui booleani (AND, OR, XOR) come detto precedentemente. L'analisi controlla la dipendenza dei risultati rispetto agli input che possono essere taint oppure untaint. Inoltre vengono studiati e implementati i casi in cui l'operazione dipende dal contenuto dell'oggetto taint o untaint nella fattispecie quando il valore è uguale a zero.
Ci si basa sulle seguendi tabelle di verità:

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

Figura 2.3: AND

Se l'oggetto A è taint:

- e l'oggetto B è uguale a 0, allora il risultato è sempre **UNTAINT** perchè non dipende dal valore di A.
- e l'oggetto B è uguale a 1, allora il risultato è sempre **TAINT** perchè l'oggeto A influenza l'operazione.

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

Figura 2.4: OR

Se l'oggetto A è taint:

- e l'oggetto B è uguale a 1, allora il risultato è sempre **untaint** perchè non dipende dal valore di A.
- e l'oggetto B è uguale a 0, allora il risultato è sempre **taint** perchè l'oggeto A influenza l'operazione.

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Figura 2.5: XOR

Se l'oggetto A è taint allora il risultato è sempre **taint** indipendentemente dal valore di B.

Esempi:

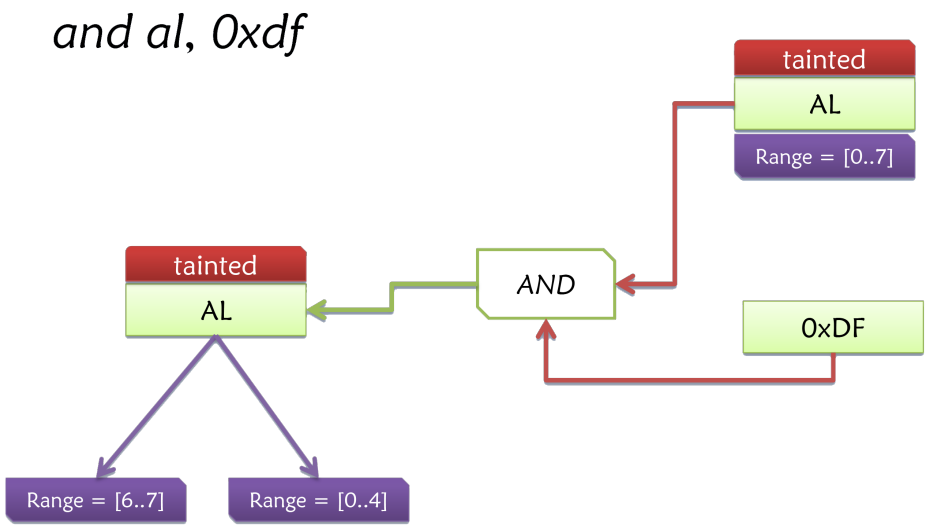

Figura 2.6: Esempio AND

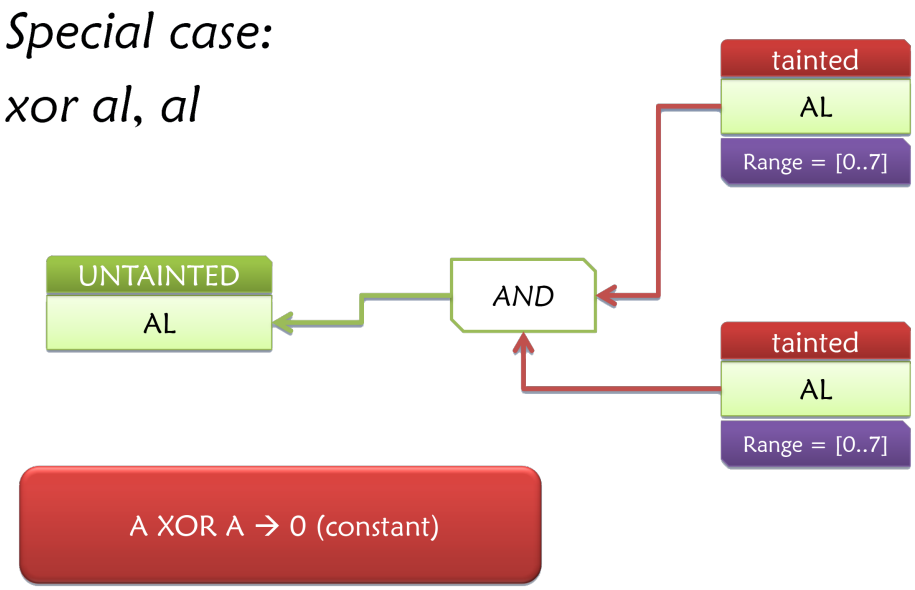

Figura 2.7: Caso Speciale XOR

2.2.3 Operatori Aritmetici

Le principali operazioni aritmetiche che andremo ad analizzare sono l'addizione, la sottrazione, la moltiplicazione e la divisione. Tutte le operazioni possono essere espresse sottoforma di operatori booleani. Per esempio, l'addizione, è espressa dall'operatore AND. In generale se uno degli operandi di una operazione aritmetica risulta essere tainted, il risultato sarà tainted. I casi speciali da controllare sono l'addizione e la sottrazione di un operando con valore 0 e la moltiplicazione per 0 e per 1. Se tali valori sono tainted anche il risultato sarà tainted altrimenti se il primo operando è untainted e il risultato finale non dipende da tale, sarà untainted.

2.2.4 Operatori su Stringhe

Le stringhe sono viste come sequenze di caratteri. In generale l'analisi sulle stringhe restituisce sempre un valore tainted se operiamo su stringhe taint o su almeno una stringa taint nel caso di composizione. Il caso speciale in questo caso è solo per concatenazione con una stringa vuota. Per un'operazione di *substring* dobbiamo controllare gli indici ed eventualmente l'operazione per ottenerli.

Capitolo 3

Descrizione dell'interprete

3.1 Analisi Progettistica

Estendere l'interprete OCAML scegliendo una delle tre semantiche studiate: operazionale, denotazionale o iterativa. La soluzione dovrá contenere il codice base imperativo e funzionale di blocchi, procedure e funzioni (escludendo l'implementazione degli oggetti).

L'estensione del linguaggio didattico dovrá prevedere:

- Il tipo stringa di caratteri: *string*
- La costante string nella semantica come sequenza di caratteri alfanumerici
- Il calcolo della lunghezza di una stringa: *len x*
- La concatenazione di stringhe: *s1@s2*
- L'operazione di sottostringa: *subs(string, index1, index2)*

La possibilitá di introdurre ulteriori estensioni/operazioni prendendo ispirazione da operazioni note in linguaggi di scripting é lasciata al programmatore.

L'estensione prevede anche l'implementazione del comando **Reflect "String"**, il quale riceve in input, come unico parametro, una stringa e richiama l'interprete sulla stessa, la quale viene interpretata come sequenza di comandi eseguibili.

3.2 Rappresentazione

Per scrivere l'interprete é stata scelta la semantica denotazionale. Per comoditá è stato realizzato uno script che carica i vari file contenenti il codice dell'interprete ml.

```
1 #use ‘‘main.ml’’;;
```

Codice 3.1: Avvio programma, "main.ml"

3.2.1 Linguaggio Imperativo

É stata adottata per una prima stesura del programma un linguaggio funzionale. La scelta ha permesso di costruire una prima versione dell'elaborato finale in modo agevole, in quanto il linguaggio funzionale permette una piú semplice e facile individuazione degli errori. Lo sviluppo del software é continuato usando un linguaggio imperativo e, riuscendo a trasporre il software funzionale prodotto.

```
1  #use "interprete/sintassi.ml";;
2
3  #use "env/env_Interfaccia.ml";;
4  #use "env/env_Semantica.ml";;
5  open Funenv;;
6
7  #use "store/store_Interfaccia.ml";;
8  #use "store/store_Semantica.ml";;
9  open Funstore;;
10
11 #use "stack/stack_Interfaccia.ml";;
12 #use "stack/stack_Semantica.ml";;
13 open SemPila;;
14
15 #use "stack/stackM_Interfaccia.ml";;
16 #use "stack/stackM_Semantica.ml";;
17 open SemMPila;;
18
19 #use "interprete/eval.ml";;
20 #use "interprete/operazioni.ml";;
21 #use "interprete/semantica.ml";;
```

Codice 3.2: Struttura Main, "main.ml"

3.2.2 Struttura Files Software

- **sintassi.ml:** contiene i domini sintattici e definisce la sintassi di tutte le operazioni e i comandi definiti.
- **env_Interfaccia.ml:** contiene la signature dell'ambiente.
- **env_Semantica.ml:** contiene l'implementazione dell'interfaccia dell'ambiente, definita in *env_interface.ml*.
- **store_Interfaccia.ml:** contiene la signature della memoria.
- **store_Semantica.ml:** contiene l'implementazione dell'interfaccia della memoria, definita in *store_interface.ml*.
- **stack_Interfaccia.ml:** contiene la signature dello stack **non modificabile**.
- **stack_Semantica.ml:** contiene l'implementazione dell'interfaccia dello stack **non modificabile**, definita in *stack_interface.ml*.
- **stackM_Interfaccia.ml:** contiene la signature dello stack **modificabile**.
- **stackM_Semantica.ml:** contiene l'implementazione dell'interfaccia dello stack **modificabile**, definita in *stackm_interface.ml*.
- **eval.ml:** contiene i domini semantici (esprimibili *eval*, dichiarabili *dval*, memorizzabili *mval*) e le conversioni di tipo.
- **operazioni.ml:** contiene l'operazione *typecheck*, usato per verificare il tipo dei parametri in ingresso ad ogni operazione, e l'implementazione tutte le altre operazioni divise per tipologia (Operazioni *base*, Operazioni sulle *stringhe*, Operazioni di supporto private e non disponibili all'esecuzione, Operazione *parser* e *parserCom* di supporto al comando *Reflect*).
- **semantica.ml:** contiene la definizione della semantica di ogni parte sintattica necessaria all'interprete.

3.3 Operazioni

Per soddisfare la richiesta di inserimento delle stringhe è stato aggiunto il tipo stringa al linguaggio, nella sintassi (type exp) e anche nella semantica (type eval esprimibile, type dval denotabile e type mval memorizzabile).

```
(* TYPE EXPRESSABLE *)
type exp =
  (* CONSTANTS *)
  | Eint of int
  | Ebool of bool
  | Estring of string
```

Codice 3.3: Espressioni, "sintassi.ml"

```
(* TYPE EVAL: EXPRESSIBLE VALUES *)
type eval =
  | Int of int
  | Bool of bool
  | Funval of efun
  | String of string
  | Novalue
```

Codice 3.4: Valori Esprimibili, "eval.ml"

Le operazioni sulle espressioni sono state implementate nel file Operazioni.ml il quale contiene anche il check di tipo e dichiarate nella sintassi con il loro numero di parametri. Nel linguaggio ci sono principalmente 3 tipi di dato: Int, Bool e ora anche String. Per eseguire delle operazioni dobbiamo essere sicuri prima di tutto che gli operandi siano del tipo giusto altrimenti è impossibile eseguire qualcosa. Esempio: una somma di stringhe non può esistere e non può essere computata.

```
(* SUPPORT FUNCTION - TYPECHECK *)
let typecheck (x, y) = match x with
  | "int" -> (match y with
    | Int(u) -> true
    | _ -> false)
  | "bool" -> (match y with
    | Bool(u) -> true
    | _ -> false)
  | "string" -> (match y with
    | String(u) -> true
    | _ -> false)
  | _ -> failwith ("not␣a␣valid␣type")
```

Codice 3.5: Check ti tipo, "operazioni.ml"

L'operazione Len restituisce un intero con la lunghezza della stringa passata come espressione. Per implementare tale operazione sono state utilizzate le librerie delle stringhe contenute in Ocaml (use string.ml). In primo luogo è stato inserito il check di tipo e convertito la funzione len della libreria nel tipo String voluto dal linguaggio.

```
1 (* COMPUTE THE LENGTH OF THE STRING X *)
2 let len x =
3   if typecheck("string",x) then
4     (match x with | String(x) -> Int(String.length x)
5                   | _ -> failwith ("len␣match␣error"))
6   else
7     failwith ("len␣type␣error")
```

Codice 3.6: Lunghezza stringa, "operazioni.ml"

Stessa implementazione anche per l'operazione concat. Date due stringhe eseguire la concatenazione e restituire un'unica stringa. In libreria l'operazione concat è particolare perché é estesa a concatenare una lista di stringhe. Il trucco utilizzato é la trasformazione della stringa da concatenare in lista, unendola a una lista di stringhe vuote. Quindi verrà eseguita la concatenazione tra una stringa e una lista che effettivamente conterrà solo una stringa valida non nulla.

```
1 (* CONCATENATE STRING X TO STRING Y *)
2 let concat (x,y) =
3   if typecheck("string",x) && typecheck("string",y) then
4     (match (x,y) with | (String(u), String(w))
5                       -> String(String.concat (u) [""";w])
6                       | _ -> failwith ("concat␣match␣error"))
7   else
8     failwith ("concat␣type␣error")
```

Codice 3.7: Concatenazione stringhe, "operazioni.ml"

L'operazione substring richiede vari controlli in seguito al check ti tipo. Si controlla che la lunghezza della stringa sia maggiore di 0 chiamando la funzione compare che controlla se un operando intero (lunghezza di s) è maggiore dell'altro (0). Dopodiché i due indici inseriti devono essere entrambi maggiori di 0, il primo minore del secondo ed entrambi minori della lunghezza della stringa.

Le specifiche richiedono che venga estratta la stringa dalla posizione i alla posizione j, mentre la substring della libreria lavora in modo diverso estraendo la stringa dalla posizione i per j caratteri. Quindi passiamo a tale funzione la posizione i come indici di start e j lo calcoliamo come differenza tra i due indici più 1. La substring restituisce la sotto stringa richiesta se tutte le condizioni sono soddisfatte.

Tutte queste operazioni sono state inserite nella semantica del linguaggio e tali possono essere richiamate dinamicamente passando il numero corretto di parametri già aventi un tipo eval (non più exp) della semantica.

```
1 (* SEMANTICS: LINK EVERY FUNCTION TAG WITH THE CORRECT OPERATION *)
2 let rec sem (e:exp) (r:dval env) (s:mval store) = match e with
3   | Estring(c) -> String(c)
4   | Len(c) -> len(sem c r s)
5   | Concat(c1,c2) -> concat(sem c1 r s, sem c2 r s)
6   | Substr(c,a,b) -> substr(sem c r s, sem a r s, sem b r s)
```

Codice 3.8: Semantica, "semantica.ml"

3.4 Comando Reflect

Tale comando presa in ingresso una stringa la converte in vere e proprie espressioni richiamando su di essa l'interprete e quindi la semantica. Tale comando è stato implementato nella semantica dei comandi del nostro linguaggio. La semantica dei comandi restituisce una memoria (store) definita precedentemente. Per visualizzare il risultato dobbiamo allocare e dichiarare una variabile di output del parse nella memoria e passare alla semantica dei comandi l'ambiente e la memoria su cui stiamo lavorando. La variabile da inizializzare e sulla quale verrà memorizzato il risultato è chiamata di default "result". Quello che dobbiamo implementare in sostanza è un vero e proprio parse, da stringa a exp.

3.4.1 Parser delle espressioni

Il parse ha bisogno di tre parametri: l'espressione (la stringa), uno stack dove memorizzare i risultati parziali (di tipo exp) e uno stack contenente l'espressione rimanente da convertire (di tipo eval). La funzione parse sfrutta la ricorsione dinamica. La soluzione implementata funziona interpretando la stringa in modo sequenziale. Sulla stringa non c'è nessun controllo iniziale di

formattazione e di contenuto. Il processo riconosce i vari comandi interni grazie alla substring che controlla i caratteri di tale stringa con le vere e proprie espressioni. Una volta riconosciute cerca i suoi operandi (ogni operazione ha un numero di operandi diverso) che possono essere ancora operazioni che restituiscono un risultato base. Per fronteggiare a tale operazione il controllo sul numero di virgole all'interno dell'operazione è limitato. Occorre utilizzare lo stack. Si effettua la push del parse della stringa rimanente fino ad arrivare a un passo base, cioè fino a quando non si trova un intero, stringa o booleano. La conversione avviene digitando l'operazione sotto forma di exp. Per ottenere il contenuto dello stack si utilizza la funzione topandpop che automaticamente esegue il top e il pop dello stack.

I passi base sono implementati in modo tale da concludere parzialmente l'operazione salvando sullo stack il risultato base estratto controllando la posizione della "," e salvando sullo stack ausiliario la stinga rimanente da scansionare. Nel caso di interi e booleani si utilizzano anche le funzioni di libreria per effettuare la conversione da stringa a intero o da stringa a booleano.

Analizzando passo a passo il problema è necessario inserire alcune condizioni per far continuare l'operazione: nel caso in cui si trovi una parentesi tonda ")" di troppo dovuta alle varie operazioni nidificate, alle "," che dividono gli operandi di una operazione.

```
1  (* "," character is ignored *)
2  else if equals(substr(String(n),Int(0),Int(0)),String(",")) then
3    parseric(String(String.sub (n) 1 (((String.length) n)-1)),stack,
4    _stackstr)
5  (* ")" character is ignored *)
6  else if equals(substr(String(n),Int(0),Int(0)),String(")")) &&
7    _(String.length(n)!=1) then
8    parseric(String(String.sub (n) 1 (((String.length) n)-1)),stack,
9    _stackstr)
```

Codice 3.9: Controlli, "operazioni.ml"

3.4.2 Parser dei comandi

La funzione parserCom utilizza gli stessi tre paramteri della funzione parser, ma viene chiamata ogni volta che viene riconosciuto che il primo token della stringa che è un comando.

Capitolo 4

Compilatore Taint

4.1 Il funzionamento

Sviluppo di una taint analysis con codice ML: determinare i flussi tainted durante la computazione di un programma P. Partizionare la memoria e l'ambiente in input (di avvio del programma P) tra dati untainted e tainted. Durante il calcolo la computazione di una operazione che compone dati tainted con dati untainted da come risultato un dato untainted se il risultato è indipendente dall'input tainted. Viceversa se l'output della operazione dipende da un dato tainted il risultato è tainted. Se una stringa s risulta tainted bloccare la computazione in quanto si ha un potenziale code injection attack. L'analisi deve fornire una memoria ed un ambiente in output con valori etichettati tainted o untainted e deve controllare dinamicamente l'esecuzione in modo da evitare code injection. Implementare successivamente un compilatore che faccia fallire l'analisi cioè trasformare il programma in uno equivalente che non attribuisce signifato alle variabili taint. Gestire i seguenti casi:

- flusso diretto, quando trovo un valore taint dentro una variabile untaint genero un'eccezione.
- flusso indiretto, posso copiare un valore senza mai assegnarlo (attraverso un ciclo while) e quindi scambiare valori taint con untaint.

Lo sviluppo in codice di quanto detto è stato realizzato su ogni espressione comando supportato dall'interprete creato per il nostro progetto. Si divide la parte dei registri e della memoria in due parti: una taint (con dati untrusted) e l'altra untaint (con dati trusted). Una volta fatta partire l'analisi vengono valutati tutti i valori e le relative etichette dei dati fino alla terminazione del

programma. Se l'analisi, che non ha ancora effetuutato nessuna operazione, restituisce un dati trusted e quindi untaint, viene eseguita l'operazione, altrimenti il programma viene bloccato in quanto potremmo avere un possibile "code injection".

La sematica utilizzata è identita a quella spiegata nel capitolo 1 per valori interi, booleani e di tipo stringa. Vediamo un esempio:

```
1 (* DATI INPUT *)
2 let ax = Ebool true ;;
3 let bx = Ebool false;;
4 let cx = Ebool true ;;
5 let dx = Ebool false;;
```

Codice 4.1: Dati, "ta_bool.ml"

Questi sono i nostri dati in ingresso al nostro programma analizzatore. Tali verranno salvati nello store e nell'ambiente di destinazione etichettati con valori taint o untaint a seconda del registro. I registri AX, BX, Ah, Al, Bh, Bl sono restituiscono valori di tipo taint mentre i rimanenti e l'area di memoria libera sono untaint. Il risultato dell'analisi viene salvato in variabile chiamata "result_taint" che andremo a valutare prima di mandare in esecuzione l'eventuale operazione.

```
1 (* INPUT *)
2 let s = And(Val(Den "ax"),Val(Den "dx"));;
3
4 (* START *)
5 let (rho,sigma) = taint(s,rho5,sigma5);;
```

Codice 4.2: Esecuzione, "ta_bool.ml"

In questa porzione di codice viene presa in ingresso la vera e propria operazione salvata nella variabile s e successivamente viene avviata l'operazione di analisi su di essa. Vengono preparate due variabili d'uscita per l'ambiente e la memoria restituita dall'analisi. Viene eseguita l'operazione di And tra un registro taint True e uno untaint False. Il risultato dipende in questo caso dal registro untaint e quindi l'operazione è untaint. I risulati vengono stampati a video come segue:

```
1 (* RESULT - OUTPUT *)
2 sem (Estring "OUTPUT") rho sigma;;
3
4 sem (Estring "AX␣-->␣tainted") rho sigma;;
5 sem (Val(Den "ax")) rho sigma;;
6
```

```
7 sem (Estring "BX␣-->␣tainted") rho sigma;;
8 sem (Val(Den "bx")) rho sigma;;
9
10 sem (Estring "CX␣-->␣untainted") rho sigma;;
11 sem (Val(Den "cx")) rho sigma;;
12
13 sem (Estring "DX␣-->␣untainted") rho sigma;;
14 sem (Val(Den "dx")) rho sigma;;
15
16 s;;
17 sem (Val(Den "result_taint")) rho sigma;;
```

Codice 4.3: Output, "ta_bool.ml"

L'esecuzione dell'analisi restituisce un valore untaint e quindi possiamo eseguire l'operazione And chiamando la semantica sulla espressione e stampare a video il risultato senza problemi. Il controllo viene eseguito sul valore contenuto nella variabile "result_taint" come segue:

```
1 if (equals(sem (Val(Den "result_taint")) rho sigma,
2        _String("untainted"))) then sem s rho sigma
3        else failwith ("Code␣Injection");;
```

Codice 4.4: Output, "ta_bool.ml"

L'analisi può essere richiamata su espressioni (come in questo esempio), ma anche su comandi o liste sempre nella stessa maniera.

4.2 Il fallimento

Come è possibile portare al fallimento una taint analysis? Tutto è possibile. Possiamo scambiare il contentuo delle variabili in due modi: copiando il valore da una all'altra oppure cicliamo i loro contenuti. Nel primo caso (la copia) non raggiungiamo una soluzione perchè la copia si trascina dietro l'etichetta e il valore relativi alla variabile. Mentre la tecnica del ciclo sui valori no. Eseguendo dei semplici cicli "while" riusciamo a scambiare i valori interni e non le etichette associate. Per esempio se un intero 10 è salvato in A, per portarlo in B occorre eseguire 10 cicli applicazdo -1 ad A e +1 a B. In 10 passi abbiamo il valore copiato in una variabile ovviamente untaint sulla quale eseguire le operazioni e bypassare ogni tipo di controllo.

É stato realizzato un compilatore che prende in ingresso un programma (operazioni o comandi), non lo modifica ma esegue cicli e operazioni addizione e sottrazioni per copiare i valori delle variabili taint in variabili untaint. per

interi, booleani e stringhe. Le nuove variabili (untaint) vengono inizializzate come segue (non possiamo conoscere a priori il tipo delle tariabili taint):

```
(* TYPECHECK PER TIPO VARIABILE DI SUPPORTO - AAX *)
let init_aax =
  if typecheck_fail("Eint",ax) then
        let aax = Eint 0 (* untaint *)
        in aax
  else if typecheck_fail("Ebool",ax) then
        let aax = Ebool true (* untaint *)
        in aax
  else if typecheck_fail("Estring",ax) then
        let aax = Estring "" (* untaint *)
        in aax
  else
        failwith ("not␣a␣valid␣type")
;;

(* INIT RX - UNTAINTED - AAX - BBX *)
let aax = init_aax;;
```

Codice 4.5: Init, "ta_fail.ml"

Una volta inizializzate con valori di default opportuni e tipo corretto in base alle variabili originali, viene avviata la fase di copia con ciclo “while”:

```
(* AVVIO COPIA DELLA VARIABILE - AAX *)
let while_ax = init_while_aax;;
let sigma = semc while_ax rho sigma;;
```

Codice 4.6: Copy, "ta_fail.ml"

Il gioco è quasi finito. Ora il compilatore dovrà caricare le cariabili di supporto untaint ogni qualvolta si faccia riferimento alle varibili taint. I risultati saranno sempre untaint e la semantica restituirà sempre un risultato. Questo implica ogni bypass sui controlli e possibili attacchi non più controllabili.

Il fallimento dell’analisi può essere eseguito su espressioni oppure su comandi sia con registri a 16 bit, sia a 8 bit.

Bibliografia

[1] Yaron Minsky, *Real world Ocaml*, Functional programming for the masses.

[2] Communication of the ACM, *Certification of programs for secure information flow*, Dorothy E. Denning, Peter J. Denning. 1977.

[3] Communication of the ACM, *A lattice model for secure information flow*, Dorothy E. Denning 1976.

[4] Georgia Institute of Technology, *Dytan: A generic dynamic taint analysis framework*, James Clause, Wanchun Li, Alessandro Orso.

[5] Dawn Song, *BitBlaze: A New Approach to Computer Security via Binary Analysis*.

[6] Scott A. Crosby, *Denial of Service via Algorithmic Complexity Attacks*.