UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA MAGISTRALE IN TECNOLOGIE DELL'INFORMAZIONE
E DELLA COMUNICAZIONE

ANALISI DINAMICA DI CODICE BINARIO

Relatore: Dott. Mattia MONGA
Correlatore: Dott. Andrea LANZI
Dott. Lorenzo MARTIGNONI

Tesi di Laurea di:
Roberto PALEARI
Matricola 685471

Anno Accademico 2005–06

Ai miei genitori

Ringraziamenti

Mi sembra doveroso incominciare la tesi ringraziando le tante persone che mi hanno supportato (...e *sopportato*) in questi ultimi anni.

Innanzi tutto ringrazio il Dott. Mattia Monga, il Dott. Lorenzo Martignoni (per avermi insegnato, tra l'altro, quanto si possa fare con "20 righe di Python"...) e il Dott. Andrea Lanzi, per avermi dato la possibilità di svolgere questa tesi e soprattutto per avermi dato una *grande* mano nel portarla avanti. Ringrazio il Dott. Lorenzo "Gigi Sullivan" Cavallaro, per essersi sempre interessato al mio lavoro e per i consigli in occasione di SESS. Grazie anche all'Ing. Silvio Ranise, per gli utili suggerimenti ed il tempo dedicatomi.

Un grazie di cuore ai miei genitori, per avermi dato così tanto e non aver mai chiesto nulla in cambio (...tenete duro eh! Prima o poi qualcosa "in cambio" arriverà!).

Naturalmente un grosso grazie anche a Laura, per essere riuscita a sopportarmi anche quando sono stato particolarmente intrattabile (...però mai come te quando sei "sotto esame"!).

Un ringraziamento particolare a tutti gli "hacker" del laboratorio LaSER (rigorosamente in ordine alfabetico): Alessandro, Dario, Davide "Brown", Emanuele, Gianpaolo "Gianz" e Luca, per tutti i consigli, le discussioni, i CTF e per la bella accoglienza che mi hanno riservato.

Concludo con un grazie a tutti i miei amici, in particolar modo ai Wild Fire (Alessandro, Fabio e Francesco), per essere stati capaci di convivere con le mie *performance* di quest'ultimo periodo.

Indice

Capitolo 1

Introduzione

Verificare la correttezza di un sistema software è un problema complesso e ancora oggi oggetto di ricerca. Mediamente, un programma informatico contiene dai cinque ai quindici difetti [89] per mille righe di codice sorgente [22]. La maggior parte di questi difetti non hanno un impatto rilevante sulla generale operatività dell'applicazione, in quanto si manifestano solamente in particolari situazioni estranee al normale flusso di esecuzione. Ciascuno di questi difetti, anche se minore, potrebbe però avere la potenzialità di compromettere la sicurezza dell'*intero* sistema. A questo si aggiunge che spesso l'unico strumento utilizzato per cercare di individuare la maggiore quantità possibile di difetti in un prodotto software è il *beta testing*: l'applicazione viene distribuita ad un limitato gruppo di utenti i quali utilizzano il programma in ambienti diversi e comunicano eventuali problemi riscontrati al produttore, che provvederà a correggerli prima del rilascio della versione definitiva. Sempre più spesso, però, le esigenze di mercato e la necessità di acquisire o mantenere il vantaggio competitivo sui concorrenti spingono gli sviluppatori a rilasciare versioni dei propri prodotti software ancora in uno stato "embrionale", affidando direttamente agli utenti il compito di sperimentare in modo più esaustivo il prodotto.

Questi fattori fanno sì che il panorama attuale veda in uso applicazioni molto complesse, poco sperimentate e quindi contenenti svariati difetti, ai quali i produttori cercano, più o meno regolarmente, di porre rimedio tramite la distribuzione agli utenti di aggiornamenti (*patch*) che si propongono di risolvere i problemi del prodotto (natural-

mente senza alcuna garanzia che non siano le stesse *patch* ad introdurre nuovi difetti). Come già accennato, però, alcuni di questi difetti possono costituire una *vulnerabilità* dell'applicazione, consentendo ad un eventuale attaccante di compromettere il sistema.

1.1 Motivazioni del lavoro

Nell'ambito della sicurezza, sono molte le soluzioni che cercano di prevenire, rilevare o risolvere i problemi descritti in precedenza; una di queste soluzioni è l'*analisi delle vulnerabilità*. Per analisi delle vulnerabilità si intende quel processo che consiste nell'analisi di un sistema allo scopo di determinare se esso contiene dei difetti che possono essere sfruttati da un attaccante al fine di compromettere la sicurezza del sistema stesso. Si tratta quindi di un processo fondamentalmente diverso rispetto ad altri approcci alla sicurezza quali il rilevamento o la prevenzione di intrusioni, in quanto si pone come principale obiettivo l'identificazione dei difetti in modo tale che questi possano poi essere corretti, mentre gli altri approcci si concentrano sul rilevamento o la prevenzione di *attacchi* che sfruttano le vulnerabilità derivanti da tali difetti.

La complessità in gioco è però notevole: un sistema operativo come Windows 2000 comprende tra i 35 e i 60 milioni di righe di codice; la ricerca manuale dei difetti di un prodotto di tali dimensioni non è praticamente possibile. Sono quindi indispensabili strumenti in grado di individuare in modo *automatico* potenziali problematiche di sicurezza nel software. Tali strumenti risultato certamente utili allo sviluppatore del software, al fine di incrementare la qualità del proprio prodotto, ma possono risultare utili anche all'acquirente del prodotto, interessato a non introdurre nei propri sistemi degli applicativi che possano costituire un punto di ingresso per un eventuale attaccante. In generale, quindi, non è possibile assumere di avere a disposizione il codice sorgente dell'applicazione, ma occorre operare sul codice binario generato in seguito alla compilazione del sorgente. Ciò complica notevolmente l'analisi, che non potrà prescindere da tutta una serie di questioni legate alla perdita di informazioni di alto livello durante il processo di compilazione [17], tra le quali: distinzione tra dati e istruzioni; esplosione della dimensione e della complessità del codice a causa dell'inserimento, da parte di compilatore e linker, di un considerevole numero di procedure

di servizio; ottimizzazioni introdotte in fase di compilazione; ricostruzione dei predicati condizionali di alto livello partendo dalle espressioni di basso livello presenti nel codice binario; riconoscimento dei costrutti di programmazione (blocchi, cicli, ...), offuscati dal processo di compilazione.

A questo si aggiunge il fatto che dimostrare formalmente l'assenza di difetti in un programma informatico mediamente complesso è un problema notoriamente indecidibile, in quanto riconducibile al problema dell'arresto [82]. Tutti gli strumenti che si pongono come obiettivo l'individuazione automatica di vulnerabilità nel software devono quindi necessariamente scontrarsi con i limiti teorici connessi a tale problematica. La disponibilità del solo codice binario non può far altro che complicare ulteriormente le cose.

Partendo da questi presupposti, molti sono gli approcci che affrontano la questione in modo approssimativo (*best-effort*) o comunque introducendo alcune semplificazioni. Il presente lavoro di tesi si concentra su uno di tali approcci: il rilevamento di vulnerabilità tramite l'applicazione di tecniche di *program analysis* direttamente sul codice binario di un programma. In particolare, il metodo descritto nei capitoli successivi si rivolge all'individuazione automatica di input per l'applicazione che siano in grado di sovrascrivere zone di memoria delicate con valori controllabili da un potenziale attaccante. Un attacco di questo tipo costituisce il principio fondamentale di tecniche di *exploiting* comunemente utilizzate, quali *stack-based buffer overflow* [5] o *heap overflow* [59].

Sebbene il recente successo degli applicativi web abbia permesso di assistere all'esplosione di nuove tipologie di attacchi informatici (SQL injection, XSS, ...) notevolmente più semplici da realizzare rispetto ai precedenti, i *memory attack* continuano a rivestire un ruolo di primo piano, testimoniato dalle oltre 50 vulnerabilità di questo tipo riportate dal sito CVE (*Common Vulnerabilities and Exposures*) nel solo mese di Gennaio 2007.

1.2 Idea

Il lavoro di tesi consiste nello studio di alcune tecniche di *program analysis* (con particolare attenzione per l'analisi dinamica) e della loro applicabilità al codice binario. In particolare, è stata progettata un'infrastruttura per l'analisi di codice binario allo scopo di individuare le potenziali vulnerabilità di un'applicazione software.

Il metodo presentato prende il nome di *smart fuzzing*: mentre nel *fuzzing* tradizionale [61] l'applicazione viene eseguita su input casuali verificando se vengono generate condizioni anomale, con l'approccio *smart fuzzing* il programma viene analizzato e la sua esecuzione monitorata; in tale modo si cerca di determinare un insieme di input in grado di condurre il flusso di esecuzione in stati potenzialmente pericolosi, ovvero in cui una vulnerabilità potrebbe manifestarsi con maggiore probabilità.

L'approccio di analisi adottato non è puramente dinamico, bensì ibrido: innanzi tutto l'applicazione viene analizzata staticamente, allo scopo di ottenere una panoramica del suo comportamento; successivamente il programma viene monitorato dinamicamente, raffinando le informazioni precedentemente raccolte e cercando di costruire un insieme di vincoli sui dati in ingresso che, una volta soddisfatti, definiscono un nuovo input in grado di provocare la manifestazione di una vulnerabilità del programma.

1.3 Organizzazione della tesi

Il lavoro di tesi è organizzato come segue.

Il Capitolo 2 fornisce le nozioni preliminari necessarie alla comprensione del lavoro. Vengono analizzati i possibili approcci alla *program analysis* insieme con i concetti e le tecniche fondamentali.

Nel corso del Capitolo 3 viene affrontato il problema della ricerca delle vulnerabilità nelle applicazioni software, soffermandosi più da vicino sulle problematiche specifiche all'analisi di codice binario. Viene inoltre presentato un semplice esempio di programma vulnerabile che sarà poi richiamato nei successivi capitoli al fine di mostrare come viene applicato l'approccio *smart fuzzing*. Sono infine definiti i principali obiettivi del lavoro di tesi.

L'architettura del modello di analisi proposto è discussa nel corso del Capitolo 4, dove è analizzata separatamente ciascuna fase prevista. Vengono evidenziati i problemi incontrati nel corso dell'analisi e le soluzioni proposte.

Il Capitolo 5 ripercorre in maniera più dettagliata le fasi del modello, approfondendone alcuni aspetti rilevanti e presentando gli algoritmi utilizzati.

Si prosegue con il Capitolo 6, dove sono discussi alcuni dettagli e limitazioni circa l'implementazione del prototipo realizzato allo scopo di sperimentare la metodologia di analisi proposta. Il capitolo si conclude introducendo alcune idee per possibili sviluppi futuri del lavoro.

Nel corso del Capitolo 7 vengono presentate alcune metodologie di analisi alternative, finalizzate in modo specifico all'individuazione delle vulnerabilità di un programma informatico oppure focalizzate sul problema più generale dell'individuazione automatica dei difetti presenti in un'applicazione software.

Il lavoro di tesi è poi concluso dal Capitolo 8.

Capitolo 2

Concetti preliminari

Il presente capitolo inizia con l'introduzione dei concetti fondamentali di *program analysis* e con la presentazione degli approcci possibili, evidenziando per ciascuno pregi e difetti. Vengono poi discussi altri argomenti indispensabili per la comprensione del lavoro di tesi.

2.1 Program analysis

Con il termine *program analysis* si fa riferimento ad un insieme di tecniche che consentono di analizzare in modo *automatico* un programma informatico, al fine di dedurne alcune proprietà.

Si tratta di metodologie tradizionalmente associate all'ambito dei compilatori, dove sono da tempo utilizzate allo scopo di generare codice per quanto possibile compatto e performante [3]. I compilatori attuali sono infatti in grado di analizzare il codice sorgente di un programma per estrarre informazioni che consentono, ad esempio, di determinare se un'istruzione di assegnamento può essere omessa, perché magari la variabile definita non viene poi utilizzata (*dead code elimination*); oppure per dedurre se un'istruzione può essere estratta dal corpo di un ciclo per inserirla nell'intestazione dello stesso, in modo da ridurre il numero di istruzioni eseguite ad ogni iterazione (*code motion*); o ancora per sostituire operazioni computazionalmente complesse con qualcosa di più efficiente ma semanticamente equivalente (*strength reduction*).

Recentemente, sono stati rivolti notevoli sforzi all'impiego di tecniche di *program analysis* per la verifica del software, come ad esempio l'individuazione di difetti nell'implementazione degli algoritmi o la ricerca di problematiche rilevanti dal punto di vista della sicurezza.

Sono possibili differenti classificazioni delle diverse tecniche di *program analysis*. Ci si concentra ora su due di queste classificazioni: la prima si focalizza su *quando* viene effettuata l'analisi, mentre la seconda si concentra sull'*oggetto* dell'analisi.

2.1.1 Analisi statica e analisi dinamica

In base a *quando* viene effettuata l'analisi del programma, possiamo tradizionalmente distinguere due approcci principali: analisi *statica* e analisi *dinamica*. Si tratta di metodologie nate in comunità differenti e che hanno per molto tempo seguito percorsi paralleli, senza mai incontrarsi. Solo di recente questa tendenza ha incominciato a venire meno, portando alla nascita delle prime analisi ibride statico-dinamiche.

2.1.1.1 Analisi statica

L'analisi statica è in grado di esaminare il codice (sorgente o binario) di un programma ed estrapolare informazioni circa *tutti* i possibili comportamenti che il software manifesterà al momento dell'esecuzione [67]. Le caratteristiche fondamentali dell'analisi statica sono due:

- *corretta* (*sound*) – i risultati dell'analisi descrivono fedelmente il comportamento del programma, in modo indipendente dall'input e quindi dalla specifica esecuzione.

- *Conservativa* – le proprietà dedotte dall'analisi sono, in linea di principio, più *deboli* di quanto siano in realtà. Per chiarire meglio questo concetto si consideri un semplice esempio. Sia f una funzione definita come $f(x) = x^2, \forall x \in \mathscr{R}$. Allora l'affermazione A : "*f restituisce un numero positivo*" è certamente vera, ma più debole di B : "*f restituisce il quadrato del suo argomento*". In particolare,

è facile osservare che $B \implies A$. Osservando f, un'analisi conservativa potrebbe dedurre l'affermazione A, o addirittura qualcosa di più generico, del tipo "f *restituisce un numero*".

Il conservativismo di questa tipologia di analisi è legato al fatto che problematiche quali *aliasing* e indirezione non sono completamente risolvibili staticamente [54].

Il classico campo di applicazione delle tecniche di analisi statica è quello delle ottimizzazioni del codice al momento della compilazione [23], ma attualmente tali metodologie vengono anche utilizzate per verifiche formali sulla correttezza del software, naturalmente limitate dalle imprecisioni dell'analisi.

2.1.1.2 Analisi dinamica

L'analisi dinamica estrae le informazioni cercate tramite l'esecuzione del programma e l'osservazione del suo comportamento. È facile notare come vantaggi e svantaggi di questo approccio siano fondamentalmente complementari rispetto a quelli dell'analisi statica, infatti si è soliti affermate che l'analisi dinamica gode delle seguenti proprietà:

- *precisa* – proprio perché le informazioni ricavate dall'analisi derivano dall'osservazione dell'esecuzione del programma, ne segue che l'analisi dinamica non soffre dei problemi che spingono l'analisi statica verso un approccio conservativo ed approssimato. Dinamicamente, ad esempio, non è necessario compiere complessi procedimenti per dedurre il valore assunto da una variabile in un certo punto del programma: è sufficiente eseguire il programma fino a quel punto e osservare il valore che la variabile assume.

- Relativa ad una *specifica* esecuzione – le informazioni ricavate dinamicamente non possono essere generalizzate fino a comprendere tutte le possibili esecuzioni, poiché dipendono dalla particolare configurazione dell'input che ha caratterizzato l'esecuzione oggetto dell'analisi.

Le tecniche di analisi dinamica, proprio perché fornendo risultati relativi ad una specifica esecuzione producono informazioni meno generali rispetto a quelle offerte

dall'analisi statica, sono state tradizionalmente denigrate dalla comunità dei linguaggi di programmazione. Infatti, ambiti come l'ottimizzazione del codice a *compile-time* richiedono trasformazioni capaci di preservare la semantica del programma, non potendo quindi accettare la fragilità dell'analisi dinamica. Attualmente, però, l'approccio dinamico è stato ripreso e largamente utilizzato in altri settori, quali, ad esempio, testing, debugging e profiling, dove, piuttosto che la solidità, è la precisione delle informazione raccolte ad assumere un peso determinante.

2.1.1.3 Sinergie

Dalle considerazioni fatte in precedenza su vantaggi e svantaggi dei due approcci, emerge quanto l'analisi statica e quella dinamica siano tra loro complementari: l'analisi statica è solida e conservativa, mentre l'approccio dinamico offre risultati più precisi ma relativi alla singola esecuzione.

In generale, quindi, le due tecniche possono essere considerate complementari sotto diversi punti di vista [10]:

completezza. L'analisi dinamica è in grado di generare "*invarianti dinamici*" [28], ovvero proprietà vere per ciascuna delle esecuzioni osservate. L'analisi statica potrebbe essere in grado di confermare o meno se tali invarianti dinamici valgono invece per ogni esecuzione. Non è comunque da escludere che i due approcci arrivino a risultati discordanti, causati da una delle due seguenti possibilità: (I) gli invarianti generati dall'analisi dinamica non sono astraibili all'intera applicazione, poiché è stato osservato un numero di esecuzioni troppo ridotto; (II) la discordanza è legata all'eccessivo conservativismo dell'analisi statica (ad esempio, perché vengono considerati cammini di esecuzione in realtà non percorribili).

Profondità. Le tecniche dinamiche concentrano la propria attenzione su un singolo cammino di esecuzione, riuscendo però a dedurre dipendenze semantiche tra entità considerevolmente distanti. Al contrario, l'analisi statica ha spesso un ristretto campo d'azione, riuscendo ad analizzare in modo efficiente e sufficientemente approfondito solamentente cammini di esecuzione piuttosto brevi.

Precisione. Uno dei principali punti di forza dell'analisi dinamica è la possibilità di osservare concretamente l'esecuzione del programma, mentre le tecniche statiche sono costrette a ragionare su un'astrazione delle possibili esecuzioni. Tale astrazione può però costituire un conveniente punto di partenza da cui procedere dinamicamente.

Una tecnica di analisi ibrida può quindi combinare gli aspetti positivi di entrambe le tipologie di analisi, ottenendo spesso risultati migliori che non applicando singolarmente uno dei due approcci. Una fase di analisi statica preliminare potrebbe, ad esempio, fornire una visione globale del comportamento dell'applicazione, poi in seguito raffinata dall'analizzatore dinamico.

In particolare, l'approccio ibrido appena descritto costituisce la soluzione adottata per la realizzazione del presente lavoro di tesi. Come si vedrà più dettagliatamente in seguito, sebbene gli sforzi siano stati concentrati prevalentemente sulla sperimentazione di nuove tecniche di analisi dinamica, si è ritenuto comunque opportuno ricorrere ad un'analisi statica preliminare dell'applicazione, in modo ottenere un insieme di informazioni approssimative ma utili per guidare le successive fasi di monitoraggio dinamico del programma.

2.1.2 Analisi del sorgente e analisi di codice binario

Una classificazione alternativa delle diverse tecniche di *program analysis* può essere effettuata spostando l'attenzione sull'*oggetto* dell'analisi. Distinguiamo quindi le due categorie seguenti.

2.1.2.1 Analisi del sorgente

L'analisi del programma è effettuata a livello del codice sorgente, potendo così accedere in modo semplice e diretto ad elementi quali i costrutti del linguaggio di programmazione (funzioni, istruzioni, espressioni, variabili, ...) e avendo a disposizione informazioni di alto livello (ad esempio, il concetto di "blocco" di codice). Si tratta di una tipologia di analisi del tutto indipendente dalla piattaforma sottostante: analizzare un programma C scritto per un sistema Intel x86 non differisce affatto dal compiere

le stesse analisi su un programma per piattaforma Alpha, a patto, naturalmente, che l'applicazione sia scritta seguendo la sintassi C standard. Evidentemente, però, l'analisi dipende dal linguaggio di alto livello utilizzato per la scrittura del programma: un analizzatore per codice C non potrà essere impiegato per esaminare un programma scritto in un altro linguaggio. Il problema di fondo è legato al fatto che linguaggi di alto livello diversi fanno solitamente riferimento ad astrazioni diverse. Si pensi, ad esempio, alle differenti astrazioni fornite dai linguaggi C e Prolog.

2.1.2.2 Analisi di codice binario

Nel caso dell'analisi di codice binario, l'oggetto dell'analisi è il codice macchina prodotto in seguito ai processi di compilazione e linking. La differenza fondamentale tra questo approccio e il precedente risiede nel fatto che in questo caso gli elementi su cui viene effettuata l'analisi sono certamente di più "basso livello" rispetto a quelli osservabili esaminando il codice sorgente: qui l'analisi è costretta a trattare *entità macchina* come istruzioni, registri del processore e locazioni di memoria.

Così come nel caso della precedente classificazione, anche qui i vantaggi dell'analisi di codice binario corrispondono agli svantaggi delle tecniche di analisi basate sul sorgente e viceversa: trattando codice macchina si ha l'indubbio vantaggio di non dipendere dal particolare linguaggio di alto livello utilizzato per la scrittura dell'applicazione, mentre non si potrà prescindere dalla piattaforma hardware sottostante, che determina, ad esempio, l'insieme delle istruzioni e dei registri disponibili e le modalità di indirizzamento possibili. In alcuni casi, inoltre, il codice sorgente potrebbe non essere disponibile.

Si rimanda alla Sezione 3.3 per una discussione più dettagliata circa le peculiarità dell'analisi di codice binario e le conseguenze connesse all'adozione di tale approccio per individuare le vulnerabilità presenti in un'applicazione software.

2.2 Concetti fondamentali di *program analysis*

2.2.1 Cenni di teoria dei grafi

Formalmente, un *grafo orientato* (o diretto) [6] G può essere rappresentato da una coppia $G = (B,E)$, dove $B = \{b_1, b_2, \ldots, b_n\}$ è un insieme di nodi (o blocchi), mentre $E = \{(b_i, b_j), (b_k, b_l), \ldots\}$ è un insieme di archi orientati. Ogni arco orientato è indicato tramite una coppia ordinata (b_i, b_j) di nodi (non necessariamente distinti) che indica che l'arco va dal nodo b_i (*coda* dell'arco) al nodo b_j (*testa* dell'arco). È quindi possibile definire una funzione Γ_G che associa ad ogni nodo un insieme, eventualmente vuoto, di *successori immediati*: $\Gamma_G(b_i) = \{b_j \mid (b_i, b_j) \in E\}$. Invertendo la funzione successore otteniamo ${\Gamma_G}^{-1}$ che definisce invece i *predecessori immediati* di un nodo: ${\Gamma_G}^{-1}(b_j) = \{b_i \mid (b_i, b_j) \in E\}$.

Un *sottografo* di un grafo orientato $G = (B,E)$ è a sua volta un grafo diretto $G' = (B',E')$, dove $B' \subset B$ e $E' \subset E$. Inoltre, la relazione $\Gamma_{G'}$ deve "restare in G'", nel senso che:

$$\forall {b_i}' \in B',\ \Gamma_{G'}({b_i}') = \{{b_j}' \mid ({b_i}', {b_j}') \in E'\}.$$

Un *cammino* in un grafo orientato G è un sottografo P con nodi ordinati ottenuti da applicazioni successive della funzione successore Γ_G. È quindi esprimibile come una sequenza di nodi $(b_1, b_2, \ldots, b_n)$, dove $b_{i+1} \in \Gamma_P(b_i)$. I lati sono impliciti: $(b_i, b_{i+1}) \in E$. Nodi e lati non sono necessariamente unici. P si dice *chiuso* se $b_n = b_1$.

La *lunghezza* $\delta(P)$ di un cammino P è il numero di lati compresi in P. In modo più formale si può affermare che dato un cammino $P = (b_1,\ b_2,\ \ldots,\ b_n)$, allora $\delta(P) = n-1$.

Dato un grafo diretto $G = (B,E)$, un nodo $q \in B$ si dice *successore* di un nodo $p \in B, p \neq q$, se esiste un cammino P in G tale che $P = (b_1, b_2, \ldots, b_n)$ con $p = b_1$ e $q = b_n$. Nella medesima situazione, si dice anche che p è *predecessore* di q. Per indicare la relazione di successore e predecessore tra q e p si scriverà $q \in succ(p)$ e $p \in pred(q)$.

In un grafo orientato, un nodo privo di predecessori è chiamato *nodo iniziale* (*entry node*), mentre definiamo *nodo terminale* (*exit node*) un nodo caratterizzato dall'assen-

za di successori. In alcune situazioni, come nel caso dei *control flow graph* descritti nella sezione seguente, la presenza di più nodi iniziali può risultare problematica. Per questo motivo spesso si ricorre all'introduzione di un nodo iniziale aggiuntivo α che sia predecessore immediato di tutti i nodi iniziali del grafo, ovvero:

$$\Gamma_G(\alpha) = \{b_i \in B \mid b_i \textit{ nodo iniziale}\} \ \wedge \ {\Gamma_G}^{-1}(\alpha) = \emptyset.$$

Allo stesso modo, è spesso utile disporre di un grafo caratterizzato da un singolo nodo terminale. Anche in questo caso, se il grafo orientato in esame dispone di più nodi terminali, è possibile introdurre un nodo terminale aggiuntivo ω successore immediato di tutti i nodi terminali del grafo:

$${\Gamma_G}^{-1}(\omega) = \{b_j \in B \mid b_j \textit{ nodo terminale}\} \ \wedge \ \Gamma_G(\omega) = \emptyset.$$

Una *componente fortemente connessa* di un grafo orientato $G = (B,E)$ è un sottografo $G' = (B',E')$ di G tale che:

$$\forall {b_i}' \in B', \ \forall {b_j}' \in B', \ \exists P_{i,j} = ({b_i}', \ldots, {b_j}')$$

ovvero esiste un cammino da ogni nodo di G' ad ogni altro nodo del sottografo. Un cammino chiuso costituisce un caso particolare di componente fortemente connessa.

2.2.1.1 Dominatori

Sia ora $G = (B,E)$ un grafo orientato con un unico nodo iniziale α ed un unico nodo terminale ω. Si ricorda che nel caso in cui il grafo di partenza disponga di più nodi iniziali o più nodi terminali, si può ricorrere al procedimento descritto in precedenza per ottenere un grafo con un singolo *entry node* ed un singolo *exit node*.

Si dice che il nodo b_i *domina* il nodo b_k se b_i è compreso in ogni cammino da α a b_k. Per indicare la relazione di dominanza verrà utilizzata la notazione proposta in [21] e si scriverà $b_i \gtrdot b_k$. Inoltre b_i *domina strettamente* b_k se $b_i \gtrdot b_k \wedge b_i \neq b_k$ e si scriverà $b_i \gg b_k$.

Si consideri l'insieme $\mathscr{P} = \{P \mid P = (\alpha, \ldots, b_k)\}$, ovvero $\mathscr{P}$ comprende tutti e

soli i cammini da α a b_k. Si può quindi definire l'*insieme dei dominatori BD* del nodo b_k come:

$$BD(b_k) = \{b_i \mid b_i \neq b_k \ \wedge \ \forall P \in \mathscr{P}, b_i \in P\}.$$

Il *dominatore immediato* del nodo b_k è il dominatore "più vicino" a b_k, ovvero quel nodo $b_i \in BD(b_k)$ tale che:

$$\delta_{min}(b_i, b_k) <= \delta_{min}(b_j, b_k), \ \forall b_j \in BD(b_k).$$

Se b_i è dominatore immediato di b_k si scrive $b_i = idom(b_k)$.

Si dimostra che ogni nodo $b_k \in B$, con $b_k \neq \alpha$, ammette uno ed un solo dominatore immediato b_i. Infatti, se esistessero due dominatori b_i e b_i' si avrebbe che $\delta_{min}(b_i, b_k) = \delta_{min}(b_i', b_k)$. Ciò comporterebbe però che $b_i = b_i'$, oppure che b_i e b_i' appartengono a due cammini distinti da α a b_k. Ma visto che per definizione un dominatore deve comparire in ogni cammino $(\alpha, \ \ldots, \ b_k)$, ne segue che b_i e b_i' devono necessariamente coincidere. Inoltre, ogni nodo b_k deve ammettere almeno un dominatore, dato che α è banalmente in ogni cammino $P \in \mathscr{P}$.

Un modo possibile per rappresentare le relazioni di dominanza tra i nodi di un grafo orientato è costituito dall'*albero di dominanza*. L'albero di dominanza di un grafo $G = (B, E)$ con nodo iniziale α è un albero con radice α il cui insieme di archi orientati è un sottoinsieme di E definito come:

$$\{(idom(b), b) \mid b \in (B - \{\alpha\})\}.$$

Quindi, dato l'albero di dominanza del grafo G e dati due nodi $b_i, b_k \in B$, si ha che b_i domina b_k se e solo se b_i è un antenato proprio di b_k nell'albero di dominanza.

In [69] viene descritto un semplice algoritmo per determinare le relazioni di dominanza fra i nodi di un grafo orientato. L'algoritmo si basa sul calcolo, per ogni nodo $b \neq \alpha$ del grafo, dell'insieme di nodi S_b tali che $\forall v \in S_b, \exists P = (\alpha, \ldots, v), b \notin P$. Generato tale insieme, i nodi in $B - \{\alpha\} - S_b$ sono tutti e soli i nodi di B dominati da b. Una volta costruiti gli insiemi dei nodi dominati da ciascun vertice del grafo, è piuttosto semplice costruire l'albero di dominanza. Siano $n = |B|$, $m = |E|$, allora

si verifica facilmente che il calcolo di uno degli insiemi S richiede tempo $O(m)$ e che tale operazione viene eseguita $n-1$ volte. La complessità dell'intero algoritmo risulta essere quindi $O(n \cdot m)$. In [55] è presentato un algoritmo più efficiente, che consente, nella sua implementazione più semplice, di determinare le relazioni di dominanza in tempo $O(m \cdot \log(n))$.

Con *frontiera di dominanza* di un nodo b_k di $G = (B,E)$ si intende il sottoinsieme di nodi $DF(b_k) \subset B$ tale che $\forall\, b_i \in DF(b_k)$, b_k domina un predecessore di b_i ma b_k non domina strettamente b_i, ovvero:

$$DF(b_k) = \{b_j \in B \mid \exists b_i \in pred(b_j), b_k \gtrdot b_i \wedge b_k \not\gg b_j\}$$

Invece che utilizzare direttamente la definizione sopra riportata per calcolare la frontiera di dominanza di ciascun nodo del grafo orientato G, in [21] è riportato un algoritmo in grado di effettuare la computazione in tempo $O(|E|+|B|^2)$ nel caso peggiore, ma in tempo lineare nel caso medio.

Viene ora introdotto il concetto di *post-dominanza* [21, 30]. Un nodo b_j si dice *post-dominare* un nodo $b_k \neq b_j$ se ogni cammino orientato $(b_k, \ldots, \omega)$ include b_j, ovvero b_j è presente in ogni cammino da b_k fino al nodo terminale del grafo orientato. Se b_j post-domina b_k si scriverà $b_j \; pdom \; b_k$. Così come nel caso della dominanza, si dice che b_j *post-domina strettamente* b_k se $b_j \; pdom \; b_k \wedge b_j \neq b_k$, mentre si definisce *post-dominatore immediato* di un nodo b_k il suo post-dominatore b_j "più vicino".

Si osservi che il calcolo delle relazioni di post-dominanza tra i nodi del grafo orientato $G = (B,E)$ si traduce nella computazione delle relazioni di dominanza tra i nodi del grafo $G^{-1} = (B',E')$ ottenuto *invertendo* [21] G, ovvero:

$$\begin{aligned} B' &= B \setminus \{\alpha, \omega\} \cup \{\alpha', \omega'\}, \;\; \omega' = \alpha \wedge \alpha' = \omega; \\ E' &= \{(b_j, b_i) \mid (b_i, b_j) \in E\}. \end{aligned}$$

G^{-1} include quindi tutti e soli i nodi di G, ma i nodi iniziale e terminale di G hanno in G^{-1} ruolo invertito. Gli archi di G^{-1} sono invece ottenuti semplicemente invertendo gli archi di G.

Il concetto di post-dominanza risulta particolarmente utile per introdurre la nozione di *dipendenza di controllo* (*control dependency*) tra due nodi b_j e b_k di G. In particolare, b_j si dice essere *control dependent* da b_k se valgono entrambe le seguenti condizioni [30]:

1. esiste un cammino $P = (b_k, \ldots, b_j)$ in G tale che b_j post-domina ogni nodo oltre a b_k in P;

2. b_j non post-domina b_k.

In altre parole, ciò significa che esiste un arco uscente da b_k che consente di raggiungere b_j ma esiste anche un arco uscente da b_k che non conduce a b_j. Nel seguito per indicare che b_j ha una dipendenza di controllo da b_k si utilizzerà la notazione $b_j \xrightarrow{c} b_k$.

2.2.1.2 Cicli

Dato un grafo orientato $G = (B,E)$, si definisce *ciclo* [40] un insieme $L \subset B$ tale da costituire una componente fortemente connessa di G.

Dato un ciclo L, chiamiano *nodo di ingresso* del ciclo ogni nodo $b_{entry} \in L$ tale che $\exists b \in {\Gamma_G}^{-1}(b_{entry}), b \notin L$, ovvero un nodo di ingresso di L è un nodo del ciclo con un predecessore immediato esterno al ciclo stesso. Inoltre i nodi di ingresso di L sono tali che supponendo $b_j \notin L$ e $b_k \in L$, allora $\forall P = (b_j, \ldots, b_k)$ esiste un nodo di ingresso b_{entry} tale che $b_{entry} \in P$; ciò significa che ogni cammino diretto da un nodo esterno al ciclo verso un nodo interno deve necessariamente attraversare un nodo di ingresso. Un ciclo che non contiene altri cicli è chiamato *ciclo interno*.

Una particolare tipologia di cicli è costituita dai *cicli naturali*, caratterizzati da due proprietà fondamentali:

1. il ciclo ha un singolo nodo di ingresso, chiamato *intestazione*. L'intestazione del ciclo domina tutti i nodi del ciclo, poiché, in caso contrario, non sarebbe l'unico nodo di ingresso.

2. Deve esistere almeno un cammino che consente di iterare nel ciclo.

Un modo per individuare i cicli naturali all'interno di un grafo orientato consiste nel ricercare gli archi il cui nodo di testa domina il nodo di coda. Archi con tale caratteristica prendono il nome di *back edge*. Dato un back edge (b_i, b_j) definiamo il ciclo naturale associato a tale arco come l'insieme di nodi che comprende b_j e tutti i nodi b_k dai quali è possibile raggiungere b_i senza passare dal nodo b_j. Dalle considerazioni appena fatte deriva l'algoritmo presentato in [3] per la costruzione dell'insieme dei nodi che costituiscono il ciclo naturale associato ad un dato *back edge*. Tale algoritmo è discusso nella sezione 5.4.

2.2.1.3 Grafi riducibili

Un grafo orientato $G = (B, E)$ con un unico nodo iniziale α ed un unico nodo terminale ω si dice *riducibile* [41, 80] se e solo se è possibile partizionare l'insieme E in due sottoinsiemi disgiunti chiamati archi "*in avanti*" (*forward edge*) e archi "*all'indietro*" (*back edge*) che godono delle seguenti proprietà:

1. l'insieme dei *forward edge* forma un grafo orientato aciclico in cui ogni nodo è raggiungibile da α, ovvero dal nodo iniziale di G.

2. L'insieme dei *back edge* è costituito da archi (b_i, b_k) tali che $b_k \gg b_i$.

In Figura 2.1 è riportato un esempio di grafo riducibile. Se si considera l'insieme di archi $\{(a,b),(b,c)\}$, il grafo diretto che si ottiene risulta aciclico, con tutti i nodi raggiungibili a partire dal nodo iniziale a. L'arco rimanente (c,b) è invece un *back edge*, dato che la sua testa b domina la coda a.

Il grafo orientato di Figura 2.2 è invece irriducibile. In generale, se conosciamo la relazione di dominanza fra i nodi del grafo possiamo determinare se un grafo diretto è riducibile rimuovendo i *back edge* e verificando se i nodi rimanenti costituiscono un grafo aciclico. L'esempio in figura, però, non ha *back edge*, dato che b non domina c ma nemmeno c domina b. Quindi affinché il grafo sia riducibile dovrebbe essere interamente aciclico, ma ciò non si verifica.

Una proprietà fondamentale dei grafi riducibili è che ogni insieme di nodi che possa essere informalmente considerato un ciclo deve necessariamente includere un *back*

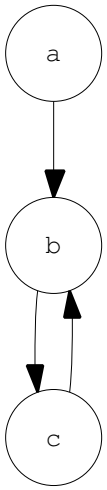

Figura 2.1: Un grafo riducibile.

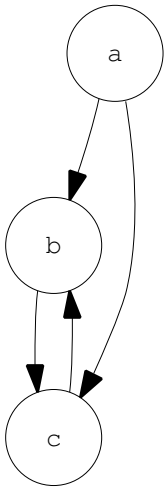

Figura 2.2: Un grafo irriducibile.

edge. Per individuare tutti i cicli di un grafo riducibile è quindi sufficiente identificare i *back edge* e determinare l'insieme dei nodi che costituiscono il ciclo naturale associato a quel particolare arco.

2.2.2 Control flow graph

Un *control flow graph* (CFG) è una struttura dati fondamentale per qualsiasi tecnica di *program analysis*. Sostanzialmente un CFG è una rappresentazione di tutti i cammini che possono essere percorsi durante l'esecuzione di un programma.

In modo più formale, possiamo definire un CFG come un grafo orientato i cui nodi rappresentano computazioni mentre gli archi indicano come il flusso di esecuzione possa spostarsi da una computazione all'altra. I nodi di un CFG prendono il nome di *basic block* e rappresentano una sequenza non interrompibile di istruzioni, con un singolo punto di ingresso (la prima istruzione che viene eseguita) ed un unico punto di uscita (l'ultima istruzione eseguita). Quando il flusso di controllo dell'applicazione raggiunge un basic block, provoca l'esecuzione della prima istruzione del blocco; le istruzioni successive sono eseguite in ordine, senza possibili interruzioni o salti al di fuori del blocco finché l'ultima istruzione non è stata raggiunta.

Si avrà che il basic block b_1 è predecessore immediato del basic block b_2 se b_2

segue b_1 in una qualche esecuzione, ovvero se si verifica una delle seguenti condizioni [3]:

1. l'ultima istruzione di b_1 è un salto condizionato o incondizionato alla prima istruzione di b_2.

2. b_2 segue immediatamente b_1 nel codice del programma e b_1 non termina con un'istruzione di salto.

Il concetto di *nodo terminale* è facilmente trasferibile ai CFG e corrisponde ai basic block in cui la computazione si arresta. Per semplicità d'esposizione, in presenza di più nodi terminali si suppone comunque di ricorrere all'introduzione di un nodo terminale artificiale ω successore di tutti i nodi terminali del grafo. Non è altrettanto lineare l'estensione della nozione di *nodo iniziale*. In un grafo orientato, infatti, il nodo iniziale è semplicemente un nodo privo di predecessori, mentre nel caso di un CFG si preferisce considerare come nodo iniziale il basic block che contiene la prima istruzione del programma o della procedura rappresentata, ovvero il blocco contenente l'*entry point* della computazione descritta dal grafo. Può però accadere che tale nodo costituisca il punto di partenza di un cammino chiuso, avendo quindi almeno un predecessore. In situazioni di questo tipo è comunque possibile supporre di poter introdurre un nodo iniziale arbitrario α che sia predecessore immediato di tutti i nodi iniziali del grafo, così come descritto nella Sezione 2.2.1.

Per partizionare una sequenza di istruzioni di un programma Q in basic block si può ricorrere al procedimento riportato in [3] e descritto qui di seguito.

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

   (a) la prima istruzione del programma è un *leader*.

   (b) Ogni istruzione che rappresenta la destinazione di un trasferimento di controllo condizionato o assoluto è un *leader*.

   (c) Ogni istruzione che segue sintatticamente un'istruzione di trasferimento di controllo condizionato o assoluto è un *leader*.

```
1  int find_max(unsigned int *v, unsigned int n)
2  {
3      unsigned int max, i;
4      int ris;
5
6      if(n <= 0)
7          ris = -1;
8      else {
9          max = v[0];
10         i = 1;
11         while(i < n) {
12             if(v[i] > max)
13                 max = v[i];
14             i = i + 1;
15         }
16         ris = max;
17     }
18     return ris;
19 }
```

Figura 2.3: Codice C della procedura `find_max`.

2. Per ogni *leader*, il suo basic block consiste nelle istruzioni comprese tra il *leader* stesso incluso e il *leader* sintatticamente successivo escluso. Eventualmente, se il *leader* in esame è l'ultimo del programma, allora il basic block si estende fino alla fine del programma.

A titolo esemplificativo, in Figura 2.3 è riportato il codice sorgente in linguaggio C di una procedura che determina il massimo elemento di un vettore di interi fornito come parametro. La funzione riceve in ingresso il vettore `v` da esaminare ed un intero `n` contenente il numero di elementi presenti in `v`. Se `v` è vuoto, allora `find_max` restituirà al chiamante `-1`, mentre in caso contrario incomincerà assumendo come massimo il primo elemento del vettore (riga 9) e successivamente procederà ad esaminare singolarmente ogni elemento di `v` (ciclo alle righe 11-15). Se verrà individuato un elemento maggiore di quello finora considerato come massimo (riga 12), allora il nuovo massimo diventerà l'elemento in esame (riga 13). Al termine del ciclo, la variabile

`max` conterrà effettivamente il massimo elemento del vettore che potrà quindi essere restituito al chiamante.

Applicando il procedimento per l'individuazione dei basic block appena descritto, si ottiene una rappresentazione del *control flow graph* della funzione simile a quella riportata in Figura 2.4. Il basic block `B1` è il nodo iniziale del CFG, dato che comprende l'*entry point* della procedura `find_max`, mentre l'unico nodo terminale è costituito dal basic block `B9`. Ogni basic block che termina con un'istruzione di salto condizionato avrà due archi in uscita: uno percorso dal flusso di esecuzione nel caso in cui la condizione valutata dal salto risulti vera, l'altro percorso quando la condizione risulta falsa. Appare chiaro, inoltre, come il ciclo *while* alle righe 11-15 del codice sorgente corrisponda ai basic block $\{$`B4`,`B5`,`B7`,`B8`$\}$: tale insieme costituisce appunto una componente fortemente connessa del grafo, in quanto partendo da un qualunque blocco dell'insieme è possibile individuare un cammino che conduce ad ogni altro blocco del ciclo. Inoltre, dato che tale ciclo dispone di un unico nodo di ingresso (`B4`) ed esiste più di un cammino che consente di iterare nel ciclo, ne segue che il costrutto *while* costituisce un ciclo naturale connesso al back edge (`B4`,`B8`).

2.2.3 Elementi di analisi del *data flow*

L'analisi del *data flow* è una particolare tipologia di *program analysis* molto comune, in cui le informazioni cercate vengono generate esaminando i *control flow graph* associati al programma o alla procedura in esame. Vengono qui introdotti alcuni concetti fondamentali propri dell'analisi del *data flow* e largamente utilizzati nei capitoli seguenti.

Se un'istruzione i utilizza in qualche modo il valore della variabile x si dice semplicemente che i *usa* x. Una *definizione* di una variabile x è invece un'istruzione i che assegna, o *potrebbe* assegnare, un valore ad x. Un'istruzione di assegnamento del tipo $x = \langle \ldots \rangle$ oppure un'istruzione che legge un valore da un dispositivo di I/O e lo memorizza nella variabile x sono chiaramente definizioni *non ambigue* di x. Esistono poi definizioni *ambigue* di x, ovvero istruzioni per le quali non è possibile determinare *staticamente* se definiscono la variabile. Per fare alcuni esempi, definizioni ambigue

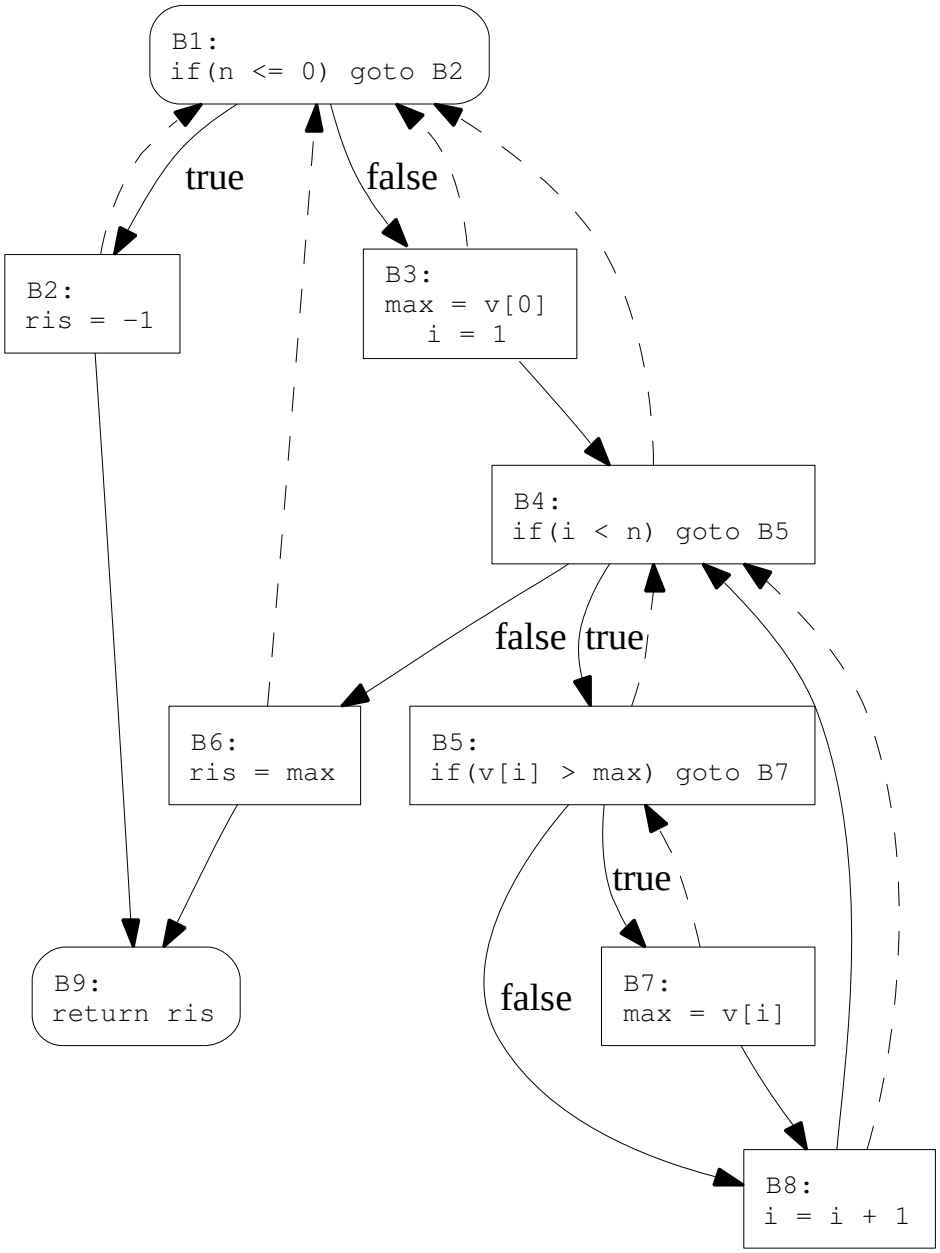

Figura 2.4: *Control flow graph* della procedura `find_max` di Figura 2.3.

di x potrebbero essere:

1. una chiamata a procedura con x come parametro; se la variabile è passata per riferimento e non per copia, allora non è possibile sapere se verrà modificata dalla procedura, a meno naturalmente di analizzare in modo dettagliato le istruzioni che costituiscono il corpo della funzione. La stessa situazione si verifica se x non è passata come parametro alla procedura ma potrebbe comunque venire modificata perché all'interno del campo di visibilità della funzione (ad esempio, se x è una variabile globale).

2. Un assegnamento attraverso una variabile puntatore che potrebbe fare riferimento ad x. Ad esempio, l'assegnamento $*w := y$ può definire x se w può puntare ad x.

Di seguito per indicare l'insieme delle variabili definite dall'istruzione i si scriverà $define(i)$, mentre con $use(i)$ si farà riferimento all'insieme delle variabili utilizzate dall'istruzione i.

Una variabile x si dice essere *live* [3] in corrispondenza di un certo punto p del CFG G se il valore di x è utilizzato lungo qualche cammino in G con origine in p. In caso contrario, x si dice essere *dead* al punto p. Si rimanda alla Sezione 5.2 per una discussione su come sia possibile calcolare le variabili *live* all'ingresso e all'uscita di un basic block.

Risulta infine utile introdurre il concetto di *grafo delle chiamate* o *call graph*. Si tratta semplicemente di un grafo orientato $C = (R, F)$ i cui nodi rappresentano le procedure del programma in esame, mentre gli archi indicano relazioni del tipo *chiamante-chiamato*. Più in dettaglio, la presenza di un arco $(r_i, r_j) \in F$ indica che nel corso dell'esecuzione della procedura r_i si ha almeno una chiamata alla procedura r_j.

2.2.3.1 Punti e cammini

All'interno di un basic block con n istruzioni si possono individuare $n+1$ *punti*, di cui uno immediatamente prima della prima istruzione del blocco (*punto iniziale* del

blocco), $n-1$ compresi tra due istruzioni consecutive ed uno immediatamente dopo l'ultima istruzione del blocco (*punto terminale* del blocco).

Un *cammino* dal punto p_1 al punto p_n è una sequenza di punti $(p_1,\ldots,p_n)$ tali che per ogni i compreso in $[1,\ldots,n-1]$ vale una delle seguenti possibilità:

1. p_i precede immediatamente un'istruzione e p_{i+1} segue immediatamente la stessa istruzione nello stesso blocco.

2. p_i è il punto terminale di un blocco e p_{i+1} è il punto iniziale di un altro blocco successore immediato del primo.

Un cammino da p_i a p_j si dice *libero da definizioni* (*definition-clear*) rispetto ad una variabile v se non comprende alcuna definizione non ambigua della variabile v.

2.2.3.2 Reaching definition

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. Il tal caso si dice che d costituisce una *reaching definition* per x al punto p.

Nell'ambito dell'analisi statica, una formulazione del concetto di *reaching definition* come quella appena presentata lascia spazio a possibili imprecisioni: si potrebbe in alcuni casi affermare che d raggiunge p mentre, in realtà, ciò non si verifica a causa della presenza di definizioni ambigue lungo il cammino dal punto che segue d a p. Adottando una definizione come quella riportata nel paragrafo precedente, si assume un atteggiamento puramente conservativo, in quanto si è in grado di garantire che trasformazioni applicate sulla base dei risultati ottenuti dall'analisi non alterino accidentalmente la semantica del programma a causa di imprecisioni dell'analisi stessa.

Quando calcolate staticamente, le *reaching definition* relative alla variabile v valutate in corrispondenza dell'istruzione i verranno nel seguito indicate con $srd(v,i)$. Nella maggior parte dei casi, i verrà identificata dal corrispondente numero di riga nel

codice sorgente del programma. Visti i problemi di ambiguità, l'atteggiamento conservativo appena sottolineato e, soprattutto, vista la necessità di fornire risultati che siano validi per tutte le possibili configurazioni dell'input, $srd(v,i)$ non potrà sempre coincidere con un insieme di cardinalità unitaria, ma spesso comprenderà più istruzioni che definiscono v.

Procedendo dinamicamente, invece, non si verificano problematiche legate a definizioni multiple o ambigue e ad ogni uso di una variabile è possibile associare una sola *reaching definition*, corrispondente all'effettiva istanza di istruzione che, nel corso dell'esecuzione in esame, ha definito la variabile osservata. In particolare, sia $G = (B,E)$ il *control flow graph* associato ad un certo programma Q. Allora definiamo *execution history* [1] un cammino in G rappresentato come sequenza di *istanze di istruzioni* relative ad una specifica esecuzione di Q. Formalmente, possiamo quindi definire una *execution history* come:

$$EH = \langle s_1, s_2, \ldots, s_n \rangle.$$

Nella definizione appena riportata, un'istanza di istruzione è indicata con $s = i^t$, dove i è un'istruzione del programma mentre l'indice t è utilizzato per distinguere occorrenze diverse della stessa istruzione. Facendo riferimento ad una *execution history* EH si definisce la *reaching definition dinamica* [1] $drd_{EH}(v,s_k)$ di una variabile v utilizzata da un'istanza di istruzione s_k come l'istanza di istruzione s_{k-j} tale che:

$$s_{k-j} \in EH \ \wedge \ v \in define(s_{k-j}) \ \wedge \ \nexists s_m, m \in (k-j,k) : v \in define(s_m).$$

Per calcolare $drd_{EH}(v,s_k)$ è sufficiente attraversare EH a ritroso partendo dall'istanza di istruzione s_k; non appena viene incontrata un'istanza di istruzione che definisce v, questa costituisce la *reaching definition* cercata.

Si supponga $s_k = i^t$. Allora è importante sottolineare come $drd_{EH}(v,s_k)$ corrisponda ad una *singola istanza* di istruzione, mentre, nel caso delle *reaching definition* statiche, $srd(v,i)$ è un *insieme di istruzioni* che può anche avere cardinalità maggiore di uno.

Per comprendere le differenze esistenti tra il caso statico e quello dinamico, si consideri ancora l'esempio di Figura 2.3 e si supponga di dover valutare le *reaching*

definition per la variabile *ris* quando utilizzata dall'istruzione alla riga 18. Le *reaching definition* statiche saranno date dall'insieme:

$$srd(ris, 18) = \{7, 16\}.$$

Dinamicamente, invece, la *reaching definition* dipenderà dalla specifica esecuzione e, di conseguenza, dalla *execution history* associata a tale esecuzione. Supponendo di eseguire la procedura di Figura 2.3 con input $\{\mathtt{v} = [5,9,3], \mathtt{n} = 3\}$ allora la *execution history* corrispondente sarà:

$$EH = \{6^1, 9^1, 10^1, 11^1, 12^1, 13^1, 14^1, 11^2, 12^2, 14^2, 16^1, 18^1\}.$$

La *reaching definition* dinamica per la variabile *ris* in corrispondenza dell'istanza di istruzione 18^1 sarà quindi:

$$drd_{EH}(ris, 18^1) = 16^1.$$

2.2.3.3 Control e data dependency

Siano i_j, i_k due istruzioni di un programma Q e siano $v \in use(i_k)$ e $i_j \in srd(v, i_k)$. Allora l'istruzione i_k si dice essere *data dependent* da i_j e (i_j^v, i_k) costituisce una *coppia di definizione-uso* rispetto alla variabile v. Analoghe considerazioni possono essere fatte relativamente al caso dinamico: se s_j e s_k sono due istanze di istruzioni di Q e $v \in use(s_k) \wedge s_j = drd(v, s_k)$, allora s_k è *data dependent* da s_j e (s_j^v, s_k) costituisce una coppia di definizione-uso. Nel seguito indicheremo con $i_k \xrightarrow{d} i_j$ la relazione di *data dependency* tra le istruzioni i_k e i_j; in modo analogo, $s_k \xrightarrow{d} s_j$ indicherà la dipendenza dell'istanza di istruzione s_k dall'istanza di istruzione s_j.

Si definisce *data dependence graph* statico di un programma Q un grafo orientato (V, E_d) dove V è l'insieme delle istruzioni di Q, mentre l'insieme di archi E_d riflette le *data dependency* esistenti tra le istruzioni. In particolare avremo che:

$$E_d = \{(i_j, i_k) \mid i_j, i_k \in V \wedge i_j \xrightarrow{d} i_k\}.$$

Naturalmente anche per il caso dinamico vale una definizione del tutto analoga, con l'unica differenza che l'insieme V comprenderà tutte le istanze di istruzioni di Q.

Si considerino ancora due istruzioni i_j e i_k di un programma Q, con i_j un'istruzione che valuta un predicato condizionale c_j e altera il flusso di esecuzione in funzione del risultato della valutazione di c_j. Allora i_k si dice essere *control dependent* da i_j se i_j è l'istruzione condizionale più prossima a i_k tale che il fatto che i_k venga o meno eseguita dipende dal risultato della valutazione di c_j [30]. Indichiamo con $i_k \xrightarrow{c} i_j$ la relazione di *control dependency* tra le istruzioni i_k e i_j. Dinamicamente si avrà che un'istanza di istruzione s_k avrà una dipendenza di controllo nei confronti di un'istanza s_j se s_j è la più vicina istanza di istruzione condizionale in grado di determinare se s_k verrà o meno eseguita.

Così come nel caso delle *data dependency*, anche per le dipendenze di controllo statiche è possibile rappresentare le relazioni esistenti tra le istruzioni di un programma Q mediante un grafo orientato (V, E_c), dove V è l'insieme delle istruzioni di Q mentre possiamo definire l'insieme degli archi del grafo come:

$$E_d = \{(i_j, i_k) \mid i_j, i_k \in V \wedge i_j \xrightarrow{c} i_k\}.$$

Anche questa volta, nel caso dinamico V sarà l'insieme delle istanze di istruzioni di Q e l'insieme degli archi orientati E_d definirà le relazioni esistenti tra un'istanza di istruzione e l'istanza di istruzione condizionale da cui dipende.

Per comprendere come sia possibile calcolare le dipendenze di controllo tra due istruzioni, si consideri un *control flow graph* $G = (B, E)$ con nodo iniziale α e nodo terminale ω e siano i_k, i_j due istruzioni del programma Q associato a G, con i_j istruzione che altera il flusso di controllo in base all'esito della valutazione di un predicato c_j. Si osservi che la relazione di *control dependency* tra le due istruzioni si traduce nella dipendenza di controllo tra i due basic block a cui le istruzioni appartengono. Infatti, siano $b_k, b_j \in B$ due basic block di G tali che $i_k \in b_k$ e $i_j \in b_j$. Il fatto che i_j sia in grado di alterare il flusso di esecuzione implica che i_j sia l'ultima istruzione del blocco b_j e che $|\Gamma_G(b_j)| > 1$. Da tali considerazioni si può dedurre che $i_k \xrightarrow{c} i_j \Rightarrow b_k \xrightarrow{c} b_j$. Vale anche il viceversa, $b_k \xrightarrow{c} b_j \Rightarrow i_k \xrightarrow{c} i_j$, a patto però di trattare con particolare

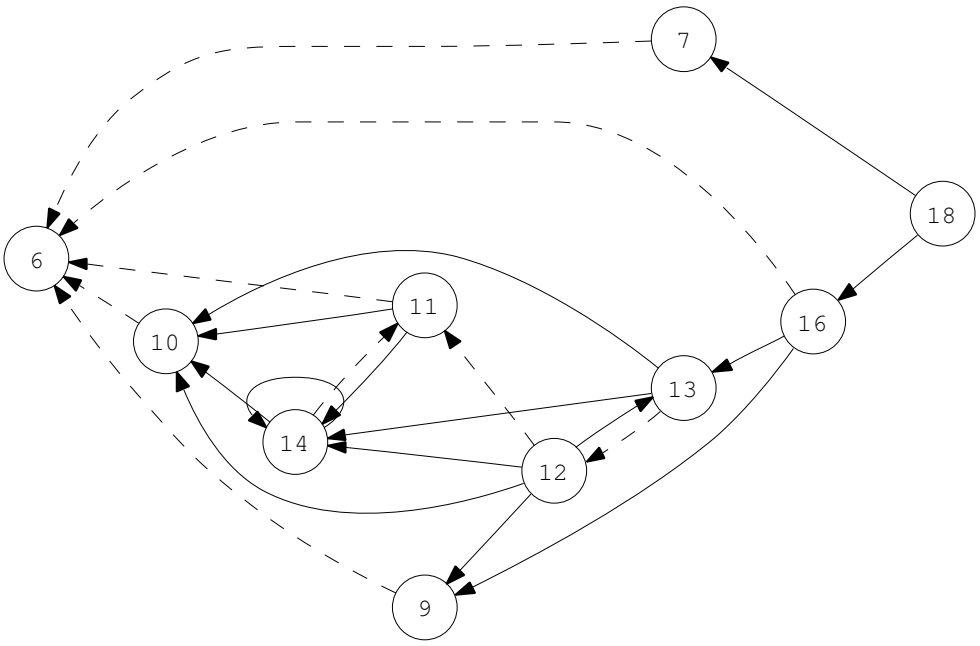

Figura 2.5: *Program dependence graph* statico relativo all'esempio di Figura 2.3.

attenzione il caso in cui $b_k = b_j$.

Il problema di derivare le dipendenze di controllo tra i basic block di G si traduce essenzialmente nel calcolo della frontiera di dominanza del grafo inverso di G. Formalmente, infatti, siano $b_k, b_j \in B$ e sia G^{-1} il grafo inverso di G; allora si può dimostrare che:

$$b_k \xrightarrow{c} b_j \quad \Leftrightarrow \quad b_k \in DF_{G^{-1}}(b_j).$$

In Figura 2.4 le dipendenze di controllo tra i basic block del grafo sono rappresentate tramite archi tratteggiati.

Un algoritmo per il calcolo delle *control dependency* che si basa su tale osservazione è descritto in [21] e presentato nella Sezione 5.3.

Prima di concludere, risulta utile introdurre il concetto di *program dependence graph* [30]. Si tratta sostanzialmente di un grafo orientato (V, E_p) che riassume le dipendenze di controllo e le *data dependency* di un programma Q. Nel caso statico, V è l'insieme delle istruzioni di Q mentre $E_p = E_d \cup E_c$. Dinamicamente, invece, V

è l'insieme di tutte le istanze di istruzioni di Q e E_p comprende sia gli archi del *data dependence graph* dinamico, sia quelli del *control dependence graph* dinamico.

A titolo esemplificativo, in Figura 2.5 è riportato il *program dependence graph* statico della procedura `find_max()` di Figura 2.3. Ogni nodo del grafo è etichettato con il numero di riga dell'istruzione rappresentata. Le dipendenze di controllo sono raffigurate tramite archi tratteggiati, mentre le *data dependency* sono indicate con un tratto continuo.

2.2.4 Program slicing

Con il termine *program slicing* [85] si fa riferimento ad una tecnica che consente di estrarre in modo automatico dall'insieme delle istruzioni di un programma un suo sottoinsieme contenente le sole istruzioni che contribuiscono a determinare il valore di una o più variabili in corrispondenza di un determinato punto del codice.

Formalmente, si definisce *criterio di slicing statico* per un programma Q una coppia (i,W), dove i è un'istruzione di Q da osservare, mentre W è un sottoinsieme delle variabili di Q su cui focalizzare l'analisi. Un *criterio di slicing dinamico* è invece una coppia (s,W), con s un'istanza di istruzione di Q.

Dato un criterio di slicing T ed un programma Q, possiamo quindi definire lo slice S per Q rispetto a T come l'insieme di tutte e sole le istruzioni di Q che hanno un qualche effetto sulle variabili osservate attraverso il criterio T.

Si consideri ora un programma Q e sia $T = (i,W)$ un criterio di slicing statico. Sia inoltre $G_p = (V,E_p)$ il *program dependence graph* statico associato a Q. Allora possiamo costruire lo *slice statico* per Q rispetto a T effettuando i seguenti passaggi [43]:

1. costruire l'insieme $H = \bigcup_{w \in W} srd(w,i)$;
2. determinare l'insieme S delle istruzioni di Q raggiungibili in G_p partendo da un'istruzione in H.

L'insieme di istruzioni S è lo slice statico cercato.

Per costruire lo *slice dinamico* [2] per il programma Q rispetto al criterio di slicing $T = (s, W)$ e relativamente ad una *execution history* EH, si determini per prima cosa il *program dependence graph* dinamico G_p^{EH}. Allora lo slice dinamico cercato comprende tutte e sole le istanze di istruzioni raggiungibili in G_p^{EH} partendo da un'istanza di istruzione dell'insieme $\bigcup_{w \in W} drd(w, s)$.

2.3 Disassembling

Con il termine *disassembling* si fa riferimento a quel procedimento tramite il quale si cerca di ricostruire una rappresentazione simbolica del codice macchina di un programma partendo dalla sua rappresentazione binaria [84]. Si tratta evidentemente di un'operazione fondamentale per ogni strumento che debba compiere una qualsiasi tipologia di analisi sul codice binario. Come si vedrà in seguito, il compito di un *disassembler* è però notevolmente complicato dalla presenza nel codice dell'applicazione di salti indiretti, dati non eseguibili (*jump table*, byte per l'allineamento, ...) inframezzati al codice del programma e istruzioni macchina con lunghezza variabile, tipico di architetture CISC come la piattaforma Intel x86.

Il modo più semplice per trasformare il codice di un programma in una sequenza di istruzioni assembly consiste nell'analizzare l'intera successione di bit che costituisce la sezione di codice del file eseguibile, procedendo in modo lineare a partire dal primo bit della sezione fino alla fine. Il disassembler incomincia quindi decodificando l'istruzione che ha inizio al primo byte della sezione di codice dell'applicazione e procede analizzando byte dopo byte fino al termine della sezione. Tale metodo è noto come *disassembly lineare* o *linear sweep* ed è ad esempio implementato nell'utility GNU `objdump` [34]. Il vantaggio principale di un approccio di questo tipo risiede nella sua semplicità, ma è evidente come eventuali byte di dati o di allineamento presenti all'interno del codice dell'applicazione possano compromettere la correttezza del risultato finale.

Per comprendere le problematiche connesse alla tecnica *linear sweep*, si consideri il frammento di codice assembly di Figura 2.6: si tratta di una semplice procedura che non fa altro che restituire al chiamante il valore 1 tramite il registro `EAX`. La procedura

Indirizzo	*Memoria*	*Istruzione*
0x8048438	eb 03	jmp 0x804843d
0x804843a	00 00 00	
0x804843d	b8 01 00 00 00	mov 0x1,%eax
0x8048442	c3	ret

Figura 2.6: Frammento di codice assembly.

però presenta un salto non condizionato seguito da alcuni byte di allineamento. In Figura 2.7 è riportato il risultato del *disassembling* del codice di Figura 2.6 utilizzando un approccio lineare: il *disassembler* ha iniziato analizzando l'istruzione con indirizzo *0x8048438* per poi proseguire in modo lineare fino alla fine del codice, andando così ad interpretare i byte di allineamento come fossero istruzioni del programma e ottenendo quindi una rappresentazione del frammento di codice semanticamente differente rispetto alla versione originale.

Indirizzo	*Memoria*	*Linear Sweep*
0x8048438	eb 03	jmp 0x804843d
0x804843a	00 00	add %al,(%eax)
0x804843c	00 b8 01 00 00 00	add %bh,0x1(%eax)
0x8048442	c3	ret

Figura 2.7: Risultato del *disassembling* del frammento di Figura 2.6 con il metodo *linear sweep*.

Il problema principale del metodo *linear sweep* è che non tiene in considerazione il flusso di controllo del codice analizzato e, di conseguenza, non è in grado di dedurre che i byte di allineamento dell'esempio precedente non possono essere raggiunti durante l'esecuzione. La soluzione più ovvia consiste nel modificare il metodo di *disassembling* in modo da considerare il flusso di controllo del programma. Intuitivamente, nel momento in cui il *disassembler* incontra un'istruzione in grado di alterare il *control flow*, determina l'insieme dei successori dell'istruzione, ovvero l'insieme delle istruzioni da cui l'esecuzione potrebbe proseguire. Nel caso di un'istruzione di salto non condizionato gli unici successori sono dati dalle possibili destinazioni del salto, mentre i successori di un salto condizionato sono le destinazione del salto in-

```
Input: addr primo indirizzo da esaminare, 𝓛 lista delle istruzioni assembly
if isVisited(addr) then
    return
while addr è un indirizzo di programma valido do
    i = decodeInstruction(addr)
    setVisited(addr)
    aggiungi i in coda a 𝓛
    if isCall(i) ∨ isJump(i) then
        foreach t ∈ getTarget(i) do
            RecursiveTraversal(t, 𝓛)
        if isCall(i) ∨ (isJump(i) ∧ isConditional(i)) then
            RecursiveTraversal(addr + getSize(i), 𝓛)
        return
    else
        addr = addr + getSize(i)
```

Figura 2.8: Algoritmo *RecursiveTraversal*.

sieme con l'istruzione sintatticamente successiva a quella in esame. Analogamente, i successori di una chiamata a funzione sono dati dalle prime istruzioni delle procedure che possono essere invocate. La variazione appena descritta prende il nome di *recursive traversal disassembling* [17] ed è l'approccio utilizzato nel modello proposto nel presente lavoro di tesi per la ricostruzione del codice assembly da analizzare.

In Figura 2.8 è riportata la procedura *RecursiveTraversal()* che descrive l'algoritmo seguito da un *disassembler* ricorsivo. Inizialmente la procedura è invocata specificando come parametro *addr* l'indirizzo corrispondente all'*entry point* del programma in esame, ovvero l'indirizzo di programma da cui deve incominciare l'esecuzione dell'applicazione, mentre il parametro $\mathscr{L}$ è una lista di istruzioni vuota. Per ciascuna istruzione di trasferimento di controllo, l'algoritmo procede analizzando ricorsivamente il cammino che inizia in corrispondenza di ciascun successore dell'istruzione. L'algoritmo *RecursiveTraversal()* utilizza le seguenti funzioni:

- *isVisited(addr)*, restituisce il valore logico "*vero*" se l'indirizzo di programma *addr* è già stato analizzato.
- *setVisited(addr)*, contrassegna l'indirizzo di programma *addr* come già analizzato.

- *decodeInstruction(addr)*, restituisce l'istruzione assembly ricostruita a partire dai dati binari memorizzati all'indirizzo *addr*.
- *isJump(i), isCall(i)*, restituiscono '*vero*" se *i* è rispettivamente un'istruzione di salto o una chiamata a funzione.
- *isConditional(i)*, restituisce "*vero*" se l'istruzione di salto *i* è di tipo condizionato.
- *getTarget(i)*, supponendo che *i* sia un'istruzione di salto o una chiamata a funzione, restituisce l'insieme dei successori immediati di *i*.
- *getSize(i)*, restituisce la dimensione in byte dell'istruzione *i*.

Se si suppone di essere in grado di determinare con esattezza l'insieme dei successori di ogni istruzione di salto o chiamata a funzione, allora l'algoritmo di Figura 2.8 garantisce che ogni istruzione del programma raggiungibile a partire dall'*entry point* verrà disassemblata correttamente. Proprio per questo motivo, il metodo *recursive traversal* è in grado di disassemblare correttamente il codice di Figura 2.6: dopo che il *disassembler* avrà esaminato l'istruzione di salto all'indirizzo *0x8048438*, l'analisi continuerà dall'istruzione con indirizzo *0x804843d* e procederà linearmente fino al termine del frammento di codice. I byte di allineamento verranno così correttamente evitati in quanto non raggiungibili in alcuna esecuzione.

Il problema fondamentale di un *disassembler* ricorsivo sorge quando vengono incontrate istruzioni di salto indirette (`jmp *%eax`) o chiamate a funzione indirette (`call *%eax`) per le quali non è possibile determinare con precisione i successori immediati. In questi casi alcune porzioni di codice potrebbero non essere analizzate in quanto raggiungibili solo indirettamente. Sfortunatamente, nel codice attuale situazioni di questo tipo si verificano piuttosto di frequente: a volte è il programmatore ad utilizzare tecniche che richiedono al momento della compilazione l'introduzione di istruzioni di trasferimento di controllo indirette (ad esempio, puntatori a funzione); più frequentemente, però, è lo stesso compilatore ad introdurre ottimizzazioni che comportano salti indiretti, ad esempio ricorrendo a *jump table* per la traduzione di alcuni costrutti *switch* o impiegando istruzioni di questo tipo per implementare meccanismi

di *dynamic linking* e *lazy binding* [56]. Appare quindi evidente che un qualunque strumento che si proponga come obiettivo l'analisi di codice binario non può prescindere dall'affrontare questioni di questo tipo.

Se si suppone di dover procedere staticamente, l'unica soluzione è ricorrere ad analisi piuttosto complesse che fanno uso di tecniche quali *program slicing* [16] o *constant propagation* [79] per determinare almeno un sottoinsieme delle possibili destinazioni delle istruzioni di salto o delle chiamate a funzione, o, in alternativa, adottare approcci meno formali che utilizzano le informazioni di rilocazione memorizzate nel file eseguibile [74] oppure una serie di euristiche [52, 39] per riuscire ad analizzare una percentuale maggiore del codice dell'applicazione.

La metodologia di analisi ibrida proposta nel presente lavoro di tesi affronta invece il problema utilizzando inizialmente un *recursive traversal disassembler* per analizzare le porzioni di codice raggiungibili a partire dall'*entry point* dell'applicazione senza trasferimenti di controllo indiretti. Successivamente, durante la fase di analisi dinamica, se l'esecuzione raggiunge nuove regioni del programma non ancora esaminate, su di esse verrà applicata nuovamente la procedura di *disassembling*.

Capitolo 3

Scenario

In questo capitolo viene presentato un esempio di programma vulnerabile, poi richiamato nel corso del lavoro di tesi per illustrare come la soluzione proposta cerca di individuarne le vulnerabilità. Si passa quindi ad una breve panoramica delle recenti ricerche nel campo dell'analisi delle vulnerabilità e ad una presentazione più dettagliata delle specifiche problematiche relative all'analisi di codice binario. Infine, viene introdotto ad alto livello l'approccio *smart fuzzing*, poi approfondito nei capitoli successivi.

3.1 Programma vulnerabile

In Figura 3.1 viene riportato un frammento di programma C contenente un difetto in grado di compromettere la sicurezza dell'applicazione.

La procedura `process_input()` riceve come parametro di input il vettore di caratteri `buf`. Il compito di tale procedura è copiare in un nuovo array tutti i caratteri "stampabili" di `buf` (ovvero con codifica ASCII nell'insieme $[32, 127)$), avendo cura di raddoppiare ogni occorrenza del carattere `'\''`. A questo scopo viene utilizzato il vettore temporaneo `newbuf`, locale alla procedura e allocato staticamente con dimensione pari a 64 caratteri (riga 3). La procedura controlla (riga 5) se l'input può essere copiato senza problemi nel vettore temporaneo, verificando che la lunghezza di `buf` sia effettivamente inferiore alla capacità di `newbuf`. Se tale condizione non è

```
1  char* process_input(char* buf) {
2    int i, j = 0;
3    char newbuf[64];
4
5    if (strlen(buf) <= 58)
6      for(i = 0; i < strlen(buf); i++) {
7        if (buf[i] >= 32 && buf[i] < 127) {
8            newbuf[j++] = buf[i];
9            if (buf[i] == '\'')
10               newbuf[j++] = buf[i];
11       }
12     }
13   newbuf[j] = '\0';
14
15   return strdup(newbuf);
16 }
```

Figura 3.1: Codice C di una procedura vulnerabile.

soddisfatta, la funzione ritorna immediatamente al chiamante senza compiere alcuna altra operazione. Nonostante tale controllo, però, la funzione è comunque vulnerabile ad un attacco di tipo *stack-based buffer overflow* [5]. Infatti, se un attaccante invocasse la procedura passando un array con un numero sufficiente di caratteri `'\''`, la duplicazione effettuata dalle righe 9-10 gli consentirebbe di scrivere in memoria oltre l'intervallo riservato al vettore `newbuf`, arrivando potenzialmente a sovrascrivere nello stack l'indirizzo di ritorno della procedura e potendo così dirottare il flusso di esecuzione verso una zona di memoria contenente istruzioni precedentemente iniettate nello spazio di indirizzamento del processo.

Per chiarire questo concetto, in Figura 3.2 è riportato un possibile layout della memoria al momento dell'esecuzione della procedura `process_input()` su architettura Intel x86. A livello assembly l'invocazione della funzione avviene utilizzando l'istruzione macchina `call` che, prima di trasferire il controllo alla procedura chiamata, provvede al salvataggio sullo stack dell'indirizzo dell'istruzione del chiamante successiva a quella di `call`. Ciò consente di mantenere in memoria un riferimento

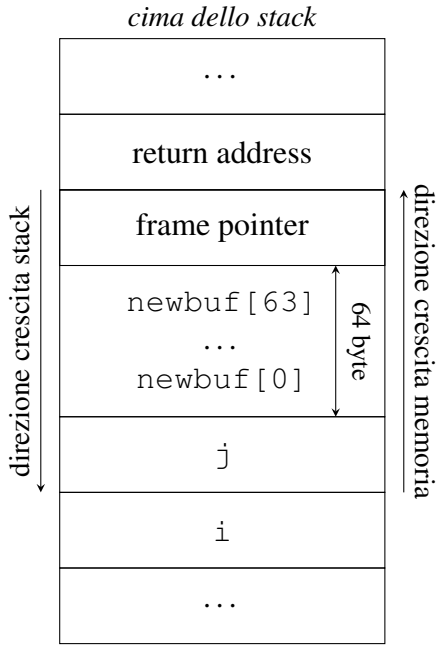

Figura 3.2: Struttura dello stack al momento dell'invocazione della procedura in Figura 3.1.

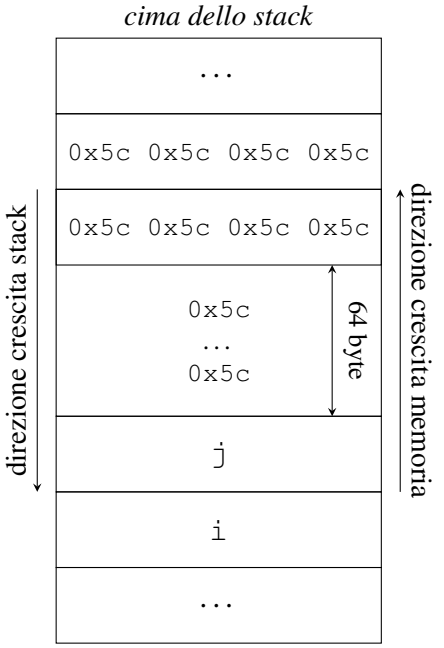

Figura 3.3: Struttura dello stack dopo l'esecuzione procedura in Figura 3.1 su una stringa di input di 36 caratteri `'\''`.

all'indirizzo a partire dal quale l'esecuzione dovrà continuare una volta terminata la procedura. Lo stesso chiamato, inoltre, provvederà, durante il prologo della funzione, a memorizzare sullo stack il contenuto del registro del processore `EBP` (*base pointer*, chiamato anche *frame pointer*). Al momento della chiamata, infatti, tale registro conterrà l'indirizzo del record di attivazione del chiamante e quindi, dovendo essere modificato dalla procedura chiamata, viene salvato sullo stack al fine di poterne ripristinare il contenuto originale durante l'epilogo della funzione. Infine, la procedura riserva sullo stack lo spazio necessario alle proprie variabili locali, che in questo caso sono `newbuf`, `i` e `j`. Si noti che a tali variabili corrisponderanno indirizzi di memoria comunque inferiori rispetto all'indirizzo precedentemente assegnato al *return address* della procedura. Per questo motivo, se un attaccante riuscisse a fare in modo che un'operazione di scrittura in un array scriva oltre la zona di memoria riservata alla variabile, allora avrebbe buone probabilità di riuscire a sovrascrivere l'indirizzo di ritorno della procedura, potendo così alterare il flusso di esecuzione del processo. Nel

nostro caso, se un attaccante riuscisse, ad esempio, a fornire in input alla procedura una stringa composta da 36 caratteri `'\''`, eluderebbe i controlli sulla lunghezza massima dell'input e, a causa della duplicazione effettuata dalle righe 9-10, arriverebbe a sovrascrivere *frame pointer* e *return address*. In Figura 3.3 è rappresentata graficamente una situazione di questo tipo. Si ricorda che il byte `0x5d` corrisponde alla codifica ASCII del carattere `'\''`.

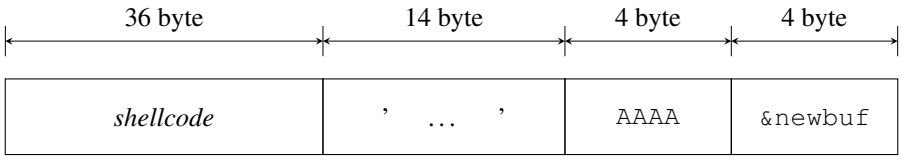

Figura 3.4: Vettore di attacco per lo sfruttamento della vulnerabilità del frammento di codice in Figura 3.1.

Complicando lievemente la struttura del parametro in ingresso è anche possibile arrivare ad eseguire codice arbitrario. Si consideri a tale proposito il vettore di attacco presentato in Figura 3.4. Il vettore ha una lunghezza totale di 58 byte, quindi sarà in grado di superare il controllo imposto dalla riga 5 e verrà copiato dalla procedura nella variabile `newbuf` avendo cura di raddoppiare ogni occorrenza del carattere `'\''`. A causa di tale duplicazione, però, i 14 caratteri `'\''` del vettore diventeranno 28 e, così facendo, il *frame pointer* e l'indirizzo di ritorno della procedura verranno sovrascritti rispettivamente con i caratteri `AAAA` e con l'indirizzo in memoria della variabile `newbuf`. Quando la procedura `process_input()` terminerà, l'esecuzione non continuerà dall'istruzione successiva alla chiamata, bensì dall'indirizzo di `newbuf` che conterrà il codice macchina iniettato dall'attaccante (*shellcode*). Come si deduce dalla Figura 3.4, lo *shellcode* iniettato dall'attaccante potrà avere una lunghezza massima di 36 byte, comunque sufficiente per l'esecuzione di una *shell* locale.

È evidente che, anche in un semplice esempio come quello appena presentato, se la procedura fosse verificata sperimentalmente eseguendola con input scelti in modo casuale e senza alcuna informazione circa il suo comportamento, difficilmente si riuscirebbe a fare in modo che tutte le condizioni necessarie a manifestare la vulnerabilità si verifichino. Questo perché, utilizzando input casuali, anche una singola condizione

di uguaglianza su un singolo byte dei dati in ingresso (come alla riga 9) può essere particolarmente difficile da soddisfare. Infatti, riuscire a rendere vera un'uguaglianza a 32 bit senza alcuna informazione sul comportamento dell'applicazione corrisponde ad "indovinare" correttamente un valore tra circa quattro miliardi di possibilità. Logicamente, la probabilità di soddisfare una sequenza di condizioni di quel tipo tende rapidamente allo zero. Al fine di essere in grado di trattare programmi anche solo mediamente complessi, è quindi indispensabile disporre di una qualche informazione sulla struttura e sul comportamento dell'applicazione analizzata.

Si ricorda che l'approccio discusso nel presente lavoro suppone di trattare direttamente codice binario. L'esempio appena presentato è stato riportato sotto forma di codice sorgente solo per chiarezza e semplicità di esposizione.

3.2 Ricerca automatica di vulnerabilità

L'identificazione automatica dei difetti di un'applicazione rilevanti dal punto di vista della sicurezza è tutt'oggi un'area di ricerca molto attiva, anche se piuttosto giovane e ancora parzialmente inesplorata.

Una prima semplice tecnica per individuare automaticamente i difetti presenti in un programma è costituita dal *fuzzing*. Con il termine *fuzz testing*, o semplicemente *fuzzing*, si fa riferimento ad una metodologia di *program testing* che prevede la generazione casuale dei dati di input dell'applicazione oggetto dell'analisi. L'obiettivo è quello di produrre casi di test che siano in grado di sollecitare il programma in modi non previsti dagli sviluppatori. L'idea originale di *fuzzing* [61] non prevede l'utilizzo di alcuna informazione relativa al comportamento dell'applicazione in esame, che viene quindi analizzata "a scatola chiusa". Ciò costituisce il limite principale dell'approccio, in quanto per percorrere il cammino di esecuzione che conduce alla manifestazione di una vulnerabilità è spesso necessario soddisfare una catena di condizioni sull'input. È quindi poco probabile generare casualmente un caso di test in grado di soddisfare un simile insieme di vincoli.

Per comprendere meglio i limiti del *fuzz testing*, si consideri il frammento di codice C riportato in Figura 3.5. La funzione `get_error_msg()` si occupa di generare un

```
1  char* get_error_msg(int error_code)
2  {
3      char *msg;
4      if (error_code != 12345678) {
5          msg = malloc(100);
6          ...
7      } else {
8          strncpy(msg, "ID non valido.", 100);
9      }
10     return msg;
11 }
```

Figura 3.5: Esempio di una procedura difficilmente analizzabile con un approccio *fuzzy* tradizionale.

messaggio associato al codice di errore passato come parametro. Si osservi, però, che se il codice di errore è uguale a *12345678*, allora la funzione cerca di copiare una stringa nella variabile *msg*, senza che quest'ultima sia stata precedentemente inizializzata. Il difetto non può quindi essere individuato a meno che la condizione valutata dall'*if* alla riga 4 non risulti falsa. Supponendo che il parametro *error_code* sia generato in modo casuale, riuscire a raggiungere l'istruzione alla riga 8 equivale ad "indovinare" correttamente il valore di *error_code* tra oltre 4 miliardi di possibilità. È evidente che la probabilità associata ad una tale eventualità è piuttosto ridotta. Se poi si considera che spesso le istruzioni di un programma che fanno si che un difetto o una vulnerabilità si manifestino dipendono da una catena di condizioni questo tipo, è chiaro che la probabilità che un simile evento si verifichi è pressoché nulla.

Dalle considerazioni appena fatte si deduce che, nonostante la tecnica del *fuzz testing* consenta di individuare un numero considerevole di problemi che non dipendono da condizioni particolarmente complesse, l'approccio non risulta adatto ad individuare difetti che si manifestano solo in corrispondenza di particolari *casi limite* [86]. Per poter trattare situazioni più articolate è indispensabile disporre di informazioni aggiuntive sull'applicazione in esame. Per una descrizione più dettagliata delle tecniche di *fuzz testing* si rimanda alla Sezione 7.1.

La naturale evoluzione delle tecniche di *fuzzing* ha portato allo sviluppo di metodologie che cercano in qualche modo di estrarre dall'applicazione esaminata una serie di informazioni che consentano di esplorare in modo più esaustivo gli stati raggiungibili durante l'esecuzione.

Un numero relativamente consistente di lavori si concentra sull'analisi del codice sorgente [87], spesso con risultati decisamente apprezzabili ed incoraggianti [88]. Nel complesso, pare che attualmente l'analisi del codice sorgente sia affrontata con tecniche ibride statico-dinamiche [27], indispensabili per superare i limiti imposti dall'approccio conservativo dell'analisi statica e dalla mancanza di visione globale dell'applicazione tipica dell'analisi dinamica. In alcuni casi, come in [15], l'analisi dinamica è stata poi integrata con motori di esecuzione simbolica per astrarre le informazioni raccolte al di sopra della specifica esecuzione.

Spostandosi poi al caso particolare dell'analisi di codice binario, qui il numero di lavori esistenti diventa ancora più ridotto. In questo settore, fatta eccezione per il presente lavoro, non pare siano state ancora esplorate le possibilità offerte da un'analisi ibrida. Si ritiene che i risultati più rilevanti siano costituiti da [66] (approccio dinamico, concentrato sulla generazione di *signature* per sistemi di rilevamento delle intrusioni, più che sull'individuazione di vulnerabilità) e [20] (approccio statico, supportato da un motore di esecuzione simbolica). Entrambi i lavori appena citati sono descritti in maggior dettaglio nel Capitolo 7.

3.3 Particolarità dell'analisi di codice binario

L'analisi di codice binario differisce, talvolta in maniera piuttosto sostanziale, dall'analisi del codice sorgente, introducendo spesso nuove questioni o complicazioni che richiedono la modifica o la completa revisione delle tradizionali tecniche di *program analysis*. Di seguito vengono presentate alcune delle problematiche relative all'analisi di codice binario.

Analisi della struttura del file eseguibile. La struttura del file contenente l'applicazione deve essere analizzata in modo dipendente dal particolare formato utilizzato al momento della compilazione (tipicamente ELF [81] su sistemi Linux e

PE [68] su piattaforma Windows). È poi necessario adottare tutta una serie di accorgimenti indispensabili per la ricostruzione dei simboli relativi ad oggetti esterni all'applicazione e importati dinamicamente al momento dell'esecuzione utilizzando una tecnica nota come *dynamic linking* [56].

Disassembly. Un primo passo fondamentale per proseguire con l'analisi è la trasformazione del codice macchina in codice assembly, prestando particolare attenzione alle problematiche derivanti da indirezione e mancanza di separazione netta tra istruzioni e dati. Si rimanda alla Sezione 2.3 per una discussione più dettagliata delle attuali tecniche di *disassembly*.

Aliasing. Un'espressione è *alias* di un'altra se identifica la medesima locazione di memoria. Un'operazione di scrittura in memoria che coinvolge la locazione a cui fa riferimento la prima espressione ha quindi ovvie ripercussioni anche sulla zona di memoria a cui si riferisce la seconda. La questione dell'*aliasing* è complessa già a livello del codice sorgente nel caso di linguaggi che, come C, prevedono il concetto di puntatore. Determinare staticamente se due espressioni sono legate da una relazione di *aliasing* è un problema non decidibile [71] e notevolmente amplificato a livello del codice binario. Nonostante la cospicua quantità di lavori sull'argomento [42], attualmente l'unica soluzione al problema pare essere adottare un approccio approssimativo e conservativo oppure procedere dinamicamente.

Perdita di informazioni. Il processo di compilazione da codice sorgente a codice macchina comporta la perdita di informazioni di medio-alto livello che devono necessariamente essere ricostruite per continuare l'analisi. Ad esempio, un semplice predicato condizionale come l'espressione `buf[i] == '\''` alla riga 9 del programma di Figura 3.1 verrà tradotto dal compilatore in un'espressione sui *flag* del processore. Al fine di poter compiere una qualche forma di "ragionamento" sull'esecuzione dell'applicazione, è importante essere in grado di ricostruire almeno in parte il predicato condizionale di alto livello. Altre informazioni perse durante la compilazione sono quelle relative alla struttura stessa

del programma: a livello di codice macchina può non essere immediato individuare, ad esempio, i limiti di una procedura, o classici costrutti di programmazione quali cicli o costrutti di selezione. Supponendo di adottare un approccio puramente dinamico, informazioni di questo tipo sono spesso impossibili da ricostruire correttamente. Per fare un esempio, si consideri un semplice costrutto *if-then*: se in tutte le esecuzioni osservate la condizione dell'*if* risulta sempre vera, allora non è possibile individuare dinamicamente dove termina il blocco *then*; ciò porterebbe erroneamente a dedurre che le istruzioni immediatamente successive al costrutto *if-then* abbiano rispetto ad esso una dipendenza di controllo, mentre invece il fatto che siano o meno eseguite è del tutto indipendente dall'esito della valutazione della condizione dell'*if*.

Ottimizzazioni. I compilatori attuali sono spesso in grado di generare codice altamente ottimizzato e performante, ma per fare ciò utilizzano spesso accorgimenti che rendono il codice più intricato e complesso da analizzare. Tecniche quali il riutilizzo dello stesso basic block come epilogo di funzioni differenti o il *dynamic linking* con *lazy binding* sono solo alcuni esempi di come il compilatore possa, al fine di massimizzare le prestazioni e minimizzare l'utilizzo di risorse, complicare ed offuscare notevolmente la struttura dell'applicazione.

Complessità. La compilazione del semplice esempio di Figura 3.1 trasforma le 16 righe di codice in circa 120 istruzioni assembly, senza contare le procedure di servizio introdotte al momento del processo linking, che portano alla generazione di un eseguibile con dimensione intorno alle 350 istruzioni macchina. Questa esplosione del numero delle istruzioni non può fare altro che complicare l'analisi e rendere difficoltoso scalare il procedimento verso applicativi più significativi e complessi.

Nonostante le numerose problematiche appena riportate, vanno però anche considerati gli aspetti positivi derivanti dall'analisi di codice binario ed i vantaggi che un simile approccio comporta rispetto alle tecniche orientate al codice sorgente.

Innanzi tutto, spesso semplicemente non si dispone dei sorgenti dell'applicazione da analizzare, come nel caso di software *closed source*. In una situazione di questo

tipo, l'unica alternativa possibile è quella di procedere con tecniche orientate al codice binario. Un altro aspetto da tenere in considerazione è che, in molti casi, le tecniche di analisi che si basano sul sorgente richiedono una qualche forma di instrumentazione del codice e la successiva ricompilazione dell'applicazione. In alcune situazioni requisiti di questo tipo non sono accettabili.

Focalizzando l'attenzione in modo specifico sull'analisi delle vulnerabilità, una problematica ben più rilevante è la seguente: un'analisi centrata sul sorgente dell'applicazione potrebbe essere incapace di individuare alcune tipologie di vulnerabilità a causa del divario esistente tra ciò che il programmatore intente al momento della stesura dell'applicazione e ciò che invece viene effettivamente eseguito dal processore. Alcune problematiche di sicurezza, infatti, potrebbero derivare da particolari legati alla piattaforma su cui l'applicazione viene eseguita, come, ad esempio, dettagli relativi a come il sistema operativo gestisce lo spazio di indirizzamento del processo, come i registri del processore vengono utilizzati dal compilatore, l'effettivo ordine di esecuzione delle istruzioni, fino alle ottimizzazioni introdotte dal compilatore ed eventuali suoi *bug*. Per chiarire meglio questo concetto si pensi ad un semplice *buffer overflow*: nella sua forma più classica, il problema deriva dal fatto che, a causa di mancati controlli sui limiti di un buffer, l'applicazione potrebbe essere indotta a sovrascrivere l'indirizzo di ritorno di una procedura con dati sotto il controllo dell'attaccante. L'effettiva posizione in memoria del buffer e dell'indirizzo di ritorno della procedura non sono però informazioni note al momento della scrittura dell'applicazione, tant'è infatti che i moderni compilatori si riservano il diritto di ridefinire secondo propri criteri l'ordine con cui le variabili locali vengono allocate, cercando di prevenire simili attacchi. Solo un'analisi condotta sul codice binario può quindi disporre delle necessarie informazioni per determinare la presenza o meno di una vulnerabilità di questo tipo. Per ulteriori dettagli su questi particolari fenomeni, si può fare riferimento a [8].

In conclusione si può affermare che, nonostante i problemi derivanti dall'analisi di codice binario siano certamente rilevanti, i vantaggi connessi ad un simile approccio risultano significativi e giustificano gli sforzi fatti in questa direzione.

3.4 Obiettivi del lavoro

Il presente lavoro di tesi si inserisce nel contesto appena descritto con l'intento di studiare le principali tecniche di analisi dinamica e la loro applicabilità al codice binario. I risultati delle ricerche sono poi confluiti nella realizzazione di un nuovo metodo per l'individuazione di vulnerabilità nelle applicazioni software, senza contare sulla disponibilità del codice sorgente del programma. Tale metodo è stato chiamato *smart fuzzing* ed è stato implementato in un prototipo sperimentale.

Nel complesso, si possono riassumere gli obiettivi del lavoro nei punti riportati di seguito.

1. Sperimentazione delle tecniche di *program analysis* su codice binario, con particolare riguardo per l'analisi dinamica. Nella maggior parte dei casi, sia l'analisi statica che l'analisi dinamica sono state applicate al codice sorgente. Come si è potuto però osservare dalle considerazioni fatte nella precedente sezione, le informazioni di cui può disporre un analizzatore che opera a livello del codice binario sono notevolmente ridotte rispetto a quando le stesse analisi sono condotte sul sorgente. Da ciò deriva la necessità di modificare, adattare o, a volte, rivedere completamente gli algoritmi di analisi quando questi devono essere utilizzati per trattare codice macchina. Nel corso della tesi verranno descritti in dettaglio i procedimenti adottati per cercare, per quanto possibile, di risolvere le problematiche dell'analisi di codice binario individuate in precedenza.

2. Realizzazione di un'infrastruttura software *open source* per l'analisi di codice binario. Sono poche le infrastrutture di *program analysis* liberamente utilizzabili e adatte ad essere impiegate per l'analisi di vulnerabilità. Le soluzioni esistenti, infatti, sono per la maggior parte orientate al codice sorgente, o comunque difficilmente estendibili e riutilizzabili. Il presente lavoro descrive un'architettura modulare per l'analisi ibrida del codice binario, poi implementata in un prototipo sperimentale. Si ritiene che un tale strumento possa costituire un utile punto di partenza per ulteriori progetti di ricerca futuri.

3. Progettazione di una nuova metodologia per l'individuazione di vulnerabilità in

modo completamente automatizzato, senza contare sulla disponibilità del sorgente o su altri elementi di alto livello (come, ad esempio, simboli o altre informazioni di debugging), ma solamente sulla possibilità di esaminare il codice macchina del programma. La soluzione proposta utilizza la *program analysis* per estrarre dall'applicazione informazioni utili a guidare il processo di ricerca delle vulnerabilità, superando così i limiti del *fuzzing* tradizionale, costretto ad una forma di testing sostanzialmente "a scatola chiusa" (*black-box*), ovvero senza alcuna informazione sulla semantica del programma analizzato.

È già stato sottolineato nel Capitolo 1 come il problema di dimostrare formalmente la correttezza di un programma mediamente complesso sia notoriamente indecidibile. Per questo motivo è importante evidenziare che il metodo presentato, così come tutti gli altri approcci con obiettivi analoghi, non cerca di dimostrare l'*assenza* di vulnerabilità in un'applicazione, ma piuttosto provare la *presenza* di una qualche vulnerabilità. In particolare, tramite l'adozione di tecniche dinamiche, la metodologia proposta è in grado di ridurre considerevolmente il numero di falsi positivi, potendo verificare, tramite l'esecuzione diretta dell'applicazione, la veridicità delle informazioni riportate.

Capitolo 4

Smart fuzzing

Viene ora analizzata in maggiore dettaglio l'architettura del modello proposto per l'individuazione di vulnerabilità nel codice binario. L'approccio ideato prende il nome di *smart fuzzing*, poiché deriva dal *fuzzing* tradizionale, supportato però da un'attenta analisi statico-dinamica del comportamento dell'applicazione, finalizzata a derivare le informazioni necessarie a guidare il processo di generazione dell'input.

4.1 Panoramica dell'approccio *smart fuzzing*

L'obiettivo del modello *smart fuzzing* è la generazione di input per l'applicazione che siano in grado di guidare il flusso di esecuzione verso casi limite [86] in cui è più probabile che una vulnerabilità possa manifestarsi.

La principale categoria di vulnerabilità che tale metodologia si propone di individuare è costituita dai cosiddetti *memory attack*: si tratta di tipologie di attacchi che consistono nella sovrascrittura di una qualche zona di memoria delicata per l'applicazione con dati controllabili da un attaccante. Un classico esempio di attacco di questo tipo è un *buffer overflow*: la mancanza di controlli su un'operazione di scrittura in memoria consente all'attaccante di scrivere oltre i limiti di un buffer, arrivando a sovrascrivere una zona di memoria (indirizzo di ritorno di una procedura, aree utilizzate per il linking dinamico, ...) che gli consente di dirottare il flusso di esecuzione verso un'area contenente codice da lui iniettato in precedenza. Allo scopo di individuare simili

vulnerabilità, l'esecuzione dell'applicazione viene monitorata, cercando di estrarre le informazioni necessarie per definire un insieme di condizioni sull'input del programma che, se soddisfatte, consentano di compromettere la sicurezza dell'applicazione. Il procedimento è iterativo: il programma è eseguito più volte con differenti insiemi di dati in ingresso; ad ogni iterazione vengono raccolte le informazioni necessarie alla costruzione dell'input successivo e alla scoperta di nuove aree di memoria delicate.

Inizialmente, l'applicazione viene eseguita con un input casuale, così come avviene nel *fuzzing* tradizionale. A ciascuna esecuzione del programma si associa un insieme di vincoli sui dati in ingresso, detto *path condition* (*PC*), che viene costantemente aggiornato in modo da rispecchiare le condizioni che devono essere verificate affinché il flusso di esecuzione raggiunga lo stato corrente dell'applicazione. Appena l'applicazione viene eseguita, tale insieme sarà vuoto ($PC = \emptyset$). Il processo viene quindi monitorato e, quando viene incontrata un'istruzione di salto condizionato, vengono eseguite una serie di analisi al fine di determinare se il predicato c associato a tale salto è in qualche modo relazionato con i dati di input. Se viene individuata una qualche dipendenza tra c e i dati in ingresso, allora l'esecuzione viene ramificata: il processo associato al ramo "*vero*" continua la propria esecuzione con un insieme di vincoli $PC' = PC \cup c$, mentre il processo associato al ramo "*falso*" riprende con $PC'' = PC \cup \neg c$. L'esecuzione di ciascun processo figlio è quindi monitorata e i relativi insiemi di vincoli vengono mantenuti aggiornati, eventualmente ramificando ulteriormente. Tale tecnica consente un notevole incremento della percentuale di cammini di esecuzione esplorati rispetto ad un approccio *fuzzing* tradizionale, aumentando così la probabilità di raggiungere stati che provocano la manifestazione di una qualche vulnerabilità.

Come esempio, si consideri come varia l'insieme *PC* delle *path condition* durante l'esecuzione del programma di Figura 3.1, supponendo di avere come input `buf = 'a\''`. Inizialmente si avrà $PC = \emptyset$. Quando l'esecuzione raggiunge la riga 5, la condizione dell'*if* viene esaminata e ne viene dedotta la relazione con l'input. L'esecuzione viene quindi ramificata. Dato che la lunghezza della variabile *buf* associata al processo corrente è minore di 58 caratteri, allora il processo potrà continuare la propria esecuzione con un insieme di vincoli $PC' = \{$`strlen(buf) <= 58`$\}$. È poi necessario generare un nuovo processo che percorra il cammino associato al

ramo "*falso*" della condizione. A questo scopo, viene generato un nuovo insieme di condizioni $PC'' = \{\texttt{strlen(buf)} > 58\}$. L'insieme viene analizzato da un risolutore di vincoli, il quale ne determina la soddisfacibilità e genera un nuovo input che rispetta i requisiti imposti (ad esempio, una vettore di 59 caratteri non nulli). Viene quindi generato un altro processo a cui è associato il nuovo input. L'esecuzione di entrambi i processi può continuare fino alla prossima istruzione di selezione (riga 7), in corrispondenza della quale si compiranno operazioni analoghe.

Si può facilmente osservare, però, che una continua ramificazione dell'esecuzione in corrispondenza dei salti condizionati dipendenti dall'input può portare in breve tempo alla generazione di un elevato numero di processi e alla saturazione delle risorse di sistema. Per questo motivo si suppone di definire un limite superiore al numero dei processi in esecuzione associati all'applicazione. Quando l'esecuzione deve essere ramificata, si verifica se il numero di processi in esecuzione ha già raggiunto il limite massimo e, in tal caso, invece che ramificare l'esecuzione, l'insieme di vincoli che dovrebbe essere assegnato al nuovo processo viene aggiunto ad una coda. Quando un processo termina, se la coda con gli insiemi di vincoli non è vuota, il primo insieme della coda viene rimosso e viene assegnato ad un nuovo processo. Così facendo è possibile limitare superiormente l'ammontare di risorse occupate.

Mediante la continua ramificazione dell'esecuzione è possibile raggiungere una considerevole percentuale dei possibili stati dell'applicazione. Inoltre, tramite l'applicazione di una serie di euristiche, l'approccio presentato è in grado di evitare cammini in cui è poco probabile che una vulnerabilità possa manifestarsi e, al contrario, cercare di prioritizzare quei cammini che potrebbero con maggiore probabilità condurre alla sovrascrittura di una zona di memoria delicata.

È evidente che le analisi richieste per implementare il procedimento appena descritto risultano considerevolmente impegnative e costose già quando vengono effettuate sul codice sorgente dell'applicazione [76] e, come già visto nella Sezione 3.3, tale complessità aumenta drasticamente quando il programma analizzato è disponibile solamente sotto forma di codice binario. A livello del codice macchina, infatti, non si dispone di predicati di alto livello, per cui le relazioni tra i salti condizionati e l'input devono essere completamente ricostruite.

4.2 Architettura dell'infrastruttura di analisi

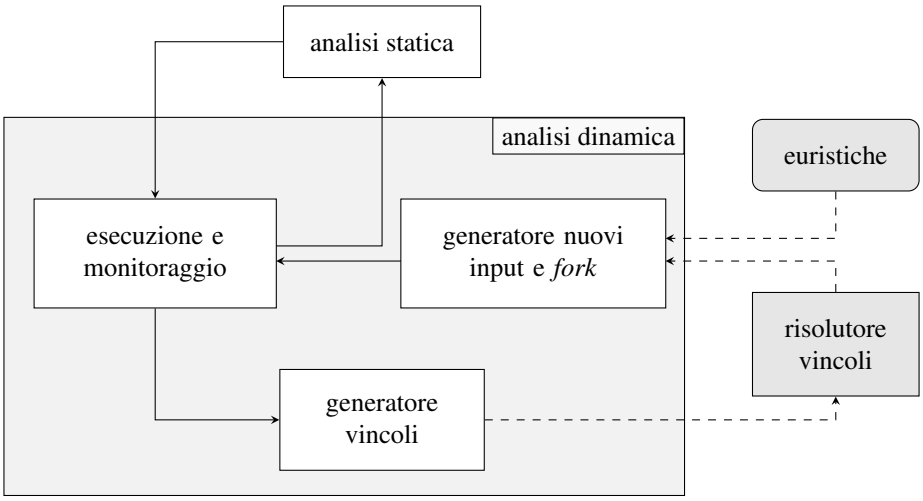

Figura 4.1: Infrastruttura di analisi.

Al fine di poter effettuare le analisi descritte nella sezione precedente, l'infrastruttura proposta nel presente lavoro è stata progettata includendo le componenti illustrate in Figura 4.1 e descritte qui di seguito.

- *Motore di analisi statica* – viene utilizzato per effettuare un'analisi preliminare del programma in modo da raccogliere un insieme di informazioni utili alla costruzione di una prima versione approssimata del CFG interprocedurale dell'applicazione. Tale componente consente anche di effettuare una prima identificazione dei cicli, poi raffinata nelle successive fasi dell'analisi.

- *Motore di analisi dinamica* – questo componente costituisce di fatto il cuore del modello proposto. Si occupa di eseguire e monitorare il programma, tenendo traccia delle dipendenze esistenti tra l'input e i salti condizionati incontrati durante l'esecuzione.

- *Insieme di euristiche* – risultano indispensabili per identificare cammini di esecuzione interessanti che potrebbero con maggiore probabilità condurre alla so-

vrascrittura di zone di memoria delicate e, quindi, a potenziali compromissioni della sicurezza dell'applicazione.

- *Generatore di vincoli* – mantiene aggiornato l'insieme dei vincoli associati all'esecuzione corrente. Quando necessario, si preoccupa inoltre di tradurre opportunamente le *path condition* in modo da poter essere poi comunicate al risolutore di vincoli.

- *Risolutore di vincoli* – dato un insieme di vincoli generati durante la fase di monitoraggio, il risolutore si occupa di stabilire se tale insieme risulta essere soddisfacibile e, in tal caso, di generare un nuovo insieme di dati in ingresso che se fornito all'applicazione consente di guidarne l'esecuzione lungo un particolare cammino.

- *Generatore nuovi input e fork* – tenendo conto dell'insieme di euristiche e delle informazioni restituite dal risolutore di vincoli, si occupa di sovraintendere il processo di ramificazione dell'esecuzione.

4.3 Analisi statica

Lo scopo principale del motore di analisi statica è quello di generare una vista conservativa ma potenzialmente inaccurata dell'intera applicazione. Tali informazioni saranno poi raffinate nella successiva fase di monitoraggio dinamico.

La struttura ad alto livello di questo componente è mostrata in Figura 4.2. Per prima cosa, il file eseguibile relativo all'applicazione viene analizzato utilizzando un modulo dipendente dal particolare formato dell'oggetto, estraendo informazioni quali il codice del programma, le librerie caricate dinamicamente e l'*entry point* dell'applicazione. A questo punto, per ogni istruzione macchina vengono effettuate una serie di fasi utili ad ottenere una rappresentazione del codice su cui siano applicabili le analisi successive. Innanzi tutto il codice macchina dev'essere trasformato in codice assembly (*disassembly*). Ciascuna istruzione assembly è quindi trasformata in una o più istruzioni intermedie, in modo da esplicitare gli effetti dell'istruzione sui registri del

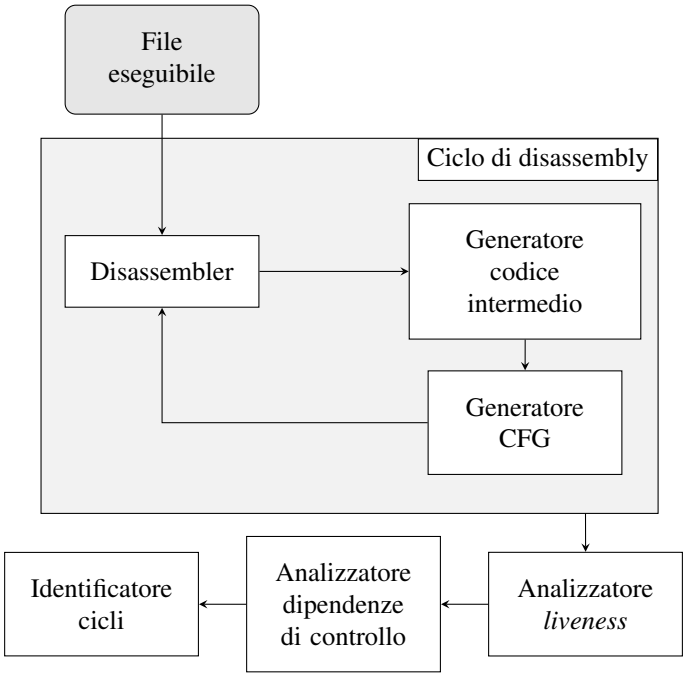

Figura 4.2: Struttura del motore di analisi statica.

processore e sulla memoria e ottenendo, inoltre, una rappresentazione del programma indipendente dalla piattaforma hardware sottostante. Infine, vengono aggiornati i CFG delle procedure facenti parte dell'applicazione, in modo da includere le istruzioni intermedie appena generate. Una volta che tutte le istruzioni del programma sono state esaminate, vengono eseguite staticamente una serie di analisi (*liveness*, dipendenze di controllo, identificazione dei cicli) necessarie per la successiva analisi dinamica. Nel prosieguo della presente sezione, ogni fase è descritta brevemente. Si rimanda ai Capitoli 2 e 5 per ulteriori dettagli tecnici circa gli algoritmi applicati in ciascuna fase.

4.3.1 Disassembly

Il codice del programma viene disassemblato utilizzando l'algoritmo ricorsivo presentato nella Sezione 2.3. Le problematiche derivanti dall'utilizzo di tale tecnica di *disassembly* sono ben note e ampiamente trattate in letteratura [84] e fondamentalmente legate all'impossibilità di esplorare regioni di codice raggiungibili solo tramite

```
1  mov 0xfffffff4(%ebp),%eax
2  add 0x8(%ebp),%eax
3  movzbl (%eax),%eax
4  cmp $0x27,%al
5  jne 0x8048415
```

Figura 4.3: Codice assembly relativo all'istruzione `if (buf[i] == '\'')`.

istruzioni di trasferimento di controllo indirette (come, ad esempio, `call *%eax`). L'insieme entro cui può variare l'indirizzo di destinazione di simili istruzioni non può essere determinato staticamente con precisione, in quanto il problema corrisponde ad effettuare una *points-to analysis* (individuazione delle locazioni a cui può far riferimento un puntatore) sulla variabile utilizzata per specificare la destinazione; si tratta di una questione notoriamente indecidibile [54]. In ogni caso, adottando un approccio ibrido, è possibile evitare di gestire in questa fase situazioni di questo genere, dato che verranno opportunamente trattate in seguito dall'analizzatore dinamico.

A titolo di esempio, in Figura 4.3 viene riportato un frammento di codice assembly relativo alla riga 9 del programma di esempio in Figura 3.1. È chiaro che la particolare sequenza di codice assembly utilizzata per tradurre le istruzioni di alto livello è comunque legata allo specifico compilatore utilizzato. Nel nostro caso il frammento sopra riportato è stato generato dal compilatore `gcc`. È importante notare come il compilatore abbia negato la valutazione della condizione: mentre nel codice di alto livello se la condizione valutata dall'istruzione di selezione *if* è vera, allora viene eseguito il blocco *then* (righe 9-10), a livello assembly se la condizione valutata dall'istruzione di salto `jne` è vera, allora la sequenza di codice derivante dalla traduzione del blocco *then* viene saltata.

4.3.2 Forma intermedia

Ciascuna istruzione assembly viene trasformata in una o più *istruzioni intermedie* [18]. Si tratta di una forma di rappresentazione particolarmente semplice, che consente di ottenere una serie di vantaggi che agevolano considerevolmente le analisi successive:

- vengono esplicitati gli effetti di ciascuna istruzione sui registri del processore e sulla memoria.
- Viene ridotto considerevolmente il numero delle differenti tipologie di istruzioni. Si consideri, ad esempio, che un attuale processore Intel x86 a 32 bit supporta oltre 300 istruzioni diverse [46]. Trattare un numero così elevato di *opcode* renderebbe l'analisi del tutto impraticabile.
- Si ottiene una rappresentazione del codice dell'applicazione indipendente dal processore sottostante.

La rappresentazione intermedia adottata comprende solamente 4 differenti tipologie di istruzioni e 5 tipi di espressioni. Le classi di espressioni supportate sono le seguenti:

`Constant` rappresenta semplicemente un valore costante. Un'espressione di questo tipo verrà indicata nel seguito con `c`*n*`(`*val*`)`, dove *val* è un letterale (solitamente esadecimale) che indica il valore rappresentato, mentre *n* è la dimensione in bit di *val*. Ad esempio, il valore esadecimale a 32 bit *0x08048415* sarà indicato con l'espressione costante `c32(0x08048415)`.

`Register` un'espressione di questo tipo fa riferimento ad un particolare registro del processore. In generale, la sua forma è `r`*n*`(`*nome*`)`, dove *nome* è il nome di un registro a *n* bit. Ad esempio, quando si farà riferimento al registro a 32 bit `EAX` si scriverà `r32(EAX)`.

`OverlappingRegister` le istruzioni che possono avere come argomento un registro del processore possono non soltanto agire su un intero registro fisico, ma anche solo su una sua parte. Un'espressione che appartiene alla classe `OverlappingRegister` fa quindi riferimento ad un registro "*virtuale*" che fisicamente è mappato su una regione del corrispondente registro fisico. Per fare un esempio, sugli attuale processori Intel IA-32 di ciascun registro a 32 bit (es. `EAX`) possono essere utilizzati solo i 16 bit inferiori (es. `AX`), solo gli 8 bit inferiori (es. `AL`), oppure solamente gli 8 bit superiori della *word* meno significativa (es. `AH`). La rappresentazione utilizzata per questo tipo di espressione è del tutto

analoga a quella del caso precedente: il registro `AL`, ad esempio, verrà indicato con `r8(AL)`.

Esiste poi il caso particolare del registro di stato (`EFLAGS`) i cui singoli bit (*flag*) vengono spesso riferiti singolarmente. Per questo motivo si indicherà, ad esempio, il bit *overflow flag* del registro di stato con `r1(OF)`.

`Memory` una zona di memoria di *n* bit e con indirizzo di partenza *ind* viene indicata con `mn(ind)`, dove *ind* può essere una costante, un registro, un'espressione composita (descritta nel punto seguente) o anche un'altra zona di memoria. Quest'ultimo caso non è ottenibile dalla traduzione diretta di un'espressione assembly, ma può comunque verificarsi in seguito alle trasformazioni applicate nel corso dell'analisi.

Per fare un esempio, il byte di memoria il cui indirizzo è dato dal contenuto del registro a 32 bit `EBP` è facilmente rappresentabile come `m8(r32(EBP))`.

`EExpression` si tratta di un'espressione composita, sia essa unaria, binaria o ternaria. Un'espressione unaria è definita ricorsivamente come $op\ (exp)$, dove op è l'operatore di negazione logica $\neg$, l'operatore di negazione *bitwise* $\sim$ oppure uno degli operatori aritmetici unari $+$ o $-$, mentre *exp* è a sua volta un'espressione intermedia di qualunque tipo. Allo stesso modo, possiamo definire un'espressione binaria come $(exp1)\ op\ (exp2)$, dove $exp1$ e $exp2$ sono espressioni, mentre op può essere un qualunque operatore binario aritmetico, logico o *bitwise*. Infine, un'espressione ternaria è un modo compatto per rappresentare costrutti di selezione del tipo *if-then-else* e ha la forma $(exp_{cond})?(exp_{vero}) : (exp_{falso})$, con exp_{cond}, exp_{vero} e exp_{falso} espressioni intermedie. Il valore dell'intera espressione sarà exp_{vero} se la condizione exp_{cond} è vera, oppure exp_{falso} nel caso contrario.

Analogamente a quanto fatto per le espressioni, viene riportata di seguito una breve descrizione delle categorie di istruzioni disponibili nel linguaggio intermedio:

`Assignment` la classe `Assignment` rappresenta una generica istruzione di assegnamento della forma *dest* $:=$ *sorg*, dove *dest* è un'espressione interme-

dia di tipo `OverlappingRegister`, `Register` o `Memory`, mentre *sorg* è un'espressione intermedia qualsiasi.

`Jump` un'istruzione `Jump` viene utilizzata per rappresentare salti condizionati o incondizionati. La sintassi è *JUMP* [*cond*] *target*, dove *target* è un'espressione che indica l'indirizzo da cui l'esecuzione deve continuare, mentre *cond* è un'espressione utilizzata come condizione del salto: se *cond* è vera, allora l'esecuzione continua dall'indirizzo indicato da *target*, mentre se *cond* è falsa la prossima istruzione ad essere eseguita sarà quella che segue sintatticamente l'istruzione di salto. Naturalmente, nel caso di salti incondizionati (es. *opcode* Intel `jmp`) *cond* è omessa.

`Call` questa tipologia di istruzione viene utilizzata per modellare una chiamata a procedura e ha sintassi *CALL target*, dove *target* è un'espressione che indica l'indirizzo della prima istruzione della procedura da invocare. È importante sottolineare che la semantica dell'istruzione intermedia `Call` è notevolmente diversa rispetto a quella dell'istruzione Intel x86 `call`. Quest'ultima, infatti, ha un comportamento piuttosto complesso, in quanto si occupa di salvare sullo stack l'indirizzo dell'istruzione del chiamante successiva alla chiamata, aggiornare il puntatore alla cima dello stack e, solo alla fine, trasferire il controllo all'indirizzo specificato. Nella nostra rappresentazione intermedia, quindi, un'istruzione assembly di `call` è di fatto tradotta in una coppia di assegnamenti che esplicitano le operazioni sullo stack e un'istruzione `Call` finale che trasferisce il controllo al chiamato.

`Ret` un'istruzione `Ret` viene utilizzata per terminare l'esecuzione di una procedura e restituire il controllo al chiamante. La sintassi è semplicemente *RET*, senza alcun argomento. Anche in questo caso valgono considerazioni analoghe a quelle fatte per l'istruzione `Call`. Semanticamente, infatti, un'istruzione assembly x86 `ret` comporta l'estrazione dell'elemento in cima allo stack, l'aggiornamento del puntatore alla cima dello stack e il trasferimento del controllo all'indirizzo indicato dall'elemento estratto. Ancora una volta, al fine di esplicitare gli effetti

di ciascuna istruzione assembly, ad una `ret` x86 vengono fatte corrispondere più istruzioni intermedie, di cui solamente l'ultima sarà di tipo `Ret`.

Utilizzando le classi di espressioni e istruzioni appena viste, è possibile trattare tutte le istruzioni assembly ricavate dalla fase di *disassembly*, traducendo ciascuna di esse in una o più istruzioni intermedie.

È importante sottolineare che la forma intermedia utilizzata non è affatto minimale, bensì ulteriormente riducibile. Estremizzando si potrebbe arrivare a ridurre l'insieme delle istruzioni fino a comprendere il solo assegnamento: un'istruzione di `Jump`, ad esempio, è interpretabile come un assegnamento al registro `EIP` (*instruction pointer*) del processore e allo stesso modo si potrebbe pensare di eliminare anche le altre classi di istruzioni. L'obiettivo della trasformazione in forma intermedia, però, non è quello di minimizzare il numero di tipologie di istruzioni necessarie a rappresentare il codice dell'applicazione, ma piuttosto ridurre la cardinalità dell'insieme degli *opcode* di un'architettura CISC come quella Intel fino a semplificare il più possibile le analisi successive. Per chiarire questo concetto, si consideri l'istruzione `Call`. Sono già state evidenziate le differenze semantiche tra un'istruzione assembly di `call` e l'istruzione intermedia `Call` ed è già stato discusso come la prima venga espansa in più istruzioni intermedie. Si potrebbe quindi pensare che, visto che l'istruzione `Call` non sottointende alcuna operazione sullo stack, questa possa essere sostituita da una semplice `Jump`. Di fatto un'operazione di questo tipo sarebbe pienamente legittima e ridurrebbe ulteriormente l'insieme delle tipologie di istruzioni intermedie necessarie. Così facendo, però, si complicherebbero notevolmente le analisi effettuate nelle fasi successive, poiché non si sarebbe più in grado di individuare i limiti di ciascuna procedura. Considerazioni del tutto analoghe possono essere fatte anche per l'istruzione `Ret`.

In Figura 4.4 è stata riportata la traduzione in forma intermedia del frammento assembly di Figura 4.3. Il lato destro delle definizioni di alcuni registri di stato è stato omesso per ragioni di spazio e perché comunque non rilevante per il prosieguo della trattazione. La numerazione sulla sinistra fa riferimento al codice assembly dell'esempio precedente. Per fare un esempio, l'istruzione intermedia 2_5 è la quinta istruzione

```
1_1   r32(EAX) := m32[(r32(EBP) + c32(0xffffffff8))]
2_1   r32(TMP) := r32(EAX)
2_2   r32(EAX) := (r32(EAX) + m32[(r32(EBP) + c32(0x8))])
2_3   r1(CF) := ...
2_4   r1(OF) := ...
2_5   r1(SF) := (c32(0x1) & (r32(EAX) >> c32(0x1f)))
2_6   r1(ZF) := (r32(EAX) == c32(0x0))
3_1   r32(EAX) := m8[r32(EAX)]
4_1   r8(TMP) := (r8(AL) - c8(0x27))
4_2   r1(CF) := ...
4_3   r1(OF) := ...
4_4   r1(SF) := (c8(0x1) & (r8(TMP) >> c8(0x7)))
4_5   r1(ZF) := (r8(TMP) == c8(0x0))
5_1   JUMP ((r1(ZF) == c1(0x0))) c32(0x80484f7)
```

Figura 4.4: Traduzione in forma intermedia del codice assembly di Figura 4.3.

del gruppo di istruzioni intermedie con cui è stata tradotta l'istruzione assembly alla riga 2 della Figura 4.3.

Ad esempio, si consideri come è stata tradotta l'istruzione `cmp $0x27,%al` (riga 4). Un'istruzione di `cmp` confronta i due operandi sottraendoli, senza memorizzare il risultato ma modificando solamente il registro di stato del processore in modo da rispecchiare l'esito del confronto. Per modellare questa espressione, la forma intermedia utilizzata prevede la memorizzazione del risultato della sottrazione in un registro temporaneo (`TMP`) fittizio. I *flag* coinvolti nell'operazione sono poi definiti utilizzando espressioni appropriate. Un esempio di questa esplicitazione degli effetti sui *flag* è dato dall'assegnamento 4_4: in seguito al confronto, il flag di segno (`SF`) sarà impostato solo se il risultato della sottrazione logica tra gli operandi sarà negativo, ovvero se il bit più significativo del registro temporaneo varrà 1.

4.3.3 Costruzione dei CFG

Ciascun gruppo di istruzioni intermedie ottenuto in seguito alla traduzione di un'istruzione assembly va ad aggiornare il CFG della procedura a cui appartiene. I basic block di ogni CFG contengono quindi le istruzioni intermedie e il raggiungimento di un'istruzione che modifica il flusso di esecuzione (`Jump`, `Call` o `Ret`) provoca la

terminazione del basic block corrente e l'inizio dell'esplorazione dei nuovi cammini di esecuzione dettati da tale istruzione. Per maggiori dettagli sull'algoritmo utilizzato per la costruzione e l'aggiornamento dei CFG, si rimanda alla sezione 5.1.

In Figura 4.5 è riportato il CFG della procedura vulnerabile `process_input()`. 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, dovuto all'esplicitazione degli effetti delle istruzioni assembly.

4.3.4 Analisi di *liveness*, dipendenze di controllo e cicli

Terminato il ciclo di *disassembly*, traduzione delle istruzioni e aggiornamento dei *control flow graph* delle procedure, si ottiene una panoramica approssimativa del comportamento dell'applicazione. L'approssimazione, come già ricordato, è legata al mancato esame delle zone di codice raggiungibili solo mediante un'istruzione di `Jump` o `Call` indiretta e verrà gestita nel corso della successiva analisi dinamica. Si passa ora ad effettuare una serie di analisi statiche che integrano le informazioni già raccolte e agevolano ulteriormente i passi successivi.

Innanzi tutto l'esplicitazione delle operazioni sul registro di stato del processore porta alla generazione di una considerevole quantità di codice inutile: molte definizioni dei *flag* sono infatti *dead*, ovvero la variabile definita non viene poi utilizzata prima della successiva ridefinizione. Tali istruzioni di assegnamento possono quindi essere rimosse preservando comunque la semantica del programma. L'algoritmo utilizzato è descritto in dettaglio nella sezione 5.2 e, in sostanza, consente di determinare, per ogni basic block, l'insieme delle definizioni dei *flag* del processore che risultano essere inutili e quindi rimovibili. Si consideri, ad esempio, la definizione del *flag* `ZF` effettuata dall'istruzione 2_6 del codice intermedio di Figura 4.4. Lo stesso *flag* viene poi ridefinito poche righe dopo dall'istruzione 4_5. Visto che esiste un unico cammino $2_6 \rightarrow 4_5$ e dato che non esiste alcuna istruzione su tale cammino che utilizza `ZF`, possiamo dedurre che l'assegnamento 2_6 è inutile (*dead*) e può quindi essere rimosso. Si è osservato che l'esecuzione di una simile analisi di *liveness* sui *flag* del processore porta, mediamente, ad una riduzione del numero delle istruzioni di ciascuna procedu-

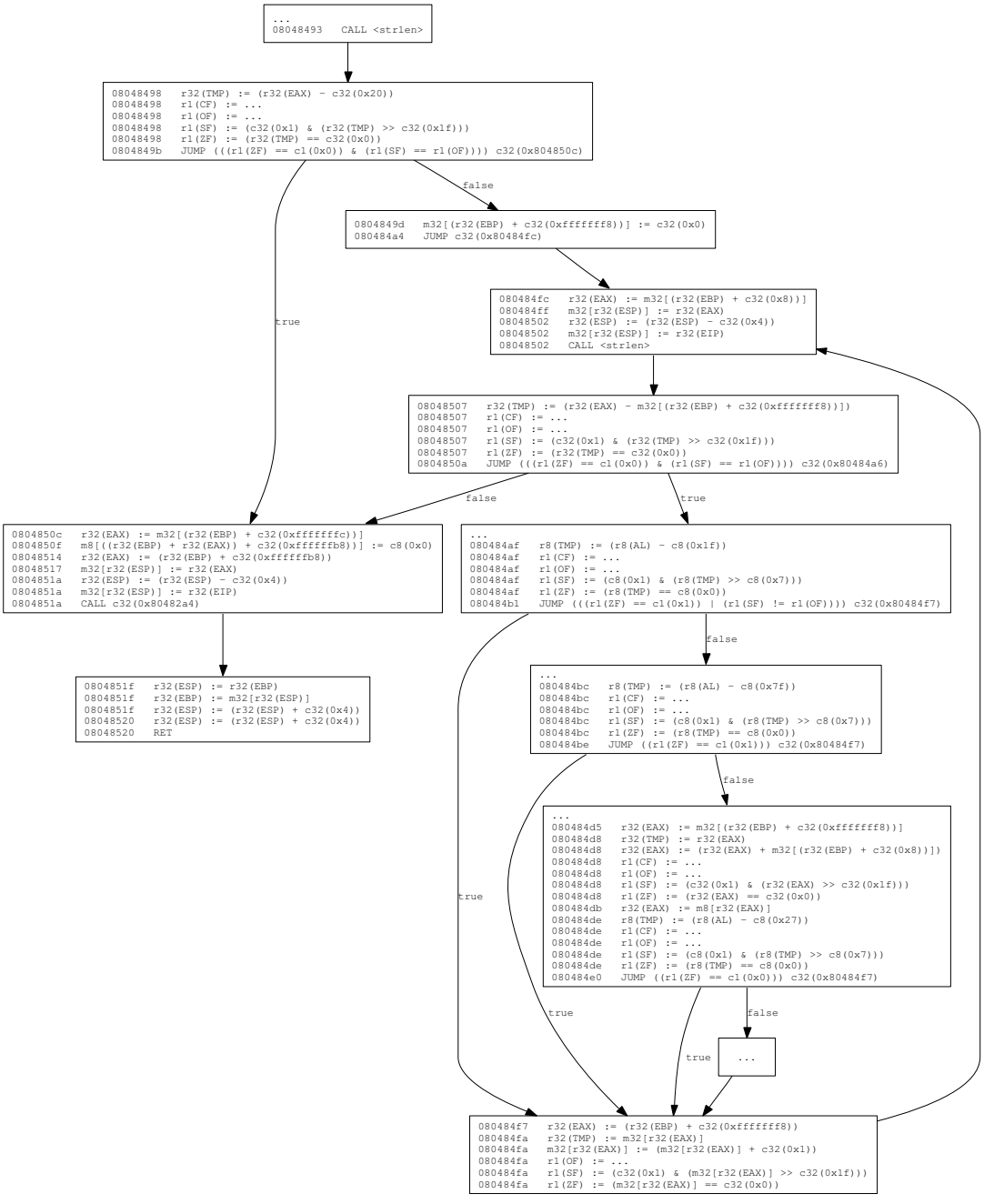

Figura 4.5: *Control flow graph* ottenuto staticamente dall'analisi dell'esempio di Figura 3.1.

ra intorno al $30-40\%$. Si è scelto di non estendere l'analisi anche ad altri registri o locazioni di memoria in quanto non si avrebbero risultati paragonabili. Il compilatore che ha generato l'applicazione, infatti, avrà già compiuto operazioni analoghe durante la compilazione del sorgente, per cui si riuscirebbero ad individuare ben poche definizioni *dead*.

È comunque importante fare una precisazione: l'applicazione di una qualsiasi analisi di *liveness* su un CFG approssimato può teoricamente portare a considerare *dead* un assegnamento solamente perché il codice del programma non è ancora stato visitato completamente. Ad esempio, se un'istruzione di assegnamento alla variabile v è seguita da un'istruzione di `Jump` indiretta (es. `JUMP (r32(EAX))`) le cui possibili destinazioni non sono ancora state individuate con precisione, allora non è possibile affermare con certezza se, in corrispondenza del salto, v è *live* oppure *dead*. Sarebbe comunque possibile adottare un approccio totalmente conservativo e supporre che ogni basic block concluso da un'istruzione di `Jump` o `Call` indiretta non ancora analizzata accuratamente abbia come successore un nodo con comportamento sconosciuto (*hell node*, [79]), che si presuppone possa utilizzare qualunque variabile e definisca, con contenuto indeterminato, ogni registro e locazione di memoria. Così facendo, però, si ridurrebbe notevolmente la precisione dell'analisi. Sperimentalmente si è osservato che, supponendo che l'eseguibile analizzato sia stato generato dalla compilazione di un sorgente di alto livello e che segua un "modello di compilazione standard" [9], la *liveness analysis* sui *flag* del processore non porta alla rimozione di codice *live*.

Si passa quindi al calcolo delle dipendenze di controllo tra i nodi del CFG. Informalmente, diciamo che un basic block b_2 ha una dipendenza di controllo [30] da un altro basic block b_1 se il fatto che le istruzioni di b_2 vengano o meno eseguite dipende dalla condizione valutata in b_1. Per ciascuna procedura dell'applicazione, il relativo CFG viene analizzato al fine di calcolare le dipendenze di controllo di ogni *basic block*. L'algoritmo utilizzato è quello proposto in [21] e basato sulla computazione della frontiera di dominanza del CFG inverso. Verrà poi illustrato nella Sezione 5.3.

Un punto fondamentale dell'approccio presentato è costituito dall'analisi dei cicli, in quanto, come si vedrà più dettagliatamente in seguito, consentirà in alcune situazioni di agevolare la ricerca delle vulnerabilità tramite l'astrazione del comportamento

di un ciclo. Una prima parte di questo processo può essere effettuata staticamente: i CFG sono esaminati in modo da identificare i cicli insieme con i corrispondenti punti di ingresso e di uscita, utilizzando l'algoritmo descritto nella Sezione 5.4. Ciò consentirà in seguito all'analizzatore dinamico di determinare quando sta per cominciare una nuova iterazione di un ciclo, oppure quando l'esecuzione sta per uscire da un costrutto di iterazione.

4.4 Analisi dinamica

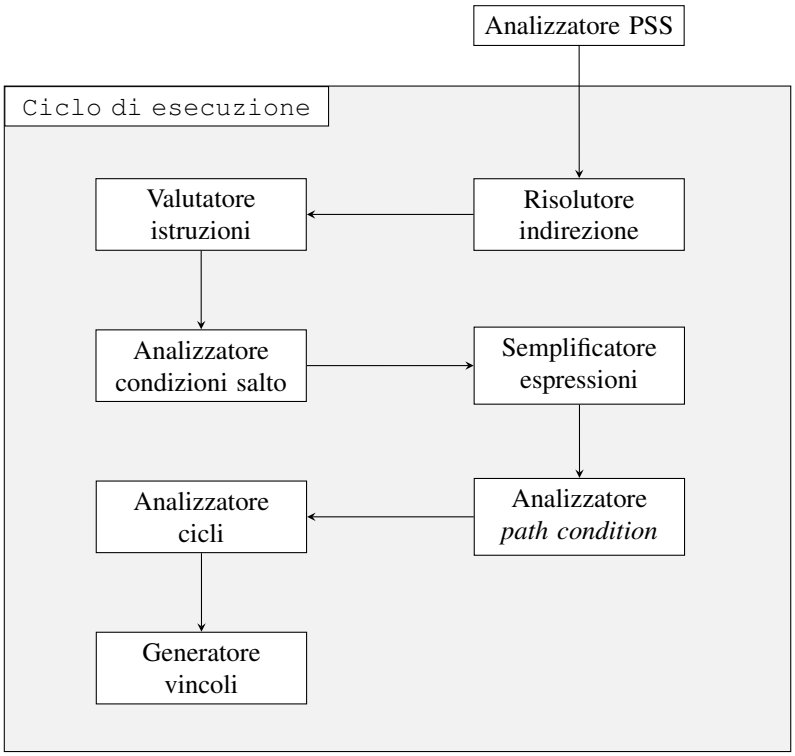

Figura 4.6: Struttura del motore di analisi dinamica.

Il compito del motore di analisi dinamica è quello di eseguire e monitorare l'applicazione al fine di dedurre le informazioni necessarie alla generazione di un insieme di vincoli che si ritiene possano condurre alla sovrascrittura di una zona di memoria delicata con dati provenienti dall'input e, quindi, sotto il controllo di un eventuale

attaccante. Questo modulo opera sulla base delle informazioni prodotte dalla precedente analisi statica, raffinandole e arricchendole ulteriormente con i dati raccolti dinamicamente. È però opportuno ricordare che le informazioni ricavate staticamente sono indipendenti dalla specifica esecuzione, mentre l'analisi dinamica è legata ad un'esecuzione particolare e, quindi, ad una particolare configurazione delle variabili in ingresso. Le informazioni ricavate dinamicamente non sono quindi generalizzabili ad ogni esecuzione del programma e, di conseguenza, nel caso in cui una ramificazione dell'esecuzione porti a rieseguire l'applicazione su un nuovo input, le analisi dinamiche dovranno essere effettuate nuovamente.

Il processo di analisi dinamica è schematizzato in Figura 4.6. Prima di iniziare l'esecuzione del nuovo processo, il suo stato iniziale viene esaminato. Il ciclo di esecuzione può quindi incominciare: il processo è eseguito istruzione per istruzione, avendo cura di ripetere la precedente analisi statica sulle nuove regioni di codice scoperte in seguito alla risoluzione delle destinazioni di salti o chiamate a funzione indirette. Non appena eseguita, ogni istanza di istruzione è memorizzata in una *execution history* e quindi analizzata e semplificata in modo da esplicitarne gli effettivi usi e definizioni. Si passa quindi all'analisi delle condizioni di salto, primo passo fondamentale verso la ricostruzione dei predicati di alto livello legati ai salti condizionati. Successivamente, si procede con la semplificazione delle espressioni mediante trasformazioni logico-matematiche o sintattiche. Si arriva quindi alla generazione delle *path condition*, ovvero all'esplicitazione delle relazioni esistenti tra salti condizionati e dati in ingresso; si tratta di un passo indispensabile per ottenere espressioni di alto livello utilizzabili poi per la definizione dei vincoli sull'input. Vengono quindi analizzati dinamicamente i cicli incontrati nel corso dell'esecuzione, cercando di astrarne il comportamento in modo da dedurre il numero di iterazioni necessarie per raggiungere particolari stati dell'applicazione. Infine, quando è necessario ramificare l'esecuzione per esplorare nuovi cammini, il motore di analisi dinamica fa uso di un generatore di vincoli per esaminare le informazioni raccolte e costruire un insieme di condizioni sull'input adatte a condurre l'applicazione fino allo stato desiderato.

4.4.1 Analisi dello stato iniziale del processo

Prima di poter proseguire con il monitoraggio dell'esecuzione del processo, è necessario raccogliere alcune informazioni circa il suo stato iniziale. Tali informazioni verranno memorizzate all'interno di una struttura dati chiamata *process startup state* (*PSS*).

Innanzi tutto è importante analizzare lo spazio di indirizzamento del processo al fine di estrarre eventuali parametri da riga di comando e variabili d'ambiente. Si tratta, infatti, di dati in ingresso all'applicazione che potrebbero venire manipolati da un attaccante. Nella pratica, questa particolare operazione è piuttosto delicata, in quanto legata alla struttura della memoria assegnata al processo dal sistema operativo. È quindi evidente che questo modulo non potrà prescindere dal particolare sistema operativo utilizzato, in quanto sistemi operativi differenti potrebbero organizzare tale area di memoria in modo diverso e addirittura versioni differenti dello stesso sistema potrebbero introdurre varianti sostanziali (come ad esempio è accaduto in Linux con l'introduzione della randomizzazione dello spazio di indirizzamento virtuale a partire dal kernel 2.6.12).

Anche lo stato dei registri del processore viene memorizzato nel *PSS*. In questo modo se durante l'esecuzione dell'applicazione viene incontrata un'istruzione che utilizza un registro non ancora definito da un assegnamento, si potrà fare direttamente riferimento al valore di quel registro così come è stato memorizzato nello stato iniziale del processo.

In realtà si può avere un problema del tutto analogo con le locazioni di memoria: può accadere che un'istruzione utilizzi il valore memorizzato in una locazione di memoria non ancora esplicitamente definita. Questo può essere sintomo di un errore di programmazione (utilizzo di una variabile non ancora inizializzata) oppure, più probabilmente, il valore desiderato è quello memorizzato dal sistema operativo al momento della creazione del processo (come, ad esempio, nel caso di una variabile globale inizializzata, memorizzata su Linux nella sezione `.data`). In questo caso, però, non è conveniente salvare nel *PSS* il contenuto iniziale dell'*intero* spazio di indirizzamento del processo. Per questo motivo la questione viene risolta dinamicamente: nel momen-

to in cui un'istruzione cerca di recuperare il contenuto di una zona di memoria, se tale zona non è stata precedentemente definita da un'istruzione, allora ne viene salvato il contenuto nel *PSS*. In questo modo lo stato iniziale del processo conterrà informazioni circa le sole locazioni di memoria utilizzate dall'applicazione senza prima essere state esplicitamente definite da un'istruzione di assegnamento.

4.4.2 Monitoraggio dell'esecuzione

Terminata l'analisi dello stato iniziale del processo, può incominciare il ciclo di esecuzione e monitoraggio dell'applicazione. Ogni istanza di istruzione intermedia appartenente al gruppo associato all'istruzione assembly appena eseguita viene aggiunta in coda all'*execution history* relativa al processo. Tale struttura risulta particolarmente utile alle analisi successive come, ad esempio, per calcolare con precisione le *dynamic reaching definition* [1] e formulare interrogazioni circa il flusso dei dati nel corso dell'esecuzione. Iniziano quindi una serie di analisi dinamiche finalizzate alla costruzione delle *path condition* connesse al cammino di esecuzione corrente e all'eventuale ramificazione dell'esecuzione.

4.4.2.1 Risoluzione di salti e chiamate a funzione indirette

Nel caso in cui l'istruzione assembly corrente sia una `Jump` o una `Call` indiretta, è necessario valutare l'indirizzo di destinazione a cui il controllo sta per essere trasferito. Si osservi che dinamicamente questo comporta semplicemente la valutazione del contenuto di un registro o di una zona di memoria, entrambe operazioni possibili tramite l'osservazione diretta della locazione desiderata. In ogni caso, è possibile che, risolvendo l'indirizzo da cui l'esecuzione dovrà riprendere, ci si trovi a dover eseguire una nuova zona di codice che non è stata analizzata nella fase statica, poiché non raggiungibile tramite l'impiego del *disassembler* ricorsivo utilizzato. Per questo motivo può essere necessario invocare nuovamente l'analizzatore statico al fine di esaminare la regione di codice appena scoperta e aggiornare di conseguenza i CFG.

4.4.2.2 Valutazione delle istruzioni

Operando a *run-time*, è possibile determinare con precisione le aree di memoria utilizzate e definite da ciascuna istruzione. Per questo motivo, per ciascuna istanza di istruzione, ogni espressione utilizzata viene esaminata e *attualizzata*.

Per fare un esempio, l'istruzione 3_1 del codice intermedio di Figura 4.4 utilizza chiaramente il registro `EAX`. In realtà, però, l'istruzione usa anche la locazione di memoria a 8 bit con indirizzo `EAX`. L'indirizzo di tale zona di memoria non è però determinabile staticamente, a meno di complesse e potenzialmente inaccurate analisi sulle definizioni di `EAX` che raggiungono quel punto del programma. Procedendo dinamicamente, però, possiamo asserire con certezza che, ad esempio, quando l'istruzione 3_1 viene raggiunta per la prima volta, `EAX` contiene il valore *0xBFFFFAD1*, per cui l'istruzione utilizza anche la locazione di memoria `m8[c32(0xBFFFFAD1)]`. È importante notare che tale valutazione non deve essere fatta per ogni istruzione, bensì per ciascuna *istanza* di istruzione. Nel nostro esempio, l'istruzione 3_1 potrebbe essere eseguita più di una volta, magari perché parte del corpo di un ciclo. Ad ogni esecuzione di 3_1 il registro `EAX` potrebbe contenere un valore differente, quindi l'istruzione potrebbe referenziare zone di memoria sempre diverse.

Per ogni istanza di istruzione s, la valutazione delle istruzioni e l'attualizzazione delle espressioni consentono quindi di determinare con precisione e senza bisogno di ricorrere a stime conservative gli insiemi $\mathit{define}(s)$ e $\mathit{use}(s)$ contenenti rispettivamente le variabili (registri o locazioni di memoria) definite e utilizzate da s.

4.4.2.3 Analisi delle condizioni di salto

Ogni volta che un'istruzione di salto condizionato viene eseguita si effettua l'analisi delle condizioni di salto, al fine di ricostruire una formulazione ad alto livello del predicato valutato dall'espressione. Si è già sottolineato in precedenza come il problema fondamentale sia legato al fatto che, a livello del codice macchina, le condizioni di salto sono specificate come espressione sui *flag* del processore, quindi è necessario definire un procedimento che consenta l'estrapolazione della semantica del predicato. Per chiarire la questione, si consideri l'istruzione di salto condizionato alla riga 5 del

frammento di codice assembly in Figura 4.3. Si tratta di un'istruzione di salto `jne` (*jump-if-not-equal*) che trasferisce il controllo all'indirizzo indicato come argomento se il *flag* `ZF` non è impostato. Come si può osservare dall'istruzione 5_1 di Figura 4.4, la traduzione in forma intermedia ha esplicitato la condizione valutata su `ZF`.

È stato sviluppato un algoritmo per l'analisi dei salti condizionati, basato principalmente sull'analisi delle *reaching definition* dei registri *flag* e sull'applicazione di un modello che esprime la semantica dell'espressione di salto. Tale algoritmo è descritto in dettaglio nella Sezione 5.5. Nel caso dell'esempio precedente, l'applicazione dell'algoritmo sviluppato arriva a trasformare la condizione sul *flag* `ZF` in un'espressione del tipo `r8(AL) != c8(0x27)`. Si noti che, nel momento in cui l'istruzione di salto viene raggiunta, il registro `AL` contiene effettivamente il carattere `buf[i]`, mentre la costante $0x27$ rappresenta, in codifica ASCII, il valore corrispondente al carattere `'\''`.

4.4.2.4 Semplificazione delle espressioni

Le espressioni utilizzate dalle istruzioni intermedie sono spesso piuttosto complesse e articolate, in particolar modo quelle associate ad assegnamenti generati durante la trasformazione in forma intermedia allo scopo di esplicitare l'effetto di un'istruzione assembly sul registro di stato del processore. Allo scopo di agevolare le analisi successive, ciascuna espressione viene semplificata applicando una serie di trasformazioni riassunte nelle categorie riportate di seguito.

Logiche. Si tratta di semplici semplificazione logiche che consentono, però, di ridurre considerevolmente la complessità di un'espressione.

Per fare un esempio, un'espressione logica del tipo $exp == exp$ è sempre banalmente verificata e può quindi essere sostituita con la costante 1, a cui è associato il valore logico "*vero*".

Matematiche. Sono trasformazioni finalizzate alla semplificazione delle espressioni tramite l'impiego di proprietà aritmetiche di base.

Questa categoria comprende, ad esempio, la semplificazione di operazioni aritmetiche che coinvolgono l'elemento neutro dell'operatore ($exp + 0 \rightarrow exp$),

l'applicazione di proprietà associative che possono condurre ad un'ulteriore riduzione della complessità ($(exp + c_1) + c_2 \rightarrow exp + (c_1 + c_2)$, con exp espressione e c_1, c_2 valori costanti) e la combinazione di valori costanti ($c_1 + c_2 \rightarrow c_3$, con c_1, c_2, c_3 valori costanti e $c_3 = c_1 + c_2$).

Sintattiche. Questa categoria comprende semplificazioni che non sono finalizzate direttamente alla riduzione della complessità di un'espressione, ma piuttosto alla trasformazione dell'espressione in una forma canonica, sulla quale basare le analisi successive.

Per fare un esempio, un'espressione del tipo $c + exp$, con c costante e exp espressione, verrà trasformata applicando la proprietà commutativa dell'operatore di somma aritmetica, arrivando ad ottenere la nuova espressione $exp + c$.

Nonostante la semplificazione delle espressioni avrebbe potuto già avere luogo al momento della trasformazione del codice assembly in forma intermedia, tale operazione è stata posticipata fino a dopo l'analisi delle condizioni di salto. Il motivo di questa scelta è legato al fatto che, come già sottolineato in precedenza, per la ricostruzione dei predicati di alto livello connessi ai salti condizionati si ricorre all'applicazione di modelli (*template*) che riassumono la semantica del predicato condizionale. Per compiere tale operazione è necessario esaminare gli operandi delle istruzioni di assegnamento ai registri *flag* del processore coinvolti nella condizione di basso livello associata al salto. L'applicazione delle trasformazioni di semplificazione potrebbe quindi alterare il lato destro dell'istruzione di assegnamento, andando così a complicare ulteriormente la procedura.

4.4.2.5 Analisi delle *path condition*

Finora si è discusso su come il predicato di basso livello associato ad un salto condizionato possa essere trasformato in un'espressione di più alto livello semanticamente equivalente alla condizione di partenza. Tale operazione, però, non è ancora sufficiente per poter dedurre se e quale porzione dei dati in ingresso è utilizzata in una condizione di salto. Si tratta, evidentemente, di un requisito fondamentale per la successiva generazione di un insieme di vincoli da utilizzare per definire un nuovo input che guidi

l'applicazione fino allo stato desiderato. In sostanza, quindi, le espressioni risultanti dalla precedente analisi delle condizioni di salto devono essere ulteriormente raffinate e trasformate in vincoli sui dati in ingresso (*path condition*).

L'algoritmo sviluppato allo scopo di dedurre le relazioni tra condizione di salto e input è descritto in dettaglio nella Sezione 5.6. Tale algoritmo si basa principalmente sull'analisi dell'espressione utilizzata per esprimere la condizione, già trasformata dalle fasi precedenti. Nell'espressione vengono propagate ricorsivamente le *reaching definition* relative a ciascuna variabile utilizzata, arrestando la propagazione nel momento in cui si incontra una variabile connessa ai dati in ingresso. Alla fine del procedimento si otterrà una nuova espressione che utilizza solamente valori costanti e, eventualmente, variabili legate all'input.

Per chiarire l'importanza di questo passo, si consideri ad esempio l'espressione `r8(AL) != c8(0x27)` ottenuta in seguito all'analisi delle condizioni di salto e si supponga che il procedimento sopra descritto venga compiuto in corrispondenza della prima iterazione del ciclo *for* di Figura 3.1. Supponendo che il parametro `buf` sia connesso ai dati di input dell'applicazione, il procedimento di propagazione che caratterizza l'analisi delle *path condition* trasforma la condizione in una nuova espressione del tipo `m32(c32(0xBFFFFAD1))!= c8(0x27)`, dove la locazione di memoria a 32 bit con indirizzo *0xBFFFFAD1* corrisponde effettivamente al primo byte del vettore `buf`. A meno dei simboli, si è arrivati alla completa ricostruzione dell'espressione di alto livello `buf[i] == '\''` valutata dal costrutto *if* alla riga 9 del frammento di codice C di Figura 3.1.

Una volta completata la ricostruzione delle condizioni di alto livello, l'insieme dei vincoli dedotti viene esaminato allo scopo di individuare sottoinsiemi di condizioni astraibili in una rappresentazione più compatta ma semanticamente equivalente. Infatti, nonostante le analisi condotte fino a questo punto abbiano consentito di esplicitare le relazioni esistenti tra i predicati condizionali e i dati in ingresso all'applicazione, la granularità dei predicati è comunque quella propria del codice assembly. Per questo motivo, all'interno dell'insieme dei vincoli vengono ricercate sequenze di condizioni traducibili in concetti di più alto livello, come "*lunghezza di una stringa*" o "*confronto tra due zone di memoria*".

Per fare chiarezza, si consideri ancora una volta la funzione di Figura 3.1 e si supponga di eseguire l'applicazione con input `buf = 'abc'`. Allora, non appena terminato l'esame del codice assembly che si occupa della valutazione del predicato condizionale alla riga 5 del sorgente, si riusciranno a dedurre delle condizioni sull'input del tipo:

```
buf[0] != 0 ∧ buf[1] != 0 ∧ buf[2] != 0 ∧ buf[3] == 0.
```

A livello assembly, infatti, non si dispone sempre delle informazioni necessarie per riconoscere la chiamata alla funzione di libreria `strlen`: i simboli potrebbero essere completamente mancanti oppure il compilatore potrebbe ricorrere ad ottimizzazioni che rendono particolarmente complesso il compito di riconoscere l'invocazione della procedura (in particolare, `gcc` spesso sostituisce un'istruzione del tipo `call <strlen>` con il corpo stesso della funzione, utilizzando una tecnica nota come *function inlining*). Se da un lato il problema di identificare funzioni di libreria in corrispondenza di simboli mancanti o incompleti può essere comunque almeno in parte risolto [38, 26], gestire operazioni di *inlining* da parte del compilatore è piuttosto complicato e, soprattutto, l'impiego di tecniche per il riconoscimento delle funzioni di libreria potrebbe non consentire di identificare reimplementazioni da parte dell'utente di procedure standard. L'unica soluzione pare quindi consistere nel ricostruire la *semantica* della funzione di alto livello a partire dalle osservazioni fatte sul codice assembly. Ritornando all'esempio appena considerato, si è in grado di riconoscere la semantica di una sequenza di uguaglianze e disuguaglianze come quelle sopra riportate e rappresentarla in una forma più compatta del tipo `strlen(buf) == 3`.

4.4.2.6 Analisi dei cicli

L'analisi dei costrutti di iterazione è una questione piuttosto complessa: il numero di iterazioni compiute da un ciclo potrebbe essere indeterminato e il comportamento di ciascuna iterazione potrebbe differire in modo sostanziale dalle altre, andando, ad esempio, a definire zone di memoria completamente diverse. Molti approcci che hanno affrontato il problema da un punto di vista puramente statico sono stati costretti ad

introdurre forti semplificazioni. Per fare un esempio, la tecnica di analisi statica descritta in [20] suppone di esaminare ogni ciclo considerando al più tre iterazioni, oltre le quali il comportamento del ciclo diventa "indefinito". Procedendo dinamicamente, invece, la questione è, almeno in parte, più semplice. Supponendo di avere in precedenza identificato il costrutto di iterazione, è infatti possibile determinare quando sta per incominciare una nuova iterazione oppure quando l'esecuzione sta per uscire dal ciclo.

In particolare, il modello *smart fuzzing* impiega una tecnica di analisi che, tramite il monitoraggio di ogni iterazione, consente di astrarre il comportamento del ciclo dallo specifico insieme dei dati in ingresso connesso ad una particolare esecuzione. Innanzi tutto vengono osservate dinamicamente le relazioni esistenti tra la condizione di terminazione del ciclo e i dati in ingresso all'applicazione. Si procede quindi ad identificare accuratamente le variabili di induzione e a dedurre come queste possano influenzare l'indirizzo di destinazione di un'istruzione di scrittura in memoria. In base alle informazioni raccolte si è in grado, in alcuni casi, di generare un insieme di vincoli che definiscono un nuovo input potenzialmente pericoloso per la sicurezza dell'applicazione. Si osservi che la tecnica di analisi proposta consente di produrre un input pericoloso senza avere la necessità di eseguire il ciclo in questione compiendo ogni possibile numero di iterazioni.

Per perseguire tale obiettivo, l'analisi ideata prevede i seguenti passi:

1. innanzi tutto i cicli devono essere identificati. Questa fase è effettuata staticamente ed è già stata descritta nella Sezione 4.3.4. Consiste fondamentalmente nell'individuazione nel CFG di ciascuna procedura dell'insieme di basic block che costituisce un costrutto di iterazione, avendo cura di identificare *entry point* e *exit point* del ciclo.

2. Successivamente, la condizione del ciclo viene analizzata dinamicamente allo scopo di individuare eventuali relazioni con i dati in ingresso. È evidente, infatti, che non ha senso procedere con l'analisi di cicli eseguiti un numero costante di volte o, comunque, quando il numero di iterazioni effettuate non dipende dai dati in ingresso.

3. Con il termine *variabile di induzione* si intende una variabile che ad ogni iterazione viene incrementata o decrementata di una quantità che costituisce un invariante del ciclo, oppure una funzione lineare di un'altra variabile di induzione [3].

   Innanzi tutto, per ogni iterazione, le istanze di istruzioni appartenenti al corpo del ciclo e registrate in precedenza nella *execution history* vengono esaminate, estraendo tutti gli assegnamenti con caratteristiche tali da definire una possibile variabile di induzione. Da tale insieme vengono quindi rimosse le istruzioni che definiscono variabili non utilizzate per la valutazione della condizione di terminazione del ciclo.

4. Si dispone a questo punto di un insieme di variabili di induzione effettivamente utilizzate dalla condizione di terminazione del ciclo. Se il ciclo comprende una qualche istruzione di assegnamento che definisce una locazione di memoria il cui indirizzo è funzione di una variabile di induzione, allora si procede a stimare il numero minimo di iterazioni necessarie a sovrascrivere la zona di memoria più vicina contenente dati critici per la sicurezza dell'applicazione.

Stima del numero minimo di iterazioni. Per stimare il numero minimo di iterazioni da utilizzare per sovrascrivere la zona di memoria delicata più vicina, è possibile adottare il procedimento descritto di seguito. Sia v una variabile di induzione utilizzata nella condizione di terminazione del ciclo e impiegata per definire l'indirizzo di destinazione di un'operazione di scrittura in memoria del tipo `ml` $(b+v) := \langle \ldots \rangle$, con b indirizzo base. Sia inoltre λ una costante tale che vale la relazione $v_{i+1} = v_i + \lambda$, per ogni iterazione i. Si supponga $\lambda > 0$ (considerazioni del tutto analoghe possono essere fatte nel caso $\lambda < 0$). Sia infine $[\tau_k, \tau_k + \varepsilon_k]$ la zona di memoria delicata più vicina a b, ovvero quella regione di memoria tale che:

$$[\tau_k, \tau_k + \varepsilon_k] = \min(\{[\tau, \tau + \varepsilon] \in \mathscr{T} \mid \tau \geq b\}).$$

```
1 void copy1(char *src)
2 {
3   int i;
4   char dest[20];
5
6   for(i=0; i<=strlen(src); i++)
7     dest[i] = src[i];
8 }
```

Figura 4.7: Ciclo gestibile con l'approccio descritto nella Sezione 4.4.2.6.

```
1 void copy2(char *src)
2 {
3   int i = 0;
4   char dest[20];
5
6   while(*src)
7     dest[i++] = *src++;
8 }
```

Figura 4.8: Ciclo non gestibile con l'approccio descritto nella Sezione 4.4.2.6.

Dove $\mathscr{T}$ è l'insieme delle locazioni di memoria delicate. Allora il numero minimo n_{min} di iterazioni da effettuare può essere stimato tramite la formula:

$$n_{min} = \left\lceil \frac{\tau_k - b}{\lambda} \right\rceil .$$

Si presti attenzione al fatto che, per semplicità, nel procedimento appena riportato si suppone che la dimensione in bit l della regione di memoria definita sia maggiore o uguale a $8 \cdot \lambda$.

A titolo esemplificativo, si consideri il semplice frammento di codice C di Figura 4.7. La procedura `copy1()` si occupa solamente di copiare il contenuto della stringa `src`, passata come parametro alla funzione, nella variabile locale `dest`. Chiaramente, a causa della mancanza di opportuni controlli preliminari, se la stringa `src` superasse i 20 caratteri, allora il ciclo *for* andrebbe a scrivere in memoria oltre la zona riservata a `dest`, potendo quindi arrivare a modificare l'indirizzo di ritorno della procedura. Se si esaminasse il codice della funzione senza una tecnica di analisi simile a quella descritta nel paragrafo precedente, allora si potrebbe comunque riuscire a condurre l'applicazione in uno stato di esecuzione in cui si manifesta la vulnerabilità appena evidenziata. Per arrivare ad un tale risultato senza una dettagliata analisi del comportamento del ciclo, bisognerebbe però provare ad eseguire la procedura prima con un input che rende immediatamente falsa la condizione alla riga 6, quindi con dei dati in ingresso che portano all'esecuzione del ciclo una sola volta, poi ancora con un

input in grado di indurre due iterazioni sul corpo del ciclo e così via, fino ad arrivare ad eseguire l'istruzione alla riga 7 per circa 28 volte e provocare quindi la sovrascrittura dell'indirizzo di ritorno della funzione `copy1()`. Utilizzando il metodo presentato in precedenza, è invece sufficiente analizzare una particolare esecuzione della procedura di Figura 4.7 che comprende almeno un'iterazione del ciclo *for*. Dinamicamente si potrà individuare la relazione esistente tra la condizione del ciclo e i dati in ingresso: il corpo del ciclo è eseguito un numero di volte pari alla lunghezza della stringa `src`, escluso il carattere finale di terminazione. Si procederà quindi con l'identificazione delle variabili di induzione del ciclo. In questo caso si ha una sola variabile di induzione, `i`, incrementata ad ogni iterazione. Infine, esaminando il corpo del ciclo, verrà individuata un'istruzione di assegnamento che definisce un byte di memoria con indirizzo della forma `&dest + i`, dove `&dest` è l'indirizzo del primo byte del vettore `dest`. Si potrà quindi dedurre che aumentando la lunghezza della stringa `src` si induce il ciclo *for* a compiere un numero maggiore di iterazioni, potendo quindi sovrascrivere una qualunque locazione dello stack con indirizzo maggiore o uguale a `&dest`.

Con il metodo di analisi proposto si ha l'indubbio vantaggio di poter astrarre il comportamento del ciclo dallo specifico insieme di dati in ingresso. Analizzando una sola iterazione è possibile, in alcuni casi, generare un insieme di vincoli che definiscono dei dati in input capaci di compromettere la sicurezza dell'applicazione tramite la sovrascrittura di una zona di memoria delicata con dati sotto il controllo di un ipotetico attaccante.

Sfortunatamente la tecnica appena descritta non è in grado di astrarre il comportamento di tutti i possibili costrutti di iterazione. A titolo esemplificativo, si consideri la procedura `copy2()` riportata in Figura 4.8. La semantica di `copy2()` è del tutto equivalente a quella di `copy1()`: entrambe le procedure copiano la stringa di caratteri passata come parametro di ingresso (`src`) in un vettore locale alla funzione (`dest`). Quando però il ciclo alle righe 6–7 di `copy2()` viene analizzato, verranno inizialmente individuate due variabili di induzione distinte, entrambe incrementate in corrispondenza di ciascuna iterazione. La prima variabile di induzione corrisponde alla locazione di memoria contenente l'intero `i`. La seconda corrisponde invece al

puntatore `src`, anch'esso incrementato ad ogni iterazione (riga 7). L'unica variabile di induzione ad essere utilizzata nella condizione di terminazione del ciclo (riga 6) è `src`, che non viene però poi impiegata per definire l'indirizzo di destinazione di un'operazione di scrittura in memoria. Per questo motivo, l'analizzatore non sarà in grado di determinare a priori il numero di iterazioni necessarie per sovrascrivere, con dati controllabili dall'attaccante, una locazione di memoria contenente dati delicati.

4.4.2.7 Generazione dei vincoli e dei nuovi input

Una volta compiute le analisi sopra descritte, si è in grado di generare un insieme di vincoli tali da consentire l'esplorazione di nuovi cammini di esecuzione. Si supponga, ad esempio, di considerare la funzione di Figura 3.1 con input `buf = 'a\''` e di avere appena terminato l'esecuzione della riga 8 in corrispondenza della prima iterazione del ciclo *for*. Allora i vincoli sull'input connessi all'esecuzione corrente saranno del tipo:

$$PC_0 = \{\texttt{strlen(buf) <= 58 } \wedge \texttt{ buf[0] >= 32 } \wedge \texttt{ buf[0] < 127}\}.$$

Una volta compiute le analisi sulla riga 9 del programma, verrà dedotta una condizione sui dati in ingresso del tipo `buf[0] != '\''`. Al fine di poter esplorare entrambi i cammini di esecuzione connessi al predicato condizionale della riga 9, vengono generati due insiemi di vincoli corrispondenti ai due possibili valori di verità della condizione valutata:

$$\begin{aligned} PC_{true} &= PC_0 \cup \{\texttt{buf[0] != '\''}\}; \\ PC_{false} &= PC_0 \cup \{\texttt{buf[0] == '\''}\}. \end{aligned}$$

I due insiemi di vincoli vengono inviati ad un risolutore (*constraint solver*) che si occupa innanzi tutto di determinare la soddisfacibilità dell'insieme di clausole. Se le condizioni non costituiscono un insieme soddisfacibile, allora si può procedere direttamente a scartare il cammino corrispondente in quanto non percorribile. In caso contrario, il risolutore fornisce le informazioni necessarie per poter definire un nuo-

vo input per l'applicazione in grado di condurre l'esecuzione fino allo stato associato al corrispondente insieme di vincoli. Sarà poi compito del generatore di nuovi input trattare opportunamente le informazioni ottenute dal risolutore, considerando anche un insieme di euristiche progettate per privilegiare quei cammini che conducono verso stati in cui è più probabile che una vulnerabilità si manifesti.

4.5 Euristiche

Vengono ora presentate una serie di euristiche utili ad identificare le zone di memoria da monitorare e i cammini di esecuzione che potrebbero con maggiore probabilità condurre verso uno stato dell'applicazione in cui una vulnerabilità si manifesta.

4.5.1 Corruzione di aree di memoria delicate

Nel corso del monitoraggio dinamico dell'esecuzione dell'applicazione, ogni zona di memoria contenente dati sotto il controllo di un ipotetico attaccante viene contrassegnata come *contaminata* (*tainted*). Si inizia con un insieme minimale di locazioni contaminate che corrispondono alle zone di memoria contenenti i dati di input. Nel caso in cui venga analizzato un intero programma, si potranno considerare come contaminate le aree di memoria che ospitano i parametri da linea di comando e le variabili d'ambiente. Se invece l'analisi è limitata ad una particolare procedura, allora il concetto di "*zona contaminata*" dipende dalla semantica del programma. Limitando l'analisi ad una specifica funzione non si potrà infatti disporre di una visione globale del comportamento dell'applicazione. Per questo motivo sarà necessario indicare allo strumento quali tra i parametri e le variabili globali utilizzate dalla procedura in esame devono essere inizialmente considerate come contaminate. In seguito, nel corso dell'esecuzione, il metodo presentato prevede di tenere traccia di come l'insieme iniziale di locazioni contaminate si estenda arrivando ad includere altre zone di memoria.

In modo analogo l'analizzatore dinamico mantiene un insieme di zone di memoria considerate *delicate*. Si tratta di aree che, se fossero sovrascritte con dati contaminati, potrebbero consentire il dirottamento del flusso di esecuzione verso una zona di memo-

ria contente codice controllato dall'attaccante. Anche in questo caso l'insieme iniziale delle zone delicate è piuttosto ridotto. Ad esempio, nel caso in cui venga monitorato un intero programma tale insieme potrà contenere l'indirizzo di ritorno della procedura principale `main()`. Con il prosieguo dell'esecuzione l'insieme verrà costantemente aggiornato, aggiungendo nuove zone di memoria e rimuovendone di esistenti.

Per fare un esempio, nel momento in cui viene invocata la procedura *r* con un'istruzione assembly del tipo `call <r>`, il chiamante aggiunge in cima allo stack l'indirizzo di ritorno corrispondente all'istruzione sintatticamente successiva a quella di `call`. Tale indirizzo identifica una zona di memoria delicata: se nel corso di *r* l'area individuata dall'indirizzo di ritorno venisse sovrascritta con dati contaminati, allora il flusso di esecuzione verrebbe alterato. La pericolosità di una simile locazione di memoria è però limitata nel tempo al periodo durante il quale la procedura chiamata rimane attiva e, nel momento in cui *r* termina, tale indirizzo cessa di identificare una zona di memoria delicata, dato che l'area potrebbe venire utilizzata per memorizzare altri dati.

Si riassumono qui di seguito alcune categorie di locazioni di memoria che possono essere considerate delicate.

Indirizzo di ritorno. Si è già discusso su come l'indirizzo di ritorno di una procedura costituisca una zona delicata in quanto bersaglio molto comune per attacchi di tipo *buffer overflow*. Come già ricordato, è fondamentale che tale area di memoria venga considerata delicata solo fino a quando la procedura corrispondente non termina.

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* [59] 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*" [75].

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`).

Per poter rilevare sovrascritture di aree delicate con dati contaminati o dirottamenti del flusso di controllo potenzialmente dannosi, durante la fase di monitoraggio dell'esecuzione dell'applicazione vengono individuate tutte quelle istanze di istruzioni intermedie *s* che rispecchiano una delle seguenti condizioni:

- *s* è un'istruzione di salto o una chiamata a funzione indiretta, la cui destinazione è stata precedentemente contrassegnata come contaminata; esempi di istru-

zioni di questo tipo sono `JUMP r32(EAX)` oppure `CALL r32(EAX)`, con `r32(EAX)` considerato, in quel punto del programma, come contaminato.

- *s* è un assegnamento della forma `mn(addr) := expr`, dove *addr* e *expr* sono espressioni che indicano rispettivamente l'indirizzo della zona di memoria da definire e il valore da assegnare. Un'istanza di istruzione intermedia di questo tipo può portare alla compromissione della sicurezza dell'applicazione se:
  - *addr* è stata contrassegnata come contaminata, oppure
  - *addr* identifica una zona di memoria delicata mentre *expr* è contaminata.

Inoltre, al fine di essere in grado di rilevare la corruzione di alcune aree di memoria non riconosciute come delicate (come, ad esempio, nel caso delle variabili locali di una funzione), se durante l'esecuzione l'applicazione solleva un segnale del tipo *segmentation fault*, questo viene considerato come possibile sintomo dell'alterazione del flusso di esecuzione verso una zona di memoria che non contiene codice o che semplicemente non è ancora stata allocata, oppure potrebbe essere causato da un tentativo di scrittura in memoria ad un indirizzo non valido. Si tratta in ogni caso di una situazione problematica, sulla quale conviene comunque indagare procedendo in modo manuale.

4.5.2 Cammini di esecuzione critici

Si possono definire una serie di euristiche per la scelta del percorso di esecuzione da seguire, in modo da percorrere con maggiore probabilità cammini che consentono di raggiungere porzioni di codice potenzialmente pericolose e, al contrario, evitare di visitare regioni che non si ritiene possano portare alla compromissione della sicurezza dell'applicazione. Vengono riportati di seguito alcuni esempi di euristiche di questo tipo.

1. Per ogni istruzione di salto condizionato, cercare di esplorare il ramo che consente di raggiungere una porzione del codice dell'applicazione ancora inesplorata.

2. Evitare l'esplorazione di cammini privi di assegnamenti a zone di memoria con indirizzo non costante e chiamate a funzione.

3. Prediligere l'esplorazione di cammini che comprendono istruzioni che definiscono zone di memoria con indirizzo non costante.

4. Quando non è possibile astrarre il comportamento di un ciclo utilizzando le tecniche di analisi presentate nella Sezione 4.4.2.6, cercare di fare in modo che il ciclo effettui un numero "considerevole" di iterazioni. Per ogni iterazione, se non sono osservate operazioni di scrittura in memoria dirette verso zone delicate, è possibile, ad una prima approssimazione, evitare di esplorare ulteriormente il ciclo. Per fare un esempio, si consideri un ciclo *for* che manipola solo alcune variabili semplici locali alla procedura. In questo caso, indipendentemente dal numero di iterazioni complessive, ogni iterazione definirà al più le locazioni di memoria i cui indirizzi corrispondono alle variabili locali e non si osserveranno assegnamenti con indirizzi di destinazione alterati ad ogni iterazione, come invece potrebbe accadere nel caso di operazioni su variabili di tipo array. Un comportamento di questo tipo non risulta quindi essere rilevante per perseguire gli obiettivi dell'analisi.

   Un esempio di applicabilità di questa euristica è dato dalla procedura `copy2()` di Figura 4.8. Si ricorda che tale procedura non è gestibile con la tecnica di analisi dei cicli discussa nella Sezione 4.4.2.6, in quanto la variabile di induzione utilizzata nella condizione di terminazione del ciclo non è poi impiegata per definire l'indirizzo di destinazione di un'operazione di scrittura in memoria. In base all'euristica appena riportata, quando l'analizzatore fallisce nell'astrarre il comportamento del ciclo *while* di `copy2()`, procede analizzando la condizione di terminazione (riga 6) insieme con le *path condition* ad essa connesse. In questo modo può determinare che il ciclo è eseguito `strlen(src)` volte e, di conseguenza, aumentando la lunghezza della stringa di input `src` è possibile incrementare il numero di iterazioni effettuate. L'analizzatore può quindi provare ad indurre il programma a compiere un numero "elevato" di iterazioni del ciclo *while*. Per stimare in modo conservativo il numero di iterazioni necessarie si potrebbe considerare la distanza tra l'indirizzo base della locazione di memoria definita indirettamente e la zona di memoria delicata più vicina.

Nel caso della procedura `copy2()`, il numero di iterazioni può essere valutato in $|\texttt{\&dest} - \texttt{\&}\mathit{retaddr}|$, dove `&`*retaddr* è l'indirizzo della locazione di memoria contenente il *return address* di `copy2()`.

5. Evitare rami "*senza uscita*", ovvero cammini di esecuzione conclusi da chiamate a sistema che provocano la terminazione del processo e che risultano privi di istruzioni "interessanti" (assegnamenti alla memoria con indirizzo non costante, chiamate a funzione, ...).

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

4.5.3 Gestione delle funzioni di libreria

Le librerie standard (come, ad esempio, la *GNU C Library* [35]) costituiscono componenti particolarmente complessi da analizzare, in quanto comprendono spesso codice piuttosto intricato a causa di ottimizzazioni pesanti introdotte in fase di compilazione o frammenti scritti in codice macchina per ragioni di efficienza. D'altra parte, però, le funzioni di librerie rappresentano una percentuale considerevole dell'insieme delle istruzioni eseguite da una qualsiasi applicazione software. Per fare un esempio, l'esecuzione del solo comando `/bin/ls` in una directory con 5 file e 5 sottodirectory utilizza circa 30 funzioni di libreria differenti, invocate complessivamente oltre 280 volte. Appare quindi evidente che, nonostante la complessità, è comunque indispendabile essere in grado di analizzare anche codice appartenente a procedure di questo tipo.

Sia per ragioni di efficienza e sia per cercare di evitare di trattare codice così complesso, è conveniente provare ad utilizzare tecniche alternative che consentono, una volta riconosciuta la funzione di libreria invocata, di evitare per quanto possibile di analizzare il corpo della procedura, sostituendolo con una sequenza di istruzioni intermedie che ne riassume il comportamento e ne esplicita gli effetti di interesse per l'analisi in corso.

Tecniche analoghe a quella appena menzionata vengono utilizzate in contesti diversi e sono spesso conosciute sotto il nome di *function summarization*. Per fare un esempio, in [87] viene presentata una metodologia di *dynamic taint analysis* basata sull'instrumentazione del codice sorgente e finalizzata all'individuazione di vulnerabilità legate alla mancata validazione dei dati in ingresso. In tale lavoro la *function summarization* è impiegata per trattare le funzioni di libreria che non possono essere instrumentate a causa della non disponibilità del codice sorgente o per l'impossibilità di ricompilare l'intero insieme delle librerie da cui dipende l'applicazione analizzata. Ogni chiamata ad una procedura di libreria è sostituita quindi da una specifica *wrapper function* che esplicita gli effetti che eventuali parametri di input *tainted* possono avere sui parametri di output della funzione.

Si osservi, però, che, mentre nel caso del codice sorgente è solitamente semplice identificare le funzioni di libreria invocate, a livello del codice binario tale operazione risulta in alcuni casi tutt'altro che banale. Nel caso di un file eseguibile che utilizza tecniche di *dynamic linking* [56] è spesso possibile utilizzare le informazioni di rilocazione per riconoscere con precisione la procedura di libreria invocata da un'istruzione di `call`. Se il codice delle procedure di libreria utilizzate è già incluso all'interno del file eseguibile (*static linking*), o se per qualche ragione non è possibile fare affidamento su simboli ed informazioni di rilocazione, allora la questione risulta sensibilmente più complicata. Anche in questo caso, però, si è spesso in grado di impiegare alcune tecniche [26, 38] che consentono di identificare con discreta accuratezza le funzioni chiamate nel corso dell'esecuzione.

Una volta identificata la specifica funzione di libreria, si può procedere alla sostituzione del corpo della procedura con una sequenza di istruzioni intermedie che ne sintetizzano il comportamento. A questo punto è sufficiente eseguire il corpo della procedura di libreria senza analizzare il comportamento di ogni singola istruzione. Una volta che il controllo ritorna al chiamante, si può provvedere ad aggiungere alla *execution history* l'insieme di istanze di istruzioni intermedie che corrisponde alla *function summarization* associata alla funzione appena invocata, avendo cura di attualizzare le espressioni utilizzate dalle istruzioni in modo da adattarle all'esecuzione corrente.

È naturalmente indispensabile fare attenzione a non introdurre codice che possa al-

```
0x08048324    push %ebp
0x08048325    mov %esp, %ebp
0x08048327    push %ebx
0x08048328    push %ecx
0x08048329    push %edx

    /* stringa di destinazione */
0x0804832a    mov 0x8(%ebp), %eax
0x0804832d    mov %eax, %ecx

    /* stringa sorgente */
0x0804832f    mov 0xc(%ebp), %ebx

              ciclo:
0x08048332    movb (%ebx), %dl
0x08048334    movb %dl, (%ecx)
0x08048336    inc %ebx
0x08048337    inc %ecx
0x08048338    test %dl, %dl
0x0804833a    jne loop

              fine:
0x0804833c    pop %edx
0x0804833d    pop %ecx
0x0804833e    pop %ebx
0x0804833f    pop %ebp
0x08048340    ret
```

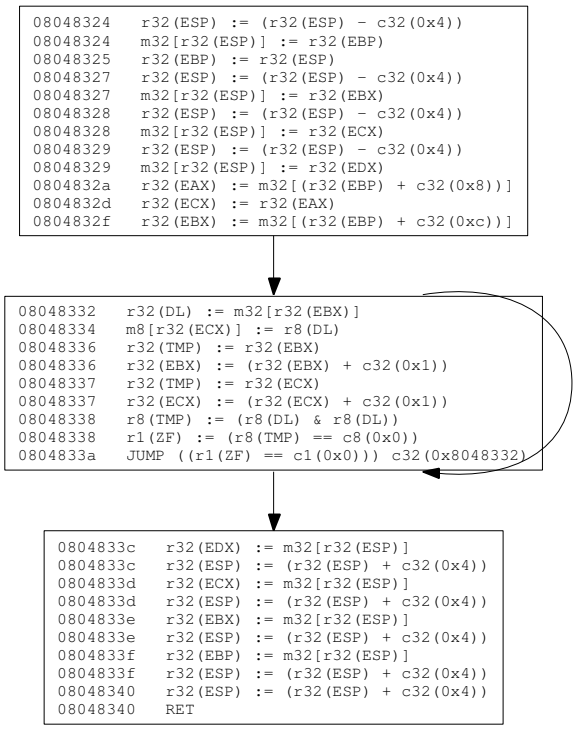

Figura 4.9: Sorgente assembly e CFG relativi ad una possibile *summarization* per la procedura di libreria `strcpy()`.

terare la semantica dell'applicazione e influire sulle istruzioni successive alla chiamata a funzione. Come già sottolineato in precedenza, si presume di trattare con applicazioni che seguono un "modello di compilazione standard" [9], per cui è possibile fare affidamento sul rispetto di alcune convenzioni che facilitano la sostituzione del corpo della procedura con istruzioni equivalenti. Ad esempio, si suppone di poter utilizzare lo stack sia per accedere ai parametri passati dal chiamante, sia per memorizzare temporaneamente i registri alterati dal codice introdotto, in modo da poterli poi ripristinare al termine della funzione e preservarne così il valore iniziale.

In Figura 4.9 sono riportati il codice assembly e il corrispondente CFG relativi ad una possibile *summarization* della funzione di libreria `strcpy()`. In questo caso,

vista la semplicità della procedura in questione, è stato possibile riprodurne esattamente il comportamento. È facile notare come la sequenza di istruzioni utilizzata per la *summarization* non influisca in alcun modo su eventuali istruzioni successive alla chiamata a funzione, dato che i registri utilizzati sono salvati sullo stack all'inizio della procedura e ripristinati immediatamente prima di restituire il controllo al chiamante. Si osservi però che, nonostante la semplicità dell'esempio riportato, la sostituzione del corpo di `strcpy()` con la *function summary* corrispondente può risultare decisamente conveniente: se, ad esempio, l'applicazione analizzata fa uso del linking dinamico, utilizzando la *summarization* si evita di dover seguire inutilmente l'intero processo di *lazy binding*.

L'euristica appena presentata è particolarmente utile quando impiegata come tecnica di astrazione, in quanto consente di concentrare l'analisi solamente sugli aspetti di una particolare funzione che risultano di interesse per l'analisi in corso. Si consideri a riguardo la chiamata di libreria `printf()`. Si è già discusso nella Sezione 4.5.1 di come sia opportuno monitorare gli argomenti di una simile procedura al fine di individuare vulnerabilità di tipo *format bug*. Nel complesso, però, `printf()` è una funzione piuttosto complicata ed articolata ma, soprattutto, una volta verificata la provenienza dei dati in ingresso il corpo della procedura è di poco interesse per il prosieguo dell'analisi. Per questo motivo se il chiamante, come solitamente accade, non utilizza il valore di ritorno della funzione, allora è sufficiente associare alla procedura `printf()` una *function summary* costituita da un insieme vuoto di istruzioni, riducendo sensibilmente l'*overhead* legato all'analisi.

4.6 Risolutore di vincoli

Il problema che rimane da affrontare riguarda come trasformare un insieme di *path condition* in un nuovo input per l'applicazione che soddisfi i vincoli imposti. A questo scopo si ricorre all'utilizzo di un *risolutore di vincoli* (o *constraint solver*) che, dato un insieme di condizioni sull'input *PC*, è in grado di compiere due operazioni fondamentali:

1. determinare se *PC* è soddisfacibile. Così facendo è possibile stabilire se un

particolare cammino di esecuzione non deve essere considerato perché non percorribile.

2. Se *PC* è soddisfacibile, generare un nuovo insieme di dati che soddisfa le condizioni imposte da *PC*. In questo modo, quando l'applicazione in esame sarà eseguita sull'input appena generato, verrà indotta a percorrere il cammino di esecuzione associato a *PC*.

Formalmente, il risolutore di cui necessita il metodo di analisi finora descritto 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 soddisfacibilità di formule logiche che esprimono vincoli relativi ad un prodotto software o hardware. Si tratta di strumenti frequentemente utilizzati per la verifica formale di programmi informatici. Per poter essere impiegata per risolvere gli insiemi di vincoli generati dalle precedenti fasi di analisi, una procedura di decisione deve necessariamente consentire di definire variabili non tipizzate che possano essere impiegate per rappresentare le locazioni di memoria trattate dal codice dell'applicazione in esame e consentirne la manipolazione tramite operazioni aritmetiche (eventualmente anche operazioni non lineari, come moltiplicazione, divisione e modulo), operazioni *bitwise* e relazionali ($<, \leq, \ldots$).

In passato sono state spesso utilizzate variazioni del *framework* proposto da Nelson e Oppen [64] per combinare tra loro procedure di decisione specializzate nella trattazione di specifiche teorie in modo da ottenere un'unica procedura in grado di gestire la combinazione delle teorie iniziali. Esempi di implementazioni di tale approccio sono dati da CVCL [11] e Yices [24].

Il limite principale del modello proposto da Nelson e Oppen va ricercato essenzialmente nel fatto che ogni qual volta una procedura di decisione specializzata è in grado di dedurre l'uguaglianza tra due espressioni, deve necessariamente esplicitare tale relazione e comunicarla alle altre procedure, introducendo così un *overhead* spesso considerevole. Per questo motivo alcuni *solver*, come Cogen [19], UCLID [53] o STP [15],

preferiscono tralasciare l'approccio Nelson-Oppen e adottare tecniche completamente differenti.

Si consideri ad esempio STP. Si tratta di del *solver* utilizzato all'interno di EXE, uno strumento per la ricerca di vulnerabilità nel codice sorgente discusso in maggiore dettaglio nella Sezione 7.2.3. L'approccio adottato da STP è notevolmente diverso rispetto ai *solver* Nelson-Oppen. STP opera infatti trasformando le variabili in ingresso in vettori di bit e traducendo ciascuna operazione sulle variabili in più operazioni atomiche a livello dei singoli bit. I vincoli sono quindi preprocessati tramite l'applicazione di identità logico-matematiche per poi essere trasformati, per mezzo di un procedimento solitamente noto come "*eager translation*", in una formula logica puramente proposizionale. Una volta ottenuto un insieme di formule proposizionali sui bit logicamente equivalenti alle condizioni in ingresso, STP lo comunica ad un *SAT solver* (STP utilizza il SAT solver MiniSAT [25]) che, trattandosi sostanzialmente di formule booleane, è in grado, in caso di soddisfacibilità, di dedurre una possibile configurazione dei bit tale da soddisfare i vincoli imposti inizialmente.

STP è quindi un esempio di procedura di decisione che presenta tutte le caratteristiche richieste per poter essere impiegata per risolvere gli insiemi di *path condition* prodotti dalle analisi descritte nelle sezioni precedenti. Inoltre gli autori hanno provveduto ad implementare all'interno del *solver* una serie di euristiche utili ad incrementare in modo sostanziale l'efficienza delle operazioni che coinvolgono array, oltre ad alcune semplificazioni in grado di ridurre la complessità dei vincoli in ingresso.

Per tutte queste ragioni, STP appare uno strumento particolarmente adatto ad essere integrato con il modello di "*smart fuzzer*" discusso nel presente capitolo.

Capitolo 5

Algoritmi di analisi

In questo capitolo vengono ripercorse le fasi del modello proposto già discusse nel corso del Capitolo 4, approfondendone alcuni aspetti rilevanti. In particolare verranno riportati e descritti gli algoritmi utilizzati: in alcuni casi (*liveness*, dipendenze di controllo e identificazione dei cicli) si tratta di algoritmi già noti in letteratura e qui adattati per l'analisi di codice binario, mentre altri (CFG statici, analisi dei salti e analisi delle *path condition*) rappresentano un contributo originale del presente lavoro di tesi.

5.1 Costruzione dei CFG statici

Una delle operazioni fondamentali da compiere per poter analizzare un programma consiste nel ricostruire il *control flow graph* associato all'applicazione.

In generale, la generazione di un *control flow graph* a partire da codice macchina o da codice assembly è notevolmente più complessa rispetto a quando la stessa operazione è compiuta sul codice sorgente. In particolare, la problematica principale è legata alla presenza di istruzioni di trasferimento di controllo indirette: si tratta di istruzioni di salto o di chiamata a funzione con indirizzo di destinazione definito tramite un registro del processore o una locazione di memoria. Staticamente spesso non si è in grado di determinare con esattezza l'insieme dei possibili indirizzi di destinazione da cui potrebbe riprendere l'esecuzione dopo una tale istruzione. Il problema sussiste anche nel caso del codice sorgente (ad esempio, se vengono utilizzati puntatori a funzione) ma

a livello del codice binario il fenomeno risulta considerevolmente amplificato. In situazioni di questo tipo spesso l'analizzatore è costretto ad agire in modo conservativo e supporre che un salto o una chiamata indiretta possano raggiungere *tutte* le istruzioni del programma [79, 23]. In alternativa, analisi più dettagliate sono in alcuni casi in grado di ridurre l'insieme delle istruzioni raggiungibili esaminando con più attenzione i possibili valori entro cui possono variare il registro o l'area di memoria che definiscono la destinazione del salto o della chiamata a funzione.

Nel caso del modello proposto, si è però preferito sfruttare le potenzialità offerte da un approccio ibrido e ignorare nel corso dell'analisi statica gli effetti di trasferimenti di controllo indiretti. Si arriverà così a produrre un'approssimazione potenzialmente incompleta del CFG delle procedure del programma in esame. I grafi saranno poi completati e raffinati nel corso della successiva analisi dinamica, quando si potranno risolvere i problemi legati all'indirezione semplicemente osservando il comportamento dell'applicazione nella specifica esecuzione.

In Figura 5.1 è riportato l'algoritmo *BuildCFG* sviluppato allo scopo di ricostruire il CFG associato ad una data procedura. L'idea di base è quella di considerare singolarmente ogni istruzione intermedia della procedura in esame. Nel caso in cui l'istruzione non alteri il flusso di controllo, allora è sufficiente aggiungerla al basic block corrente. In caso contrario, invece, è necessario valutare anche l'effetto del trasferimento di controllo, che può comportare la creazione di nuovi blocchi, lo *splitting* (divisione) di blocchi esistenti oppure, nel caso di chiamate a funzioni, la scoperta di nuove procedure da analizzare in seguito.

Inizialmente l'algoritmo verrà applicato alla procedura contenente l'*entry point* dell'applicazione. Tale operazione porterà alla scoperta degli indirizzi delle funzioni chiamate direttamente dalla procedura principale, sulle quali potrà essere ripetuta l'analisi. Il procedimento continua fino a quanto non esistono più procedure da analizzare. Così facendo si è in grado di generare un insieme di *control flow graph*, ciascuno associato ad una specifica funzione raggiungibile a partire dalla procedura iniziale tramine una catena di chiamate dirette.

Più in dettaglio, l'algoritmo proposto riceve in ingresso l'indirizzo *addr* corrispondente alla prima istruzione della procedura da esaminare, un basic block *b* e un CFG

Input: un indirizzo di programma $addr$, un basic block b e un *control flow graph* G

if $b == \perp$ **then**
  $b =$ *nuovo basic block*
if $G == \perp$ **then**
  $G =$ *nuovo basic block*

foreach indirizzo a, partendo da $addr$ **do**
  $ii = decodeInstruction(a)$
  foreach $i \in ii$ **do**
    if G non ha ancora un *entry point* **then**
      aggiungi b come *entry point* di G
    $addToBB(i,b)$
    if $isCall(i) \wedge isDirect(i)$ **then**
      $updateCallGraph(addr, getTarget(i))$
      $newb =$ *nuovo basic block*
      $addToCFG(newb,G)$
      $addLink(b,newb)$
      $BuildCFG(a + getSize(i), newb, G)$
      return
    else if $isJump(i)$ **then**
      $targets = \emptyset$
      if $isDirect(i)$ **then**
        $targets = targets \cup \{getTarget(i)\}$
      if $isConditional(i)$ **then**
        $targets = targets \cup \{a + getSize(i)\}$
      foreach $t \in targets$ **do**
        if t è già stato visitato **then**
          $newb = SplitBB(G, getBB(t), t)$
          $addLink(b,newb)$
        else
          $newb =$ *nuovo basic block*
          $addToCFG(newb,G)$
          $addLink(b,newb)$
          $BuildCFG(t,newb,G)$
      return
    else if $isRet(i)$ **then**
      aggiungi b come *exit point* di G
      return

Figura 5.1: Algoritmo *BuildCFG*.

G. Quando la procedura *BuildCFG()* è invocata per la prima volta, sia *b* che *G* saranno inizializzati al valore indefinito $\perp$. Incominciando dall'indirizzo *addr* e ripetendo l'operazione per ciascun indirizzo di programma *a*, l'istruzione assembly associata ad *a* viene disassemblata e successivamente tradotta in un insieme di istruzioni intermedie *ii*. Ciascuna istruzione $i \in ii$ deve quindi essere analizzata individualmente. Innanzi tutto si valuta se *i* costituisce il punto di ingresso della procedura, in quanto prima istruzione del grafo. Dopo aver aggiunto *i* al basic block corrente, si procede analizzando i possibili effetti dell'istruzione sul flusso di controllo dell'applicazione. Si distinguono tre casi principali:

1. se *i* è una chiamata diretta a funzione, allora per prima cosa viene aggiornato il grafo delle chiamate associato all'applicazione. Il concetto di grafo delle chiamate è stato introdotto nel corso della Sezione 2.2.3. L'algoritmo si limita ad aggiornare il grafo delle chiamate, senza analizzare ricorsivamente la procedura invocata. In ogni caso, una chiamata a funzione provoca la terminazione del basic block corrente e la creazione di un nuovo blocco che conterrà le istruzioni sintatticamente successive a quella di `Call`. La procedura *BuildCFG()* è quindi invocata ricorsivamente sul ramo di esecuzione successivo alla chiamata. Al ritorno dalla chiamata ricorsiva, *BuildCFG()* termina.

2. Se *i* è un'istruzione di salto, allora viene costruito l'insieme dei possibili indirizzi di destinazione. Nel caso di un salto diretto, tale insieme comprende certamente la destinazione del salto. Inoltre, se *i* è un salto condizionato, allora l'esecuzione potrebbe anche riprendere dall'istruzione successiva a *i*. Si osservi che, nel caso in cui un indirizzo di destinazione sia già stato incluso nel CFG, allora il trasferimento di controllo potrebbe comportare lo *splitting* del basic block contenente tale indirizzo qualora la destinazione del salto non corrisponda alla prima istruzione del blocco. Questa operazione è svolta dalla procedura *SplitBB()* descritta in seguito. Dopo aver aggiornato opportunamente l'insieme degli archi del grafo *G*, nel caso in cui una destinazione non sia ancora stata esplorata viene richiamata ricorsivamente *BuildCFG()* sul nuovo ramo. Così come per le istruzione di

`Call`, anche in questo caso terminata l'esplorazione ricorsiva delle destinazioni del salto *BuildCFG()* restituisce il controllo al chiamante.

3. Infine, se *i* è un'istruzione di `Ret` il basic block corrente è aggiunto tra gli *exit point* del grafo e l'algoritmo si arresta.

Vengono descritte di seguito le funzioni utilizzate dall'algoritmo di Figura 5.1:

- *decodeInstruction(a)*, si occupa di disassemblare il codice del programma in corrispondenza dell'indirizzo a e trasformare l'istruzione assembly ricavata in una o più istruzioni intermedie. Restituisce l'insieme delle istruzioni intermedie ottenute dalla trasformazione.
- *addToBB(i, b)*, aggiunge l'istruzione *i* tra quelle facenti parte del basic block *b*.
- *addLink(b_i, b_j)*, aggiunge il basic block b_j tra i successori immediati di b_i e b_i tra i predecessori immediati di b_j.
- *addToCFG(b, G)*, aggiunge il basic block *b* al *control flow graph G*.
- *updateCallGraph(a_i, a_j)*, aggiorna il grafo delle chiamate associato all'applicazione in esame, aggiungendo l'arco (r_i, r_j) che rappresenta l'invocazione della procedura r_j con indirizzo a_j da parte della procedura r_i con indirizzo di partenza a_i.
- *isJump(i), isCall(i), isRet(i)*, restituiscono il valore di verità *vero* se l'istruzione intermedia *i* è rispettivamente un'istruzione di `Jump`, `Call` o `Ret`.
- *getTarget(i)*, restituisce l'indirizzo di destinazione di un'istruzione di trasferimento di controllo (`Jump` o `Call`).
- *isDirect(i)*, restituisce il valore di verità "*vero*" se l'istruzione *i* è un'istruzione di salto o una chiamata a funzione diretta. Ciò implica che *getTarget(i)* sia definito e sia un'espressione di tipo `Constant`.
- *isConditional(i)*, restituisce "*vero*" se l'istruzione di salto *i* è di tipo condizionato.

Input: un *control flow graph* G, un basic block b e un indirizzo di programma $addr$
Output: un nuovo basic block risultante dallo *splitting*, oppure b se non è necessario alcuno *split*

if $addr == getLowAddr(b)$ **then**
  return b
else
  $newb = nuovo\ basic\ block$
  foreach $i \in b, getAddress(i) \geq addr$ **do**
    $remove(i,b)$
    $add(i,newb)$
  $addLink(b,newb)$
  $addToCfg(newb,G)$
  return $newb$

Figura 5.2: Algoritmo *SplitBB*.

- *getSize(i)*, restituisce la dimensione in byte dell'istruzione i.
- *getBB(i)*, restituisce il basic block a cui appartiene l'istruzione i.

Un'ultima funzione utilizzata da *BuildCFG()* è *SplitBB(G, b, addr)*, che si occupa di dividere il basic block b di G nei due basic block b_i e b_j, tali che ogni istruzione di b_i ha indirizzo $a < addr$ mentre ogni istruzione di b_j ha indirizzo $a \geq addr$. La procedura si occupa anche di aggiornare di conseguenza l'insieme dei basic block associati al grafo G. In seguito all'esecuzione di *SplitBB()* si avrà che:

$$\begin{aligned} {\Gamma_G}^{-1}(b_i) &= {\Gamma_G}^{-1}(b); \\ \Gamma_G(b_j) &= \Gamma_G(b); \\ {\Gamma_G}^{-1}(b_j) &= \{b_i\}; \\ \Gamma_G(b_i) &= \{b_j\}. \end{aligned}$$

Si osservi che le equazioni appena riportate indicano b_i come unico predecessore immediato di b_j. In realtà ${\Gamma_G}^{-1}(b_j)$ include anche il basic block che comprende l'istruzione di salto che ha causato lo *splitting*. Tale predecessore viene però aggiunto dalla stessa *BuildCFG()* in seguito all'invocazione di *SplitBB()*. Il comportamento di

SplitBB() è riassunto dall'algoritmo di Figura 5.2. *SplitBB()* utilizza a sua volta alcune semplici procedure:

- *getAddress(i)*, restituisce l'indirizzo di programma associato all'istruzione *i*.
- *getLowAddr(b)*, restituisce il minimo tra gli indirizzi delle istruzioni facenti parte del basic block *b*.
- *add(i, b), remove(i, b)*, si occupano rispettivamente di aggiungere e rimuovere l'istruzione *i* dal basic block *b*.

Prima di concludere la presente sezione, si osservi che l'algoritmo proposto per la costruzione del *control flow graph* associato ad una data procedura utilizza una tecnica di *disassembly* di tipo *recursive traversal*. L'esplorazione del codice della procedure in esame avviene infatti seguendo ricorsivamente il flusso di esecuzione. Come già evidenziato nella Sezione 2.3, tale tecnica consente di evitare di confondere tra loro dati e istruzioni del programma.

5.2 Analisi di *liveness*

Nel corso della Sezione 4.3.4 si è discusso circa come un'analisi della *liveness* delle istruzioni di assegnamento ai registri *flag* del processore possa consentire la rimozione di numerose istruzioni introdotte in seguito all'esplicitazione dei *side effect* del codice assembly che però non risultano influire sulla semantica del programma. Viene ora presentato il metodo riportato in [3] per determinare gli insiemi delle variabili *live* all'ingresso e all'uscita di ciascun basic block di un CFG. Una volta note tali informazioni, è possibile procedere alla rimozione di tutte quelle istruzioni intermedie che definiscono variabili che risultano essere *dead* all'uscita di un blocco.

Formalmente, sia $G = (B, E)$ un *control flow graph* e si consideri un basic block $b \in B$. Allora si indica con *in(b)* l'insieme delle variabili *live* in corrispondenza del punto che precede immediatamente la prima istruzione di *b*, mentre *out(b)* è l'insieme delle variabili *live* al punto immediatamente successivo all'ultima istruzione di *b*. Sia inoltre *def(b)* l'insieme delle variabili definite in modo non ambiguo in *b* prima di un

Input: un *control flow graph* $G = (B,E)$ e gli insiemi $def(b)$ e $use(b)$, $\forall b \in B$

foreach $b \in B$ **do**
  $in(b) = \emptyset$
$changed = true$
while $changed == true$ **do**
  $changed = false$
  foreach $b \in B$ **do**
    $out(b) = \bigcup_{s \in \Gamma_G(b)} in(s)$
    $oldin = in(b)$
    $in(b) = use(b) \cup (out(b) - def(b))$
    if $oldin \neq in(b)$ **then**
      $changed = true$

return *out*

Figura 5.3: Algoritmo per determinare le variabili *live* all'uscita di ogni basic block di un *control flow graph*.

qualsiasi uso e sia *use(b)* l'insieme delle variabili utilizzate in *b* prima di una qualunque definizione. Allora le seguenti equazioni esprimono la relazione esistente tra gli insiemi *def, use* e le incognite *in, out*:

$$\begin{aligned} in(b) &= use(b) \cup (out(b) - def(b)); \\ out(b) &= \bigcup_{s \in \Gamma_G(b)} in(s). \end{aligned}$$

La prima equazione stabilisce che una variabile è *live* all'ingresso di *b* se è utilizzata in *b* prima di essere definita, oppure se è live all'uscita di *b* senza essere ridefinita da un'istruzione del blocco. La seconda equazione, invece, asserisce che una variabile è *live* all'uscita di un basic block se e solo se risulta essere *live* all'ingresso di almeno uno dei successori immediati del blocco stesso.

In Figura 5.3 è riportato un algoritmo che utilizza le equazioni appena presentate per calcolare l'insieme *out(b)* per ogni basic block *b* del CFG. Il procedimento adottato dall'algoritmo è piuttosto semplice: per ciascun basic block *b*, *in(b)* inizialmente coincide con l'insieme vuoto; vengono quindi applicate iterativamente le equazioni che definiscono *in* e *out* fino a quando gli insiemi *in* (e quindi anche gli insiemi *out*) non risultano convergere. La terminazione dell'algoritmo è garantita dal fatto che

Input: un *control flow graph G*

$rg = reverseGraph(G)$
$rdf = dominanceFrontier(rg)$

foreach $b \in getNodes(rg)$ **do**
  $cd(b) = \emptyset$

foreach $b_i \in getNodes(rg)$ **do**
  foreach $b_j \in rdf(b_i)$ **do**
    $cd(b_j) = cd(b_j) \cup \{b_i\}$

return *cd*

Figura 5.4: Algoritmo per il calcolo delle dipendenze di controllo di un CFG.

la cardinalità di ciascun insieme *in(b)* può solo aumentare e, dato che l'insieme delle variabili del programma è finito, prima o poi la condizione del ciclo *while* dovrà necessariamente diventare falsa.

Intuitivamente, il numero di iterazioni compiute dal ciclo *while* è superiormente limitato dalla lunghezza massima di un cammino privo di cicli, $|B| - 1$. Si può quindi dimostrare che, nel caso peggiore, la complessità dell'algoritmo è $O(|B|^2)$, anche se è stato verificato sperimentalmente che il numero di iterazioni compiute dal ciclo *while* è di fatto molto inferiore a tale stima.

5.3 Dipendenze di controllo

Nel corso della Sezione 2.2.3.3 si è discusso su come il problema di determinare le dipendenze di controllo tra i basic block di un *control flow graph G* si possa sostanzialmente ridurre al calcolo della frontiera di dominanza di G^{-1}. Tale tecnica è implementata nell'algoritmo proposto in [21] e riportato in Figura 5.4.

L'algoritmo riceve in ingresso un CFG e procede calcolandone il grafo inverso *rg* e la frontiera di dominanza del grafo inverso *rdf*. Si continua generando in modo incrementale le dipendenze di controllo *cd(b)* per ciascun nodo *b* del grafo in input. Di seguito vengono descritte le procedure utilizzate nel corso algoritmo:

- *getNodes(G)*, dato un grafo orientato $G = (B, E)$, restituisce l'insieme B dei nodi di G.

```
Input: un grafo orientato G e un back edge (n,d)
procedure insert(m)
begin
    if m ∉ loop then
        loop = loop ∪ {m}
        push(stack,m)
end

stack = ∅
loop = {d}
insert(n)
while stack ≠ ∅ do
    m = pop(stack)
    foreach p ∈ pred(m) do
        insert(p)
return loop
```

Figura 5.5: Algoritmo per l'identificazione di un ciclo naturale.

- *reverseGraph(G)*, restituisce l'inverso del grafo orientato G.
- *dominanceFrontier(G)*, calcola la frontiera di dominanza del grafo orientato $G = (B,E)$ passato come argomento. Viene restituita una mappa *df* che associa a ciascun nodo in B un insieme (eventualmente vuoto) contenente tutti e soli i nodi di B che ne costituiscono la frontiera di dominanza. Si rimanda a [21] per un algoritmo per il calcolo della frontiera di dominanza che, nel caso medio, ha complessità lineare rispetto al numero dei nodi del grafo in esame.

L'algoritmo termina restituendo la mappa *cd* che associa ciascun nodo *b* del grafo in ingresso con un insieme di nodi (eventualmente vuoto) da cui *b* risulta essere *control dependent*.

5.4 Identificazione dei cicli

In Figura 5.5 è riportato l'algoritmo utilizzato nell'infrastruttura di analisi descritta nel presente lavoro di tesi al fine di individuare i cicli presenti all'interno del *control flow*

graph associato ad una procedura.

L'algoritmo qui presentato è descritto in [3] e consente, dato un grafo orientato $G = (B, E)$ ed un suo *back edge* $(n, d) \in E$, di determinare l'insieme $loop \subseteq B$ dei nodi di G che costituiscono il ciclo naturale associato a (n, d).

A partire dal nodo n, vengono considerati tutti i nodi $m \neq d$ già appartenenti al ciclo e si aggiungono a *loop* i predecessori p di m. Si osservi che, dato che il nodo d è inserito inizialmente tra i nodi del ciclo, i suoi predecessori non vengono mai esaminati e quindi sono considerati solamente quei nodi a partire dai quali è possibile raggiungere n senza passare da d, in accordo con la definizione di ciclo naturale presentata nella Sezione 2.2.1.2. Al termine della computazione, *loop* comprende tutti e soli i nodi che costituiscono il ciclo naturale cercato.

Le uniche procedure utilizzate dall'algoritmo sono quelle relative alla gestione della struttura a pila *stack*: *push* si occupa di aggiungere un elemento in testa alla pila, mentre *pop* restituisce l'elemento in testa alla struttura a pila passata come argomento.

Come già ricordato, l'algoritmo appena presentato consente di determinare i cicli naturali associati ad un dato back edge. Se il grafo G è riducibile, allora un approccio di questo tipo permette di determinare tutti e soli i cicli di G. In alcuni casi, però, una condizione di questo tipo può sembrare eccessivamente restrittiva in quanto potrebbe essere necessario dover trattare cicli irriducibili (cicli con più nodi di ingresso) ottenuti, ad esempio, dall'analisi di codice binario particolarmente ottimizzato. In simili situazioni si può ricorrere ad algoritmi differenti [77, 40, 72] che consentono l'identificazione dei cicli di un grafo orientato anche in presenza di situazioni di irriducibilità.

5.5 Analisi delle condizioni di salto

In Figura 5.6 è riportato l'algoritmo sviluppato per analizzare le istanze di istruzioni di salto condizionato. Come già trattato nella Sezione 4.4.2.3, il problema che tale algoritmo si propone di risolvere è la traduzione della condizione di un'istanza di istruzione di `Jump` j, ottenuta direttamente dalla trasformazione in forma intermedia di un'istruzione assembly, in un'espressione di più alto livello. In particolare, la condizione di j utilizzerà esclusivamente i *flag* del processore. L'idea alla base dell'algoritmo consiste

Input: j, un'istanza di istruzione di salto condizionato

foreach $f \in use(getRhs(j))$ **do**
    $replace(getRhs(j), f, getRhs(drd(f, j)))$
    foreach $v \in use(getRhs(j))$ **do**
        if $isTemporary(v)$ **then**
            $replace(getRhs(j), v, getRhs(drd(v, j)))$

$replace(getRhs(j), getRhs(j), applyTemplate(getRhs(j)))$

Figura 5.6: Algoritmo *RecoverPredicate*.

nel valutare le *reaching definition* di ogni *flag* utilizzato da j e impiegare le istruzioni di assegnamento così ricavate per applicare un modello che sintetizza la semantica dell'istruzione di salto.

Più in dettaglio, l'algoritmo di Figura 5.6 utilizza le seguenti funzioni:

- $getRhs(i)$, restituisce il lato destro dell'istruzione i. Nel caso di un'istruzione di salto condizionato, *getRhs(i)* restituisce la condizione di salto.
- $replace(g, e, e')$, sostituisce nell'espressione g tutte le occorrenze della sottoespressione e con e'.
- $applyTemplate(e)$, restituisce il risultato della semplificazione dell'espressione e tramite l'applicazione di un modello che riassume la semantica dell'istruzione di salto condizionato a cui appartiene.
- $isTemporary(v)$, restituisce il valore di verità "*vero*" se la variabile v rappresenta il registro temporaneo `TMP`.

A scopo esemplificativo, si consideri l'istruzione 5_1 di Figura 4.4. La traduzione in forma intermedia ha già esplicitato la condizione valutata: `r1(ZF) == c1(0x0)`. Supponiamo ora che l'algoritmo per l'analisi delle condizioni di salto venga applicato ad un'istanza di tale istruzione. Innanzi tutto viene eseguito il ciclo *foreach* più esterno, allo scopo di determinare gli assegnamenti che definiscono i *flag* utilizzati dall'espressione condizionale. Nel nostro caso l'unico *flag* utilizzato è quello di *zero* (`ZF`), per cui

Input: j, un'istanza di istruzione di assegnamento o di salto condizionato; $\mathscr{I}$, l'insieme delle variabili connesse ai dati di input

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
    $replace(getRhs(j), v, getRhs(PropagateDRD(\overline{drd}(v, j))))$

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

return j

Figura 5.7: Algoritmo *PropagateDRD*.

il ciclo compirà una sola iterazione. La *reaching definition* dinamica relativa alla variabile `r1(ZF)` corrisponde ad un'istanza dell'istruzione 4_5, per cui la prima chiamata alla procedura *replace()* trasformerà la condizione di salto nell'espressione

```
(r8(TMP) == c8(0x0)) == c1(0x0).
```

Il *foreach* più interno si occupa di sostituire, nell'espressione finora costruita, ogni occorrenza del registro fittizio `TMP` con la rispettiva *reaching definition* dinamica. Nel nostro caso l'unica occorrenza di `TMP` verrà sostituita con il lato destro dell'assegnamento 4_1, ottenendo l'espressione

```
((r8(AL) - c8(0x27)) == c8(0x0)) == c1(0x0).
```

A questo punto la procedura *applyTemplate()* sarà in grado di applicare il modello che riassume la semantica di un salto condizionato `jnz`, restituendo l'espressione semplificata `r8(AL) != c8(0x27)` che prenderà il posto della condizione di salto originale.

5.6 Analisi delle *path condition*

Si ricorda che l'analisi delle *path condition* è finalizzata alla trasformazione della condizione di un'istanza di istruzione di `Jump`, già esaminata dall'analizzatore di salto, in un'espressione sui dati in ingresso all'applicazione. L'algoritmo che si occupa di tale operazione è riportato in Figura 5.7. L'approccio adottato consiste nel propagare all'interno della condizione di salto le definizioni delle variabili utilizzate, arrestando il processo di propagazione nel momento in cui viene incontrata una variabile contenente dati provenienti dall'input. Le funzioni utilizzate dall'algoritmo sono le seguenti:

- $getRhs(i)$, restituisce il lato destro dell'istruzione i. Nel caso di un'istruzione di salto condizionato, $getRhs(i)$ restituisce la condizione di salto.
- $replace(g,e,e')$, sostituisce nell'espressione g tutte le occorrenze della sottoespressione e con e'.
- $\overline{drd}(v,s_k)$, restituisce la *reaching definition* dinamica della variabile v valutata in corrispondenza dell'istanza di istruzione s_k, gestendo anche eventuali problematiche derivanti da definizioni parzialmente sovrapposte. Infatti, nel caso peggiore, alla definizione di una locazione di memoria di n byte con indirizzo base m potrebbero contribuire fino a n differenti istanze di istruzioni intermedie di assegnamento, con ciascuna che definisce un singolo byte all'interno dell'intervallo $[m,m+(n-1)]$. La procedura $\overline{drd}$ si occupa di tale questione, per cui $r = \overline{drd}(v,s_k)$ potrebbe non corrispondere ad una reale istruzione del programma in quanto, nel caso di sovrapposizioni, la funzione restituisce un'istruzione di assegnamento virtuale che combina gli effetti di tutte le istanze di istruzioni di assegnamento reali che concorrono nella definizione della variabile v.

Per fare un esempio, si consideri l'espressione `r8(AL) != c8(0x27)` con cui è stata tradotta dall'analizzatore di salto la condizione dell'istruzione 5_1 e si supponga l'analisi venga condotta in corrispondenza della prima iterazione del ciclo *for* di Figura 3.1. Sia j l'istanza dell'istruzione 5_1 relativa a tale iterazione. Si supponga inoltre che l'insieme $\mathscr{I}$ delle variabili connesse ai dati di input contenga la sola area

di memoria corrispondente al vettore `buf`. Dato che l'unica variabile utilizzata dall'espressione è rappresentata dal registro `AL`, il ciclo *foreach* dell'algoritmo compirà un'unica iterazione. Il registro `AL` non compare nell'insieme $\mathscr{I}$ dei dati in ingresso, per cui la condizione del primo *if* risulta falsa. Viene quindi valutata la *reaching definition* dinamica $\overline{drd}(\texttt{r8(AL)}, j)$. Tale definizione esiste e corrisponde ad un'istanza dell'istruzione 3_1, per cui anche la condizione del secondo *if* appare falsa e verrà quindi eseguito il corpo dell'*else* che provoca l'alterazione della condizione di partenza sostituendo l'occorrenza del registro `AL` con il risultato dell'esecuzione ricorsiva dell'algoritmo sull'istanza dell'istruzione 3_1.

La propagazione ricorsiva delle *reaching definition* continua fino a trasformare l'espressione di partenza in `m32(c32(0xBFFFFAD1)) != c8(0x27)`. La locazione di memoria a 32 bit con indirizzo *0xBFFFFAD1* corrisponde infatti al primo byte del vettore `buf` e, di conseguenza, la condizione valutata dal primo *if* risulta vera e provoca l'arresto della ricorsione.

Si osservi che, anche nel caso in cui non fosse esistita alcuna relazione tra l'espressione condizionale e i dati di input, il processo ricorsivo si sarebbe comunque arrestato a causa della condizione valutata dal secondo *if*. Ciò garantisce quindi la terminazione dell'algoritmo di Figura 5.7.

A livello implementativo, l'algoritmo può essere reso sensibilmente più efficiente tramite il calcolo preventivo dello *slice* dinamico relativo alle variabili utilizzate dalla condizione di salto di partenza, in modo da poter poi ricercare le *reaching definition* richieste dall'algoritmo tra le sole istruzioni che fanno parte dello *slice*.

Capitolo 6

Implementazione e limitazioni

Nel presente capitolo sono discussi alcuni dettagli relativi all'implementazione del modello descritto in precedenza. Vengono in seguito evidenziate le caratteristiche principali del prototipo sviluppato insieme con alcune limitazioni e idee per futuri sviluppi del lavoro.

6.1 Prototipo

Il modello di *smart fuzzer* ampiamente discusso nel corso del Capitolo 4 è stato parzialmente implementato in un prototipo sperimentale.

In particolare, il prototipo realizzato include tutte le funzionalità proprie dell'analizzatore statico e dinamico, comprendendo quindi i moduli illustrati in Figura 4.2 e 4.6. Attualmente lo strumento è in grado di analizzare file eseguibili in formato ELF [81] contenenti codice per piattaforma Intel IA-32 e destinati ad essere eseguiti su sistema operativo Linux. In ogni caso, grazie all'approccio modulare seguito durante le fasi di progettazione e realizzazione del prototipo, risulta piuttosto semplice estendere lo strumento in modo da supportare formati, architetture e sistemi operativi differenti. Una volta esaminato il file contenente il programma da analizzare, il prototipo può procedere con le attività di *disassembly* e traduzione in forma intermedia, per poi passare alle fasi di analisi statica e dinamica discusse in precedenza, arrivando così a dedurre le *path condition* associate ad un particolare cammino di esecuzione. Anche

se sono già stati condotti alcuni test preliminari, rimane comunque da implementare un modulo per interfacciare lo strumento con un risolutore di vincoli simile a quelli discussi nella Sezione 4.6.

Lo strumento realizzato è quindi in grado di compiere tutte le analisi necessarie e generare un insieme di *path condition*, ma la risoluzione dei vincoli deve ancora essere effettuata manualmente. Nel complesso, il prototipo consiste di oltre 20000 righe di codice Python [70], a cui si aggiungono circa 100 di codice Yapps [7] per il *parsing* delle espressioni contenute nelle istruzioni *assembly* e 1000 righe di codice C per l'interfacciamento con la *system call* Linux `ptrace()` (utilizzata per il monitoraggio dinamico del processo analizzato) e con la libreria GNU `libopcodes` [34] (utilizzata per trasformare il codice macchina in codice *assembly*).

6.1.1 Monitoraggio del processo

Per poter compiere le analisi dinamiche descritte finora è ovviamente fondamentale disporre di un'infrastruttura che consenta di monitorare costantemente lo stato del processo analizzato. Un simile *framework* deve poter essere utilizzato per esaminare i registri del processore e lo spazio di indirizzamento virtuale assegnato al processo, permettendo inoltre di controllare il procedere dell'esecuzione.

La fase di monitoraggio dinamico dell'esecuzione dell'applicazione è attualmente effettuata utilizzano un'interfaccia di debugging basata sulla *system call* `ptrace()`, tramite la quale è possibile recuperare informazioni dettagliate circa lo stato del processo analizzato.

Sono state considerate due possibili soluzioni per implementare l'interfaccia di debugging. La prima prevede la realizzazione di *breakpoint software* [33] tramite la *syscall* `ptrace()`, in modo da poter arrestare l'esecuzione in corrispondenza dell'indirizzo di programma desiderato. La tecnica dei *breakpoint software* consiste nel sostituire nella memoria del processo analizzato il primo byte dell'istruzione in corrispondenza della quale si desidera arrestare l'esecuzione con l'*opcode* associato all'istruzione macchina Intel `int 3`, avendo cura di memorizzare il dato sovrascritto in una variabile temporanea. Quando il processore eseguirà l'istruzione `int 3` passerà

il controllo al gestore delle *breakpoint exception* [47] il quale, a sua volta, notificherà un "*debugger*" che, in questo caso, corrisponde allo strumento di analisi. Nonstante un simile procedimento sia in grado di produrre gli effetti voluti, presenta comunque alcuni inconvenienti. Per prima cosa, l'impostazione di un breakpoint richiede almeno sei invocazioni della chiamata di sistema `ptrace()`, riassumibili nei seguenti passi:

1. salvataggio del contenuto della locazione di memoria contenente il primo byte dell'istruzione in corrispondenza della quale si desidera arrestare l'esecuzione.

2. Sovrascrittura della locazione precedentemente salvata con l'*opcode* dell'istruzione macchina `int 3`.

3. Richiesta al sistema operativo di riprendere l'esecuzione del processo in esame.

4. Lettura del registro `EIP`, che conterrà l'indirizzo dell'istruzione successiva all'interruzione software.

5. Decremento del registro `EIP`, in modo da riportare l'esecuzione all'indirizzo corrispondente all'istruzione dove il processo deve essere arrestato.

6. Ripristino dello stato originale della locazione sovrascritta in precedenza.

È quindi evidente che ciascuno dei passi appena descritti richiede per lo meno due *context switch* tra *user space* e *kernel space* per l'esecuzione della chiamata a sistema, andando così ad incidere in modo significativo sull'efficienza dello strumento di analisi. In secondo luogo, si osservi che un breakpoint software modifica lo spazio di indirizzamento del processo analizzato e, di conseguenza, potrebbe non risultare compatibile con applicazioni che implementano alcune semplici tecniche di *anti-debugging* come, ad esempio, un *checksum* del proprio codice.

I motivi appena citati hanno spinto verso l'implementazione di una soluzione di debugging differente, che sfrutta le funzionalità hardware offerte dagli attuali processori Intel. Più in dettaglio, per arrestare inizialmente l'esecuzione dell'applicazione in corrispondenza dell'*entry point* o all'indirizzo da cui l'analisi deve cominciare vengono utilizzati i *breakpoint hardware*. Si tratta di una tecnica che consente di delegare

interamente la gestione dei breakpoint al processore. Il prototipo non si deve preoccupare di compiere tutte le operazioni necessarie ad impostare un breakpoint software, riprendere l'esecuzione e successivamente ripristinare la locazione di memoria alterata in precedenza. Lo strumento di analisi deve solo avere cura di impostare correttamente alcuni registri della CPU in modo da definire quando l'esecuzione dell'applicazione in esame deve essere sospesa.

Una volta che l'esecuzione è stata arrestata in corrispondenza dell'indirizzo da cui iniziare l'analisi, è necessario procedere ad eseguire ed esaminare un'istruzione alla volta. Per fare ciò non risulta certamente conveniente impostare manualmente un breakpoint sull'istruzione successiva, riprendere l'esecuzione, compiere le analisi necessarie e ripetere iterativamente il procedimento. È invece più vantaggioso chiedere alla stessa CPU di sospendere l'esecuzione e notificare il debugger dopo ogni istruzione eseguita. Per fare ciò è sufficiente portare il processore in modalità *single-step* [48] tramite l'impostazione del flag `TF` (*trap flag*) del registro `EFLAGS`. Così facendo, dopo l'esecuzione di ogni istruzione il processore solleverà una *single-step exception* che verrà opportunamente notificata al debugger. Non è però possibile impostare manualmente il flag `TF`, dato che al momento della riprese dell'esecuzione del processo (tramite una chiamata a `ptrace(PTRACE_CONT)`) tale bit viene impostato nuovamente a zero dal kernel. Per questo motivo è comunque necessario ricorrere ad un'invocazione di `ptrace()` specificando una richiesta del tipo `PTRACE_SINGLESTEP`, in corrispondenza della quale il sistema operativo provvede ad impostare il *flag* `TF`, riprendere l'esecuzione del processo monitorato e restituire il controllo al debugger quando l'applicazione viene nuovamente interrotta.

Si osservi, però, che il prototipo realizzato deve comunque ricorrere alla chiamata a sistema `ptrace()` ogni qual volta debba accedere ai registri del processo esaminato o al suo spazio di indirizzamento virtuale. Come si vedrà nella sezione seguente, ciò introduce un notevole *overhead* che appesantisce in modo considerevole la fase di monitoraggio.

`fase statica`	
disassembly e CFG statici	6.376 s
rimozione assegnamenti flag	7.401 s
dipendenze di controllo	0.147 s
identificazione cicli	0.053 s
	13.977 s

`fase dinamica`	
analisi PSS iniziale	0.190 s
monitoraggio processo	22.090 s
ripetizione fase statica	0.344 s
analisi dei salti	3.396 s
semplificazione espressioni	2.405 s
analisi *path condition*	11.974 s
analisi dei cicli	0.134 s
	40.533 s

`statistiche`	
cicli	45
salti condizionati	812
istruzioni	7979
execution history	11249
semplificazioni	7666

Tabella 6.1: Misurazioni relative all'analisi effettuata su `/bin/ls`.

6.1.2 Prestazioni

Per discutere circa le prestazioni del prototipo realizzato e alcuni problemi ancora presenti nello strumento, sono state condotte alcune misurazioni relative all'analisi di una versione modificata dell'utility Linux `ls`. I risultati ottenuti sono riassunti nella Tabella 6.1. L'esperimento è stato condotto utilizzando una macchina con processore Intel Pentium 4 a 2.8 Ghz, con 512 KB di cache e 512 MB di memoria RAM. I dati riportati corrispondono alla media aritmetica dei valori registrati su 10 esecuzioni del programma.

6.1.2.1 Vulnerabilità

Al fine di verificare l'efficacia del procedimento proposto, il programma `ls` è stato modificato introducendo appositamente una semplice vulnerabilità. In particolare, all'interno del flusso di esecuzione principale è stata aggiunta un'operazione di copia tra vettori di caratteri che non effettua alcun controllo sull'effettiva capienza del vettore destinazione, fissata staticamente a 512 byte. Il vettore sorgente contiene una stringa

corrispondente al percorso di un file da visualizzare, così come è stato specificato dall'utente tramite riga di comando. In questo modo, se un attaccante fosse in grado di eseguire il programma `ls` modificato fornendo un percorso con lunghezza maggiore di 512 byte, riuscirebbe a scrivere in memoria oltre i limiti dell'array destinazione.

L'applicazione è stata eseguita specificando da riga di comando un singolo argomento, corrispondente ad una directory contenente 5 file e 5 sottodirectory. Una volta terminata l'analisi, lo strumento è stato in grado di dedurre che se la lunghezza del nome della directory specificata avesse superato i 516 caratteri, allora si sarebbe potuto sovrascrivere l'indirizzo di ritorno di una procedura. Il risultato è stato confermato dall'analisi manuale. Infatti, i primi 512 byte della stringa di input vengono copiati all'interno del vettore di caratteri, mentre i successivi 4 byte sovrascrivono il *base pointer* della procedura che, nell'implementazione corrente, non è considerato un dato sensibile. I 4 byte che seguono vanno effettivamente a sovrascrivere il *return address* di una procedura.

Si osservi che il prototipo è stato in grado di dedurre una probabile vulnerabilità fin dalla prima esecuzione. Ciò è stato possibile grazie alla tecnica di analisi dei cicli discussa nella Sezione 4.4.2.6.

Come si vede dai dati riportati nella Tabella 6.1, l'intera analisi ha richiesto circa 63 secondi.

6.1.2.2 Analisi statica

I dati riportati mostrano come le prestazioni relative alla fase di analisi statica siano dominate dalle attività di "*disassembly* e costruzione dei CFG" e "rimozione del codice *dead*". La prima voce comprende le operazioni di *disassembly* del codice macchina, generazione della forma intermedia e costruzione dei *control flow graph* a partire dalle informazioni derivabili staticamente. Si tratta, in sostanza, dell'implementazione dell'algoritmo descritto nella Sezione 5.1. La voce successiva include invece sia l'analisi di *liveness* condotta sugli assegnamenti ai *flag* del processore, sia la rimozione del codice *dead*. Rispetto a questi due fattori iniziali, le restanti voci dell'analisi statica appaiono trascurabili.

È importante sottolineare come le prestazioni relative alle analisi condotte statica-

mente non costituiscano elementi particolarmente critici, in quanto tali attività devono essere condotte una sola volta prima della reale esecuzione dell'applicazione. Inoltre, le informazioni dedotte staticamente, proprio perché indipendenti dalla specifica esecuzione, possono essere memorizzate e utilizzate nelle successive analisi della stessa applicazione. In realtà l'analisi dinamica richiama poi nuovamente l'analizzatore statico per esaminare regioni di codice raggiungibili solo tramite trasferimenti indiretti. Tali informazioni, però, non devono essere dedotte per ciascuna esecuzione e possono essere anch'esse salvate e riutilizzate nelle analisi future.

6.1.2.3 Analisi dinamica

La fase di analisi dinamica costituisce il punto più problematico dell'intero procedimento. Si osservi, innanzi tutto, che i dati riportati in tabella sono relativi ad una sola esecuzione dell'applicazione in quanto, a causa della mancata integrazione con il risolutore di vincoli, il prototipo non è ancora in grado di esplorare automaticamente gli stati del programma.

La prima voce, relativa alla generazione dello stato iniziale del processo, insieme con l'analisi dei cicli costituiscono fattori pressoché trascurabili. Il monitoraggio del processo rappresenta invece il fattore dominate dell'intera fase. Questa voce comprende in sostanza tre sottoattività:

1. *esecuzione* – il processo viene eseguito in modalità *single step*, ripetendo per ciascuna istanza di istruzione le analisi dinamiche e avendo cura di invocare l'analizzatore statico sulle nuove regioni di codice.

2. *Valutazione* – ogni istanza di istruzione dev'essere valutata singolarmente come descritto nel corso della Sezione 4.4.2.2 e aggiunta in coda alla *execution history*.

3. *Aggiornamento PSS* – così come già discusso nella Sezione 4.4.1, nel momento in cui un'istanza di istruzione accede ad una zona di memoria non ancora definita, viene memorizzato il contenuto della locazione nel PSS. In questo modo si evita di dover salvare nel PSS l'intero spazio di indirizzamento virtuale del processo.

Ciascuna di queste attività risulta piuttosto dispendiosa in termini di tempo. L'impiego dei *breakpoint hardware* insieme con il *single-stepping* supportato dal processore ha significativamente ridotto l'*overhead* connesso all'esecuzione dell'applicazione, ma è comunque necessario effettuare numerose chiamate a `ptrace()` per valutare gli operandi di tutte le espressioni. Infine, l'aggiornamento del PSS richiede che, per ogni locazione di memoria utilizzata da ciascuna istanza di istruzione, ne venga calcolata la *reaching definition* dinamica. Nonostante l'implementazione dell'algoritmo adottato faccia uso di una *cache* interna per evitare di elaborare più volte la stessa richiesta, l'operazione rimane comunque abbastanza costosa. Se la *reaching definition* non esiste, allora il contenuto della cella di memoria deve essere recuperato con un'ulteriore chiamata a `ptrace()` e poi aggiunto all'interno del PSS.

L'analisi dinamica procede con l'analisi delle condizioni di salto. Anche in questo caso si ricorre alle *dynamic reaching definition* per determinare l'istanza di istruzione che definisce i *flag* coinvolti nell'espressione condizionale associata al salto. A differenza dell'attività di aggiornamento del PSS, in questo caso la ricerca della definizione sarà piuttosto breve e solitamente confinata all'interno dello stesso basic block in cui compare l'istruzione di salto.

La semplificazione delle espressioni consiste nella continua applicazione di trasformazioni logiche, matematiche e sintattiche fino a quando l'espressione non può essere ridotta ulteriormente. In questo caso l'*overhead* è dovuto alla necessità di semplificare ricorsivamente le sottoespressioni di ogni espressione utilizzata dalle istanze di istruzione.

L'analisi delle *path condition* è particolarmente onerosa, seconda solo all'attività di monitoraggio dinamico. Qui l'operazione più gravosa è costituita dall'applicazione ricorsiva dell'algoritmo presentato nella Sezione 5.6. Tra l'altro, ogni invocazione della procedura *PropagateDRD()* richiede anche la computazione delle *reaching definition* relative alle variabili utilizzate dall'istruzione in esame.

6.1.2.4 Possibili miglioramenti

Come ricordato in precedenza, gran parte del prototipo è scritto in Python, un linguaggio ad oggetti interpretato che grazie alla tipizzazione dinamica è particolarmente

adatto alla prototipazione rapida. Il sorgente Python viene inizialmente compilato in una forma di *bytecode* che deve poi essere interpretato in fase di esecuzione. Inoltre, l'analisi di un'applicazione mediamente complessa comporta la creazione di un numero considerevole di oggetti, uno per ciascuna istanza di istruzione e sottoespressione utilizzata. La memoria richiesta da tali istanze dev'essere allocata dinamicamente dall'interprete. Questi fattori vanno ad incidere in modo considerevole sull'efficienza dello strumento, portando a prestazioni certamente inferiori rispetto a quanto si sarebbe potuto ottenere da un linguaggio compilato in modo nativo. D'altra parte, le caratteristiche e la libreria di classi del linguaggio Python hanno agevolato notevolmente lo sviluppo del prototipo, permettendo di tralasciare i dettagli relativi all'implementazione delle varie strutture dati utilizzate. Con un maggiore sforzo implementativo si potrebbe quindi riscrivere alcuni componenti critici in codice C, riducendo così almeno parte dell'*overhead* introdotto dall'interprete Python.

Un altro elemento che limita considerevolmente l'efficienza del prototipo è rappresentato dalla chiamata a sistema `ptrace()`, utilizzata per esaminare lo stato del processo monitorato. Una possibile alternativa consiste nell'instrumentare dinamicamente il codice del programma esaminato in modo che sia la stessa applicazione ad eseguire il codice necessario a generare le informazioni richieste dall'analisi. Una soluzione di questo tipo potrebbe essere realizzata facendo uso di Valgrind [65], un'infrastruttura per l'instrumentazione dinamica del codice binario. In questo modo si ritiene di poter ridurre considerevolmente l'*overhead* legato ai *context switch* tra kernel e spazio utente. Rimane comunque da valutare sperimentalmente in quale misura lo strumento possa beneficiare di un simile miglioramento.

Per concludere, nonostante le misurazioni presentate nella sezione precedente abbiano evidenziato tempi di esecuzione piuttosto lunghi, sembra comunque che il prototipo realizzato possa essere notevolmente migliorato implementando gli accorgimenti appena discussi. In ogni caso, si consideri che l'intero modello di *smart fuzzer* non è pensato per essere utilizzato per monitorare in tempo reale l'esecuzione di un'applicazione, bensì per analizzare *off-line* un programma e individuare possibili vulnerabilità.

6.2 Limitazioni e sviluppi futuri

In questa sezione vengono riportate alcune limitazioni relative alle tecniche di analisi utilizzate nel modello discusso finora, proponendo possibili soluzioni che costituiscono sviluppi futuri del lavoro.

Una prima importante limitazione del metodo descritto è rappresentata dall'impossibilità di analizzare codice capace di modificarsi autonomamente. Alcune applicazioni ricorrono infatti alla generazione dinamica di porzioni del proprio codice, spesso per ragioni di efficienza o per offuscare il proprio comportamento. Per fare un esempio, molti esemplari di virus si diffondono distribuendo una versione cifrata del proprio codice, che viene decifrata dinamicamente quando il programma è eseguito sulla macchina bersaglio. A volte anche lo stesso compilatore `gcc` introduce istruzioni per la generazione automatica di brevi frammenti di codice detti *trampoline*, utilizzati per implementare il meccanismo delle funzioni annidate [32]. Attualmente il modello non è in grado di gestire comportamenti di questo tipo. Si suppone infatti che, una volta disassemblata, un'istruzione non possa essere successivamente modificata. Per poter trattare codice automodificante si potrebbe ad esempio monitorare il comportamento dell'applicazione al fine di individuare operazioni di scrittura all'interno della sezione di codice del processo. Così facendo si potrebbe determinare quali istruzioni sono influenzate dall'operazione di scrittura, potendo quindi calcolare nuovamente solo le informazioni necessarie. Spesso, però, le sezioni del processo che contengono codice eseguibile non sono scrivibili, per cui l'applicazione posiziona il codice generato direttamente sullo stack, rendendo così più difficile ricorrere a soluzioni come quella appena proposta. In ogni caso è facile immaginare che un qualunque approccio che cerchi di trattare applicazioni che alterano dinamicamente il proprio codice vada ad incidere negativamente sulle prestazioni del modello.

Un'ulteriore limitazione del modello proposto risiede nella tecnica utilizzata per identificare staticamente i cicli presenti nel programma. L'algoritmo presentato nella Sezione 5.4 è infatti in grado di trattare esclusivamente cicli riducibili, ovvero costrutti di iterazione con un unico nodo di ingresso che domina ogni altro nodo del ciclo. Sfortunatamente, però, a livello del codice binario capita di incontrare procedure che

comprendono cicli con *entry point* multipli. In situazioni di questo tipo è necessario ricorrere ad algoritmi più complessi di quello utilizzato correntemente. In futuro si prevede di risolvere il problema ricorrendo all'algoritmo discusso in [77], un'estensione dell'algoritmo proposto da Tarjan [80] in grado di identificare sia cicli riducibili che irriducibili.

Risulterebbe piuttosto utile migliorare l'algoritmo di *disassembly* utilizzato per tradurre il codice macchina dell'applicazione in codice *assembly*. Come già discusso, attualmente il modello utilizza la tecnica *recursive traversal*, capace di risolvere alcune delle problematiche proprie dell'approccio *linear sweep*. Il procedimento potrebbe però essere ulteriormente migliorato, cercando di massimizzare la percentuale di codice riconosciuta staticamente. Ciò consentirebbe, ad esempio, di ridurre la probabilità di rimuovere assegnamenti *live* durante l'analisi della *liveness* delle istanze di istruzioni intermedie. Si potrebbe utilizzare un approccio ibrido *linear-traversal* per combinare i vantaggi di entrambi i metodi, così come descritto in [74]. In aggiunta, si potrebbero adottare le euristiche proposte in [84, 74] per cercare di individuare porzioni di codice raggiungibili solo staticamente e produrre risultati corretti anche in presenza di applicazioni che ricorrono all'utilizzo di alcune delle tecniche di offuscamento presentate in [57].

Infine, si ritiene che un ultimo aspetto che debba ancora essere notevolmente sviluppato riguarda l'analisi dei costrutti di iterazione (Sezione 4.4.2.6). In primo luogo si potrebbe cercare di migliorare la tecnica di analisi in modo da essere in grado di dedurre le relazioni esistenti tra le diverse variabili di induzione presenti all'interno del ciclo. Per fare un esempio, si consideri ancora una volta la procedura `copy2()` di Figura 4.8. L'analizzatore non è attualmente in grado di astrarre il comportamento del ciclo *while* dato che l'unica variabile di induzione utilizzata nella condizione di terminazione del ciclo, `src`, non è poi impiegata per definire l'indirizzo di destinazione di un'operazione di scrittura in memoria. Una seconda variabile di induzione, `i`, è invece utilizzata come indice di un array, ma non contribuisce a definire la condizione del *while*. In realtà le due variabili sono fortemente correlate, in quando entrambe vengono incrementate nel corso di ogni iterazione del ciclo. Per questo motivo, se si fosse in grado di dedurre la relazione esistente tra `src` e `i`, si potrebbe stabilire che

```
1  void sum(char *src)
2  {
3      char dest[10];
4      int s, i;
5
6      s = 0;
7      i = 0;
8
9      do {
10         s += *src;
11         dest[i] = s;
12
13         i++;
14         src++;
15     } while(s < 100 && *src);
16 }
```

Figura 6.1: Esempio di calcolo di valore "aggregato".

agendo sulla lunghezza della stringa `src` è comunque possibile controllare l'indirizzo di memoria definito dall'istruzione alla riga 7.

Una seconda problematica relativa all'analisi dei cicli è costituita dal calcolo di valori "aggregati" sui dati contaminati dall'input. Per chiarire questo concetto è opportuno ricorrere ad un semplice esempio. Si consideri la procedura `sum()` di Figura 6.1: si tratta di una semplice funzione che somma i byte del vettore di input `src` memorizzando le somme parziali nell'array locale `dest`, arrestando l'operazione al primo byte nullo di `src` o comunque quando la somma parziale risulta maggiore o uguale a 100. Nonostante la metodologia di analisi proposta sia in grado di determinare la dipendenza esistente tra la condizione di terminazione del ciclo e la lunghezza della stringa `src`, non è attualmente capace di determinare il legame esistente tra la condizione del ciclo e la variabile `s`. Rimane quindi da valutare se sia possibile migliorare l'attuale tecnica di analisi dei cicli in modo da poter trattare anche situazioni di questo.

Capitolo 7

Lavori correlati

Vengono ora presentati alcuni lavori correlati al modello discusso finora. Il presente capitolo inizia trattando l'approccio "*fuzzy*" tradizionale, per poi passare ad analizzare alcune recenti pubblicazioni relative all'applicazione delle metodologie di *program analysis* per l'individuazione di problematiche rilevanti dal punto di vista della sicurezza. Per ciascuna alternativa discussa, vengono sottolineati i vantaggi e gli svantaggi del metodo proposto, insieme con le affinità e le differenze rispetto al modello studiato nel presente lavoro di tesi. Il capitolo si conclude con una breve panoramica sui risultati ottenuti dalla ricerca nell'ambito della generazione automatica o semiautomatica dei casi di test.

7.1 Fuzzing

Con il termine *fuzz testing*, o semplicemente *fuzzing*, si fa riferimento ad una tecnica di *program testing* particolarmente semplice, che prevede la generazione di casi di test in modo casuale. L'idea alla base di tale approccio consiste nel collegare l'applicazione in esame ad una sorgente di dati casuali o pseudo-casuali. Nel caso in cui uno di questi input sia in grado di arrestare in modo imprevisto la computazione (*crash*) o fare in modo che l'esecuzione entri in ciclo infinito (o almeno in un ciclo che si può ragionevolmente supporre infinito), ciò viene considerato come sintomo della presenza di un difetto nel programma. Si tratta quindi di una forma di *black-box testing*, in quanto

l'applicazione analizzata è trattata come fosse una "scatola nera", di cui non si dispone di alcuna informazione circa il comportamento interno. Le uniche informazioni di cui si suppone di poter disporre sono le specifiche che descrivono come l'applicazione *dovrebbe* comportarsi.

L'idea originale di *fuzz testing* fu proposta nel 1990 da Barton Miller e altri in [61]. In tale lavoro gli autori applicano la tecnica del *fuzzing* a circa 90 differenti utility a linea di comando su 7 versioni di UNIX, riuscendo ad individuare difetti in oltre il 24% dei programmi esaminati. Negli anni seguenti, il gruppo di Miller ripeté l'esperimento con insiemi di programmi differenti, su sistemi UNIX [62], Mac OS X [60] e Windows [31], con esiti particolarmente incoraggianti.

La generazione casuale di casi di test è stata per qualche tempo una pratica duramente criticata in letteratura. Nel suo libro sul *software testing* [63], Myers considera il *random testing* come la metodologia "*meno efficace di tutte*" e poco incline a consentire di individuare un numero rilevante di difetti. Nonostante ciò, tenendo conto della semplicità della tecnica di testing proposta e considerando che i *fuzzer* di Miller non utilizzano alcuna specifica sul funzionamento dei programmi esaminati, i risultati ottenuti dagli autori appaiono quanto meno sorprendenti e il *fuzzing* è oggi parte del processo di sviluppo del software di numerose aziende, tra cui la stessa Microsoft [58]. Lo stesso Miller, però, sottolinea che simili tecniche di testing non possono avere pretese di esaustività e sono solamente in grado di completare, non certo sostituire, approcci più formali.

La tecnica di *black-box fuzzing* descritta finora ha certamente come vantaggi la semplicità e la possibilità di automatizzare l'intero procedimento. D'altra parte, però, è evidente il principale inconveniente legato ad un approccio di questo tipo: l'impiego di dati di input completamente casuali e la mancanza di informazioni sul comportamento dell'applicazione limitano notevolmente la percentuale del codice coinvolto nel processo di testing.

Un primo tentativo di superare i limiti naturali dell'approccio *fuzzy* tradizionale consiste nel generare i casi di test tenendo conto della struttura che i dati di input devono rispettare. Ad esempio, l'esame di un server web tramite richieste HTTP completamente casuali risulterebbe ben poco efficace, dato che la probabilità di ottenere

anche una singola richiesta sintatticamente corretta sarebbe piuttosto ridotta. Al contrario, avrebbe maggiore utilità un *fuzzer* in grado di variare in modo casuale solo alcuni campi, continuando però a produrre richieste sintatticamente valide. Da analoghe considerazioni si sono sviluppati numerosi strumenti di *fuzzing* dipendenti dal protocollo seguito dalla specifica applicazione, alcuni dei quali incentrati sull'analisi di protocolli di rete [83], altri focalizzati sulla ricerca di vulnerabilità in applicazioni web [78], altri ancora in grado di applicare le tecniche *fuzzy* alla struttura stessa dei file eseguibili [44], fino ad arrivare a veri e propri *framework* per la generazione di nuovi strumenti di testing [4].

7.2 Ricerca automatica di vulnerabilità

Nonostante gli strumenti *protocol-dependent* presentati nella sezione precedente risultino notevolmente più efficaci rispetto a quando il programma è analizzato utilizzando un *fuzzer* tradizionale, questi non fanno altro che risolvere parzialmente il problema di fondo dell'approccio *fuzzy*. Infatti, applicazioni anche mediamente complesse hanno tipicamente un *control flow* piuttosto articolato, per cui appare poco probabile riuscire ad indurre la manifestazione di una vulnerabilità quando questa dipende da una complessa rete di condizioni sui dati di input.

Per questo motivo la ricerca attuale ha iniziato a considerare approcci più articolati, in grado di dedurre e analizzare il *control flow* dell'applicazione al fine di riuscire ad esplorare con maggiore efficacia gli stati che il programma può raggiungere. Di seguito vengono descritti alcuni modelli che si inseriscono in questo contesto e che, malgrado si pongano obiettivi analoghi a quelli discussi nel corso della Sezione 3.4, sviluppano idee in qualche modo differenti rispetto a quelle proposte nel presente lavoro di tesi.

Solo uno dei lavori presentati di seguito [20] è focalizzato in modo specifico sulla ricerca automatica delle vulnerabilità, mentre un altro [66] si concentra sul rilevamento di un attacco informatico tramite l'analisi dell'esecuzione di un'applicazione. Sono poi discusse alcune metodologie [15, 36] che si inseriscono nel contesto più generale dell'individuazione dei difetti presenti in un'applicazione software, impiegando però tecniche molto simili a quelle utilizzate per l'identificazione di problematiche di sicu-

rezza, quali, ad esempio, la generazione di casi di test in grado di esplorare in modo esaustivo i cammini di esecuzione possibili.

7.2.1 Static detection of vulnerabilities in x86 executables

In [20] gli autori presentano un approccio per l'identificazione delle vulnerabilità di un'applicazione software a partire dal file eseguibile. Il lavoro descritto si pone come obiettivo l'individuazione di possibili utilizzi di dati controllabili dall'attaccante come argomento di procedure delicate. In particolare, il modello proposto si sofferma sulle chiamate di libreria `system()` e `popen()`, utilizzate nei sistemi Linux per eseguire il comando identificato dalla stringa passata come argomento.

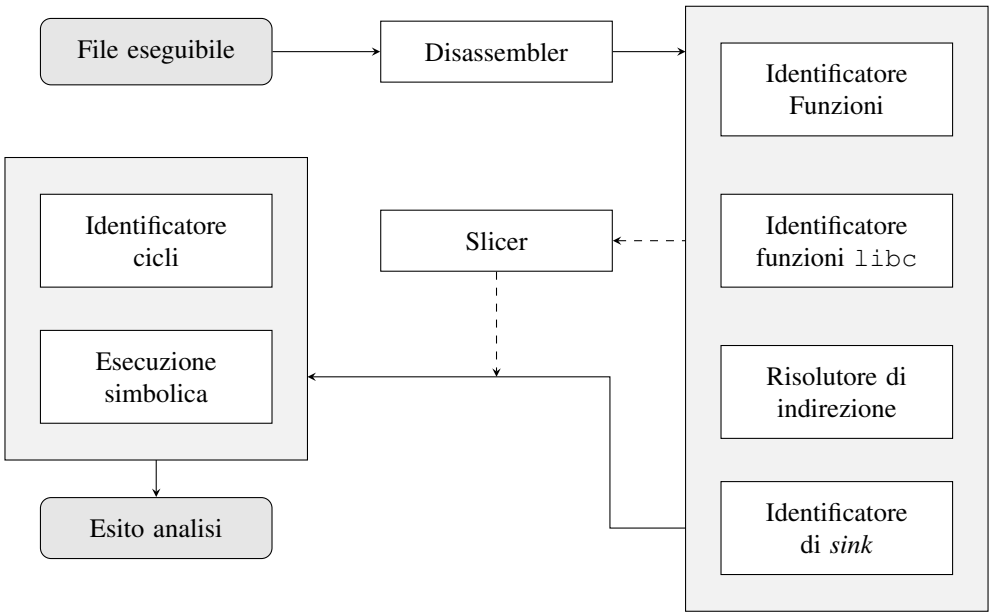

Figura 7.1: Diagramma a blocchi del modello di analisi descritto in [20].

La tecnica adottata dagli autori prevede due fasi principali, riassunte nel diagramma a blocchi riportato in Figura 7.1. Innanzi tutto tramite analisi statica vengono estratte dall'eseguibile le informazioni necessarie per le operazioni successive. In seguito, il modello prevede una fase di esecuzione simbolica che consente di individuare possi-

bili cammini in grado di portare all'utilizzo di dati pericolosi come argomenti delle funzioni `system()` o `popen()`.

Più in dettaglio, la fase di analisi statica inizia con il *disassembly* dell'eseguibile. Si procede quindi cercando di risolvere salti e chiamate indirette mediante l'applicazione di una serie di euristiche, in parte dipendenti dallo specifico compilatore utilizzato per la generazione dell'eseguibile. Vengono quindi generati i CFG associati alle procedure individuate finora, procedendo poi all'identificazione di cicli e chiamate ricorsive. La fase si conclude con la risoluzione dei nomi delle funzioni di libreria tramite l'impiego delle informazioni di rilocazione memorizzate all'interno del file. Ciò consente di identificare i *sink*, ovvero le locazioni in corrispondenza delle quali vengono invocate le funzioni `system()` o `popen()`, su cui si concentreranno le analisi successive. Così come per l'approccio *smart fuzzing* descritto nei capitoli precedenti, anche qui, al fine di incrementare l'efficienza dell'intero procedimento, vengono impiegate tecniche di *slicing* per estrarre le istruzioni che possono avere una qualche influenza sugli argomenti delle chiamate di libreria oggetto dell'analisi.

A questo punto si dispone delle informazioni necessarie per proseguire con l'esecuzione simbolica [49] dell'applicazione. Ciò consiste nel simulare l'esecuzione del programma, supponendo come input un insieme di simboli che rappresentano valori arbitrari dei dati in ingresso. Inizialmente all'applicazione è associato un insieme di condizioni $PC = \emptyset$. In corrispondenza di un'istruzione in grado di alterare il flusso di esecuzione in base alla valutazione di una condizione c, l'esecuzione viene ramificata. Al ramo associato al valore di verità "*vero*" viene assegnato un nuovo insieme di vincoli $PC' = PC \cup c$, mentre l'insieme di condizioni relative al ramo "*falso*" sarà $PC'' = PC \cup \neg c$. Ogni volta che viene aggiunta una nuova condizione all'insieme dei vincoli, ne vengono valutati gli effetti sui dati in ingresso. In alcuni casi ciò può portare a determinare la non soddisfacibilità di un certo insieme di condizioni, mentre in altre situazioni potrebbe permettere di concretizzare uno o più dati simbolici associati ad una particolare esecuzione simulata.

Nonostante alcune similitudini relative alle metodologie impiegate per trattare codice binario, l'approccio descritto in [20] differisce in maniera sostanziale rispetto a quanto proposto nel presente lavoro di tesi. Innanzi tutto le tecniche di analisi utilizzate

sono puramente statiche: le informazioni necessarie vengono ricavate senza eseguire realmente l'applicazione, ma al più interpretando il codice del programma tramite il motore di esecuzione simbolica. Se da un lato ciò comporta tutti i vantaggi connessi alla possibilità di ottenere informazioni indipendenti dalla specifica esecuzione, dall'altro però introduce necessariamente una serie di limitazioni evidenziate dagli stessi autori e legate alle imprecisioni dell'analisi. Per prima cosa il modello proposto non tiene in alcun modo in considerazione eventuali relazioni di *aliasing* tra le espressioni. Se a livello del codice sorgente una limitazione di questo tipo può apparire ragionevole, a livello del codice binario una simile assunzione pare piuttosto forte e incline a produrre un considerevole numero di falsi negativi e falsi positivi. In secondo luogo, le tecniche di analisi statica utilizzate dagli autori non consentono di determinare in modo accurato la condizione di terminazione di un ciclo. Per questo motivo l'analisi dei cicli è limitata a tre iterazioni, oltre le quali tutti i registri e le locazioni di memoria modificate dal corpo del ciclo sono considerate indefinite. Analogamente, l'analisi non è in grado di trattare chiamate ricorsive.

In altri termini, la limitazione fondamentale dell'approccio appena descritto risiede nell'imprecisione propria delle tecniche di analisi statica, certamente amplificata quando tali metodi sono applicati sul codice binario. Questo obbliga ad accettare un tasso di falsi positivi piuttosto elevato e un discreto numero di falsi negativi: i primi sono dovuti all'eccessivo conservativismo dell'analisi, mentre i secondi potrebbero essere legati alla mancata identificazione di cammini percorribili, a causa, ad esempio, dell'incapacità di risolvere alcuni trasferimenti di controllo indiretti. Si osservi che adottando un approccio ibrido i due fenomeni possono essere notevolmente contenuti, dato che l'analisi dinamica può essere impiegata sia per raffinare le informazioni conservative ottenute staticamente, sia per verificare concretamente la validità dei risultati prodotti andando ad eseguire l'applicazione sui dati in ingresso ritenuti potenzialmente pericolosi.

7.2.2 TaintCheck

TaintCheck [66] è uno strumento in grado di analizzare dinamicamente il codice binario di un'applicazione al fine di determinare in modo automatico quando un dato di input è in grado di condurre l'esecuzione verso la manifestazione di una vulnerabilità. Tale procedimento viene impiegato per individuare attacchi informatici che sfruttano nuove vulnerabilità e generare automaticamente una *signature* da utilizzarsi per identificare altri attacchi dello stesso tipo.

Nonstante le finalità di TaintCheck siano differenti rispetto agli obiettivi considerati dal presente lavoro di tesi, le tecniche adottate risultano comunque molto simili. Anche TaintCheck, infatti, effettua una *dynamic taint analysis* dell'applicazione in esame: il codice binario del programma viene instrumentato utilizzando l'architettura offerta da Valgrind [65] per poi essere eseguito in un ambiente controllato. TaintCheck inizialmente considera i dati provenienti dalla rete o da altre fonti di input come contaminati (*tainted*), tenendo poi traccia di come tali dati vanno ad influenzare altre variabili del programma. Nel caso in cui un dato *tainted* venga utilizzato in modo da compromettere la sicurezza dell'applicazione (ad esempio, come destinazione di un salto indiretto), allora l'esecuzione viene arrestata e viene generata una *signature* per l'attacco appena rilevato.

La differenza fondamentale con l'approccio *smart fuzzing* risiede nel fatto che TaintCheck non deve preoccuparsi di esplorare tutti i possibili cammini di esecuzione possibili, dato che ciò non è richiesto per perseguire le finalità che gli autori si sono prefissi. Di conseguenza, TaintCheck non deve ricostruire accuratamente le *path condition* associate alla specifica esecuzione, generare un nuovo input in grado di esplorare un nuovo stato del programma in esame o trattare in modo particolare i costrutti di iterazione, ma può limitarsi a rilevare eventuali attacchi e determinare la porzione dell'input che ha portato alla compromissione della sicurezza del sistema.

7.2.3 EXE

Il lavoro presentato in [15, 14] da Engler e altri costituisce, per quando ne sappiamo, il risultato migliore ottenuto finora nell'ambito dell'identificazione automatica dei difetti

presenti in un'applicazione software.

Nell'articolo appena citato, gli autori descrivono EXE ("*EXecution generated Executions*"), uno strumento in grado di analizzare un programma e generare automaticamente un insieme di input in grado di indurre la manifestazione di un difetto. EXE opera a livello del codice sorgente, focalizzandosi sull'analisi di codice C. L'unico intervento manuale richiesto consiste nell'instrumentazione del codice dell'applicazione tramite l'introduzione di alcune istruzioni necessarie per segnalare allo strumento quali variabili devono essere considerate come "*simboliche*". Il programma deve quindi essere ricompilato utilizzando il compilatore fornito con EXE, che si occupa di introdurre controlli in corrispondenza di ciascun assegnamento, espressione e salto condizionato, in modo da monitorare il percorso seguito dai dati simbolici. Un'operazione che coinvolge dati esclusivamente concreti (ovvero non simbolici) viene eseguita come se il programma non fosse instrumentato. Se però un qualche operando è simbolico, allora l'operazione non è eseguita realmente ma viene analizzata dal motore di EXE in modo da determinarne gli effetti sul risultato. Se nel corso dell'esecuzione viene incontrata un'istruzione in grado di alterare il flusso di controllo in base alla valutazione di un'espressione condizionale, allora EXE procede a ramificare l'esecuzione in modo del tutto analogo a quanto descritto nel Capitolo 4. Per la risoluzione degli insiemi di vincoli associati a ciascuna esecuzione, viene utilizzato il *constraint solver* STP già discusso nella Sezione 4.6 e sviluppato dagli stessi autori di EXE.

L'efficacia del metodo proposto è testimoniata dai sorprendenti risultati ottenuti dagli autori, che sono stati in grado di scoprire numerose problematiche (alcune delle quali anche rilevanti dal punto di vista della sicurezza) all'interno di codice complesso, maturo e sperimentato per lungo tempo, come, ad esempio, lo stesso kernel Linux [88] o il *packet filter* BPF utilizzato nei sistemi FreeBSD e Linux. Per ciascuno degli esempi riportati, EXE è stato in grado di generare automaticamente dei casi di test che inducono la manifestazione del difetto.

È importante sottolineare che mentre approcci come quello descritto nella Sezione 7.2.1 prevedono un'esecuzione *simulata* dell'applicazione tramite l'interpretazione software di ogni singola istruzione assembly, il metodo descritto da Engler suppone di eseguire concretamente il codice, gestendo in modo specifico solamente le istruzioni

che trattano dati simbolici e procedendo a ramificare l'esecuzione quando necessario. Ciò consente di incrementare notevolmente la precisione dei risultati ottenibili da questo tipo di analisi ed elimina molti degli inconvenienti presenti nel lavoro trattato nella sezione precedente.

D'altra parte, i limiti di EXE vanno essenzialmente individuati nell'analisi del codice sorgente dell'applicazione e nella necessità di instrumentare il programma esaminato. Innanzi tutto, la necessità di instrumentare manualmente il codice del programma e successivamente ricompilare l'intera applicazione può risultare, in alcuni contesti, piuttosto pesante. In secondo luogo, si è già discusso nella Sezione 3.3 circa i problemi legati all'analisi del codice sorgente, riassumibili nell'impossibilità di analizzare applicazioni *closed source* e di individuare difetti che dipendono da dettagli propri della piattaforma su cui viene eseguita l'applicazione. Si osservi, ad esempio, che nel caso in cui il programma analizzato faccia uso di funzioni di libreria per le quali non si dispone del codice sorgente, allora EXE si troverebbe costretto ad utilizzare simili procedure "*a scatola chiusa*", senza poter contare su alcuna informazione circa il loro comportamento interno e senza essere quindi in grado di individuare difetti che in qualche modo dipendono da tali funzioni.

Nonostante gli svantaggi appena discussi, EXE è probabilmente la soluzione più valida finora proposta e certamente quella che è riuscita ad ottenere i risultati più incoraggianti.

7.2.4 DART

La metodologia di testing automatico del software descritta in [36] è sotto molti punti di vista simile all'approccio adottato da EXE e descritto nella precedente sezione. Gli autori, descrivono DART ("*Directed Automated Random Testing*"), uno strumento che si prefigge come obiettivo principale la generazione in modo completamente automatico di casi di test in grado di condurre l'applicazione esaminata lungo ogni possibile cammino di esecuzione.

Così come EXE, anche DART si concentra sull'analisi di applicazioni scritte in linguaggio C. Il codice sorgente del programma viene instrumentato, ricompilato e

inizialmente eseguito su un input casuale. Durante l'esecuzione dell'applicazione vengono esaminate le condizioni associate ad istruzioni di salto condizionato (ad esempio, costrutti *if*), generando un insieme di vincoli che, una volta soddisfatti, permettono di definire un nuovo insieme di dati di input che consente di condurre l'esecuzione del programma verso lo stato voluto.

Analogamente a quanto previsto dal modello discusso nel presente lavoro di tesi, anche DART trasforma le istruzioni dell'applicazione in una forma intermedia sulla quale effettuare le analisi necessarie. Le tipologie di istruzioni intermedie previste sono quattro: istruzioni condizionali, istruzioni di assegnamento, *halt* (terminazione del programma) e *abort* (errore, come nel caso della funzione C `abort()`).

La metodologia di analisi proposta dagli autori prevede sia l'impiego di tecniche di esecuzione simbolica, sia il ricorso all'esecuzione concreta. In particolare, durante l'esecuzione DART tiene traccia dell'espressione simbolica lineare che definisce il valore di ciascuna variabile del programma analizzato. Se il valore di una certa variabile non può essere espresso tramite una combinazione lineare delle altre variabili del programma, allora DART ripiega sull'esecuzione concreta, valutando dinamicamente il valore della variabile in esame. Lo stesso avviene nel caso in cui l'applicazione cerchi di dereferenziare un puntatore il cui valore dipende da un dato di input. Al termine di ciascuna esecuzione, viene analizzata la *execution history* corrente insieme con le condizioni valutate dai salti incontrati finora e viene generato un sistema di vincoli che, una volta risolto, consente di definire un input in grado di percorrere un nuovo cammino di esecuzione, scelto in modo da esplorare in profondità l'albero dei cammini possibili. Il sistema di vincoli viene inviato ad un *constraint solver* che si occupa di risolverlo e, se soddisfacibile, di restituire una nuova configurazione delle variabili in ingresso che soddisfa le condizioni imposte dal sistema.

Una volta esplorati tutti i cammini possibili, DART valuta se nel corso di una qualche esecuzione ha dovuto concretizzare il valore di una variabile. Se ciò non si è verificato, allora il processo di analisi può considerarsi concluso. In caso contrario, al fine di garantire un risultato conservativo, DART applica nuovamente l'intero procedimento di analisi. In questo modo, se DART termina, allora ciò implica che è stato percorso ogni possibile cammino di esecuzione, senza dover mai ripiegare sull'esecu-

zione concreta. Se in qualche circostanza l'analisi è dovuta ricorrere alla valutazione dinamica del valore di una variabile, allora l'intero procedimento non terminerà.

Simili accorgimenti permettono agli autori di garantire formalmente che DART termina senza incontrare alcuna istruzione di *abort* se e solo se non esiste alcun cammino di esecuzione che consente di raggiungere una simile istruzione. Se DART non termina, allora non si dispone di alcuna informazione certa sulla correttezza del programma in esame.

L'impiego dell'esecuzione simbolica e il ricorso all'esecuzione concreta solo quando l'espressione associata ad una variabile cade al di fuori della teoria supportata dal *constraint solver* permette quindi a DART di superare in qualche modo i limiti imposti dal *prover* utilizzato e di garantire comunque la correttezza dell'analisi.

Le differenze tra l'approccio suggerito dagli autori di DART e quello proposto da Engler e altri con EXE sono riassumibili in due aspetti fondamentali. Per prima cosa, EXE dispone di una procedura di decisione *ad-hoc*, appositamente progettata per trattare le espressioni ottenute dall'analisi di codice C, mentre il non determinismo derivante dalla presenza di condizioni non lineari pare limitare l'efficacia di DART. D'altra parte, però, non è realistico supporre che il *solver* STP utilizzato da EXE sia effettivamente in grado di gestire *ogni* possibile insieme di vincoli. Si supponga, ad esempio, di incontrare una condizione di alto livello del tipo *md5(buffer) = 123456*. Appare certamente poco probabile che STP sia in grado di determinare una configurazione dell'array *buffer* che consenta di verificare l'uguaglianza. Nonstante le limitazioni che ne derivano, il ricorso ad un approccio non deterministico come quello adottato da DART risulta in alcune circostanze indispensabile per trattare codice che non può essere gestito in modo formale. In secondo luogo, mentre gli autori di EXE non discutono le problematiche relative all'analisi di funzioni di libreria per le quali non si dispone del codice sorgente, gli autori di DART trattano situazioni di questo tipo eseguendo concretamente la procedura, potendo eventualmente simulare un valore di ritorno casuale. Un approccio di questo tipo non risolve certamente il problema, ma per lo meno consente di stimolare l'applicazione in maniera più approfondita.

Confrontando invece DART con la metodologia di analisi proposta nel presente lavoro di tesi, emergono certamente alcuni punti in comune, come la tecnica di analisi

basata sulla traduzione del programma in forma intermedia, la ricostruzione delle *path condition* associate ad un certo cammino di esecuzione e la tecnica di esplorazione dei cammini possibili. Naturalmente DART analizza codice sorgente, per cui le problematiche che si trova a dover fronteggiare sono spesso differenti rispetto a quando le stesse analisi sono condotte sul codice binario. Infine, un aspetto discusso durante la presentazione del modello *smart fuzzing* e non approfondito dagli autori di DART riguarda l'analisi dei cicli: senza l'adozione di alcuna tecnica per l'astrazione del comportamento di un ciclo o almeno una qualche euristica che limiti il numero delle iterazioni effettuate, DART pare destinato ad arenarsi indefinitiviamente all'interno del corpo di un costrutto di iterazione che presenta una condizione di terminazione dipendente dai dati di input.

7.3 Generazione automatica dei casi di test

Il problema di individuare in modo automatico le vulnerabilità presenti in un programma informatico attraverso l'esplorazione sistematica dei suoi possibili stati di esecuzione può essere inquadrato nel più generico contesto del *testing* dei prodotti software e, più in particolare, nell'ambito della generazione dei casi di test. Una vulnerabilità infatti non è altro che una particolare tipologia di difetto capace di compromettere la sicurezza dell'applicazione, a volte senza nemmeno manifestarsi all'utente tramite l'alterazione delle funzionalità del programma in esame. La metodologia di analisi discussa nei capitoli precedenti e i lavori presentati nella Sezione 7.2 non fanno altro che generare alcuni input che si ritiene possano consentire di esplorare nuovi stati dell'applicazione, utilizzando alcune euristiche per indirizzare l'analisi verso l'esplorazione di stati in qualche modo "interessanti".

Oltre a considerare i lavori focalizzati in modo più o meno specifico sull'individuazione delle vulnerabilità, sembra quindi utile discutere brevemente alcuni risultati significativi ottenuti dalla ricerca relativamente alla generazione automatica o semiautomatica dei casi di test.

Da un punto di vista puramente teorico, l'unico procedimento che consente di dimostrare sperimentalmente la correttezza di un prodotto software consiste nel veri-

ficare i risultati prodotti dall'applicazione in corrispondenza di *ogni* possibile configurazione delle variabili di input. Naturalmente una soluzione di questo tipo non è praticamente attuabile, in quanto spesso lo spazio degli input di un programma si dimostra essere concettualmente infinito. Per questo motivo è solitamente necessario selezionare un insieme limitato di possibili dati di input (o *casi di test*) che soddisfano alcuni criteri [63]. In particolare, i criteri *strutturali* impongono che alcuni elementi fondamentali del programma siano valutati in modo esaustivo. Per fare alcuni esempi, il criterio di *statement coverage* cerca di massimizzare il numero di istruzioni del programma visitate nel corso delle esecuzioni effettuate, mentre il criterio di *branch coverage* mira a definire un insieme di casi di test che consenta di visitare ogni ramo associato ad un'istruzione condizionale. Il criterio adottato dal metodo di analisi proposto nel presente lavoro e utilizzato anche dagli approcci alternativi descritti nella sezione precedente è invece più ambizioso, in quanto mira a visitare ogni possibile cammino di esecuzione che risulta essere percorribile partendo dalla prima istruzione del programma in esame (*path coverage*). Si tratta però di un obiettivo difficilmente raggiungibile nel caso di programmi che fanno uso di costrutti iterativi, per cui è comunque indispensabile ricorrere ad una qualche euristica che consenta di limitare il numero di iterazioni effettuate in corrispondenza di ciascun ciclo.

Viene ora affrontata una breve discussione circa le recenti evoluzioni relative agli strumenti per la generazione automatica dei casi di test.

Generatori *random*. Negli ultimi anni sono state proposte diverse soluzioni alternative per la generazione automatica dei casi di test. Un primo approccio è descritto da Bird e Munoz in [12] e consiste nella generazione casuale dei casi di test, in modo sotto alcuni punti di vista analogo a quanto fatto dai *fuzzer protocol-dependent*. Il metodo prevede la definizione accurata della struttura che un caso di test deve rispettare, precisando l'insieme entro cui ciascun elemento può variare, in modo tale da poter produrre casi sintatticamente corretti.

I generatori casuali possono solo disporre delle specifiche che definiscono il comportamente dell'applicazione analizzata, senza poter contare su alcuna altra informazione. Inoltre è necessario definire un generatore differente per ciascuna applicazione

da analizzare, a meno naturalmente di formalizzare la rappresentazione delle specifiche in modo da consentirne un'elaborazione automatica. L'efficacia dei generatori casuali pare quindi essere sensibilmente limitata dalle problematiche appena enunciate.

Generatori *path-oriented*. Un approccio alternativo che consente di derivare casi di test ancor più significativi è costituito dai generatori *path-oriented*. Si tratta di strumenti che, partendo da un dato programma e da un criterio di test, sono in grado di produrre un insieme di casi di test che rispettano il criterio specificato.

In particolare, in un generatore *path-oriented* si possono solitamente distinguere tre fasi principali: (I) costruzione del *control flow graph* dell'applicazione; (II) identificazione dei cammini di esecuzione che rispettano il criterio di test; (III) generazione dei casi di test che consentono di percorrere i cammini individuati.

Alcuni strumenti di questo tipo [13] impiegano tecniche di esecuzione simbolica per dedurre l'insieme di vincoli sui dati di input associati ad un particolare cammino di esecuzione. I problemi principali di una simile tecnica sono legati a questioni relative al proliferare delle espressioni simboliche intermedie, alla precisione nella ricostruzione dei vincoli anche in presenza di array o puntatori e alla capacità di risolvere *constraint* anche non lineari.

Una tecnica alternativa è detta *execution-oriented* [51] e consiste nel valutare dinamicamente i valori assunti dalle variabili del programma. Un caso di test che consente di percorrere un particolare cammino di esecuzione può poi essere generato ricercando una possibile configurazione delle variabili in ingresso tramite l'applicazione di algoritmi per la ricerca del minimo di opportune funzioni associate ai predicati condizionali. L'intero procedimento può essere reso più efficiente ricorrerendo alla *dynamic data flow analysis* per ricostruire le dipendenze esistenti tra predicati condizionali e dati di input, in modo da concentrare gli algoritmi di minimizzazione solo sulle variabili utilizzate in un'espressione condizionale che risultano dipendere effettivamente dai dati di input. Nonostante le metodologie *execution-oriented* consentano di superare molti dei limiti propri degli approcci che fanno uso dell'esecuzione simbolica, soffrono comunque di problemi di efficienza dovuti alla necessità di eseguire il programma in esame un numero spesso considerevole di volte.

Generatori *goal-oriented*. Il problema principale di tutte le tipologie di generatori *path-oriented* è legato ai cammini non percorribili [45]. Dato che la selezione di un cammino di esecuzione avviene solitamente esaminando esclusivamente il *control flow graph* del programma in esame, capita spesso che vengano scelti cammini non percorribili, sprecando così una quantità considerevole di risorse computazionali e riducendo l'efficienza dell'intero procedimento.

È quindi stata proposta una terza categoria di generatori di casi di test, ottenuta a partire dai generatori *path-oriented* tramite l'eliminazione della fase preliminare di selezione dei cammini che soddisfano un dato criterio. L'idea alla base di tali strumenti consiste nel concentrare l'analisi solamente sui rami del CFG in grado di influire realmente sul raggiungimento dell'obiettivo definito dal criterio di test.

Più in dettaglio, in questa nuova categoria di generatori il cammino di esecuzione è costruito dinamicamente mediante l'esecuzione concreta del programma in esame e l'impiego di tecniche di *dynamic data flow analysis*. In particolare, all'applicazione è inizialmente associata una configurazione casuale delle variabili in ingresso. Quando viene raggiunta un'istruzione in grado di alterare il flusso di controllo in base al risultato della valutazione di un'espressione condizionale, vengono esaminate le *control dependency* esistenti tra il nodo che si intende raggiungere e l'istruzione condizionale stessa. Così facendo è possibile stabilire se, al fine di soddisfare il criterio di test, conviene proseguire lungo il cammino delineato dall'input attuale oppure se è opportuno ricorrere alla generazione di un caso di test alternativo che consenta di raggiungere un differente successore dell'istruzione corrente. Il procedimento continua fino a quando non viene raggiunto il nodo indicato dal criterio di test. I generatori ottenuti dall'implementazione dell'approccio appena descritto sono detti *goal-oriented* e sono stati proposti per la prima volta da Korel in [50].

Generatori *chain-oriented*. Sia i generatori *path-oriented* che quelli *goal-oriented* guidano il processo di ricerca utilizzando esclusivamente il *control flow graph* del programma analizzato. Così facendo, però, in alcune situazioni risulta particolarmente difficile generare un caso di test in grado di raggiungere uno specifico nodo. Infatti, focalizzando l'analisi sul CFG dell'applicazione si può disporre di informazioni circa

le dipendenze di controllo esistenti tra i nodi del grafo, ma non si possono considerare relazioni a livello delle singole istruzioni quali le *data dependency*. Per questi motivi, l'evoluzione naturale degli approcci precedenti porta a considerare anche le informazioni derivabili dalle relazioni di *data dependency*, in modo da raffinare ulteriormente il processo di ricerca di un input che soddisfi il criterio di test.

Ferguson e Korel formalizzarono queste intuizioni in [29], definendo una tecnica per la generazione dei casi di test nota come *chaining approach*. Uno strumento di questo tipo è sostanzialmente un'estensione di un generatore *goal-oriented*. Quando un generatore *chain-oriented* non è in grado di determinare un input in grado di raggiungere un successore q di un'istruzione condizionale p (detta *problem node*), viene determinato l'insieme $D = \bigcup_{u \in use(p)} srd(u,p)$ che comprende tutte e sole le *static reaching definition* delle variabili utilizzate dal predicato valutato da p. Si supponga sia $D = \{d_1, \ldots, d_n\}$. Allora il metodo prevede di ripetere il processo di ricerca imponendo però che l'istruzione d_1 venga visitata e che il cammino da d_1 fino a p sia *definition-clear* rispetto a $define(d_1) \cap use(p)$. La risoluzione di questo sottoproblema può portare alla scoperta di un input che soddisfa il criterio iniziale, a ripetere ricorsivamente il procedimento in seguito alla scoperta di un nuovo *problem node* p', oppure a determinare l'insoddisfacibilità dell'insieme di vincoli. Nell'ultimo caso, il metodo prevede che venga considerata la seconda definizione d_2 e venga ripetuta la ricerca dell'input. Il procedimento continua fino a quando non viene trovato un caso di test valido oppure viene determinata l'insoddisfacibilità dell'insieme di vincoli relativi all'ultima definizione d_n. Si osservi che se si limita il processo ricorsivo ad una profondità massima k, allora con $k = 0$ si ottiene un generatore *goal-oriented*.

Generatori *constraint-based*. Un ultimo metodo, radicalmente differente da quelli appena discussi, è presentato da Gotlieb e altri in [37] e sfrutta i progressi fatti nell'ambito della risoluzione automatica di sistemi di vincoli.

Il procedimento proposto prevede due fasi principali. Per prima cosa vengono determinate la forma SSA (*Static Single Assignment* [73]) del programma e le dipendenze di controllo esistenti. Il codice dell'applicazione viene quindi tradotto staticamente in un sistema di *constraint* che ne riassume la semantica. La seconda fase

consiste nella risoluzione di tale sistema in modo da determinare se esiste almeno un cammino percorribile in grado di raggiungere l'obiettivo fissato dal criterio di test e, in tal caso, viene utilizzato un metodo di ricerca basato sulla risoluzione formale dei vincoli insieme con tecniche di enumerazione dei valori assumibili dalle variabili logiche per determinare una possibile configurazione dei parametri di input in grado di consentire il raggiungimento del nodo obiettivo. Il metodo di ricerca è corretto da opportune euristiche che consentono di aumentare sensibilmente l'efficienza dell'intero procedimento.

Dalla precedente discussione emergono alcune analogie significative tra il modello *smart fuzzing* e gli strumenti appena descritti.

Innanzi tutto, si osservi che l'approccio *smart fuzzing* prevede l'utilizzo di tecniche di *data flow analysis* analoghe a quelle impiegate dai generatori *execution-oriented*. D'altra parte, però, tutti gli strumenti *path-oriented* suppongono di identificare staticamente i cammini che rispettano il criterio di test, rischiando così di considerare anche cammini non percorribili e riducendo quindi l'efficacia dell'analisi. Il modello *smart fuzzing*, invece, grazie alla ricostruzione delle *path condition* e all'integrazione con il risolutore di vincoli, è in grado di determinare dinamicamente la percorribilità di un cammino di esecuzione.

In secondo luogo, esistono diverse similitudini tra il procedimento adottato dai generatori *chain-oriented* e quanto previsto dal modello *smart fuzzing*. Per fare un esempio, entrambi considerano sia le *data dependency* che le *control dependency* per guidare il processo di esplorazione dei cammini di esecuzione.

Bisogna comunque ricordare che esiste una differenza fondamentale tra il problema considerato dal presente lavoro di tesi e quello affrontato dagli strumenti discussi in precedenza. Il modello *smart fuzzing* ha infatti come obiettivo l'esplorazione esaustiva degli stati dell'applicazione analizzata, secondo una strategia analoga a quanto previsto dai criteri di testing di tipo *path coverage*. I generatori di casi di test si prefiggono invece come obiettivo il raggiungimento di un particolare *elemento* del programma, sia questo un'istruzione o un arco di un CFG. Si tratta di una differenza spesso sottile, ma che porta allo sviluppo di tecniche di analisi piuttosto differenti. Si consideri ad

esempio come i generatori *chain-oriented* ricorrono alla valutazione delle *reaching definition* statiche associate ad un'istruzione condizionale per determinare una sequenza di assegnamenti tale da consentire di generare una configurazione degli input adatta a raggiungere l'elemento obiettivo. Un procedimento di questo tipo non è richiesto dal metodo *smart fuzzing*, in quanto il fine di quest'ultimo è riuscire a percorrere in modo sistematico ogni possibile cammino di esecuzione.

Capitolo 8

Conclusioni

Nel presente lavoro di tesi è stato presentato un modello per l'individuazione delle vulnerabilità presenti in un'applicazione software tramite l'impiego di tecniche di *program analysis* ibride statico-dinamiche. Il metodo proposto non richiede alcun intervento da parte dell'utente, compiendo ogni operazione in modo completamente automatico. Le analisi sono condotte interamente sul codice binario, potendo così trattare anche applicazioni per le quali non si dispone del codice sorgente ed essendo inoltre in grado di individuare problematiche di sicurezza legate a dettagli dipendenti dalla piattaforma su cui viene eseguito il programma in esame.

Il modello proposto prende il nome di *smart fuzzing*, in quando costituisce un'evoluzione degli strumenti di *fuzz testing* tradizionali. Invece che generare casi di test in modo completamente casuale, la metodologia proposta prevede di dedurre dinamicamente le condizioni sull'input associate ad un particolare cammino di esecuzione. Ciò permette, insieme ad un'attenta analisi del comportamento dell'applicazione, di determinare un nuovo insieme di dati in ingresso in grado di condurre l'esecuzione verso uno stato in cui è più probabile che una vulnerabilità si manifesti.

Al fine di incrementare l'efficienza del procedimento di analisi, sono state proposte alcune euristiche per la scelta dei cammini di esecuzione. Dato che spesso le procedure contenute nelle librerie standard sono particolarmente complesse da ottimizzare, sono state proposte alcune tecniche che permettono di sostituire tali funzioni con un ridotto insieme di istruzioni che ne riassumono il comportamento.

Nonostante il modello di analisi sia di tipo ibrido, i principali contributi del presente lavoro sono connessi alle metodologie di analisi dinamica. In particolare, costituiscono contributi originali le tecniche per la ricostruzione delle relazioni esistenti tra predicati condizionali e dati di input e per astrarre il comportamento di un ciclo in modo da poter individuare una possibile vulnerabilità senza dover iterare ogni possibile numero di volte. Inoltre, nonostante alcune delle tecniche di analisi adottate fossero già note in letteratura, la loro applicazione al codice binario ha spesso richiesto sostanziali modifiche.

Le tecniche di analisi discusse sono state implementate in un prototipo sperimentale, in grado di analizzare file eseguibili in formato ELF destinati a piattaforme Intel x86, senza contare sulla disponibilità di codice sorgente, simboli o altre informazioni di debugging. Alcune misurazioni preliminari condotte su una versione modificata dell'utility Linux `ls` hanno consentito di verificare l'efficacia della metodologia proposta.

Sono state evidenziate alcune problematiche relative alle tecniche di analisi trattate e all'efficienza del prototipo sviluppato. Per ciascuna di esse, sono state proposte delle possibili soluzioni che si ritiene possano costituire possibili sviluppi futuri del lavoro.

L'individuazione automatica delle vulnerabilità nei programmi informatici è un'area di ricerca ancora piuttosto giovane ed inesplorata. Da un punto di vista formale, dimostrare la correttezza di un'applicazione software è una questione notoriamente indecidibile, per cui tutte le metodologie di analisi che si pongono obiettivi analoghi a quelli del presente lavoro di tesi sono necessariamente costrette a scontrarsi con i limiti teorici del problema. Naturalmente la presenza del solo codice binario e l'assenza di qualunque informazione di alto livello non possono fare altro che introdurre ulteriori complicazioni. Il presente lavoro di tesi ha proposto alcune tecniche di analisi che, nonostante non siano in grado di risolvere completamente il problema, cercano comunque di sfruttare i recenti progressi nella *program analysis* per individuare le problematiche di sicurezza di un'applicazione. Nel complesso si ritiene che il modello *smart fuzzing* possa costituire 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] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, NY, USA, June 1990.

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

[4] D. Aitel. The Advantages of Block-Based Protocol Analysis for Security Testing. Technical report, Immunity Inc., February 2002.

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

[6] F. E. Allen. Control Flow Analysis. *SIGPLAN Notices*, 5:1–19, 1970.

[7] Amit Patel. Yet Another Python Parser System. [Online; ultimo accesso 12-Marzo-2007].

[8] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, Zurich, Switzerland, October 2005.

[9] G. Balakrishnan, T. W. Reps, and J. Lim. Intermediate-representation recovery from low-level code. In *Proceedings of the 2006 ACM SIGPLAN Workshop*

on Partial Evaluation and Semantics-based Program Manipulation, Charleston, South Carolina, USA, January 2006.

[10] T. Ball. The Concept of Dynamic Analysis. In *Proceedings European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC 1999)*, pages 216–234. Springer Verlag, Sept. 1999.

[11] C. W. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *CAV*, pages 515–518, 2004.

[12] D. L. Bird and C. U. Munoz. Automatic Generation of Random Self-checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[13] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM Press.

[14] C. Cadar and D. R. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *SPIN*, pages 2–23, 2005.

[15] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM Press.

[16] C. Cifuentes and M. V. Emmerik. Recovery of Jump Table Case Statements from Binary Code. *Science of Computer Programming*, 40(2–3):171–188, 2001.

[17] C. Cifuentes and K. J. Gough. Decompilation of Binary Programs. *Software—Practice and Experience*, 25(7):811–829, July 1995.

[18] C. Cifuentes and S. Sendall. Specifying the Semantics of Machine Instructions. In *6th International Workshop on Program Comprehension - IWPC*, pages 126–133, Ischia, Italy, June 1998. IEEE Computer Society.

[19] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate Theorem Proving for Program Verification. In *Proceedings of CAV 2005*, volume 576 of *Lecture Notes in Computer Science*, pages 296–300. Springer Verlag, 2005.

[20] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static Detection of Vulnerabilities in x86 Executables. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, December 2006.

[21] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[22] N. Davis and J. Mullaney. The Team Software Process (TSP) in Practice: A Summary of Recent Results. Technical report, Carnegie Mellon Software Engineering Institute, September 2003.

[23] S. Debray, W. Evans, R. Muth, and B. de Sutter. Compiler Techniques for Code Compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[24] B. Dutertre and L. de Moura. System Description: Yices 1.0. Technical report, Computer Science Laboratory, SRI International, 2006.

[25] N. Eén and N. Sörensson. An Extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[26] 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.

[27] M. D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.

[28] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[29] R. Ferguson and B. Korel. The Chaining Approach for Software Test Data Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):63–86, 1996.

[30] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[31] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. Technical report, University of Wisconsin, July 2000.

[32] GNU Project – Free Software Foundation. GCC: The GNU Compiler Collection, GCC Internals. [Online; ultimo accesso 14-Marzo-2007].

[33] GNU Project – Free Software Foundation. GDB: The GNU Project Debugger, GDB Internals. [Online; ultimo accesso 13-Marzo-2007].

[34] GNU Project – Free Software Foundation. GNU Binary Utilities. [Online; ultimo accesso 28-Febbraio-2007].

[35] GNU Project – Free Software Foundation. GNU C Library. [Online; ultimo accesso 28-Febbraio-2007].

[36] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[37] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 53–62, New York, NY, USA, 1998.

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

[39] L. C. Harris and B. P. Miller. Practical Analysis of Stripped Binary Code. In *SIGARCH Computer Architecture News*, volume 33, pages 63–68, New York, NY, USA, 2005. ACM Press.

[40] P. Havlak. Nesting of Reducible and Irreducible Loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, 1997.

[41] M. S. Hecht and J. D. Ullman. Characterizations of Reducible Flow Graphs. *Journal of the ACM (JACM)*, 21(3):367–375, 1974.

[42] M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[43] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM Press.

[44] iDefense Labs. FileFuzz. [Online; ultimo accesso 05-Marzo-2007].

[45] D. C. Ince. The Automatic Generation of Test Data. *The Computer Journal*, 30(1):63–69, 1987.

[46] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Nov. 2006. Instruction Set Reference.

[47] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Nov. 2006. Basic Architecture.

[48] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Nov. 2006. System Programming Guide.

[49] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[50] B. Korel. A Dynamic Approach of Test Data Generation. In *Proceedings of the International Conference on Software Maintenance*, pages 311–317, San Diego, CA, USA, Nov. 1990. IEEE Computer Society.

[51] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[52] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of USENIX Security 2004*, pages 255–270, San Diego, CA, August 2004.

[53] S. K. Lahiri and S. A. Seshia. The UCLID Decision Procedure. In *CAV'04: Proceedings of the 16th International Conference on Computer Aided Verification*, pages 475–478, 2004.

[54] W. Landi. Undecidability of Static Analysis. *LOPLAS*, 1(4):323–337, 1992.

[55] T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

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

[57] Linn and Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *SIGSAC: 10th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2003.

[58] S. B. Lipner. The Trustworthy Computing Security Development Lifecycle. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 2–13. IEEE Computer Society, 2004.

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

[60] B. P. Miller, G. Cooksey, and F. Moore. An Empirical Study of the Robustness of MacOS Applications Using Random Testing. In *RT '06: Proceedings of the 1st international workshop on Random testing*, pages 46–54, New York, NY, USA, 2006. ACM Press.

[61] B. P. Miller, L. Fredrikson, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Comm. of the ACM*, 33(12):32, December 1990.

[62] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical report, University of Wisconsin-Madison, April 1995.

[63] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1978.

[64] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[65] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Proceedings of the Third Workshop on Runtime Verification (RV '03)*, Boulder, Colorado, USA, July 2003.

[66] J. Newsome and D. Song. Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.

[67] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[68] M. Pietrek. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. *Microsoft Systems Journal*, 9(3), Mar. 1994.

[69] P. W. Purdom, Jr., and E. F. Moore. Immediate Predominators in a Directed Graph [H]. *Communications of the ACM*, 15(8):777–778, 1972.

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

[71] G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, Sept. 1994.

[72] G. Ramalingam. Identifying Loops in Almost Linear Time. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 21, 1999.

[73] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th Symposium on Principles of Programming Languages*, pages 12–27, New York, NY, USA, 1988.

[74] B. Schwarz, S. K. Debray, and G. R. Andrews. Disassembly of Executable Code Revisited. In *9th Working Conference on Reverse Engineering (WCRE)*, pages 45–54, Richmond, VA, USA, 2002. IEEE Computer Society.

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

[76] G. Snelting, T. Robschink, and J. Krinke. Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[77] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, 1996.

[78] N. Surribas. Wapiti – Web Application Vulnerability Scanner. [Online; ultimo accesso 05-Marzo-2007].

[79] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA*, June 2000.

[80] R. Tarjan. Testing Flow Graph Reducibility. In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 96–107, New York, NY, USA, 1973. ACM Press.

[81] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. May 1995.

[82] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Procedings of the London Mathematical Society*, 42(2):230–265, 1936.

[83] University of Oulu. PROTOS – security testing of protocol implementations. [Online; ultimo accesso 05-Marzo-2007].

[84] G. Vigna. *Malware Detection*, chapter Static Disassembly and Code Analysis. Advances in Information Security. Springer, 2007.

[85] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th International Conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

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

[87] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[88] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. R. Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *IEEE Symposium on Security and Privacy*, pages 243–257. IEEE Computer Society, 2006.

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