UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA TRIENNALE IN INFORMATICA

ANALISI E VERIFICA DI UN FRAMEWORK PER L'ANALISI DINAMICA DI CODICE BINARIO

Relatore: Dott. Mattia MONGA
Correlatore: Dott. Roberto PALEARI

Tesi di Laurea di:
Luca GIANCANE
Matricola 675309

Anno Accademico 2006–07

Ai miei genitori

Ringraziamenti

Innanzitutto ringrazio il Dott. Mattia Monga, per avermi dato l'opportunità di svolgere questo lavoro di tesi e il Dott. Roberto Paleari, per il grande aiuto nel portarla avanti e per la costante disponibilità (ISPELL!!!).

Un ringraziamento speciale ai miei genitori, senza i quali non sarebbe stato possibile tutto questo. Grazie soprattutto per aver creduto in me, per il sostegno e per avermi sopportato nei miei momenti no (visto?Ce l'ho fatta!Vi voglio bene).

Ringrazio tutte le persone che indirettamente mi hanno aiutato, tra queste il mio fratellone Marco e la mia sorellina Silvia.

Un ringraziamento speciale agli Artificial_Studios, Aristide (JoY§TiCk) e Stefano (StefanoAI), i migliori compagni che avrei potuto trovare e soprattutto grandi amici, con cui ho condiviso questi tre anni di "gioie e dolori". Grazie per avermi sopportato durante i miei momenti di "sclero" (immagino sappiate di cosa parlo...) e per essermi stati sempre vicini. Probabilmente senza di voi non sarei arrivato fino a qui(grazie ragazzi!).

Ringrazio i componenti del LaSER, in ordine scientifico inverso: Gianz(ti ho già risposto...NO!!!), Ema(chuppaaaa!), Dario(scusa se credevo ti chiamassi Igor), David Brown(www.skebby.it), e Ale(fondamentalmente), per l'accoglienza(all'inizio un po fredda), i grandi insegnamenti, le risate e per tutti i caffè che mi avete fatto fare.

Ringrazio i miei colleghi di corso, tra cui in ordine alfabetico: Andrea(Galeina), Fabio(DiLa), Giuliano(Gollum), Laura(laura.prog.io), Marco(Skunk), Marco(metalman), Silvita, Simone (Ssj4), ... , per le piacevoli ore di studio trascorse in università.

Ringrazio Roberta per avermi sopportato durante i miei periodi di stress universitario e per avermi incoraggiato quando serviva.

Grazie!

Indice

Capitolo 1

Introduzione

Alcuni errori di programmazione possono essere sfruttati per sovvertire il comportamento di un'applicazione ed eventualmente prendere il controllo di interi sistemi. Tali errori rendono quindi un programma vulnerabile ad attacchi compiuti da utenti malintenzionati. Qualsiasi applicazione di una certa complessità contiene, con buona probabilità al suo interno una o più vulnerabilità. Dal momento del rilascio di un prodotto software al rilevamento di un errore che causi una vulnerabilità trascorre un lasso di tempo indeterminato. Una volta rilevata una vulnerabilità e dopo che tale baco di programmazione viene pubblicato, i produttori dell'applicazione sono soliti rilasciare una *patch* (file creato per risolvere uno o più difetti di programmazione). Da tale momento è cura dei singoli utenti applicare le modifiche rilasciate per difendersi da eventuali attacchi. In poche parole quindi, la colpa da attribuire a tali problemi di sicurezza informatica non è solo ai programmatori, ma anche ad utenti superficiali. Tuttavia la situazione diventa più complessa quando le vulnerabilità sono causate da errori di progettazione. In tale caso infatti, la soluzione a un errore pubblicato consiste nella maggior parte dei casi nel rivedere e riprogettare l'intera applicazione. Questo non sempre è possibile, e comunque comporta una tempistica e un impegno economico maggiore. Per tali motivi, lo studio effettuato sulle applicazioni, che conduce al rilevamento delle vulnerabilità sopra citate, risulta fondamentale. Il lavoro di tesi presentato si prefigge lo studio di un *framework* allo scopo di rilevare vulnerabilità a partire da codice binario. L'identificazione automatica dei difetti di programmazione relativi ai

file binari è un area di ricerca ancora giovane ma promettente. Il *framework* studiato, che prende il nome di *smart fuzzer*, si basa su una tecnica denominata *fuzzing*, che consiste nell'utilizzare come input di una data applicazione dati totalmente random, allo scopo di causare errori di esecuzione. Basandosi esclusivamente su questo tipo di approccio, a causa della totale randomizzazione dei dati di input, si rischia di non essere in grado di soddisfare ogni possibile condizione presente nell'applicazione. Risulta quindi essenziale integrare l'approccio sopra spiegato, rendendo l'analisi "dedicata", ossia specifica per un'applicazione. Per fare ciò il *framework* studiato utilizza tecniche di *program analysis* ibride per il rilevamento di tali difetti di programmazione. Unisce infatti tecniche statiche e dinamiche, allo scopo di una più completa e precisa analisi del file binario da cui bisogna estrapolare vulnerabilità. In parole povere l'approccio *smart fuzzer* consiste nell'analizzare il programma e nel monitorare l'esecuzione, al fine di estrapolare un insieme di input in grado di indirizzare il flusso di esecuzione verso stati in cui potrebbero manifestarsi delle vulnerabilità. Si compone di due fasi: una prima fase di analisi statica che fornisce informazioni generali sul comportamento di una applicazione, dopodiché tramite un'analisi dinamica viene monitorato il flusso di esecuzione, raffinando le informazioni raccolte in precedenza. Le informazioni ottenute verranno poi utilizzate per la creazione di nuovi input, che forniti all'applicazione, sono in grado di sovrascrivere zone di memoria delicate. Molto importante notare che l'approccio utilizzato effettua un'analisi su codice binario. Questo rende l'analisi libera dai problemi di distribuzione del codice, in quanto non tutte le applicazioni, ad esempio prodotti commerciali, risultano *open source*. Questa però non risulta essere l'unica motivazione, in quanto in molti casi un'analisi svolta su codice sorgente non permette il rilevamento di problematiche di sicurezza causate dai dettagli legati alla piattaforma su cui viene eseguita l'applicazione.

Il presente lavoro di tesi si basa principalmente sullo studio della fase di analisi dinamica di tale *framework*, focalizzandosi sull'analisi delle *path condition*, ossia quella fase in cui vengono creati i vincoli per la costruzione di nuovi input.

Il contributo originale di questo lavoro di tesi, consiste nel rilevamento delle funzioni inline, ossia quelle funzioni che non hanno simboli di rilocazione associati, o nel caso di static linking dove le funzioni di libreria non hanno simboli associati. La

soluzione proposta è stata implementa direttamente all'interno del *framework* tramite la creazione di alcuni algoritmi.

Il secondo contributo consiste nell'integrazione di tali algoritmi nella fase di analisi delle *path condition*. Questo permette la creazione di nuovi input a partire dai vincoli ricavati dalle funzioni rilevate nel contributo precedente.

Il presente lavoro di tesi è organizzato nel seguente modo:

Capitolo 2: Background

In questo capitolo vengono presentati alcuni dei concetti utilizzati nel lavoro proposto. Tali concetti risultano essenziali per la comprensione del lavoro svolto. Vengono introdotti i concetti di control flow graph, reaching definition (fondamentale per la comprensione della fasi di analisi delle path condition) e i principali tipi di attacchi che sfruttano i cosiddetti *memory error*.

Capitolo 3: Panoramica

In tale capitolo viene fornita una panoramica sul *framework* utilizzato come piattaforma per l'analisi di codice binario. Vengono spiegate le varie fasi che compongono il processo di analisi. A scopo esplicativo vengono forniti vari esempi, tra cui un prototipo di codice vulnerabile. Oltre all'analisi statico dinamica vengono esposte alcune euristiche utilizzate per l'individuazione delle zone di memoria da monitorare.

Capitolo 4: Analisi delle path condition

In questo capitolo vengono illustrati i principali contributi di questo lavoro di tesi. Viene spiegata la tecnica di riconoscimento di funzioni inline e una panoramica sull'analisi delle path condition. Le tecniche descritte in questo capitolo vengono accompagnate da relativi esempi.

Capitolo 5: Conclusioni

L'ultimo capitolo è composto dalle conclusioni ricavate alla fine dello svolgimento del lavoro di tesi.

Capitolo 2

Concetti preliminari

In questo capitolo verranno introdotti alcuni concetti indispensabili per la comprensione di questo lavoro di tesi.

2.1 Control flow graph

Un control flow graph è una struttura dati utilizzata nelle tecniche di *program analisys*. Formalmente si tratta di un grafo i cui nodi rappresentano computazioni e gli archi rappresentano il flusso di controllo. In poche parole è una rappresentazione di tutti i cammini di una applicazione che potrebbero essere attraversati durante un'esecuzione. Ogni nodo del control flow graph rappresenta un basic block, cioè una sequenza di statement (istruzioni) consecutive in cui il flusso non è interrotto e/o non attraversa una branch. Tali nodi sono caratterizzati dall'avere un'unico punto di ingresso (prima istruzione eseguita) e un'unico punto di uscita (ultima istruzione eseguita). Quando il flusso di controllo di un applicazione raggiunge un basic block, questo causa l'esecuzione della prima istruzioni contenuta nel blocco; le istruzioni successive vengono eseguite in ordine, senza che avvengano interruzioni o salti al di fuori del blocco fino al raggiungimento dell'ultima istruzione.

Un basic blocco b_1 si dice predecessore immediato di b_2 se b_2 segue b_1 in una possibile esecuzione, ossia se si verifica una delle seguenti condizioni [3]:

1. l'ultima istruzione del basic block b_1 rappresenta un istruzione di salto condizionato o incondizionato con destinazione la prima istruzione di b_2.

2. il basic block b_2 segue immediatamente b_1 nel codice del programma e la terminazione di b_1 non è causata da un'istruzione di salto.

In un classico grafo orientato il nodo di ingresso è semplicemente un nodo privo di predecessori, mentre in un CFG si considera come nodo iniziale il basic block contenente l'*entry point* dell'applicazione, ossia la prima istruzione eseguita dal programma o dalla procedura rappresentata. Viene considerato come nodo terminale del CFG il nodo contenente l'istruzione che provoca l'arresto della computazione. Il procedimento utilizzato per partizionare una sequenza di istruzioni di un programma P in basic block è riportato in [3] e descritto qui di seguito:

1. Prima di tutto è necessario determinare l'insieme dei *leader*, ossia le istruzioni di P che occuperanno il primo posto all'interno dei basic block. Le regole da considerare sono le seguenti:

   - il primo statement di un blocco è considerato *leader*;
   - ogni statement corrispondente al bersaglio di un'istruzione di salto, viene considerato *leader*;
   - ogni statement che segue un istruzione di salto viene considerato *leader*.

2. un basic block consiste nelle istruzioni comprese tra un *leader* stesso incluso e il *leader* sintatticamente successivo escluso. Nel caso in cui il *leader* analizzato fosse l'ultimo del programma, il basic block che lo contiene si estenderebbe fino alla fine dell'applicazione.

2.1.1 Reaching definition

Prima di spiegare il concetto di reaching definition, verranno spiegati i concetti di punto e cammino. All'interno di un basic block con n istruzioni vi sono $n+1$ punti: il

punto iniziale che corrisponde al punto immediatamente prima della prima istruzione del blocco, il punto terminale del blocco situato immediatamente dopo l'ultima istruzione del blocco e *n-1* punti compresi tra due istruzioni consecutive. Un cammino tra due punti è composta da una sequenza di punti, tali che per ogni i compreso in [1,....,*n*-i] valga una delle seguenti possibilità:

1. p_i è un punto immediatamente precedente ad un'istruzione e il punto p_{i+1} segue immediatamente la stessa istruzione nello stesso blocco.

2. p_i punto terminale di un blocco, e di conseguenza p_{i+1} è il punto iniziale del blocco immediatamente successivo.

Un cammino tra due punti si dice *libero da definizioni* o *definition-clear* rispetto ad una variabile, se in tale cammino non vi è nessuna definizione *non ambigua* della variabile presa in considerazione. Per definizione *non ambigua* di una varibile x si intende un'istruzione di assegnamento del tipo $x = < ... >$, oppure un'istruzione che legge un valore da un dispositivo di I/O e lo memorizza nella variabile x. Esistono poi definizioni *ambigue* di una variabile x, rappresentate da istruzioni per le quali non è possibile determinare *staticamente* se definiscono la variabile, ad esempio un assegnamento attraverso una variabile puntatore che potrebbe fare riferimento ad x.

Una definizione d della variabile x *raggiunge* un certo punto p se esiste un cammino dal punto che segue immediatamente d fino a p lungo il quale non si hanno definizioni non ambigue di x, ovvero se esiste un cammino *definition-clear* dal punto che segue immediatamente d fino a p. In tal caso si dice che d costituisce una *reaching definition* per x al punto p. Formalmente può essere rappresentata da insiemi di coppie:

$$\{(x, p) \mid x \text{ è una variabile del programma e p è un punto del programma}\}$$

Il significato della coppia (x, p) nell'insieme associato al punto q, è che l'assegnamento di x nel punto p "raggiunge" il punto q. Considerando la definizione sopra data in un'analisi statica (Vedasi capitolo 3), risulta essere poco accurata, in quanto si potrebbe in alcuni casi non avere la definizione di una variabile che raggiunge un punto eseguendo un cammino libero da definizioni ambigue. Di conseguenza quando si calcolano staticamente le *reaching definition* relative ad una variabile corrispondente ad un'istruzione, si avrà un insieme di istruzioni che definiscono tale variabile.

A differenza in un'analisi dinamica dove non si verificano problemi di ambiguità, è possibile associare un unica *reaching definition* ad un istanza di istruzione. In particolare consideriamo un *control flow graph* $G = (B,E)$ associato ad un certo programma P. Definiamo una *execution history* [1], ossia un cammino in G rappresentato come una lista di istanze di istruzioni del programma Q registrate durante la sua esecuzione nell'ordine in cui esse sono state effettivamente eseguite. Formalmente definiamo una *execution history* come $EH = < x_1, x_2, ..., x_n >$ in cui x_1 indica la *i-esima* istruzione del programma. Nel caso un'istruzione venga eseguita più volte, per esempio se si trova all'interno di cicli, le diverse istanze vengono differenziate da un numero in apice. Utilizzando la definizione appena riportata, un'istanza di istruzione verrà indicata con $s = i^n$, dove i rappresenta un'istruzione del progamma e n l'indice utilizzato per differenziare le diverse occorrenze della stessa istruzione. Utilizzando una *execution history* EH si definisce la *reaching definition dinamica* [1] denotata con $drd_E H(v, x_k)$, di una variabile v utilizzata da un'istanza di istruzione x_k, come l'istanza di istruzione $x_k - j$, tale che:

$$x_{k-j} \in EH \ \wedge \ v \in define(x_k - j) \ \wedge \ \nexists x_m, m \in (k-j,k) : v \in define(x_m)$$

Per calcolare la *dinamic reaching definition* della variabile v utilizzata nell'istruzione x_k, è sufficiente scorrere l'*execution history* EH a ritroso partendo da x_k, fino al rilevamento di un'istanza di istruzione che definisce v, che di conseguenza costituisce la *reaching definition* ricercata.

2.2 Memory error

In questa sezione verranno descritti i difetti di programmazione che portano ad una corruzione della memoria. Questi sono spesso causati da una errata gestione dell'input, come ad esempio la mancanza di controllo sulla lunghezza di una stringa proveniente dai dati di input. È molto importante analizzare questo tipo di errori, in quanto potrebbero consentire ad un ipotetico attaccante la generazione di un input che consenta di sovrascrivere zone di memoria, dirottando il flusso di esecuzione verso istruzioni da esso controllate. Quindi un'ipotetica malgestione dell'input potrebbe causare vulne-

rabilità che consentano ad un attaccante l'esecuzione di codice arbitrario. Di seguito vengono descritti i principali attacchi che sfruttano tale tipologia di errore:

Stack-based buffer overflow : il buffer overflow [4] è una vulnerabilità di sicurezza che può affliggere un programma software. Consiste nel fatto che tale programma non controlla in anticipo la lunghezza dei dati in arrivo, ma si limita a scrivere il loro valore in un buffer di lunghezza prestabilita, confidando che l'utente non immetta più dati di quanti esso ne possa contenere: questo può accadere se il programma è stato scritto usando funzioni di libreria di input/output che non fanno controlli sulle dimensioni dei dati trasferiti.

Quando quindi, per errore o per malizia, vengono inviati più dati della capienza del buffer destinato a contenerli, i dati in eccesso vanno a sovrascrivere le variabili interne del programma, o altre zone sensibili del suo stack; come conseguenza di ciò, a seconda di cosa è stato sovrascritto e con quali valori, il programma può dare risultati errati, bloccarsi, o (se è un driver di sistema o lo stesso sistema operativo) bloccare il computer. Conoscendo molto bene il programma in questione, il sistema operativo e il tipo di computer su cui viene eseguita l'applicazione, si può precalcolare una serie di dati malevoli che inviata per provocare un buffer overflow consenta ad un malintenzionato di prendere il controllo del programma (e a volte, tramite questo, dell'intero computer).

Questo tipo di debolezza dei programmi è noto da molto tempo, ma solo di recente la sua conoscenza si è diffusa tanto da permettere anche a dei *cracker* dilettanti di sfruttarla per bloccare o prendere il controllo di altri computer collegati in rete. Non tutti i programmi sono vulnerabili a questo tipo di inconveniente: perchè un dato programma sia a rischio è necessario:

1. che il programma preveda l'input di dati di lunghezza variabile e non nota a priori;
2. che li immagazzini entro buffer allocati nel suo spazio di memoria vicini ad altre strutture dati vitali per il programma stesso;
3. che il programmatore non abbia implementato alcun mezzo di controllo sulla correttezza dell'input.

La prima condizione è facilmente verificabile, dalle specifiche del programma; le altre due invece sono interne ad esso e riguardano la sua completezza in senso teorico. Lo stack overflow consiste ugualmente nella sovrascrittura dell'area dati del programma, ma questa volta la causa è l'attività del programma stesso: chiamando con dei parametri particolari una funzione ricorsiva del programma, questa accumula chiamate in sospeso sullo stack fino a riempirlo completamente e inizia a sovrascrivere la memoria vicina.

Heap overflow : è un altro nome per indicare un buffer overflow che avvenga nell'area dati dello heap. A differenza che nello stack, dove la memoria viene allocata staticamente, nello heap essa viene allocata in modo dinamico dalle applicazioni a run-time e tipicamente contiene le variabili allocate dinamicamente.
Gli heap overflow [10] solitamente vengono utilizzati da ipotetici attaccanti, per violare la sicurezza dei programmi scritti in modo non impeccabile. L'attacco avviene come segue: se una applicazione copia dei dati senza preventivamente controllare se trovano posto nella variabile di destinazione, l'attaccante può fornire al programma un insieme di dati troppo grande per essere gestito correttamente, andando così a sovrascrivere i metadati (cioè le informazioni di gestione) dello heap, contenuti nella locazione di memoria immediatamente seguente a quella in cui vengono posizionati i dati. In questo modo, l'attaccante può sovrascrivere una locazione arbitraria di memoria, con una piccola quantità di dati. Nella maggior parte degli ambienti, questo può fornire all'attaccante il controllo dell'esecuzione del programma.
La vulnerabilità Microsoft JPEG GDI+ MS04-028 è un esempio del pericolo che uno heap overflow può rappresentare per un utente informatico.

Format bug : i format bug [12] sono problemi di sicurezza relativamente recenti che hanno aperto un nuovo fronte da combattere in milioni di righe di codice scritto in passato. La vulnerabilità è causata da un errore nell'utilizzare una funzione che usa stringhe di formato, in C la più nota è naturalmente `printf()`. Un attaccante potrebbe inserire come parametro di questa funzione una stringa in grado di sovrascrivere locazioni di memoria e dirottare il flusso di esecuzione.

In breve, si tratta di passare a una funzione che riceve stringhe di formato, tipicamente una `printf()` del linguaggio C, una stringa che in realtà contiene una serie di parametri di specifica dell'input (tipicamente si usano gli specificatori di formato `%s` e `%x` per esaminare il contenuto della memoria e `%n` per sovrascrivere parti della memoria, in particolare dello stack, questo permette attacchi di tipo stack overflow e return to `libc`).

Off by one : è un attacco basato sullo stack-based buffer overflow, con la differenza che ha effetto anche quando il limite del bersaglio può essere oltrepassato di un solo byte. In poche parole quando si esegue una chiamata a funzione, la prima istruzione provvede a salvare sullo stack il registro `ebp` (*base pointer*). Nel caso in cui il buffer bersaglio sia collocato appena sotto il registro all'interno dello stack, allora un ipotetico attaccante potrebbe sovrascrivere con un valore arbitrario l'ultimo byte di `ebp`. Quindi non è possibile sovrascrivere l'indirizzo di ritorno della funzione, ma si può tramite la tecnica spiegata convincere il processore che ci si trovi in una locazione di memoria diversa da quella aspettata. Nel caso in cui un ipotetico attaccante riuscisse a inserire all'interno di questa zona di memoria l'indirizzo di una procedura iniettata in precedenza, potrebbe eseguire codice albitrario e quindi compiere qualsiasi operazione [7].

Capitolo 3

Panoramica

L'identificazione automatica di vulnerabilità nel codice binario avviene tramite l'utilizzo di tecniche di *program analysis* (vedasi capitolo 2) statico-dinamiche, implementate in un prototipo sperimentale. In questo capitolo verranno descritte ad alto livello la struttura ed il funzionamento del *framework* di analisi studiato.

3.1 *Smart fuzzing*

L'infrastruttura di analisi considerata nel lavoro di tesi, prende il nome di "*smart fuzzer*". Lo scopo di questo modello è guidare il flusso di esecuzione di un'applicazione verso casi limite, dove la probabilità di rilevare una vulnerabilità aumenta. Le principali vulnerabilità che il modello si prefigge di rilevare sono i cosiddetti "*memory error*". Queste vulnerabilità, danno ad un ipotetico "attaccante" la possibilità di sovrascrivere zone di memoria delicate per l'applicazione. Un esempio di tale attacco è il *buffer overflow*[4]. Consiste nel fatto che un dato programma non controlla in anticipo la lunghezza dei dati in arrivo, ma si limita a scrivere il loro valore in un buffer di lunghezza prestabilita, confidando che non vengano immessi più dati di quanti esso ne possa contenere. Nel caso in cui si superi la dimensione del buffer i dati in eccesso vanno a sovrascrivere le variabili interne del programma. Un "attaccante" a conoscenza del sistema su cui il programma viene eseguito e del programma in questione, potrebbe precalcolare un input malevolo, che usato per provocare un *buffer overflow*,

consenta di eseguire codice arbitrario. Per rilevare simili vulnerabilità, il modello studiato prevede di eseguire il programma più volte con input diversi, cercando ad ogni esecuzione di estrapolare informazioni utili alla generazione di nuovi input, che siano in grado di guidare l'esecuzione del programma verso cammini inesplorati.

Il modello "*smart fuzzer*" si compone di due fasi :

- analisi statica;
- analisi dinamica.

In fig 3.1 viene presentato un esempio di codice vulnerabile.

```
1  void cp(char *src)
2  {
3    char buffer[80];
4    if(strlen(src) > 10)
5    {
6      strcpy(buffer, src);
7    }
8  }
9  int main(int argc, char **argv)
10 {
11   if(argv[1][0] == 'k' && argv[1][2] == 'z')
12   {
13     cp(argv[1]);
14     return 0;
15   }
16   return 1;
17 }
```

Figura 3.1: Codice C vulnerabile

Il codice illustrato in fig 3.1 è soggetto a buffer overflow, in quanto la procedura `cp()` non effettua nessun controllo sulla dimensione dei dati in arrivo, ma si limita a copiarli in un buffer. Quindi passando alla funzione `cp()` un input maligno di dimensione opportuna si può sovrascrivere l'indirizzo di ritorno della funzione, andando ad eseguire codice arbitrario. Segue la descrizione delle fasi principali del modello.

3.1.1 Analisi statica

In questa fase si effettua un'analisi preliminare del programma. Innanzi tutto vengono estratti il *codice macchina*[1] del programma, le librerie caricate dinamicamente e l'entry point dell'applicazione. Il codice macchina viene poi trasformato in codice *assembly*[2] tramite la fase *disassemby*. Questa tecnica non permette però di esplorare regioni di codice raggiungibili solo tramite istruzioni di trasferimento di controllo indirette. Essendo che il modello adotta un approccio ibrido, è possibile ignorare questo problema, rimandandolo all'analisi dinamica. Dopo aver ottenuto il codice assembly si passa alla traduzione del codice in *forma intermedia*. Questa forma permette di esplicitare gli effetti di ciascuna istruzione sui registri e sulla memoria, rendere indipendente la rappresentazione del codice rispetto al processore e ridurre le tipologie di istruzioni utilizzando 4 tipi di istruzioni e 5 tipi di espressioni.

Le classi di espressioni supportate sono le seguenti:

1. **Constant** rappresenta un valore costante.
2. **Register** rappresenta un particolare registro del processore.
3. **OverlappingRegister** questa espressione fa riferimento ad un registro "virtuale" che fisicamente corrisponde ad una regione del corrispondente registro fisico.
4. **Memory** rappresenta una zona di memoria.
5. **EExpression** rappresenta un'espressione composta (unaria, binaria, o ternaria).

Le categorie di istruzioni disponibili sono le seguenti:

1. **Assignment** rappresenta una generica istruzione di assegnamento della forma a := b.
2. **Jump** rappresenta salti condizionati o incondizionati.

[1] Codici direttamente interpretabili ed eseguibili da parte dell'elaboratore senza necessitàị alcuna traduzione.

[2] Linguaggio di programmazione a basso livello le cui istruzioni possono essere direttamente convertite in linguaggio macchina

3. **Call** modella una chiamata a procedura.

4. **Ret** viene utilizzata per terminare una procedura.

Si passa poi alla costruzione del CFG (*control flow graph*) del programma, ossia un grafo orientato in cui i nodi sono basic block e gli archi rappresentano il flusso di esecuzione. I basic block contengono istruzioni intermedie e ogni istruzione che modifica il flusso di esecuzione (`Jump, Call o Ret`) comporta la terminazione del basic block corrente e l'esplorazione dei nuovi cammini determinati da tali istruzioni.

In Figura 3.2 viene riportato il CFG della funzione `cp()` del codice mostrato in Figura 3.1.

Alcune istruzioni intermedie sono state omesse per ragioni di spazio, ma pare evidente il notevole incremento della dimensione del codice in seguito alla traduzione in forma intermedia, dovuta all'esplicitazione degli effetti delle istruzioni assembly. Le frecce continue esplicitano il flusso di esecuzione, mentre le tratteggiate raffigurano le dipendenze di controllo.

In ultimo abbiamo la fase di *liveness analysis*, dove si effettuano una serie di analisi statiche che integrano le informazioni ottenute in precedenza. Questa analisi consiste nell'eliminazione delle definizioni non più utilizzate (*dead*). In seguito avremo il calcolo delle dipendenze di controllo tra i nodi del CFG e l'individuazione dei costrutti di iterazione.

3.1.2 Analisi dinamica

L'analisi dinamica, tramite l'esecuzione e il monitoraggio dell'applicazione, ha lo scopo di estrapolare informazioni su un insieme di vincoli per la generazione di un input, che riesca a guidare l'esecuzione verso la sovrascrittura di zone di memoria sensibili. Questa analisi, a differenza di quella statica, dipende da una particolare esecuzione, in quanto opera su specifiche variabili d'ingresso .Questo porta a non poter generalizzare i risultati ottenuti e quindi, nel caso vi siano ramificazioni che portano a rieseguire il codice su un nuovo input, l'analisi dinamica dovrà essere ripetuta. Prima di procedere con il monitoraggio si passa attraverso la fase di *analisi dello stato iniziale del processo*, in cui si analizza lo spazio di indirizzamento del processo, per estrarre eventuali

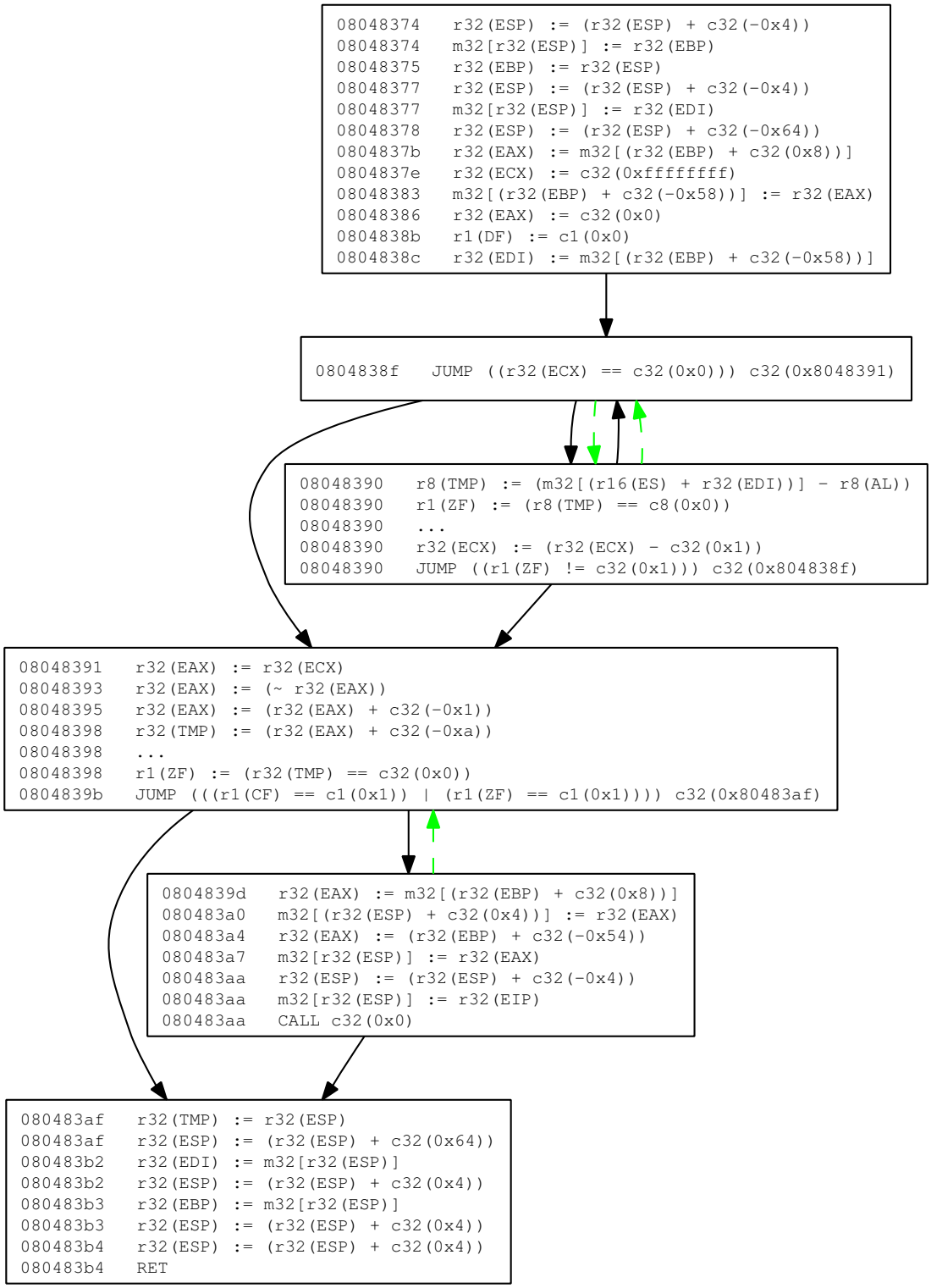

Figura 3.2: *Control flow graph* ottenuto dalla funzione cp dell'esempio in Figura 3.1

parametri da riga di comando e variabili d'ambiente. Le informazioni ottenute verranno salvate in una *struttura dati* chiamata *process startup state* (PSS). Nel PSS verrà salvato anche lo stato dei registri del processore. Conclusa la fase di analisi dello stato iniziale del proccesso si passa al *monitoraggio dell'esecuzione*, dove ogni istruzione intermedia appartenente al gruppo associato all'istruzione assembly appena eseguita viene aggiunta in coda ad una struttura dati che prende il nome di *execution history*. Dopodiché si passa ad una serie di analisi finalizzate alla costruzione delle *path condition* (vedasi capitolo 4) riguardanti l'esecuzione del cammino corrente.

- risoluzione dei salti e chiamate a funzioni indirette;
- valutazione delle istruzioni;
- analisi delle condizioni di salto;
- semplificazione delle espressioni;
- analisi delle *path condition*(vedasi capitolo 4);
- analisi dei cicli;
- generazione dei vincoli e dei nuovi input.

La *risoluzione dei salti e chiamate a funzioni indirette* si prefigge di integrare le informazioni approssimative della fase di disassembly statico. Quando analizzando le istruzioni assembly si incontra una `Jump` o `Call` indiretta, si valuta l'indirizzo di destinazione a cui il controllo sta per essere trasferito. Questa operazione può avere come conseguenza l'esecuzione di parti di codice non analizzate dalla fase di *disassembly*, e in questo caso si dovrà richiamare l'analizzatote statico sulla nuova zona di codice.

La *valutazione delle istruzioni* consiste nell'esaminare e attualizzare ogni espressione appartenente ad una istanza di istruzione, dato che, operando a *run-time* é possibile determinare con precisione le aree di memoria utilizzate da ciascuna istruzione.

L' *analisi delle condizioni di salto*, ha lo scopo di analizzare le istruzioni di salto condizionato(es. `jne` *jump-if-not-equal*), al fine di di ricostruire una formulazione ad alto livello del predicato valutato dall'espressione.

La *semplificazione delle espressioni*, agevola le analisi successive, applicando una serie di trasformazioni (logiche, matematiche e sintattiche) alle complesse espressioni utilizzate dalle istruzioni intermedie.

L'*analisi dei cicli* del modello *smart fuzzing*, tramite l'osservazione di ogni iterazione del ciclo, consente di astrarre il suo comportamento dallo specifico insieme dei dati in ingresso correlati ad una esecuzione. Innanzitutto vengono osservate dinamicamente le relazioni tra le condizioni di uscita del ciclo e i dati di ingresso all'applicazione. Dopodiché si identificano le variabili di induzione e si deduce come queste possano influenzare l'indirizzo di destinazione di un'istruzione di scrittura in memoria. In base alle informazioni raccolte si può, in alcuni casi, generare un insieme di vincoli per definire un nuovo input, pericoloso per l'applicazione.

La *generazione dei vincoli e dei nuovi input* crea un insieme di condizioni tali da consentire l'esplorazione di nuovi cammini di esecuzione. Ad esempio nel caso di un costrutto *if*, verranno generati due insiemi di vincoli, corrispondenti ai possibili valori di verità della condizione. I vincoli ottenuti verranno poi trasmessi ad un risolutore (*constraint solver*) che ne determinerà l'eventuale soddisfacibilità. Se un insieme di vincoli non risulta soddisfacibile, il cammino associato viene scartato. In caso contrario, il risolutore genererà un insieme di informazioni per la creazione di un nuovo input per l'applicazione, in grado di condurre l'esecuzione fino allo stato associato al corrispondente insieme di vincoli. Il generatore di nuovi input tratterà opportunamente le informazioni ottenute, utilizzando delle euristiche che prediliggono quei cammini dove la probabilità che una vulnerabilità si manifesti è più alta.

3.1.3 Altre analisi

Vengono ora discusse le ultime due analisi che compongono il framework in questione. Tramite queste analisi vedremo come trasformare un insieme di *path condition* in un nuovo input per l'applicazione che soddisfi i vincoli imposti, e una serie di euristiche utilizzate per determinare zone di memoria sensibili.

3.1.3.1 Euristiche

In questo paragrafo vengono presentate alcune euristiche, utilizzate per determinare i cammini dove la probabilità che una vulnerabilità si manifesti è più alta. Prima di tutto bisogna individuare le zone di memoria sensibili. Per fare ciò, durante il monitoraggio dinamico dell'esecuzione dell'applicazione, ogni zona di memoria contenente dati ipoteticamente controllabili da un attaccante, viene marcata come *contaminata (tainted)*. Un esempio di zona di memoria sensibile è l'indirizzo di ritorno, in quanto bersaglio molto comune per attacchi di tipo *buffer overflow*. È molto importante che tale area di memoria venga considerata sensibile solo fino a quando la procedura corrispondente non termina. Si riassumono qui di seguito alcune categorie di locazioni di memoria che possono essere considerate delicate.

Indirizzo di ritorno Discusso sopra.

Destinazione di salti indiretti È piuttosto comune a livello del codice assembly incontrare istruzioni di `jmp` o `call` indirette, che causano la ripresa dell'esecuzione dall'indirizzo specificato tramite un registro del processore o una locazione di memoria. L'argomento di un'istruzione di questo tipo identifica evidentemente una zona delicata. Nel caso in cui l'argomento in esame sia una locazione di memoria, allora è sufficiente considerare tale area come delicata per tutto il periodo di esecuzione che va dalla sua *reaching definition* valutata in corrispondenza dell'istruzione di salto o di chiamata a procedura, fino al momento in cui viene raggiunta l'istruzione di trasferimento di controllo. Se invece l'argomento fosse un registro x del processore (es. `jmp` $\star x$), allora si potrebbero fare osservazioni del tutto analoghe, a patto di considerare come delicata la zona di memoria utilizzata per definire il contenuto di x (anche in questo caso è sufficiente ricorrere all'analisi delle *reaching definition* che confluiscono in x).

Strutture per l'allocazione dinamica. Una particolare tipologia di attacco nota come *heap overflow* [10] altera il flusso di esecuzione dell'applicazione sfruttando la possibilità dell'attaccante di sovrascrivere alcune locazioni contenenti i metadati utilizzati per la gestione delle zone di memoria allocate dinamicamente.

Anche tali strutture possono essere considerate come delicate, ma è necessario un attento monitoraggio del gestore dello *heap* al fine di determinare con precisione quando in una certa locazione vengono memorizzati i metadati e quando, a causa della deallocazione di una variabile precedentemente allocata dinamicamente, un'area cessa di contenere dati delicati. Ad esempio, su piattaforma Linux tali operazioni possono essere effettuate tenendo traccia delle chiamate ad alcune funzioni di libreria quali `malloc()` e `free()`.

Argomenti delle chiamate a sistema. Una chiamata a sistema (nota come *system call* o, più semplicemente, *syscall*) è il meccanismo utilizzato da un'applicazione per richiedere un servizio al sistema operativo. Gli argomenti delle chiamate a sistema costituiscono spesso dati delicati. Per fare un esempio, si consideri la *syscall* Linux `sys_execve()`: tale chiamata riceve come primo argomento un puntatore ad una stringa di caratteri che rappresenta il percorso del file da eseguire. Un simile parametro rappresenta certamente una zona di memoria delicata in quanto, se sovrascritto con dati contaminati, consentirebbe ad un attaccante di eseguire un programma arbitrario.

Argomenti delle chiamate di libreria. Alcune funzioni appartenenti a librerie di sistema possono costituire un pericolo per la sicurezza dell'applicazione se invocate con parametri controllabili dall'attaccante. Si considerino, ad esempio, le procedure `system()` e `popen()` della libreria standard `glibc`, che consentono di eseguire il programma identificato dal *path* passato come parametro. Un attaccante che riuscisse a manipolare gli argomenti di funzioni di questo tipo sarebbe in grado di eseguire un qualsiasi programma residente sulla macchina. Considerare come delicate le zone di memoria che ospitano gli argomenti di tali chiamate potrebbe consentire di individuare alcune vulnerabilità particolarmente significative. Si osservi, però, che in alcuni casi monitorare i parametri delle funzioni di libreria è di fatto superfluo, visto che eventuali vulnerabilità ad esse connesse potrebbero comunque venire rilevate grazie all'analisi dei parametri delle chiamate a sistema. Nel caso della funzione `system()`, ad esempio, per poter eseguire il comando passato come argomento, la procedura dovrà prima

o poi utilizzare una *system call*, non potendo compiere tutte le operazioni direttamente da *user space*. D'altra parte, però, il monitoraggio degli argomenti delle funzioni di libreria consente spesso di incrementare l'efficienza del processo di analisi, non dovendo esaminare l'intera chiamata ma limitandosi all'esame dei parametri in ingresso. In altri casi, invece, l'analisi dei parametri delle funzioni di libreria può consentire l'individuazione di vulnerabilità non rilevabili esaminando le sole *system call*, come, ad esempio, nel caso delle funzioni della famiglia `printf()` e vulnerabilità del tipo "*format bug*" [12].

Sezioni. Intere sezioni dello spazio di indirizzamento del processo possono essere considerate delicate per tutto il periodo di esecuzione dell'applicazione. Ciò vale, ad esempio, per alcune sezioni utilizzate dal linker dinamico (come la `.got` su macchine Linux) o altre sezioni tipicamente impiegate per la memorizzazione di *code pointer* che, nel caso venissero sovrascritti con dati contaminati, potrebbero consentire il dirottamento del flusso di esecuzione (come, sempre su Linux, la sezione `.dtors`).

Vengono di seguito illustrati alcuni esempi di euristiche utilizzate per la scelta del percorso di esecuzione da eseguire, ossia quello con maggiore probabilità di raggiungere porzioni di codice potenzialmente pericolose:

1. Per ogni istruzione di salto condizionato, esplorare la ramificazione che porta in parti di codice non ancora analizzate.

2. Non esplorare cammini privi di assegnamenti a zone di memoria con indirizzo non costante e chiamate a funzione.

3. Preferire la percorrenza di cammini che comprendono istruzioni che definiscono zone di memoria con indirizzo non costante.

4. Quando non si riesce ad astrarre il comportamento di un ciclo con l'*analisi dei cicli*, fare in modo che il ciclo effettui un numero considerevole di iterazioni.

5. Evitare cammini di esecuzione conclusi da chiamate a sistema che comportano la terminazione del processo e privi di istruzioni interessanti.

6. Quando si è in grado di individuare staticamente più *reaching definition* (vedasi capitolo 2) per l'indirizzo di destinazione di un'istruzione di trasferimento di controllo (ad esempio, `JUMP r32(EAX)`), cercare di raggiungere ciascuna di esse.

3.1.3.2 Risolutore di vincoli

Il *risolutore di vincoli o constraint solver* dato un insieme di condizioni su un input PC è in grado di compiere due operazioni:

- determinare se i vincoli sono soddisfacibili, e quindi stabilire se un cammino deve essere percorso oppure no;
- generare un insieme di informazioni per la creazione di un nuovo input per l'applicazione, in grado di condurre l'esecuzione fino allo stato associato al corrispondente insieme di vincoli.

In sostanza il risolutore è una procedura di decisione capace di stabilire la soddisfacibilità di formule in cui compaiono simboli interpretati, non interpretati e uguaglianza, in grado di supportare le teorie degli interi, dei vettori di bit e degli array. Con *procedura di decisione* si intende un programma in grado di determinare la soddisfacibiliti formule logiche che esprimono vincoli relativi ad prodotto software o hardware. Si tratta di strumenti utilizzati per la verifica formale di programmi informatici.

3.1.4 Caso d'uso

Di seguito si riporta l'applicazione dell'approccio *smart fuzzing* sul codice vulnerabile mostrato in Figura 3.1. Supponendo di eseguire l'applicazione con input "abc", l'analisi rileva che la condizione di riga 11 dipende dall'input. Quindi vengono generate due PC per eseguire entrambe le ramificazioni:

$PC = 0$ inizialmente PC è vuoto.

$$PC^{I} = PC \cup src[1] \; != \; k$$

$$PC^{II} = PC \cup src[1] \; = \; k$$

L'analisi continua seguendo la ramificazione che rispetta l'input immesso. In questo caso PC^{I}, che conclude l'esecuzione senza rilevare vulnerabilità. Viene poi generato un nuovo processo per eseguire la ramificazione alternativa, che in questo caso rileva nuovamente che la condizione in riga 11 dipende dall'input. Vengono generate due PC per eseguire entrambe le ramificazione :

$$PC^{III} = PC^{II} \cup src[3] \; != \; z$$

$$PC^{IV} = PC^{II} \cup src[3] \; = \; z$$

L'analisi continua seguendo la ramificazione che rispetta l'input immesso. In questo caso PC^{III}, che conclude l'esecuzione senza rilevare vulnerabilità. Viene poi generato un nuovo processo per eseguire la ramificazione alternativa, che in questo esegue la funzione `cp()`, dove l'analisi rileva che la condizione di riga 4 dipende dall'input. Vengono generate due PC per eseguire entrambe le ramificazioni:

$$PC^{V} = PC^{IV} \cup len(src) \; <= \; 10$$

$$PC^{VI} = PC^{IV} \cup len(src) \; > \; 10$$

L'analisi continua seguendo la ramificazione che rispetta l'input immesso. In questo caso PC^{V}, che conclude l'esecuzione senza rilevare vulnerabilità. Viene poi generato un nuovo processo per eseguire la ramificazione alternativa, che in questo caso rileva una vulnerabilità agli attacchi di *buffer overflow*.

Capitolo 4

Analisi delle path condition

Prima di parlare di analisi delle path condition, verrà introdotto il problema del riconoscimento di funzioni inline.

É importante affrontare questo problema in quanto la sua risoluzione permette la creazione di *path condition* riguardanti tali funzioni e quindi informazioni più accurate per la creazione di un nuovo input per l'applicazione.

Come esempio per spiegare il problema useremo la funzione di libreria `strlen()` utilizzata in riga 4 del codice in Figura 3.1. Questa funzione calcola la lunghezza di una stringa, ossia il numero di caratteri che la compongono. Nello specifico dell'esempio, la funzione `strlen()` calcola la lunghezza della stringa `src` (inteso come il numero di caratteri dell'array puntato da `src`) escluso il terminatore '`\0`' e verifica se tale lunghezza sia maggiore di 10.

Dopodiché, sarà illustrato tramite l'analisi delle *path condition* il metodo con cui vengono ricavati i vincoli per la creazione di un nuovo input.

4.1 Funzioni di libreria

Le librerie standard (ad esempio la *GNU C Library*), assumono comportamenti particolarmente complessi da analizzare, a causa di codice particolarmente intricato, pesanti ottimizzazioni in fasi di compilazione e parti di codice scritte in linguaggio assembly dal programmatore per ragioni di efficienza.

D'altra parte, le funzioni di libreria vengono largamente utilizzate dai programmatori durante la fase di coding nel ciclo di vita di un software. Quindi rappresentano una percentuale considerevole dell'insieme delle istruzioni eseguite da una qualsiasi applicazione software. Per fare un esempio, l'esecuzione del comando /bin/ls in una directory con 5 file e 5 sottodirectory utilizza circa 30 funzioni di libreria differenti, invocate complessivamente 280 volte. Risulta quindi indispensabile riuscire ad analizzare questo tipo di funzioni, nonostante la loro complessità.

Nel framework di *smart fuzzing*, per ragioni di efficienza, una volta riconosciute e individuate tali funzioni, si cerca di sostituire il corpo della procedura con una serie di istruzioni intermedie che ne riassumano il comportamento e che ne esplicitino gli effetti di interesse per l'analisi in corso, evitando così di andare ad analizzare l'intera funzione. Tecniche simili vengono utilizzate in contesti diversi e prendono il nome di *function summarization*.

A questo punto rimane il problema di identificare tali funzioni. Questa non è un'operazione banale quando ci si trova ad analizzare codice binario, a differenza delle analisi fatte sui codici sorgenti, dove il rilevamento delle funzioni di libreria invocate è solitamente semplice. Nel caso in cui il file eseguibile analizzato usi tecniche di *dynamic linking*[9], è spesso possibile utilizzare le informazioni di rilocazione per riconoscere con precisione la procedura di libreria invocata da un'istruzione di `call`. Ad esempio la funzione `strcpy()`:

$$char * strcpy(char * dest,\ const\ char * src);$$

Copia la stringa `src` sull'area puntata da `dest` la quale deve essere sufficientemente ampia ad accogliere tutti i caratteri di `src` compreso il terminatore '`\0`'. Verrà tradotta in fase di disassemblamento come una chiamata a funzione, e quindi è possibile tramite i simboli di rilocazione risalire al nome della funzione. Una volta riconosciuta la funzione, si può passare alla sostituzione del corpo della procedura con una serie di istruzioni intermedie. È quindi sufficiente a questo punto eseguire il corpo della procedura di libreria senza analizzare il comportamento di ogni singola istruzione. Una volta che il controllo torna al chiamante, si provvede ad aggiungere all'*execution history* l'insieme di istanze di istruzioni intermedie che corrisponde alla *function summariza-*

```
1   0x08048374 <cp+00>:   push %ebp
2   0x08048375 <cp+01>:   mov %esp,%ebp
3   0x08048377 <cp+03>:   push %edi
4   0x08048378 <cp+04>:   sub $0x64,%esp
5   0x0804837b <cp+07>:   mov 0x8(%ebp),%eax
6   0x0804837e <cp+10>:   mov $0xffffffff,%ecx
7   0x08048383 <cp+15>:   mov %eax,0xffffffa8(%ebp)
8   0x08048386 <cp+18>:   mov $0x0,%eax
9   0x0804838b <cp+23>:   cld
10  0x0804838c <cp+24>:   mov 0xffffffa8(%ebp),%edi
11  0x0804838f <cp+27>:   repnz scas %es:(%edi),%al
12  0x08048391 <cp+29>:   mov %ecx,%eax
13  0x08048393 <cp+31>:   not %eax
14  0x08048395 <cp+33>:   sub $0x1,%eax
15  0x08048398 <cp+36>:   cmp $0xa,%eax
16  0x0804839b <cp+39>:   jbe 0x80483af <cp+59>
17  0x0804839d <cp+41>:   mov 0x8(%ebp),%eax
18  0x080483a0 <cp+44>:   mov %eax,0x4(%esp)
19  0x080483a4 <cp+48>:   lea 0xffffffac(%ebp),%eax
20  0x080483a7 <cp+51>:   mov %eax,(%esp)
21  0x080483aa <cp+54>:   call 0x80482d8 <strcpy@plt>
22  0x080483af <cp+59>:   add $0x64,%esp
23  0x080483b2 <cp+62>:   pop %edi
24  0x080483b3 <cp+63>:   pop %ebp
25  0x080483b4 <cp+64>:   ret
```

Figura 4.1: Disassemblato della funzione `cp()` di figura 3.1.

tion associata alla funzione appena invocata, avendo cura di attualizzare le espressioni utilizzate dalle istruzioni, in modo da adattarle all'esecuzione corrente.

Nel caso in cui il codice delle procedure di libreria utilizzate fosse gia incluso all'interno del file eseguibile (*static linking*), o non fosse possibile fare affidamento su simboli ed informazioni di rilocazione, il riconoscimento di tali funzioni non sarebbe più un problema banale, anche se già in parte risolto [6, 5]. Ad esempio, analizzando il disassemblato della funzione `cp()` del codice in figura 3.1 riportato in figura 4.1, si nota che non vi è nessuna chiamata o riferimento alla funzione `strlen()`, a differenza della funzione `strcpy()`.

Riconoscere quindi una funzione, ad esempio la `strlen`, comporta un aumento di complessità dell'analisi delle istruzioni. Ad esempio, se durante tale analisi ci si trova in presenza di una istruzione del tipo $if(EAX) > 10$, capire che la definizione di

EAX dipende da una applicazione della funzione `strlen()` risulta essere un lavoro oneroso. Bisogna quindi ricorrere ad una nuova tecnica, che riconosca con accuratezza le funzioni chiamate durante l'esecuzione, in modo da evitare di analizzare istruzioni che corrispondono a tali funzioni. La tecnica utilizzata consiste nel riconoscere le funzioni inline, ossia tentare di rilevare tali procedure tramite una ispezione del codice assembly. La spiegazione dettagliata di tale tecnica segue nel prossimo paragrafo.

4.1.1 Funzioni inline

Le funzioni di libreria tradotte inline sono funzioni che non hanno simboli di rilocazione associati, o nel caso di static linking dove le funzioni di libreria non hanno simboli associati. Il riconoscimento di funzioni inline è una tecnica che consente di rendere più efficiente e semplice l'analisi del binario. Consiste nel riconoscere una funzione di libreria da un insieme di istruzioni e sostituirle con una nuova istruzione che ne riassuma e descriva il comportamento. Le fasi che compongono tale tecnica sono le seguenti:

- generazione signature;
- creazione suffix tree;
- riconoscimento funzione inline.

Prima di tutto bisogna rilevare il set di istruzioni in codice assembly che identifica tali funzioni. Dopodiché si passa ad applicare una serie di trasformazioni alle singole istruzioni appartenenti all'insieme ottenuto, in modo da renderle generiche. Il risultato delle due operazioni sopra descritte è una *signature*, ossia un insieme di informazioni che identificano univocamente una singola funzione di libreria. Le singole *signature* verranno successivamente inserite all'interno di un database di *signature*, per rendere il processo di riconoscimento di tali funzioni il più generico possibile. Lo scopo è quello di utilizzare le *signature* ottenute per la creazione di una struttura dati che prende il nome di *suffixTree* [2], utilizzata per effettuare un veloce confronto con le istruzioni analizzate del file eseguibile, allo scopo di rilevare il tipo di funzioni sopra descritte.

Le varie fasi del processo di riconoscimento delle funzioni inline verranno descritte in dettaglio nelle prossime sezioni.

4.1.1.1 Generazione *signature*

Come visto nel paragrafo precedente, una *signature* non è altro che un insieme di istruzioni che identifica una funzione di libreria non riconoscibile tramite le informazioni di rilocazione. Associare un gruppo di istruzioni ad una funzione tipo non è un'operazione semplice, necessita la conoscenza del linguaggio assembly e uno studio manuale dei vari disassemblati usati per ricercare le istruzioni che identificano una data funzione.

Come esempio per comprendere meglio l'operazione di ricerca di tali istruzioni, verrà usata la funzione di libreria `strlen()`, in quanto la sua rispettiva traduzione in codice assembly non varia al variare del suo utilizzo, a differenza di alcune funzioni tipo la procedura `memcpy()`, la cui traduzione in codice assembly varia a seconda del numero di byte da copiare, e in alcuni casi non viene tradotta come insiemi di istruzioni ma come una chiamata a funzione. La fase di generazione delle *signature* avviene in vari passi, descritti di seguito:

Passo 1: il primo passo per generare una *signature* è ricavare l'insieme di istruzioni assembly che la caratterizzano. Come esempio di codice useremo la funzione `cp()` in Figura 3.1 e la sua rispettiva traduzioni in codice assembly riportata in Figura 4.1.

Si analizza il codice assembly ricavato dal sorgente della funzione di libreria estrapolando l'insieme di istruzioni che identificano la funzione ricercata, in poche parole si elimina il prologo e l'epilogo della procedura studiata. In questo caso l'insieme di istruzioni identifica la funzione `strlen()`. Dopodiché si ricerca il corrispettivo set di istruzioni all'interno del disassemblato contenente la funzione da rilevare. In questo caso quello in figura 4.1. Successivamente si passa ad analizzare il range di istruzioni segnalato, per determinare se il loro comportamento coincide con quello della funzione (in questo esempio coincide con la funzione *strlen()*). Il set di istruzioni ricavato viene riportato in figura 4.2

```
1  0x0804837e   mov $0xffffffff,%ecx
2  0x08048383   mov %eax,0xffffffa8(%ebp)
3  0x08048386   mov $0x0,%eax
4  0x0804838b   cld
5  0x0804838c   mov 0xffffffa8(%ebp),%edi
6  0x0804838f   repnz scas %es:(%edi),%al
7  0x08048391   mov %ecx,%eax
8  0x08048393   not %eax
9  0x08048395   sub $0x1,%eax
```

Figura 4.2: Set istruzioni assembly che identificano la funzione `strlen()`

Passo 2: Ottenute le istruzioni che identificano la funzione di libreria, bisogna ricercare la loro rispettiva traduzione in codice intermedio, in quanto le istruzioni con cui verranno confrontate sono in questa forma. Per fare ciò si esegue *smart fuzzer* sul codice di esempio e si cercano le istruzioni in forma intermedia il cui indirizzo corrisponde a quello delle istruzioni che identificano la funzione. Le istruzioni in forma intermedia rilevate vengono successivamente trasformate in modo da renderle generiche. In poche parole, vengono sostituiti gli operandi dell'istruzioni che risultano essere variabili con un operando indefinito ("$\perp$"). L'identificazione degli operandi variabili, si ottiene tramite lo studio delle istruzioni durante varie esecuzioni del *framework* sul file eseguibile. Ad esempio l'istruzione 08048383 $m32[(r32(EBP) + c32(-0x58))] := r32(EAX)$ diventa 00000000 $m32[(r32(EBP) + \perp)] := r32(EAX)$, ossia viene azzerato l'indirizzo e sostituita la parte variabile con "$\perp$".

Passo 3: Questo passo consiste nel creare un istruzione che riassumi il comportamento di una data funzione. Nel caso della procedura `strlen()`, l'istruzione riassuntiva risulta essere:

$$00000000 \quad r32(\perp) := (strlen\ m32[(r32(EBP) + \perp)])$$

Ossia il registro da 32 bit avente nome indefinito, corrisponde alla lunghezza della variabile composta dal registro a 32 bit `EBP` più un operando indefinito. Le parti indefinite dell'istruzione riassuntiva verranno sostituite in fase di rico-

noscimento delle funzioni inline. Tale istruzione, che prende il nome di "info", viene poi aggiunta all'insieme delle istruzioni descritte nel passo 2.

Passo 4: L'ultimo passo consiste nella creazione di un file che contenga le informazioni sopra descritte. Questo file verrà successivamente utilizzato per la creazione di una struttura dati che consentirà un veloce rilevamento delle funzioni che necessitano il riconoscimento inline.

In figura 4.3 viene riportata la *signature* costruita seguendo i passi sopra descritti. Per ragioni di spazio l'istruzione alla riga 9 è stata omessa, in quanto non risulta determinante per le considerazioni fatte. Si noti che tale *signature* vale per un unica funzione, in quanto sia le istruzioni in forma intermedia che l'info identificano univocamente la funzione `strlen`. Seguendo i passi sopra descritti per varie funzioni, e inserendo i risultati in un database è possibile creare una struttura che sia in grado di riconoscere funzioni di libreria diverse.

```
 1    00000000 r32(ECX) := c32(0xffffffff)
 2    00000000 m32[(r32(EBP) + None)] := r32(EAX)
 3    00000000 r32(EAX) := c32(0x0)
 4    00000000 r1(DF) := c1(0x0)
 5    00000000 r32(EDI) := m32[(r32(EBP) + None)]
 6    00000000 JUMP ((r32(ECX) == c32(0x0))) None
 7    00000000 r8(TMP) := (m32[(r16(ES) + r32(EDI))] - r8(AL))
 8    00000000 r1(ZF) := (r8(TMP) == c8(0x0))
 9    00000000 r32(EDI) := ...
10    00000000 r32(ECX) := (r32(ECX) + c32(-0x1))
11    00000000 JUMP ((r1(ZF) != c32(0x1))) None
12    00000000 r32(EAX) := r32(ECX)
13    00000000 r32(EAX) := (~ r32(EAX))
14    00000000 r32(None) := (r32(EAX) + c32(-0x1))
15    INFO 00000000 r32(None) := (strlen m32[(r32(EBP) + None)])
```

Figura 4.3: Signature funzione `strlen()`

È molto importante notare che la generazione di una signature è una operazione manuale, ossia non esiste un algoritmo in grado di eseguire le operazioni sopra descritte. Questo è un problema trascurabile in quanto la *signature* viene generata un unica volta, e riutilizzata ad ogni applicazione di *smart fuzzer*.

4.1.1.2 Creazione suffixTree

La seconda fase del riconoscimento di funzioni inline, consiste nell'utilizzare il database di *signature* creato per la costruzione di una struttura dati che prende il nome di "suffixTree". Più precisamente verranno utilizzate le singole *signature* contenute nel database per la costruzione dei rami dell'albero (suffixTree). A differenza della fase precedente, la realizzazione dell'albero avviene automaticamente tramite l'utilizzo dell'algoritmo riportato in figura 4.4.

Prima della descrizione dettagliata dell'algoritmo, verranno illustrate la struttura di un possibile nodo e di una possibile foglia dell'albero. Come si può notare dalla figura 4.5, un generico nodo è composto da una istruzione in forma intermedia e da una lista di istruzioni, che identificano i nodi successivi direttamente collegati ad esso. Mentre una generica foglia è un particolare nodo composto solo da un info, ossia un istruzione che riassume il comportamento di una funzione di libreria. Una foglia quindi non ha successori, ed identifica la fine di un ramo dell'albero.

Per quanto riguarda il nodo, il recupero delle informazioni avviene tramite due chiamate: `node.instruction()` e `node.nextNode()`. La prima ritornerà una singola istruzione, mentre la seconda una lista di nodi successivi (figli), caratterizzati dalle loro istruzioni identificanti. Per quanto riguarda la foglia invece, il recupero della sua unica informazione avviene tramite l'attributo `leaf.info`, che ritornerà l'istruzione riassuntiva, utilizzata per descrivere il comportamento di una data funzione. Sempre in figura 4.5 viene riportato un esempio di suffixTree formato da due rami. In tale esempio le istruzioni che identificano i nodi sono `A, B, C`, e i cammini possibili sono due. Quindi ipotizzando che l'insieme di istruzioni `(A, B)` e `(A, C)`, che caratterizzano ognuno un ramo dell'albero, identifichino rispettivamente una funzione di libreria, è possibile tramite un confronto con le istruzioni in forma intermedia appartenenti al file eseguibile, riconoscere e sostituire tali funzioni con l'info contenuto nelle foglie dei due rami.
Passiamo ora alla descrizione dettagliata dell'algoritmo in figura 4.4 utilizzato per la costruzione di tale struttura dati. Tale algoritmo è composto da due procedure: `findNode()` e `createBranch()`, connesse tra loro. L'input che l'algoritmo si

```
Input: instruction group set (signature)

procedure findNode(instruction, root, info, group)
begin
    if instruction == root.instruction then
        findNode(group[next instruction], root, info, group)
    else
        foreach node ∈ root.nextNode( ) do
            findNode(instruction, node, info, group)
    createBranch(instruction, root, info, group)
end

procedure createBranch(instruction, root, info, group)
begin
    tmp = Node()
    tmp.instruction = instruction
    tmp.nextNode = [ ]
    root.setNextNode(tmp)
    if (group[next instruction]) != dall'ultima istruzione) then
        node = tmp
        createBranch(group[next instruction], node, info, group)
    else
        leaf = Leaf()
        leaf.info = info
        tmp.setLeaf(leaf)
end

root = Node()
foreach group ∈ Signature do
    findNode(group[first instruction], root, group[info], group)
return root
```

Figura 4.4: Algoritmo creazione suffixTree

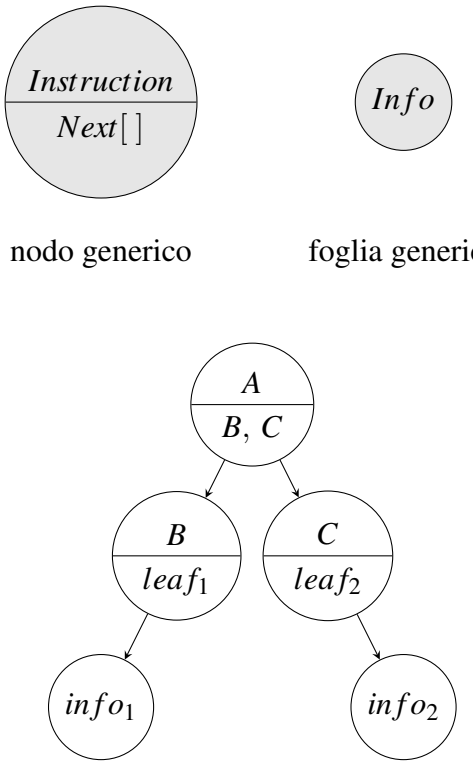

Figura 4.5: Esempio albero e struttura degli elementi

aspetta di ricevere è un file composto da una serie di liste (*signature*), contenenti ognuna un insieme di istruzioni in forma intermedia. Verranno ora descritte le procedure che formano l'algoritmo:

`Main`: la prima fase dell'algoritmo consiste nell'avviare la costruzione dell'albero. Prima di tutto crea un nodo radice inizialmente vuoto, dopodiché per ogni gruppo di istruzioni (*signature*) contenuto nel database (`Signature`), chiama la procedura `findNode()`. Tale procedura necessita come vedremo in seguito di alcuni parametri, quindi la sua invocazione avviene nel seguente modo:

$$findNode(group[first\ instruction],\ root,\ info,\ group)$$

Il valore di ritorno di tale procedura è un nodo, o più precisamente il nodo radice, che conterrà la lista dei nodi successivi. Si noti che la radice non ha un istruzione in forma intermedia che lo identifica e quindi non può esistere una foglia associata direttamente a questo nodo.

`findNode(instruction, root, info, group)`: questa procedura si prefigge di scoprire se un dato nodo è gia presente all'interno dell'albero. I parametri che tale procedura si aspetta sono in ordine: un istruzione in forma intermedia, il nodo radice, l'info del set di istruzioni che si sta analizzando e il gruppo stesso. Per ricavare quello che la procedura si prefigge di ottenere viene usata la ricorsione.

Prima di tutto effettua un controllo sul parametro istruzione. Ossia controlla se tale istruzione coincide con l'istruzione del nodo corrente. Durante la prima esecuzione tale nodo corrisponde alla radice, che come detto in precedenza non ha un'istruzione propria. Se tale confronto ha successo si richiama la procedura `findNode()` con i seguenti parametri: l'istruzione successiva del gruppo, la radice, l'info e il gruppo stesso. Altrimenti per ogni nodo appartenente alla lista dei nodi successivi del nodo in cui ci si trova, viene richiamata la procedura `findNode()` con i seguenti parametri: l'istruzione corrente, il nodo corrente, l'info e il gruppo di istruzioni correntemente esaminato. Al termine della procedura viene chiamata la procedura `createBranch()`. I parametri con cui tale procedura viene invocata variano a seconda del risultato ottenuto. Nel caso che sia stato trovato un nodo la cui istruzione coincida con quella cercata, i parametri selezionati saranno: l'istruzione da inserire, ossia quella successiva a quella trovata, il nodo da cui partire, l'istruzione riassuntiva e il gruppo di istruzioni a cui l'istruzione da inserire appartiene. Se non è stato trovato alcun nodo coincidente con l'istruzione cercata, i parametri selezionati saranno: l'istruzione non trovata e quindi da inserire, il nodo radice, l'info e il gruppo di istruzioni a cui l'istruzione da inserire appartiene.

`createBranch(instruction, root, info, group)`: Questa procedura viene utilizzata per la creazione di un ramo dell'albero. Anche la seguente

procedura come quella precedente si avvale della tecnica di ricorsione.
Prima di tutto viene creato un nodo temporaneo a cui viene associata l'istruzione derivante dai parametri, e una lista di nodi successivi vuota. Tale nodo viene poi inserito nella lista dei nodi successivi del nodo precedente, ossia quello derivante dai parametri. Dopodiché si controlla se l'istruzione associata al nodo temporaneo è l'ultima istruzione appartenente al gruppo. Se la risposta del controllo è positiva vuol dire che la costruzione del ramo deve essere conclusa con l'inserimento di una foglia. Di conseguenza l'algoritmo crea una foglia, le attribuisce l'istruzione riassuntiva e la aggiunge al nodo. Nel caso in cui la risposta del controllo risultasse essere negativa, il nodo temporaneo dovrà assumere il ruolo di nuovo nodo di partenza da cui ripartire a inserire i nodi e di conseguenza verrà richiamata la procedura `createBranch()` con i seguenti parametri: l'istruzione successiva, il nuovo nodo di partenza, l'info e il gruppo. La procedura termina con l'inserimento di una foglia.

Al termine di tale algoritmo quindi ci si trova in presenza di un albero i cui rami rappresentano ognuno una funzione di libreria. Ad esempio utilizzando la signature riportata in figura 4.3 come input per l'algoritmo appena descritto, otterremmo una albero composto da un singolo ramo, adibito al rilevamento della funzione di libreria `strlen()`. L'utilizzo della struttura descritta finora verrà esposto nel paragrafo seguente.

4.1.1.3 Riconoscimento funzione inline

In questa sezione verrà descritta la tecnica con la quale avviene il riconoscimento delle funzioni inline. Tale fase richiede la presenza della struttura dati sopra descritta e del control flow graph del file eseguibile analizzato. Per ogni *basic block* (vedasi capitolo 2) appartenente a tale flusso di controllo, vengono confrontate le singole istruzioni con quelle contenute nel suffixTree. Il confronto tra le istruzioni non avviene in maniera lessicografico, basandosi semplicemente sulle stringhe che le contengono, ma in maniera più approfondita, tramite l'utilizzo di vari controlli. Prima di tutto si verifica se la tipologia di istruzioni coincide, dopodiché si passa al con-

fronto dei singoli parametri. Ad esempio si consideri l'istruzione in forma intermedia $08048390r32(ECX) := (r32(ECX) - c32(0x1))$ e l'istruzione della *signature* in Figura 4.3 $00000000r32(ECX) := (r32(ECX) + c32(-0x1))$, si nota che entrambe sono istruzioni di tipo assegnamento, quindi si passa a confrontare i loro operandi. Prima di tutto le istruzioni verranno divise in due parti, destra e sinistra, dopodiché verranno analizzate a seconda della tipologia degli operandi. In questo caso la parte sinistra di entrambe le istruzioni corrisponde ad un registro, quindi si verificherà se tali operandi coincidono sia nella dimensione che nel nome del registro. Per quanto riguarda la parte destra la procedura risulta più complessa in quanto si tratta di due espressioni di tipo `EExpression` (vedasi capitolo 3). Prima di tutto viene normalizzata l'espressione contenuta nell'istruzione appartenente al file eseguibile, ottenendo l'espressione seguente: $(r32(ECX) + c32(-0x1))$. Dopodiché si verifica se gli operandi di entrambe le espressioni coincidono. Se questo accade, per ogni operando all'interno delle due espressioni si effettuano i controlli visti per la parte sinistra delle istruzioni, ossia si controlla se le tipologie degli operandi corrisponde e si analizza di conseguenza il loro contenuto. Seguendo il principio di confronto proposto, le due istruzioni d'esempio coincidono . Si noti che utilizzando un confronto banale tra le due stringhe, questo fallirebbe, in quanto non sarebbe in grado di determinare l'uguaglianza dell'espressioni contenute nelle istruzioni, oltre alla diversità degli indirizzi. Di seguito si riportano le tipologie di operandi analizzati dalla fase di confronto :

- `Register`: sono gli operandi di tipo registro, ad esempio $r32(EAX)$, dove il confronto consiste nel verificare se il nome e la dimensione del registro di entrambe le istruzioni corrispondono.

- `Constant`: sono operandi di tipo costante, ad esempio $c32(0xffffffff)$, dove il confronto consiste nel verificare se la dimensione della costante e il valore contenuto da tale operando, di entrambe le istruzioni corrispondono.

- `Memory`: sono operandi di tipo memoria, ad esempio $m32(ECX)$, dove il confronto consiste nel verificare se la dimensione della locazione di memoria e il valore da essa contenuto, di entrambe le istruzioni coincidono.

Le tipologie sopra descritte sono contenute all'interno di due tipi di istruzioni: espressioni semplici (Expression) ed espressioni composte (EExpression), vedasi capitolo 3. Altri tipi di istruzioni analizzate sono del tipo: `Return`, `Assignment`, `Jump`.

- `Return`: questi tipi di istruzioni si denotano con *RET* e il loro confronto è abbastanza semplice in quanto basta verificare che entrambe le istruzioni siano dello stesso tipo

- `Assignment`: sono istruzioni del tipo $A := B$. Il loro confronto consiste nell'analizzare la parte sinistra e la parte destra dell'espressioni contenute all'interno delle due istruzioni dello stesso tipo.

- `Jump`: sono istruzioni di salto. Il loro confronto consiste nell'analizzare le condizioni di salto contenute nelle istruzioni di tale tipo.

Verrà di seguito descritta la fase in cui vengono riconosciute le funzioni inline. tale fase si compone di 3 passi:

Passo 1: si confronta ogni istruzione del file eseguibile con ogni istruzione contenuta nella lista dei nodi successivi alla radice del suffixTree, seguendo la tecnica sopra descritta. Questa operazione continua fino al rilevamento di due istruzioni coincidenti o al termine delle istruzioni del file eseguibile.

Passo 2: rilevate due istruzioni coincidenti, si passa a confrontare l'istruzione successiva del file eseguibile con le istruzioni contenute nei nodi successivi. Se si rilevano due istruzioni coincidenti, si verifica se il nodo successivo corrisponde ad una foglia. In tale caso si prosegue con il passo 3. In caso contrario si prosegue ripetendo il passo 2. Nel caso in cui le istruzioni risultano differenti, il passo 2 termina e l'esecuzione ritorna al passo 1 con l'istruzione non coincidente.

Passo 3: l'istruzione contenuta all'interno della foglia, viene attualizzata, ossia vengono sostituiti gli operandi indefiniti con valori appropriati al contesto di esecuzione. Dopodiché viene creata una lista che conterrà l'istruzione attualizzata e

gli indirizzi delle istruzioni del file eseguibile che identificano la funzione rilevata. Tale lista servirà per la creazione delle path condition che fanno riferimento alle funzioni di libreria descritte in questo capitolo. Terminata questa procedura, l'esecuzione ritorna al passo 1 con l'istruzione successiva del file eseguibile.

Con questa fase si conclude il processo di riconoscimento di funzioni inline. L'utilizzo delle informazioni ottenute in quest'ultima fase verrà discusso nella prossima sezione.

4.2 Analisi delle *path condition*

L'analisi delle *path condition*, ossia espressioni che specificano la relazione tra input e cammino percorso nell'esempio corrente, è finalizzata alla trasformazione delle espressioni derivanti dall'analisi dei salti in vincoli sui dati d'ingresso. L'algoritmo utilizzato per dedurre le relazioni tra le condizioni di salto e dati di input, si basa sul risultato dell'analisi delle espressioni utilizzata per esprimere la condizione. Consiste nel propagare ricorsivamente le *reaching definition* per ogni variabile contenuta nell'espressione, fino al rilevamento di una connessione tra una variabile e i dati di input. Completata l'operazione precedente, si passa ad esaminare l'insieme dei vincoli ricavati, per individuare sottoinsiemi di condizioni astraibili in una rappresentazione più compatta, allo scopo di rendere l'insieme dei vincoli concetti di più alto livello, ad esempio *lunghezza di una stringa*. L'algoritmo che rileva la corrispondenza tra le istruzioni di salto e input viene riportato in figura 4.6. Tale algoritmo verrà spiegato in dettaglio nella prossima sezione.

4.2.1 Path condition per le funzioni inline

Per generare *path condition* riguardanti le funzioni inline, non basta analizzare le istruzioni singolarmente, in quanto le funzioni di libreria tradotte inline vengono identificate da un insieme di istruzioni. La soluzione a tale problema si ha tramite l'utilizzo delle informazioni ottenute dalla fase di riconoscimento delle funzioni inline, ossia la

Input: j, un'istanza di istruzione di assegnamento o di salto condizionato; $\mathscr{I}$, l'insieme delle variabili connesse ai dati di input; Γ, l'insieme degli indirizzi delle istruzioni appartenenti al file eseguibile identificanti una funzione inline; info, istruzione riassuntiva funzione inline

foreach $v \in use(getRhs(j))$ **do**
  if $v \in \mathscr{I}$ **then**
    non fare nulla, visto che l'espressione dipende già dai dati di input
  else if $\nexists\, r : r = \overline{drd}(v, j)$ **then**
    $replace(getRhs(j), v, PSS[v])$
  else
    if $\overline{drd}.getAddress() \in \Gamma$ **then**
      $drdTemp = RRR(\overline{drd}(v, j),\ indirizzo\ della prima\ istruzione \in \Gamma)$
      $j = \overline{drd}.setInstruction(info)$
      return j
    else
      $replace(getRhs(j), v, getRhs(PropagateDRD(\overline{drd}(v, j))))$

$replace(getRhs(j), getRhs(j), simplify(getRhs(j)))$

return j

procedure $RRR(drd, address)$
begin
  $addressTemp = indirizzo\ della\ drd\ precedente$
  if $addressTemp == address$ **then**
    return $drd\ precedente$
  else
    $RRR(drd\ precedente,\ address)$
end

Figura 4.6: Algoritmo *PropagateDRD*.

lista degli indirizzi delle istruzioni in forma intermedia che identificano una data funzione. L'algoritmo in figura 4.6 adibito a dedurre le relazioni tra condizioni di salto e input, si basa principalmente sull'analisi delle espressioni utilizzata per esprimere la condizione, già trasformata dalle fasi precedenti. Le funzioni utilizzate dall'algoritmo sono le seguenti:

- $drd(v, s_k)$, restituisce la *reaching defintion* dinamica della variabile v valutata in corrispondenza dell'istanza di istruzione s_k.
- *getAddress()*, restituisce l'indirizzo di un'istruzione.
- *setInstruction(i)*, definisce un istruzione i.
- *getRhs(i)*, restituisce il lato destro dell'istruzione i.
- *replace(g,e,e')*, sostituisce nell'espressione g tutte le occorrenze della sottoespressione e con e'.

L'algoritmo si compone di varie fasi. Prima di tutto si controlla se esiste una relazione tra i dati di input e le variabili contenute nella parte destra dell'istruzione j. Se tale controllo risulta positivo, avviene la terminazione dell'algoritmo. In caso contrario si controlla se esiste una *dynamic reaching definition* della variabile v dell'istruzione j analizzata. Se tale drd esiste, si verifica se l'indirizzo dell'istruzione contenuta fa parte del set di istruzioni che identificano una funzione inline. Tale verifica si compone di tre passi:

Passo 1: Se l'indirizzo dell'istruzione contenuta nella drd corrisponde ad uno di quelli appartenenti alla lista ottenuta nella fase di rilevamento delle funzioni inline, viene invocata la procedura `RRR` (recursive reverse research), nel seguente modo:

$$RRR(\overline{drd}(v, j),\ indirizzo\ della\ prima\ istruzione \in \Gamma\ )$$

In caso contrario continua l'esecuzione dell'algoritmo sostituendo la parte destra dell'istruzione con ogni occorrenza della sottoespressione contenente la variabile v con la *dynamic reaching definition* di v.

Passo 2: in questo passo analizziamo la procedura *RRR*. Tale procedura si occupa di risalire alla *dynamic reaching definition* che contiene la prima istruzione del gruppo al quale appartiene l'info recuperato. Il motivo per il quale bisogna risalire a tale informazione è la sostituzione del blocco di istruzioni che identificano la funzione tradotta inline appartenenti al file eseguibile, con l'istruzione riassuntiva (info). Tale procedura ritornerà quindi una drd.

Passo 3: si assegna il risultato della procedura spiegata al passo 2 ad una variabile temporanea `drdTemp`. Dopodiché si assegna tramite la funzione `setInstruction()` l'istruzione riassuntiva a tale variabile. L'algoritmo termina con il ritorno di tale variabile.

Se non esiste una drd per tale espressione l'algoritmo continua sostituendo la parte destra dell'istruzione con l'occorrenza di v del suo *PSS*. Ad esempio si consideri l'espressione $r32(EAX) \implies (0xa)$, con cui è stata tradotta la condizione dell'istruzione di salto del quarto blocco in Figura 3.2. Supponendo che esista gia una *dynamic reaching definition* per questa espressione, viene valutata la $drd(r32(EAX), j)$. Tale definizione esiste e corrisponde ad un'istanza della 3 istruzione del medesimo blocco. L'indirizzo di tale istruzione corrisponde ad uno degli indirizzi delle istruzioni che identificano la funzione inline `strlen()`. Quindi dopo essere risaliti alla drd contenente la prima istruzione del gruppo al quale appartiene l'info recuperato, l'intero blocco viene sostituito con $r32(EAX) := (strlen\ m32[(r32(EBP) + c32(-0x58))])$.

4.2.2 Esempio di path condition

Supponendo di eseguire l'applicazione riportata in Figura 3.1 con input "abc", l'analisi rileva che la condizione di riga 11 dipende dall'input, ritornando la *path condition* seguente:

$$PC = \emptyset \text{ [}path\ condition\text{ inizialmente vuoto]}$$
$$PC' = PC \cup (m8[c32(0xbfffffa9e)] != c8(0x6b))$$

Ossia la zona di memoria a 8 bit avente inizio all'indirizzo specificato nella costante a 32 bit risulta diversa da 0x6b, cioè il primo carattere dell'input è diverso da 'k'.

Eseguendo il file eseguibile con l'input proposto si è in grado di guidare l'esecuzione dell'applicazione verso porzioni di codice privi di vulnerabilità. La *path condition* ricavata esplicita la condizione con la quale si è in grado di esplorare tale ramificazione. Si utilizzano le informazioni ottenute per la creazione di un nuovo input che sia in grado di esplorare porzioni di codice non ancora analizzate, e quindi si riesegue l'applicazione con il nuovo input proposto. Supponendo di utilizzare come input "kbc", l'insieme di *path condition* ricavato è il seguente:

$$PC'' = PC \cup (m8[c32(0xbffffaae)] == c8(0x6b)) \wedge$$
$$PC''' = PC'' \cup (m8[c32(0xbffffaa0)] != c8(0x7a))$$

Ossia la zona di memoria a 8 bit avente inizio all'indirizzo specificato nella costante a 32 bit risulta uguale a 0x6b, cioè il primo carattere dell'input uguale a 'k' e la zona di memoria a 8 bit avente inizio all'indirizzo specificato nella costante a 32 bit risulta diverso da 0x7a, cioè il terzo carattere dell'input è diverso da 'z'. Dopo aver esplorato tramite il nuovo input nuove porzioni di codice, non si è stati ancora in grado di rilevare alcuna vulnerabilità. Si utilizzano le informazioni ottenute per la creazione di un nuovo input che sia in grado di esplorare porzioni di codice non ancora analizzate, e quindi si riesegue l'applicazione con il nuovo input proposto. Supponendo di utilizzare come input "kbz", l'insieme di *path condition* ricavato è il seguente:

$$PC'''' = PC'' \cup (m8[c32(0xbffffab0)] == c8(0x7a)) \wedge$$
$$PC\,''''' = PC'''' \cup (strlen\ m32[c32(0xbffffa9e)])) <= c32(0xa)$$

Ossia la zona di memoria a 8 bit avente inizio all'indirizzo specificato nella costante a 32 bit risulta uguale a 0x7a, cioè il terzo carattere dell'input è uguale a 'z', e la lunghezza della stringa contenuta nella zona di memoria a 32 bit è minore di 10, cioè la lunghezza dell'input, intesa come numero di caratteri è minore di 10. Rieseguendo l'applicazione utilizzando le informazioni ottenute, quindi con input uguale a "kbzaaaaaaaa", viene rilevata una vulnerabilità di tipo *buffer overflow*, in quanto è possibile sovrascrivere l'indirizzo $bffff8ac$ in 88 iterazioni.

Capitolo 5

Conclusioni

Nel presente lavoro di tesi viene presentato un approccio per il rilevamento automatico di vulnerabilità all'interno di codice eseguibile. Il modello proposto si avvale di tecniche di *program analysis* ibride statico-dinamiche. Le analisi vengono svolte sul codice binario, in modo tale da applicare il modello anche su applicazioni per le quali non è disponibile il codice sorgente. Il modello presentato prende il nome di *smart fuzzing*, in quanto si basa su tecniche di *fuzzing* tradizionali. L'approccio classico consiste nell'utilizzare come input di una data applicazione dati totalmente casuali, allo scopo di causare errori di esecuzione. Il modello proposto, integra l'approccio classico analizzando il programma e monitorando l'esecuzione, al fine di estrapolare un insieme di input in grado di canalizzare il flusso di esecuzione di un'applicazione in stati in cui è più probabile che una vulerabilità si manifesti, tramite tecniche statico-dinamiche.

Il presente lavoro di tesi si concentra principalmente sullo studio delle analisi dinamiche effettuate dal *framework* di *smart fuzzing*. Il contributo principale consiste nell'integrazione della fase di analisi delle *path condition* con una tecnica per il rilevamento di funzioni inline (ad esempio la funzione di libreria `strlen()`). Al momento, il *framework* studiato è in grado di effettuare tutte le analisi descritte nel presente lavoro di tesi, individuando diversi tipi di vulnerabilità. I possibili sviluppi futuri prevedono un miglioramento computazionale degli algoritmi proposti, allo scopo di rendere più efficiente e scalabile l'intero processo, riducendo il pesante *overhead* introdotto da alcune delle analisi effettuate. Un ulteriore sviluppo futuro riguarda il *porting* del

modello studiato su altre piattaforme, in quanto per il momento *smart fuzzer* funziona esclusivamente in ambiente *Linux*.

Per concludere, il modello studiato risulta essere un'innovazione nell'ambito della sicurezza informatica, in quanto i vari modelli esistenti presentano pesanti inefficienze introdotte da assunzioni fatte durante il loro sviluppo. Nel complesso si ritiene che il modello *smart fuzzing* rappresenta un valido punto di partenza da estendere e sviluppare ulteriormente nel prossimo futuro.

Bibliografia

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic Slicing in the Presence of Unconstrained Pointers. In *TAV4: Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 60–73, New York, NY, USA, 1991. ACM Press.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] Aleph One. Smashing The Stack For Fun And Profit. *Phrack Magazine*, 7(49), 1996.

[5] M. V. Emmerik. Identifying Library Functions in Executable Files Using Patterns. In *ASWEC '98: Proceedings of the Australian Software Engineering Conference*, page 90, Washington, DC, USA, 1998. IEEE Computer Society.

[6] I. Guilfanov. Fast Library Identification and Recognition Technology, 1997. [Online; ultimo accesso 28-Febbraio-2007].

[7] Klog. The Frame Pointer Overwrite. *Phrack Magazine*, 9(55), 1999.

[8] A. Lanzi, L. Martignoni, M. Monga, and R. Paleari. A smart fuzzer for x86 executables. In *SESS'07: Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*, Minneapolis, MN, USA, May 2007. ACM.

[9] J. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

[10] Michel Kaempf. Smashing The Heap For Fun And Profit. *Phrack Magazine*, 11(57), 2001.

[11] Python Software Foundation. The Python Programming Language. [Online; ultimo accesso 28-gennaio-2007].

[12] Scut, Team Teso. Exploiting Format String Vulnerabilities. March 2001.

[13] Wikipedia. Corner case — Wikipedia, The Free Encyclopedia, 2006. [Online; ultimo accesso 28-Febbraio-2007].

[14] T. W. Yellman. Failures and related topics. 48:6–8, Mar. 1999.