POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Analisi della complessità di tecniche di offuscamento

Relatore
Prof. Cataldo Basile

Corelatore:
Prof. Antonio Lioy

Laureando
Stefano Biora

Luglio 2019

Ringraziamenti

Vorrei ringraziare in modo speciale la mia famiglia: Paola Carla Negrisolo e Gianmario Biora. Mi hanno reso consapevole sull'importanza degli studi e mi hanno sempre incoraggiato e supportato in ogni momento.

Vorrei inoltre ringraziare Daniele Canavese e Leonardo Regano per avermi aiutato e consigliato nella stesura di questo documento.

Indice

Introduzione

Ogni anno che passa, il settore legato alla sicurezza informatica diventa sempre più importante. Difatti ogni giorno i sistemi informatici sono sottoposti a numerosi attacchi sempre più complicati ed articolati dovuti semplicemente a voler creare dei disservizi oppure per poter sottrarre informazioni ritenute preziose per le aziende o per privati [1]. Nel report generato dall'Associazione italiana per la Sicurezza Informatica (Clusit)[1] i dati analizzati mettono in risalto il notevole aumento degli attacchi informatici, avvenuti nell'anno 2017 rispetto agli anni precedenti: un aumento del 240% rispetto al 2011 ed un amento del 7% rispetto al 2016 [2].

Una soluzione legata alla protezione dei sistemi informatici che sta acquisendo una parte di fondamentale importanza riguarda la protezione del software [3]. Solitamente, quando si pensa ad un attacco informatico si tende a pensare che l'attaccante sia una persona esterna al sistema sottoposto ad attacco assumendo che le persone interne a tale sistema siano utenti fidati e in assenza di intenzioni malevoli. In questo scenario, assumendo appunto che gli attaccanti siano figure esterne, gli attacchi progettati potrebbero mirare a voler creare dei disservizi quali una fuga di dati ritenuti sensibili che potrebbero nuocere al business di una azienda, la divulgazione o la cancellazione di dati personali e privati relativi agli utenti della azienda sottoposta ad attacco, oppure la non raggiungibilità di un server per impedirgli di erogare i propri servizi (attacco DoS)[2]. Se, invece, assumessimo che gli attaccanti siano utenti in grado di accedere al sistema (MATE)[3], la tipologia di attacchi risulta completamente diversa. In questo scenario appena descritto la protezione del software ricopre un ruolo di notevole importanza, difatti un attaccante potrebbe:

- violare la proprietà intellettuale legata ad un parte di codice appropriandosi in maniera illecita degli algoritmi sviluppati;
- tramite il processo del *reverse engineering*, capire il funzionamento del codice;
- copiare e divulgare, senza alcuna autorizzazione, codici o parti di esso violando le leggi sulla copyright;
- manomettere e modificare una parte del codice affichè esso esegua istruzioni per cui non era stato progettato;
- usufruire a suo vantaggio di alcune vulnerabilità presenti nel codice.

Un soluzione adottata in merito alla protezione del software riguarda la tecnica dell'offuscamento di codice [4], sia esso sorgente o binario. Lo scopo principale degli offuscamenti risulta essere la modifica del codice o del flusso di esecuzione del codice stesso, mantenendo inalterato il comportamento originario, per aumentarne la complessità in modo da rendere più complicato capirlo. Esistono differenti tecniche di offuscamento:

- le più semplici permettono di modificare il codice in modo che esso risulti incomprensibile alla lettura di un attaccante umano evitando in tal modo che quest'ultimo sia in grado di

[1] https://clusit.it/.

[2] Denial of service.

[3] Man at the end

concepirne il flusso ed il funzionamento.Alcune di queste tecniche sono, ad esempio, l'introduzione nel codice di *junk code*, tecnica utilizzata far credere che il codice sia più complesso di quanto lo è realmente, aggiungendo parti di codice che non hanno alcun significato a livello funzionale oppure semplicemente applicando la tecnica del *renaming*, utilizzata per modificare il nome di determinate variabili affinché esso risulti privo di significato;

- altre tecniche di offuscamento risultano invece più complesse poiché mirano, modificando il flusso di esecuzione del programma, a rendere il codice incomprensibile non solo da un punto di vista umano ma anche dall'elaborazione automatica di particolari strumenti, chiamati deoffuscatori [5]. Questi strumenti, tramite tecniche di deoffuscamento, risultano in grado di effettuare un *reverse engineering* del codice offuscato permettendo di risalire al codice in chiaro o almeno ad una versione molto simile di esse, abbattendo in questo modo la protezione voluta e cercata [6].

Le tecniche di offuscamento più complesse introducono un *overhead*, occupando più memoria e consumando più banda. Risulta pertanto necessario effettuare degli studi approfonditi sul tipo di protezione che si vuole applicare affinché essa non incida sulle prestazioni del sistema. Inoltre bisogna anche analizzare l'*effort* richiesto da un attaccante per effettuare il *reverse engineering* sulla protezione applicata e risalire al codice in chiaro sfruttando cosi le eventuali vulnerabilità: una protezione che può essere abbattuta in pochi giorni non è da considerarsi una buona protezione [7].

Nell'ambito del *software engineering*, solitamente, per semplificare la gestione del software a lungo termine, si tende ad evitare di scrivere funzioni o pezzi di codice troppo complessi. Per identificare tali parti di codice, i programmatori tipicamente si affidano alle metriche di complessità: tali metriche forniscono una stima di quanto un codice risulti essere complesso [8]. Siccome gli offuscamenti, apportando modiche software, sono in grado di modificare la complessità associata al codice, le metriche di complessità risultano essere un possibile efficace metodo per identificare il grado di sicurezza apportato da un determinato offuscamento. Tali metriche infatti, rappresentano degli indici quantitativi e qualitativi in grado di rappresentare una stima di quanto un codice risulti complesso in termini di interazioni tra le diverse componenti che costituiscono il codice stesso e di fornire una descrizione oggettiva su determinati aspetti del codice analizzato quali possono essere ad esempio il numero di linee di codice, il numero di determinate istruzioni oppure il numero di percorsi linearmente indipendenti.

Con il termine asset si indica ogni genere di dato, dispositivo o altro componente che supporta attività legate alle informazioni. In particolare, nell'ambito della protezione del software, questo termine viene utilizzato per definire porzioni di codice contenenti dati ed informazioni ritenuti sensibili come possono essere ad esempio dati da considerarsi confidenziali, password o algoritmi proprietari. Generalmente tali asset sono soggetti a protezione, proprio a causa delle informazioni in essi contenuti: per cui se durante l'analisi del software si dovessero trovare zone di codice protette molto probabilmente queste risulterebbero essere asset. Se dallo studio dei valori delle metriche calcolate sul codice risultasse la presenza di determinati pattern ricorrenti, associati a particolari metriche di riferimento, in grado di indicare e porre in risalto porzioni di codice protette o il tipo di protezione applicata, ecco che un attaccante sarebbe in grado, in poco tempo, di concentrarsi sugli asset individuati oppure di identificare il livello di protezione che si era voluto applicare anzichè cercare particolari pattern guardando il codice disassemblato od il control flow graph [9] [10]. Tali valori che rappresentano un determinato pattern di codice ricorrente e che permettono di identificare determinati aspetti caratteristici propri di ogni sistema analizzato, in ambito informatico, vengono denominati *fingerprint* [11]. Lo studio delle *fingerprint* risulta quindi utile per poter analizzare sia i comportamenti introdotti da ogni tipo di offuscamento in riferimento alla variazione ed al valore assunto dalle metriche estratte sia per poter determinare se alcune di queste metriche risultano essere più significative di altre.

Nell'articolo "Towards Optimally Hiding Protected Assets in Software Applications" [11], le *fingerprint* sono state ampiamente discusse e dettagliate, ed è stato inoltre teorizzato che esse siano legate alle metriche. Nello studio effettuato, si è dimostrata empiricamente la presenza delle *fingerprint* estraendo le metriche dai programmi a disposizione ed osservando se davvero esse fossero in grado di identificare la presenza di pattern nel codice. Dai test effettuati, infatti, si è potuto analizzare e scoprire (come ci si attendeva) che ogni offuscamento riporta dei cambiamenti specifici e identificabili su determinate metriche di interesse. Tuttavia è anche emerso che alcuni

tipi di protezioni risultano più difficili da individuare rispetto ad altre: per alcune protezioni infatti le *fingerprint* risultano molto evidenti, in altri casi invece no. Dai test in oggetto risulta anche che non tutte le metriche rappresentano delle *fingerprint* di riferimento infatti ogni offuscamento influisce in modo diverso sul codice, in base al tipo di protezione ad esso associato.

Un'ulteriore tematica di importante rilievo quando si parla di protezione di codice riguarda la scelta della tipologia del miglior offuscamento da poter applicare per assicurare il maggior livello di sicurezza possibile. Ad oggi, questa analisi, viene effettuata richiedendo l'intervento di esperti del settore: tale procedimento viene svolto manualmente, richiede molta esperienza da parte degli esperti e risulta essere molto lungo. Per cercare di automatizzare ed accelerare lo studio sulla scelta della tipologia del miglior offuscamento si è ipotizzato di ricorrere ad un'ulteriore metrica: la potency [7], tramite la quale si potrebbe ricavare una stima del grado di sicurezza che ogni protezione è in grado di applicare. Per mezzo di tale metrica, calcolata a partire dalle metriche di complessità estratte dai codici analizzati, si riesce ad esprimere in maniera numerica un coefficiente in grado di rappresentare il livello di protezione apportato da un determinato offuscamento: maggiore è il valore di questo coefficiente, maggiore risulta il livello di protezione apportato dall'offuscamento scelto; minore risulta il valore di questo coefficiente, minore è il livello di protezione associato.

Siccome ogni metrica descrive diversi aspetti della complessità e della sicurezza sulla parte di codice analizzata, i test sulla *potency* sono stati eseguiti su metriche diverse: in questo modo si sono potuti condurre degli studi in grado di analizzare il livello di protezione applicato ad una funzione potendo osservare differenti aspetti legati al suo grado di sicurezza. Poiché è emerso che la *potency* risulta essere un ottimo indicatore, a seguito della scelta delle corrette metriche da prendere come riferimento per il calcolo, nel caso si voglia decidere quale offuscamento apportare si può ricorrere all'utilizzo di tale metrica per decidere quale offuscamento, tra quelli disponibili, si potrebbe applicare per soddisfare il livello di protezione necessario.

Inoltre, a seguito di numerosi interventi con gli esperti del progetto *ASPIRE*[4] è emerso che solamente due metriche, attualmente, risultano essere di notevole importanza per il calcolo della *potency*: la metrica di McCabe e le metriche di Halstead calcolate su file binari. Si è quindi cercato e provato a dimostrare la presenza di ulteriori *potency* (calcolate su metriche differenti) che potessero risultare altrettanto rilevanti mettendole a confronto con le *potency* calcolate sulle metriche riportate dagli esperti del progetto ASPIRE.

Nello studio riportato verranno dettagliatamente descritte ed analizzate alcune specifiche metriche di complessità e successivamente verranno affrontati gli offuscamenti ponendo l'attenzione sul tipo di protezione che ognuno di essi è in grado di apportare al codice soggetto a tale protezione. In seguito si dettaglieranno i processi e le metodologie per mezzo delle quali si è pensato di automatizzare sia l'applicazione di offuscamenti su un set di programmi da analizzare sia l'estrazione delle metriche su file in chiaro ed offuscati, che siano essi protetti o in chiaro. Infine, una volta raccolti tutti i dati, si riporteranno gli studi e le analisi eseguite sull'utilizzo e sul significato delle *fingerprint* e delle *potency*.

Organizzazione della tesi

Si riporta la struttura della tesi:

- Capitolo 1: vengono descritte le metriche software specificando la loro importanza, a cosa servono ed il loro significato. Vengono inoltre analizzati alcuni strumenti il cui scopo è estrarre le metriche sia da file sorgenti che da binari;
- Capitolo 2: viene fornita una definizione di offuscamento spiegandone obiettivo e scopo, ed alcuni di essi vengono analizzati nel dettaglio mostrando i cambiamenti che sono in grado di apportare nel codice. Infine vengono analizzati alcuni strumenti in grado di apportare tali offuscamenti sia su file sorgenti che su binari;

[4] https://aspire-fp7.eu/.

- Capitolo 3: viene fornita una definizione di benchmark suite e dell'utilizzo fatto in questo studio. Vengono infine descritti alcuni strumenti che mettono a disposizione tali suite;
- Capitolo 4: vengono enunciati i mitivi principali che hanno portato a questo studio e successivamente viene spiegata e dettagliata la progettazione ideata per l'implementazione di uno script in grado di automatizzare la raccolta dati;
- Capitolo 5: viene descritta ed analizzata l'implementazione dello script sviluppato riportandone una descrizione ad alto livello in grado di spiegare concettualmente come comunicano tra di loro le varie parti dello script;
- Capitolo 6: si descrivono i procedimenti con i quali si è deciso di analizzare i dati estratti e raccolti per mezzo dello strumento creato;
- Capitolo 7: vengono mostrati e discussi i risultati dei test eseguiti soffermandosi sull'importanza dell'uso delle fingerprint e della potency;
- Capitolo 8: vengono riportate le osservazioni conclusive ed alcune proposte di possibili progetti e sviluppi futuri relativi allo studio effettuato;
- Appendice A: viene spiegato come compilare il file delle annotazioni necessario allo strumento Diablo per apportare l'offuscamento selezionato ed estrarre le metriche sulle sezioni di codice desiderate;
- Appendice B: vengono descritti il manuale utente ed il manuale sviluppatore dello script sviluppato. Inoltre vengono anche riportati i manuali programmatore relativi agli script implementati per l'analisi dei dati;
- Appendice C: viene fornita una spiegazione a basso livello di come è stato implementato lo script utilizzato.
- Appendice D: vengono riportati tutti i grafici generati ed analizzati utilizzati per l'identificazione delle fingerprint;
- Appendice E: vengono riportati tutti i grafici generati ed analizzati riguardo alla ricerca di nuove metriche utili al calcolo della potency.

Capitolo 1

Metriche software

L'obiettivo di questo capitolo è fornire una definizione di metriche per far capire cosa esse siano e di spiegare la loro importanza nell'implementazione del codice. Si riporta anche una lista di strumenti di estrazione in grado di estrarre metriche sia da file sorgenti sia da file binari, elencandone vantaggi e limitazioni.

1.1 Definizione

Una metrica è un valore numerico con il quale si può stimare la misura di determinate caratteristiche o proprietà software fornendo una rappresentazione quantitativa di ciò che si desidera osservare.

Nel nostro caso ci concentreremo sia su metriche software, che metriche di complessità. Le metriche software ci serviranno per avere una proiezione numerica di determinate proprietà del software analizzato. Esempi di metriche sono il numero di linee di codice, il numero di commenti, il numero di speciche istruzioni condizionali. Le metriche di complessità servono per ottenere una stima di quanto sia complesso, in termini di interazioni tra entita, il codice che sottoporremo al nostro studio. Esempi di metriche di complessità sono la complessità ciclomatica e le metriche di complessità di Halstead.

Il calcolo delle metriche può essere apportato a livelli differenti: all'intero programma, alle classi, alle funzioni.

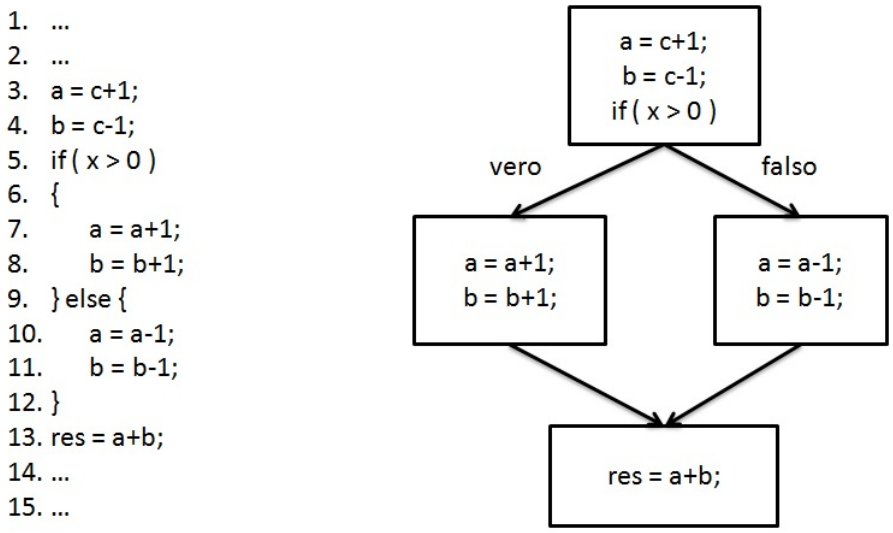
1.1.1 Control Flow Graph

Molte metriche di complessità, descritte in questo capitolo, usate da molti offuscamenti (Capitolo 2) si basano sull'utilizzo, la modifica, ed il controllo del Control Flow Graph (CFG).

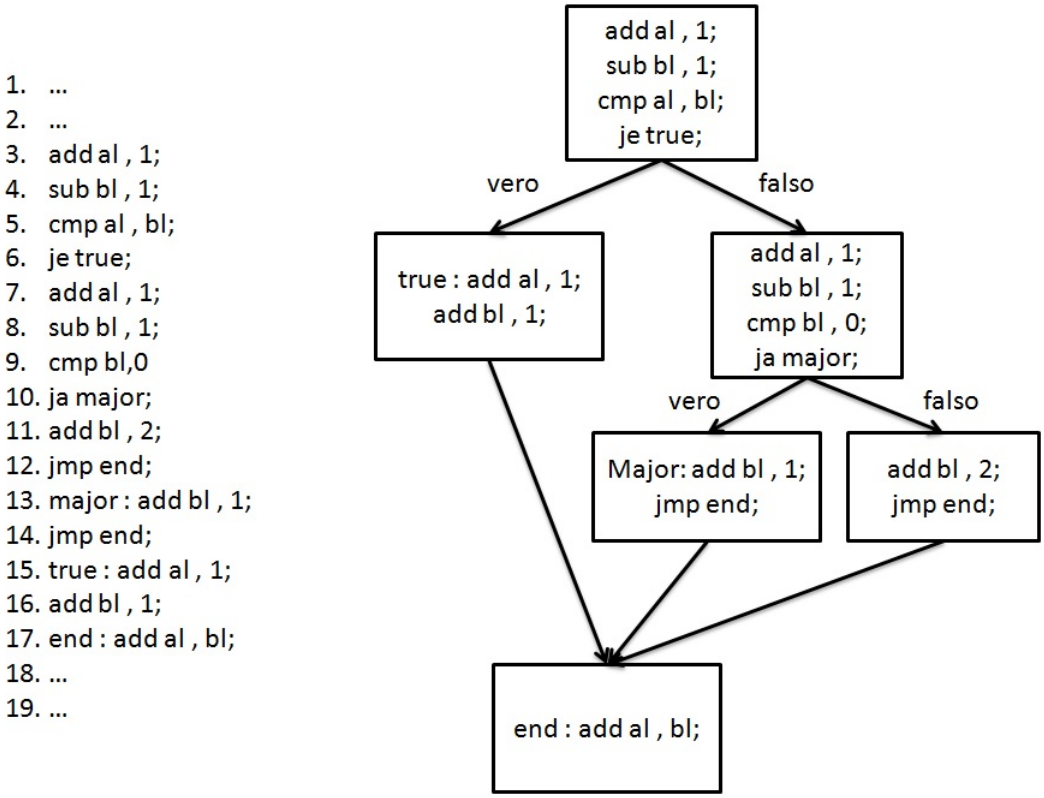
Un Control Flow Graph (CFG) è una rappresentazione grafica del flusso di controllo di un programma tramite l'analisi di tutti i percorsi possibili che possono essere effettuati durante la sua esecuzione [9] [10]. Gli elementi di base di un CFG sono due:

- nodi: rappresentano i basic block, ovvero sequenze di codice consecutivo senza alcuna ramificazione presente, fatta eccezione per le ramificazioni in entrata e in uscita [12];
- archi: rappresentano il possibile flusso di controllo tra la fine di un nodo e l'inizio di un altro.

La Figura 1.1 mostra alcuni esempi di Control Flow Graph, in particolare la Figura 1.1a mostra il codice ed il CFG ad esso relativo per una parte di programma scritta nel linguaggio C mentre la Figura 1.1b mostra il codice ed il CFG per una parte di programma scritta nel linguaggio assembly.



(a) Esempio di Control Flow Graph per codice C.



(b) Esempio di Control Flow Graph per codice assembly.

Figura 1.1: Esempi di Control Flow Graph.

Dato un grafo G [13], composto da due sottoinsiemi (V,A) in cui V rappresenta l'insieme di tutti i nodi del grafo (ad esempio V={a, b, c,d}) ed A rappresenta l'insieme di tutti gli archi (ad esempio V={(a,b), (a,c), (b,d), (c,d)}), si riportano le seguenti principali definizioni:

- grafo orientato: rappresentato da coppie di nodi ordinate, per cui la coppia (a,b) risulta diversa dalla coppia (b,a);

- grafo non orientato: rappresentato da coppie di nodi non ordinate, per cui la coppia (a,b) risulta uguale dalla coppia (b,a);
- cammino: sottoinsieme di nodi ed archi per mezzo dei quali è possibile arrivare ad un nodo b partendo da un nodo a;
- ciclo: sottoinsieme di nodi e di archi per mezzo dei quali è possibile partendo da un nodo a ritornare allo stesso nodo a;
- grafo connesso: se per ogni coppia di nodi (u,v) appartenenti all'insieme V esiste un cammino che collega u a v;
- componente fortemente connessa: in un grafo orientato un nodo u è fortemente connesso ad un nodo v se il nodo u è raggiungibile da v e v è raggiungibile da u. Si chiama componente fortemente connessa un qualsiasi sottoinsieme di nodi ed archi tali per cui ogni nodo di questo sottoinsieme è mutualmente raggiungibile da qualsiasi altro nodo di quello stesso sottoinsieme;
- arco diretto: arco orientato;
- arco indiretto: arco non orientato.

1.1.2 Complessità ciclomatica

La complessità ciclomatica o complessità condizionale è stata sviluppata da Thomas J. McCabe nel 1976 ed è una metrica software utilizzata per misurare il numero di cammini linearmente indipendenti partendo dal Control Flow Graph (CFG) di un file sorgente [14].

Matematicamente, la complessità ciclomatica è definita dalla Formula 1.1

$$v(G) = e - n + 2p \tag{1.1}$$

dove:

- $v(G)$ rappresenta la complessità ciclomatica del CFG;
- e rappresenta il numero di archi del grafo;
- n rappresenta il numero di nodi del grafo;
- p rappresenta il numero di componenti connesse[1].

Prendiamo la Figura 1.2 che mostra un esempio di un possibile CFG.

- Il numero di archi è $e = 10$.
- Il numero di nodi è $n = 9$.
- Il numero di componenti connesse è $p = 1$.

Dalla Formula 1.1 mostrata in precedenza possiamo quindi risalire alla complessità ciclomatica del grafo dato:

$$v(G) = 10 - 9 + 2 \cdot 1 = 3$$

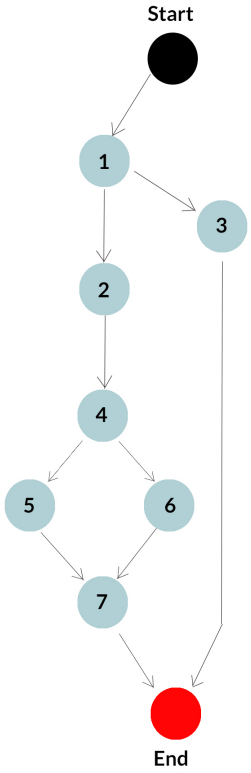
Il numero 3 indica quindi il numero di percorsi linearmente indipendenti che possiamo facilmente verificare in quanto:

1. (Start, 1, 2, 4, 5, 7, End);

[1] Numero di sottografi in cui ogni coppia di vertici v, u è connessa da un cammino.

```
int foo(int i){
        if (i>10){
                i=i*10;
                if(i>500){
                        i=i/2;
                }else{
                        i=i/3;
                }
                i=i*i:
        }else{
                i=0;
        }
        return i;
}
```

(a) Programma scritto in C.



(b) Control Flow Graph.

Figura 1.2: Esempio di CFG.

2. (Start, 1, 2, 4, 6, 7, End);
3. (Start, 1, 3, End).

Nel qual caso invece che il grafo fosse un grafo fortemente connesso[2] (cfr. Sezione 1.1.1), la complessità ciclomatica sarà così calcolata:

$$v(G) = e - n + p$$

Se il codice analizzato non dovesse avere nessuna istruzione condizionale come *IF, FOR, WHILE* la complessità ciclomatica del grafo sarà pari a 1, ovvero esisterà un solo cammino linearmente indipendente.

1.1.3 Metriche di Halstead

Queste metriche sono state introdotte da Maurice Howard Halstead nel 1977 con l'intento di identificare proprietà misurabili del software e di ricavare relazioni tra di esse [15]. Queste metriche possono essere calcolate sia su interi programmi sia su metodi, funzioni, moduli o classi del programma preso in considerazione. Bisogna prima calcolare quattro valori:

- η_1: numero di operandi distinti;
- η_2: numero di operatori distinti;
- N_1: numero totale di operandi;

[2]CFG che contiene una sola componente fortemente connessa.

- N_2: numero totale di operatori.

Dopodichè si possono calcolare le seguenti metriche:

- *vocabolario del programma*: somma del numero di operatori ed operandi distinti, $\eta = \eta_1 + \eta_2$;
- *lunghezza del programma*: somma del numero di operatori ed operandi totali, $N = N_1 + N_2$;
- *lunghezza del programma calcolata*: calcolo della *lunghezza del programma* senza conoscere il numero totale di operandi N_1 ed il numero totale di operatori N_2, $\hat{\mathrm{N}} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$;
- *volume*: rappresenta la dimensione dell'implementazione di un algoritmo misurando in bit il contenuto delle informazioni del programma $V = N \times \log_2 \eta$;
- *difficoltà*: rappresenta una stima della difficoltà nell'implementare la parte di programma analizzata, $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$;
- *effort*: rappresenta una stima dello sforzo richiesto per implementare la parte di programma analizzata, $E = D \times V$;
- *Tempo richiesto per programmare*: rappresenta una stima del tempo necessario per implementare la parte di programma analizzata, $T = \frac{E}{18} secondi$;
- *numero di bug*: rappresenta una stima del numero di errori nell'implementazione della parte di programma analizzata, $B = \frac{E^{\frac{2}{3}}}{3000}$ oppure più facilmente $B = \frac{V}{3000}$.

Prendiamo come esempio la funzione riportata nella Figura 1.3. Procediamo adesso col

```
int foo(){
        int a,b,sum;
        a=1;
        b=1;
        sum=a+b;
        print("Sum = %d",sum);
        return;
}
```

Figura 1.3: Esempio di funzione.

calcolare i valori di η_1, η_2, N_1, N_2:

- $\eta_1 = 10$ {foo, (), , ';', ',', int, =, +, print, return}
- $\eta_2 = 5$ {a, b, sum, 1, 'Sum = %d'}
- $N_1 = 21$
- $N_2 = 12$

Avendo ora questi quattro elementi a disposizione, possiamo procedere col calcolo delle metriche:

- *vocabolario del programma*: $\eta = 10 + 5 = 15$;
- *lunghezza del programma*: $N = 21 + 12 = 33$;
- *lunghezza del programma calcolata*: $\hat{\mathrm{N}} = 10 \log_2 10 + 5 \log_2 5 = 44{,}8$;
- *volume*: $V = 33 \times \log_2 15 = 128{,}7$;
- *difficoltà*: $D = \frac{10}{2} \times \frac{12}{5} = 12$;
- *effort*: $E = 12 \times 128{,}7 = 1544{,}4$;
- *tempo richiesto per programmare*: $T = \frac{1544{,}4}{18}\mathrm{s} = 85{,}8\mathrm{s}$;
- *numero di bug*:$B = \frac{128{,}7}{3000=0{,}0429}$.

1.2 Strumenti di estrazione delle metriche

In rete è possibile trovare svariati tool, sia commerciali sia non. Questi strumenti, dato un file sorgente od un file binario, estraggono le metriche software.

1.2.1 Frama-c

Frama-c[3] è una piattaforma open source dedicata all'analisi del codice sorgente implementato con il linguaggio C. È disponibile su Windows e su Unix con diverse versioni scaricabili. Tra le varie caratteristiche messe a disposizione, permette di ottenere un report sulle metriche software sia a livello di intero programma sia a livello di singole funzioni. Le metriche che estrae sono:

- numero di linee di codice;
- numero di punti decisionali come per esempio `if`, `for`, `while`;
- numero di varibili globali;
- numero di istruzioni `if`;
- numero di cicli `loop`;
- numero di istruzioni `goto`;
- numero di assegnazioni;
- numero di `exit point`;
- numero di funzioni contenute nel programma[4];
- numero di funzioni chiamate;
- numero di *pointer dereferencing*[5];
- valore della complessità ciclomatica.

1.2.2 cccc

cccc[6] è un tool gratuito disponibile sia per Windows che per Unix per l'estrazione di metriche software. È stato progettato per analizzare file sorgenti scritti nel linguaggio C, C++ e Java. L'ultima versione è stata rilasciata nel 2006. Le metriche che calcola sono:

- numero di linee di codice;
- valore della complessità ciclomatica;
- numero di linee di commento;
- rapporto tra numero di linee di codice e numero di linee di commento;
- rapporto tra il valore della complessità ciclomatica e il numero di linee di commento;
- numero di metodi per classe;
- numero di metodi pubblici per classe;

[3] https://frama-c.com/.

[4] Nel caso si analizzasse il programma a livello di funzione, tale parametro ritorna il valore 1

[5] Il numero di puntatori utilizzati all'intero della funzione.

[6] http://cccc.sourceforge.net/.

- profondità dell'albero di ereditarietà delle classi;
- numero di sottoclassi;
- numero di accoppiamenti tra oggetti, ovvero il numero di altri moduli, siano essi client o supplier, che risultano accoppiati al modulo corrente;
- misura del flusso di informazioni tra differenti moduli[7]

1.2.3 OClint

OClint[8] è un tool open source che si basa sull'analisi statica del codice per migliorare e ridurre i difetti del codice sorgente scritto nei linguaggi C e C++. Sono disponibili diverse versioni ed è utilizzabile solamente sulle piattaforme Linux e macOS X. Le metriche calcolate sono:

- valore della complessità ciclomatica;
- numero di percorsi di esecuzione aciclici in un determinato metodo;
- numero di tutte le istruzioni escludendo i commenti, le istruzioni vuote, i blocchi vuoti, le parentesi graffe o i punti e virgola dopo la chiusura delle parentesi;
- numero di livelli di nidificazione.

Le metriche riportate vengono utilizzate dal programma per verificare che il loro valore non superi il valore di una soglia, impostabile per ogni metrica, dall'utente: se una metricha dovesse violare tale regola allora sul report generato verrà segnalato un errore mostrando il valore assunto dalla metrica in questione.

1.2.4 SourceMeter

SourceMeter[9] è un tool per l'analisi statica del codice sorgente per differenti linguaggi di programmazione: C/C++, Java, C#, Python e progetti RPG[10]. In base al tipo di caratteristiche del quale si vuole usufruire sono scaricabili una versione gratuita e due a pagamento. Disponibile sia per Windows che per Linux. Le metriche calcolate sono suddivise in sei categorie:

- *Cohesion*: misura l'estensione della coerenza tra gli elementi del codice sorgente;
- *Complexity*: misura la complessità di determinati elementi del codice sorgente;
- *Coupling*: misura la quantità di interdipendenze tra gli elementi del codice sorgente;
- *Documentation*: misura la quantità di commenti e documentazione degli elementi del codice sorgente;
- *Inheritance*: misura i diversi aspetti della gerarchia di ereditarietà;
- *Size*: misura le proprietà di base del sistema analizzato in termini di diverse cardinalità come il numero di righe di codice, il numero di classi o il numero di metodi.

[7] Metrica calcolata in funzione del fan-in e fan-out rispetto ai moduli presi in considerazione.

[8] http://oclint.org/.

[9] https://www.sourcemeter.com/.

[10] Role-playing game.

1.2.5 CMT++

CMT++[11], Complexity Measurement Tool, è un tool per l'analisi statica del codice sorgente per i linguaggi di programmazzione C, C++ e C#. È disponibile sia per Windows per Linux. Permette di calcolare le seguenti metriche:

- valore della complessità ciclomatica;
- massimo valore della complessità ciclomatica;
- media del valore della complessità ciclomatica;
- numero di linee fisiche;
- numero di linee di programma, escluse linee vuote o di commento;
- numero di linee vuote;
- numero di linee di commento;
- lunghezza di Halstead;
- numero di operatori;
- numero di operandi;
- dimensione del vocabolario;
- numero di operatori unici;
- numero di operandi unici;
- volume del programma;
- numero di bug;
- difficoltà;
- effort;
- livello di programma;
- tempo di implementazione;
- indice di manutenibilità[12] senza commenti;
- indice di manutenibilità12 con commenti;
- indice di manutenibilità12 totale.

1.2.6 CppDepend

CppDepend[13] è un tool gratuito per l'analisi statica del codice sorgente per misurare quantitativamente la qualità del codice implementato con i linguaggi di programmazione C e C++. Disponibile solamente per Windows. Permette di calcolare 82 metriche dividendole in sette categorie *Metrics on Application*, *Metrics on Projects*, *Metrics on Namespaces*, *Metrics on Types*, *Metrics on Methods*, *Metrics on Fields*, *Halstead Metrics*. Tra le più importanti ricordiamo:

[11] http://www.testwell.fi/cmtdesc.html.

[12] È una metrica software per misurare quanto il codice sorgente sia facile da modificare e supportare.

[13] https://www.cppdepend.com/.

- il numero di linee di codice;
- il numero di linee di commenti;
- la percentuale di commenti definita dalla formula

$$100 \cdot \text{NbLinesOfComment}/(\text{NbLinesOfComment} + \text{NbLinesOfCode});$$

- il numero di namespace;
- il numero di tipi di dato che possono essere classi astratte o concrete, `struct`, `enumeration`;
- il numero di metodi;
- il numero di campi;
- il valore della complessità ciclomatica;
- il numero totale di operatori N_1;
- il numero totale di operandi N_2;
- il numero totale di operatori distinti η_1;
- il numero totale di operandi distinti η_2;
- la lunghezza del programma definita dalla formula $N = N_1 + N_2$;
- il volume del programma definito dalla formula $V = N * log2(|eta_1 + \eta_2)$;
- il livello del programma definito dalla formula $L = (2/\eta_1) * (\eta_2/N_2)$;
- la difficoltà del programma definita dalla formula $D = (\eta_1/2) * (N_2/\eta_2)$;
- l'effort del programma definito dalla formula $E = D * V$;
- il tempo di implementazione del programma definito dalla formula $T = E/18s$;

1.2.7 AdLint

AdLint[14] è un tool open source, gratuito, per l'analisi statica del codice sorgente. È disponibile per diverse piattaforme quali Windows, macOS X e GNU/Linux. Le metriche calcolate sono le seguenti:

- numero di istruzioni;
- numero di funzioni;
- numero di istruzioni non raggiungibili;
- numero di linee;
- numero di parametri;
- numero di variabili non usate;
- numero di posizione della funzione chiamata;
- numero di posizione della chiamata dalla funzione;
- numero di istuzioni `goto`;
- numero di `return` in una funzione;

[14] http://adlint.sourceforge.net/pmwiki/upload.d/Main/users_guide_en.html.

- numero di istruzioni `if` senza clausola `else`;
- il livello massimo di profondità della struttura di controllo di una funzione;
- il numero di possibili percorsi di esecuzione di una funzione;
- valore della complessità ciclomatica.

1.2.8 Diablo

Diablo[15] è un framework di riscrittura binaria utilizzato per diversi scopi tra cui l'analisi statica del codice binario generato. Disponibile per ambiente Unix. Le metriche sono:

- numero di istruzioni assembly;
- numero di operandi sorgente[16];
- numero di operandi destinazione[17];
- lunghezza di Halstead;
- numero ramificazioni del Control Flow Graph (CFG);
- numero di ramificazioni indirette: l'indirizzo di destinazione non è noto staticamente ma è definito run-time da calcoli effettuati durante l'esecuzione del programma. Ad esempio su codice assembler:
  - una ramificazione diretta è data dall'istruzione `jump 0x00a0`;
  - una ramificazione indiretta può essere rappresentata dalla serie di istruzioni:

```
mov ah, 0x00;
mov al, 0xa0;
jump ax;
```

- valore della complessità ciclomatica.

1.3 Confronto e risultati

La Tabella 1.1 mostra un riassunto[18] degli strumenti di estrazione delle metriche riportati nella Sezione 1.2.

Nella scelta degli strumenti di estrazione, si è deciso di concentrarsi su strumenti che fossero in grado di rispettare le seguenti condizioni:

- operare su ambiente Unix;
- essere open source;
- analizzare file *.c*;
- estrarre alcune metriche significative tra cui la complessità ciclomatica ed le metriche di Halstead.

[15] https://github.com/csl-ugent/diablo.

[16] Gli operandi presi come input per un'istruzione

[17] Gli operandi presi come output per un'istruzione

[18] La complessità ciclomatica è stata abbreviata con la scritta *CC*.

Strumenti di estrazione	Ambiente	Codice sorgente/ binario	Open source	N° metriche calcolate	File .c	CC	Metriche di Halstead
Frama-c	Window Unix	sorgente	si	12	si	si	si
cccc	Window Unix	sorgente	si	13	si	si	no
OClint	Unix macOS X	sorgente	si	4	si	si	no
SourceMeter	Window Linux	sorgente	no	divise in 6 categorie	si	si	no
CMT++	Window Linux	sorgente	no	22	si	si	si
CppDepend	Window	sorgente	no	82	si	si	si
AdLint	Window Linux GNU/Linux	sorgente	si	14	si	si	no
Diablo	Unix	binario	si	7	no	si	si

Tabella 1.1: Riassunto degli strumenti di estrazione metriche analizzati.

Inoltre si è anche cercato di porre l'attenzione su strumenti in grado di estrarre metriche sia su file sorgenti che su file binari.

Dati quindi questi requisiti di partenza, dopo un'attenta ed accurata analisi, la scelta finale è ricaduta su due strumenti:

- `Frama-c` 1.2.1. Estrae le metriche su file sorgenti. Per eseguire *Frama-c*, lanciare da terminale il seguente comando: `frama-c -metrics -metrics-by-function input_file > output_file`. Un esempio del file di output è riportato nella Figura 1.4:

```
Stats for function <Path_to_file/name_function>
================================================================================
Sloc = 127
Decision point = 18
Global variables = 0
If = 17
Loop = 9
Goto = 2
Assignment = 42
Exit point = 1
Function = 1
Function call = 28
Pointer dereferencing = 11
Cyclomatic complexity = 19
```

Figura 1.4: Esempio file di output di Frama-c .

- `Diablo` 1.2.8. Estrae le metriche da file binari. Per eseguire *Diablo* lanciare da terminale il seguente comando: `path_to_diablo-obfuscator -Z off -S --annotation-file annotations.json -O . -o output_file input_file`. Si riporta nella Figura 1.5 un esempio del file di output generato da *Diablo* contenente le metriche calcolate. Nell'Appendice A vengono riportati maggiori dettagli in merito al file delle annotazioni denominato come *annotations.json*.

```
#region_idx,nr_ins_static,nr_src_oper_static,nr_dst_oper_static,
    halstead_program_size_static,nr_edges_static,nr_indirect_edges_CFIM_static
    ,cyclomatic_complexity_static
1, 14, 15, 18, 47, 3, 0, 3,
2, 69, 80, 79, 228, 11, 0, 6,
3, 42, 58, 46, 146, 0, 0, 2,
4, 27, 28, 33, 88, 6, 0, 5,
5, 17, 12, 21, 50, 0, 0, 4,
6, 38, 34, 26, 98, 0, 0, 2,
```

Figura 1.5: Esempio file di output di Diablo .

Capitolo 2

Offuscamento del codice

Questo capitolo tratta di tecniche di offuscamento, soluzioni utilizzate per proteggere il proprio codice. In particolare verranno analizzate alcune tra le molte tecniche di offuscamento e verranno introdotte delle formule matematiche il cui scopo servirà per valutare il livello di sicurezza apportato dalla tecnica di offuscamento scelta. Infine verranno riportati e confrontati vari strumenti di offuscamento, elencandone pregi e difetti.

2.1 Definizione

Nell'ambito della sicurezza informatica con il termine *offuscamento* si intende la modifica del codice sorgente o binario affinché la comprensione dello stesso risulti ardua o molto complessa sia per un lettore umano sia per gli strumenti di *deoffuscamento*[1] [5] [6]. In questo contesto si userà il termine *protezione* come sinonimo di *offuscamento*, anche se in generale non è propriamente corretto in quanto esistono altre tecniche di protezione che non offuscano il codice, per esempio l'*anti-debugging* che impedisce l'uso dei debugger. Collberg fornisce una definizione oggettiva sul cosa sia una trasformazione offuscante: data una trasformazione τ, un codice sorgente P e il codice sorgente del programma trasformato P' come mostrato nella Formula 2.1

$$P \xrightarrow{\tau} P' \tag{2.1}$$

si dice che τ sia una trasformazione offuscante se P e P' hanno lo stesso comportamento [7]. In particolare:

- se P fallisce nel terminare oppure termina con delle condizioni di errore, allora P' potrebbe o non potrebbe terminare;
- altrimenti P' dovrà terminare e produrre lo stesso risultato di P.

L'operazione di offuscamento del codice può essere applicata o manualmente dal programmatore stesso oppure mediante l'utilizzo di strumenti appositi di offuscamento. Inoltre non è necessario proteggere sempre tutto il codice. Spesso si utilizza offuscare una parte di esso, che sia una funzione o un'insieme di istruzioni, più soggetta a vulnerabilità e ad azioni malevole da parte di terzi con lo scopo di compromettere il corretto funzionamento del programma stesso o di appropiarsi di dati che dovrebbero rimanere segreti.

Possono esserci diverse ragioni per cui applicare una protezione al codice, tra cui:

[1] Strumenti in grado di applicare l'operazione inversa dell'*offuscamento*, ovvero dato un codice offuscato riportare il codice in chiaro senza alcuna protezione.

- proteggere la *proprietà intellettuale*, quindi proteggere le idee avute e realizzate, rendendo più complicato il *reverse engineering* col fine di evitare l'utilizzo del codice implementato da terze parti non autorizzate;
- rendere complicata la modifica del codice con intenzioni malevole quali ad esempio l'inserimento di malware.

2.2 Tecniche di offuscamento

Esistono differenti tecniche di offuscamento da poter utilizzare per proteggere il codice [4]. Alcune di queste sono utilizzate semplicemente per rendere più complicata la comprensione del codice da parte di un lettore umano:

- il *renaming delle variabili*, che consiste nel rinominare le variabili utilizzate per evitare che i loro nomi originali aiutino il lettore nella comprensione del codice che sta analizzando;
- l'introduzione di *junk code*, ovvero l'introduzione di codice *"spazzatura"* che non altera il comportamento del programma, ma che potrebbe far perdere molto tempo nella sua comprensione in quanto consiste nell'introduzione di codice fittizio, che non ha alcuno scopo ed aumenta solo il numero di linee di codice, con il fine di nascondere la logica di esecuzione del programma stesso.

Soluzioni come quelle appena riportate, non sono utili contro strumenti di deoffuscamento, i quali sono in grado di applicare il *reverse engineering* su tali protezioni ed in poco tempo risalire al codice sorgente originale. Esistono tuttavia soluzioni che possono offrire un maggiore livello di protezione contro i deoffuscatori e contro la lettura da parte di un lettore umano. Possiamo suddividere questi offuscamenti in due gruppi:

- trasformazioni del control flow graph (cfr. Sezione 1.1.1): unendo basic block indipendenti tra di loro, dividendo basic block che dovrebbero essere logicalmente connessi, modificando l'ordine dei blocchi logici da eseguire ottenendo però sempre lo stesso comportamento del codice originale;
- trasformazioni dei dati, che offuscano la struttura stessa del dato da proteggere, per esempio cambiandone la codifica (esadecimale, binario, base64), dividendo il contenuto di una variabile in più variabili oppure riordinando il contenuto di un vettore.

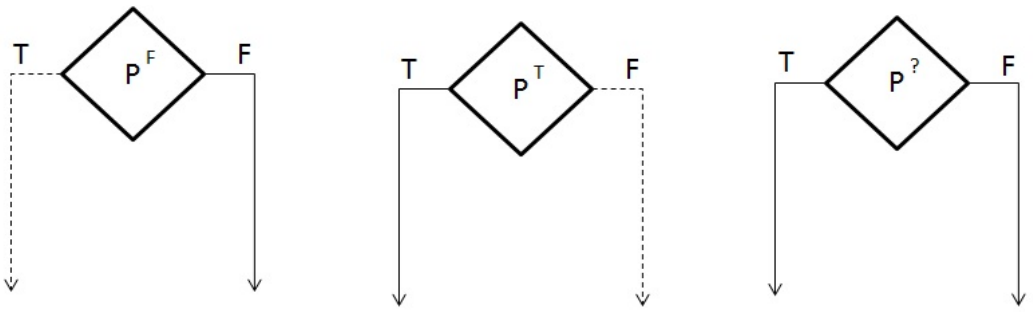
2.2.1 Predicati opachi

I *predicati opachi* appartengono al gruppo delle trasformazioni del control flow graph [7] [16]. Un predicato è un'espressione booleana e si definisce opaco se un deoffuscatore può calcolare con molta difficoltà il suo valore mentre quest'ultimo è noto all'offuscatore. Allo stesso modo si può definire il significato di una variabile opaca, ovvero una variabile che ha determinate proprietà conosciute a priori dall'offuscatore ma che sono difficili da dedurre da un deoffuscatore.

Come riportato da Collberg [7], una variabile V si dice opaca se ha una proprietà q che è conosciuta al momento dell'offuscamento e si indicherà con V^{q}. Se tale proprietà è valida solo in un determinato punto del programma, che indicheremo con p, allora la variabile si indicherà con V_p^{q}. Un predicato P si dice opaco se il suo esito è noto al momento dell'offuscamento. Si indicherà con P^{T} se tale predicato sarà sempre valutato con *true*, con P^{F} se sarà sempre valutato con *false*, con $P^{?}$ se il valore assunto dal predicato può essere sia *true* che *false*. Se il valore del predicato è legato ad un determinato punto del programma, allora si userà la notazione P_p.

La Figura 2.1 mostra degli esempi di costrutti opachi ovvero costrutti condizionali[2] la cui condizione si basa sull'uso di predicati opachi:

[2]Blocchi condizionali quali `if`, `while`, `switch`.



(a) (b) (c)

Figura 2.1: Esempi di predicati opachi.

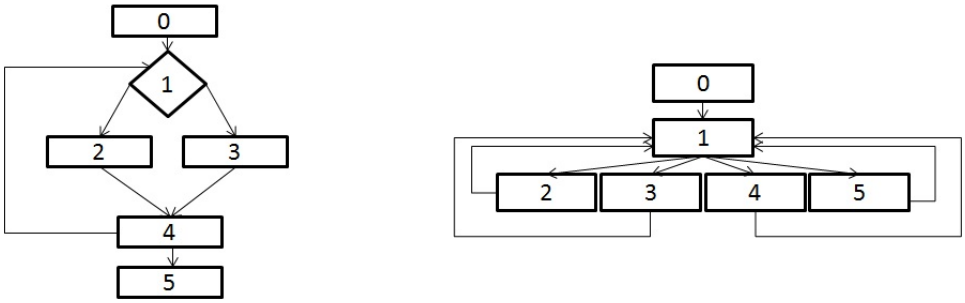
- la Figura 2.1a mostra un costrutto opaco il cui corrispondente predicato opaco sarà sempre valutato come *true*;
- la Figura 2.1b mostra un costrutto opaco il cui corrispondente predicato opaco sarà sempre valutato come *false*;
- la Figura 2.1c mostra un costrutto opaco il cui corrispondente predicato opaco potrà essere valutato sia come *true* sia come *false*.

Le linee solide indicano il percorso che potrebbe essere eseguito mentre le linee tratteggiate indicano il percorso che non sarà mai scelto.

2.2.2 Control flow flattening

La trasformazione *control flow flattening* appartiene al gruppo delle trasformazioni del control flow graph [7] [17]. Questo tipo di offuscamento *appiattisce* il control flow del programma che si vuole proteggere rendendo più complicata la comprensione del susseguirsi dei basic block (cfr. Sezione 1.1.1). L'operazione del *flattening* spesso viene utilizzata inserendo un ciclo infinito tramite l'istruzione `switch` per controllare il corretto flusso di esecuzione del programma.

Osservando la Figura 2.2a, si può notare come dal control flow graph del codice originale si possa facilmente dedurre che i blocchi numero 2 e numero 4 seguano il blocco logico numero 1, alquanto più complicato giungere alla medesima conclusione osservando il control flow graph del programma protetto in Figura 2.2b poiché non sembra esserci una corrispondenza diretta.



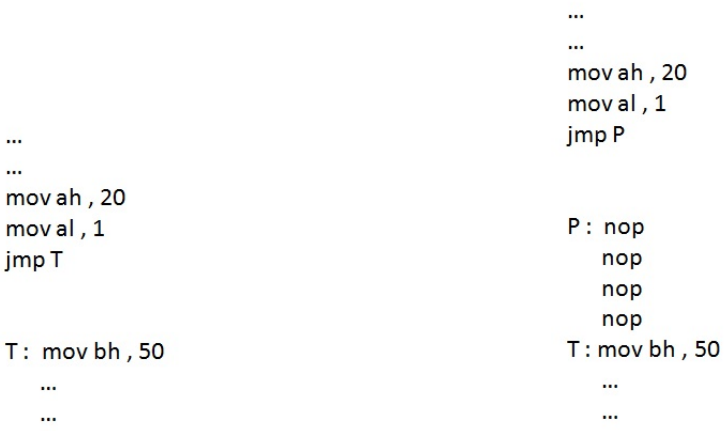
(a) Programma in chiaro (b) Programma offuscato

Figura 2.2: Esempio di control flow flattening.

2.2.3 Anti branch analysis

L'*anti-branch analysis* appartiene al gruppo delle trasformazioni del control flow graph e l'obiettivo di questa trasformazione è rendere più complicato, per un deoffuscatore, determinare quale sarà la prossima istruzione da eseguire [7] [18].

Per capirne meglio il significato, si può osservare la Figura 2.3 che mostra il codice assembler di un codice non protetto e lo stesso codice dopo aver applicato la trasformazione *anti branch analysis*. Nel codice in chiaro, Figura 2.3a, il salto alla `label T` è facilmente intuibile dall'istruzione `jmp T`. Nel codice offuscato invece, Figura 2.3b, il salto alla `label T` viene effettuato passando dalla `label P` (istruzione `jmp P`). Seguendo il flusso di esecuzione del programma si può notare che una volta eseguite le istruzioni associate alla `label P`, verranno eseguite le istruzioni associate alla `label T`.



(a) Programma in chiaro

(b) Programma offuscato

Figura 2.3: Esempio di anti branch analysis.

Si noti che nell'esempio riportato in Figura 2.3b, l'insieme di istruzioni associate alla `label P` sono istruzioni nulle (`NOP`) e che il salto alla `label P` è in chiaro (istruzione `jmp P`). Per un miglior offuscamento, si potrebbe integrare il programma utilizzando istruzioni non nulle associate alla `label P` oppure utilizzando i predicati opachi per effettuare il salto all'etichetta di destinazione.

2.2.4 Encoding

L'encoding appartiene al gruppo delle trasformazioni dei dati. Lo scopo principale è cambiare la codifica di stringhe e numeri per rendere più complicata la comprensione dei dati trattati. Di seguito si riportano due esempi di *encoding*, uno per le stringhe ed uno per i dati numerici:

- la stringa "prova" viene codificata in *base64*, diventando "`cHJvdmE=`";
- il dato intero "132654" viene codificato in *esadecimale*, diventando "$2062E$" (codifica funzionante solo per il codice sorgente, non per il binario).

In Figura 2.4 viene riportato un ulteriore esempio trattando un'espressione algebrica intera. Nel codice in chiaro il risultato dell'espressione è facilmente intuibile ed anche manualmente si potrebbe ricavare il valore risultante, al contrario di quanto avviene nel codice sottoposto a protezione.

2.3 Potency

Per definire quanto una trasformazione τ (cfr. Formula 2.1) sia sicura o meno, si può utilizzare la formula legata alla *potency*. Difatti il calcolo della *potency* fornisce un valore oggettivo di

```
a= 10;
b=40;
c=75;
res= a+b+c;
```

(a) Espressione in chiaro

```
a = 10;
b = 40;
c = 75;
res = ((((a | b) << 1) - (a ^ b)) ^ c) + (((((a | b) << 1) - (a ^ b)) & c) + ((((a | b) << 1) - (a ^ b)) & c));
```

(b) Espressione offuscata

Figura 2.4: Esempio di encoding.

quanto un dato offuscamento protegga il codice sorgente originale dalla comprensione di un lettore umano [7]. Dati:

- P: il codice sorgente a cui applicare una trasformazione;
- P': il codice sorgente a cui è stata applicata una trasformazione;
- τ: una trasformazione in grado di mantenere il comportamento di P in P', si veda Equazione 2.1;
- $E(P)$ e $E(P')$: le complessità di P e P' rispettivamente, in cui $E()$ rappresenta una metrica di complessità come descritto nel Capitolo 1.

Allora il calcolo di $\tau_{pot}(P)$, *potency* di τ rispetto a P, è mostrato dalla Formula 2.2 e rappresenta il cambiamento del valore della complessità da P a P'.

$$\tau_{pot}(P) = E(P')/E(P) - 1 \tag{2.2}$$

In base al risultato ottenuto si può affermare che:

- se $\tau_{pot}(P) > 0$ allora τ è un valido offuscamento;
- se $\tau_{pot}(P) = 0$ allora non è stato apportato un offuscamento poiché non c'è stato alcun incremento della complessità;
- se $\tau_{pot}(P) < 0$ allora τ è un deoffuscamento poiché si è decrementata la complessità, per cui si sta semplificando il codice.

2.4 Potency combinata

La potency combinata è una metrica calcolata come la somma di alcune metriche [19], scelte a priori, moltiplicate per un peso arbitrario dato un determinato offuscamento:

$$P_{\sigma,a} = \sum_{m \in M} w_m * \pi_{\sigma,m,a} \tag{2.3}$$

Dove:

- a rappresenta un asset;
- m rappresenta la metrica presa in considerazione;
- σ rappresenta l'offuscamento considerato;
- $\pi_{\sigma,m,a}$ è la potency di σ su a per la metrica m;
- M rappresenta l'insieme di tutte le metriche calcolate;
- w_m rappresenta il peso arbitrario che si vuole associare alla metrica m.

Per lo script *Potency.py* si è deciso di porre i pesi w_m uguali a 1 per ogni metrica presa in considerazione:

- nr_ins_static;
- nr_src_oper_static;
- nr_dst_oper_static;
- halstead_program_size_static;
- nr_edges_static;
- nr_indirect_edges_CFIM_static;
- cyclomatic_complexity_static.

Per mezzo di questa metrica, in grado di aggregare potency calcolate su metriche diverse, si possono analizzare diversi aspetti legati alla complessità applicata da un determinato offuscamento analizzando in questo modo il livello di sicurezza introdotto da diversi punti di vista.

2.5 Strumenti per l'offuscamento

Di seguito una lista di strumenti in grado di offuscare il codice, riportandone le caratteristiche principali.

2.5.1 Appfuscator

Appfuscator[3] è uno strumento per l'offuscamento di codice sorgente dedicato alla protezione dei linguaggi di programmazione supportati dalla piattaforma *.NET*[4] ed altri linguaggi di programmazione quali ad esempio python e perl. È disponibile solamente per la piattaforma Windows ed esistono sia versioni a pagamento, sia una versione completamente gratuita. Per la versione gratuita gli offuscamenti resi disponibili sono i seguenti:

- la capacità di rinominare alcuni elementi del codice;
- la cifratura delle costanti nel codice, come ad esempio stringhe ed interi.

2.5.2 CObfuscator

CObfuscator[5] è uno strumento di offuscamento disponibile sia per Windows sia per Unix utilizzato per la protezione del codice sorgente scritto in C/C++. Lo strumento è gratuito per entrambe le piattaforme. Gli offuscamenti messi a disposizioni sono:

- la capacità di rinominare alcuni elementi del codice;
- l'encoding;
- l'appiattimento del control flow graph, tecnica riportata nella Sezione 2.2.2.

2.5.3 CodeMorph

CodeMorph[6] è uno strumento di offuscamento disponibile per la piattaforma Windows in grado di applicare offuscamenti per il codice sorgente scritto nei linguaggi C e C++. È uno strumento a pagamento avendo però a disposizione delle versioni di prova gratuite. L'unica protezione fornita risulta essere l'aggiunta di *junk code* nei file del codice sorgente.

---

[3] http://appfuscator.com/.

[4] Tali linguaggi risultano essere ad esempio C#, managed C++, visual basic .NET

[5] http://cipherbox.blogspot.com/2010/02/code-obfuscator-for-c-program-control.html.

[6] http://www.sourceformat.com/code-obfuscator.htm.

2.5.4 Tigress

Tigress[7] è uno strumento open source che permette l'offuscamento del codice sorgente scritto in C e C++ ed è disponibile per la piattaforma Unix. Questo stumento mette a disposizione diverse metodologie di offuscamento, tra cui:

- l'appiattimento del control flow graph, tecnica riportata nella Sezione 2.2.2;
- l'unione di due funzioni in una sola;
- la divisione di una funzione in funzioni più piccole;
- la sostituzione di espressioni aritmetiche composte da numeri interi con espressioni più complesse, come mostrato nella Sezione 2.2.4;
- la sostituzione di variabili definite come interi con rappresentazioni più complesse, come mostrato nella Sezione 2.2.4;
- la creazione e l'uso di predicati e variabili opache, come riportato nella Sezione 2.2.1;
- la trasformazione dell'*anti branch analysis*, come riportato nella Sezione 2.2.3.

2.5.5 Snob

Snob[8], acronimo di *Simple Name Obfuscator*, è uno strumento di offuscamento disponibile per Windows ed è gratuito. La sua funzionalità si può applicare a diversi linguaggi di programmazione quali Javascript, python, perl e altri linguaggi interpretati. La protezione messa a disposizione da questo strumento, a livello di codice sorgente, è la rimozione di commenti e la capacità di rinominare gli identificatori con stringhe prive di significato.

2.5.6 StarForce

StarForce[9] è uno strumento di offuscamento per i linguaggi di programmazione C e C++ in grado di operare sia a livello di codice sorgente sia a livello binario. È uno strumento a pagamento ma con una versione di prova gratuita, disponibile per Windows, Unix, macOS. Offre più di 30 metodi di protezione, tra cui:

- l'encoding;
- l'aggiunta di *junk code*;
- l'unione di sezioni di codice selezionate;

2.5.7 cxxo

cxxo[10] è uno strumento di offuscamento del codice sorgente scritto in C/C++, disponibile per le piattaforme Windows, Unix e macOS. È uno strumento a pagamento ma con una versione di prova gratuita. Offre differenti modalità di offuscamento tra cui:

- la capacità di rinominare le variabili;
- l'encoding;
- la rimozione di commenti e linee vuote;
- la possibilità di compattare il codice su un'unica linea.

[7] http://tigress.cs.arizona.edu/index.html.

[8] http://www.macroexpressions.com/snob.html.

[9] http://www.star-force.com/products/starforce-obfuscator/.

[10] http://stunnix.com/prod/cxxo/.

2.5.8 Diablo

Diablo[11] è un framework di riscrittura binaria disponibile per ambiente Unix. Lo strumento mette a disposizione diverse funzionalità tra cui tre metologie di offuscamento:

- l'appiattimento del control flow graph, tecnica riportata nella Sezione 2.2.2;
- la creazione e l'uso di predicati e variabili opache, il cui scopo è mostrato nella Sezione 2.2.1;
- la trasformazione dell'*anti branch analysis*, come riportato nella Sezione 2.2.3.

2.6 Confronto e risultati

Nella Tabella 2.1 mostra un riassunto degli strumenti di offuscamento descritti e dettagliati nella Sezione 2.5.

Strumenti di offuscamento	Ambiente	Codice sorgente/ binario	File.c	Anti branch analysis	Control flow flattening	Predicati opachi
Appfuscator	Window	sorgente	no	no	no	no
CObfuscator	Window Unix	sorgente	si	no	si	no
CodeMorph	Windows	sorgente	si	no	no	no
Tigress	Unix	sorgente	si	si	si	si
Snob	Window	sorgente	no	no	no	no
StarForce	Window Unix macOS	sorgente binario	si	no	no	no
cxxo	Window Unix macOS	sorgente	si	no	no	no
Diablo	Unix	binario	no	si	si	si

Tabella 2.1: Riassunto degli strumenti di offuscamento analizzati.

Nella scelta degli strumenti di offuscamento, si è deciso di concentrarsi su strumenti che fossero in grado di rispettare le seguenti condizioni:

- operare su ambiente Unix;
- applicare le protezioni su codice sorgente scritto *.c* o a livello binario;
- applicare offuscamenti più complessi della capacità di rinominare le variabili e della aggiunta di codice fittizio, come l'uso dell'*anti branch analysis*, del *control flow flattening*, dei *predicati opachi*.

A fronte di questi quattro prerequisiti, dopo un'accurata analisi, la scelta degli strumenti da utilizzare è ricaduta su: *Tigress* e *Diablo*.

[11] https://github.com/csl-ugent/diablo.

Tigress

Lo strumento *Tigress*, presentato in Sezione 2.5.4), è uno strumento di offuscamento in grado di applicare le trasformazioni sul codice sorgente. Per eseguire *Tigress* da terminale, il comando da usare risulta essere

```
tigress --Transform=[trasformazione] [parametri_trasformazione] [file]
```

in cui:

- il parametro `[trasformazione]` rappresenta la trasformazione che si vuole applicare;
- il parametro `[parametri_trasformazione]` rappresenta la lista di parametri associati alla trasformazione da voler applicare. Ad esempio, supponendo di scegliere la trasformazione *control flow flattening* si può indicare il tipo di dispatch[12] da voler utilizzare potendo scegliere tra diversi metodi quali: `switch`, `goto`, `call`; altrimenti, scegliendo la trasformazione che utilizza i *predicati opachi* si può indicare il tipo di struttura dell'operatore opaco da voler utilizzare come: `struct`, `list`, `input`.
- il parametro `[file]` rappresenta il file al quale si vuole applicare la trasformazione selezionata.

Diablo

Lo strumento *Diablo*, presentato in Sezione 2.5.8 è uno strumento di offuscamento in grado di applicare le trasformazioni sul codice binario. Per eseguire *Diablo* da terminale, il comando da usare è: `[path\_to\_diablo-obfuscator] -[Z off] -[S] --annotation-file [annotations.json] -[O .] -o [output\_file] [input\_file]`. I parametri principali risultano essere:

- `[path_to_diablo-obfuscator]`, per indicare il *path* al compilatore da voler utilizzare;
- `annotations.json`, per rappresenta il percorso al file delle annotazioni;
- `[output_file]`, per indicare il file in cui memorizzare i risultati ottenuti;
- `[input_file]`, per indicare il file da passare come input.

In Appendice A vengono riportati maggiori dettagli in merito al file delle annotazioni denominato come `annotations.json`: il tipo di trasformazioni da voler applicare al file passato in input verranno indicate all'interno di questo file.

[12] Metodo con il quale si sceglie di saltare da un punto del programma ad un altro.

Capitolo 3

Benchmark suite

In questo capitolo inizialmente verrà fornita un definizione di benchmark comprenderne obiettivi, scopi e funzionalità, Successivamente verranno introdotte le benchmark suite e ne saranno analizzate e confrontate alcune di esse. Infine si darà una spiegazione di quale, tra le benchmark suite presentate, è stata scelta elencandone: i motivi principali, il fine per cui si necessità l'utilizzo di tale suite e gli scopi per cui verrà utilizzata.

3.1 Benchmark: una definizione

In informatica il termine *benchmark* viene utilizzato per definire un insieme di test atti a misurare e classificare, tramite indici quantitativi o qualitativi restituiti come risultato, le prestazioni di determinati componenti hardware quali ad esempio CPU[1] e GPU[2], oppure componenti software come potrebbero essere web server o applicazioni client [20] [21]. In particolare ogni *benchamrk* viene appositamente ideato col fine di testare l'operatività per cui è stato progettato un dato componente in termini di velocità e correttezza delle operazioni da eseguire e la capacità di operare essendo soggetto a carichi di lavoro molti elevati [22] [23] [24] [25] [26]. Tra i test più conosciuti si riportano:

- *stress test*: utilizzato per testare la capacità del componente di operare sotto enormi quantità di carico di lavoro;
- *spike test*: utilizzato per verificare le prestazioni di un componente sottoposto a numerose richieste simultanee e sottoposto a differenti livelli di carico di lavoro in differenti intervalli temporali;
- *load test*: utilizzato per studiare le prestazioni del componente sottoposto ad un determinato carico di lavoro;
- *soak test*: utilizzato per verificare che il componente sottoposto ad un carico di lavoro costante nel tempo sia in grado di mantenere le prestazioni desiderate.

Solitamente i parametri sottoposti a monitoraggio per evidenziare criticità e malfunzionamenti sono:

- il throughput per la velocità di risposta;
- la correttezza della risposta;
- l'utilizzo della CPU;
- l'utilizzo della memoria.

[1] Central processing unit, unità di elaborazione centrale.

[2] Graphics processing unit, unità di elaborazione grafica.

3.2 Benchmarck suite analizzate

I *benchmarck suite* sono delle raccolte di benchmark, messe a disposizione per poter testare le specifiche caratteristiche dei propri componenti hardware o software. In base alla proprietà da voler testare, si sceglie il benchmark opportuno e si valutano i risultati ottenuti.

3.2.1 SPEC

Standard Performance Evaluation Corporation o SPEC[3] è una società costituita per stabilire, mantenere e sostenere benchmark standardizzati per valutare le prestazioni e l'efficienza energetica della nuova generazione di sistemi informatici. Vengono messe a disposizione diverse tipologie di suite:

- *cloud*: benchmark progettati per sollecitare il provisioning e gli aspetti di runtime di un cloud presentato come IaaS [4];
- *CPU*: benchmark progettati per fornire misurazioni delle prestazioni CPU;
- *graphics and workstation performance*: benchmark progettati per misurare le prestazioni grafiche e le prestazioni delle workstation;
- *Java client/server*: benchmark progettati per misurare le prestazioni in base alle più recenti funzionalità delle applicazioni Java;
- *storage*: benchmark progettati per valutare le prestazioni utilizzando il throughput dei file server e il tempo di risposta;
- *power*: benchmark atti alla valutazione delle caratteristiche di potenza assorbita dall'hardware;
- *virtualization*: benchmark per valutare le prestazioni dei server virtualizzati.

La benchmark suite di SPEC è a pagamento e non è open-source.

3.2.2 Phoronix test suite

Phoronix test suite[5] è una piattaforma che offre più di 450 profili di test e oltre 100 suite di test supportati dal sito `OpenBenchmarking.org`[6] e permette di aggiungere nuovi test. È disponibile per Windows, Linux, macOS, GNU Hurd, Solaris e BSD Operating Systems e tale piattaforma risulta è sia gratuita sia open-source. Tra le varie suite messe a disposizione, si possono trovare:

- compiler: benchmark per testare le prestazioni del compilatore;
- computational: benchmark per la misurazione delle prestazioni computazionali ed aritmetiche del sistema;
- CPU: benchmark progettati per testare le prestazioni della CPU;
- desktop graphics: benchmark progettati per testare le prestazioni grafiche del desktop;
- workstation: benchmark progettati per misurare le prestazioni delle workstation;
- memory test: benchmark progettati per testare le prestazioni della RAM del computer;
- server motherboard: benchamrk progettati per testare le prestazioni della scheda madre per un ambiente server.

[3] https://www.spec.org/benchmarks.html.

[4] Infrastucture as a Service.

[5] https://www.phoronix-test-suite.com/.

[6] https://openbenchmarking.org/.

3.3 Conclusioni ed utilizzo

Nel nostro caso, la necessità di una benchmark suite è dovuta al bisogno di disporre di una fonte da cui poter scaricare una notevole/modesta quantità di file sorgenti (implementati col linguaggio di programmazione C) e non al fatto di voler analizzare specifiche caratteristiche di componenti hardware o software.

Tra le due suite descritte nella Sezione 3.2, SPEC e Phoronix test suite, si è deciso di utilizzare Phoronix test suite poiché:

- è gratuita;
- è open-source;
- offre una vasta gamma di programmi scaricabili;
- è disponibile su ambiente Linux;
- offre la possibilità di scaricare i benchmark messi a disposizione come file sorgenti ed inoltre molti di questi file risultano essere implementati con il linguaggio *C*;
- ogni benchmark scaricato è dotato di *makefile*[7]. Esistono tuttavia alcune eccezioni date da benchmark composti da file sorgenti che non hanno bisogno di dipendenze complesse per essere compilati come, ad esempio, i benchmark composti da un singolo file sorgente.

Avendo a disposizione il *makefile* si possono inoltre creare i compilati solo per determinati *target*[8] scelti oppure apportare modifiche ai parametri di compilazione per adattare i compilati alle proprie esigenze.

[7] Un *makefile* è un file di testo che contiene le istruzione per eseguire la compilazione dei file sorgenti ad esso associati.

[8] All'interno dei *makefile* i *target* sono gruppi di comandi che verranno eseguiti in blocco.

Capitolo 4

Progettazione della soluzione

In questo capitolo verranno enunciati i motivi principali che hanno portato a questo studio ed alle metodologie con le quali si è deciso di raccogliere i dati necessari. In particolare si discuterà della progettazione messa in atto per la creazione di uno script in grado di automatizzare la raccolta di metriche a livello sorgente e binario sia di file non protetti che offuscati. Verranno infine mostrati e descritti l'architettura ed il workflow dello script e si riporterà come si è pensato di organizzare i dati estratti. Il nome dello script è: *Metrics_analysis.py*.

4.1 Obiettivi

Vengono di seguito esposti i motivi principali dello studio eseguito.

4.1.1 Fingerprint

Il termine *fingerprint* viene utilizzato per indicare un pattern di codice riconoscibile o un comportamento particolare assunto da un programma in esecuzione [11]. Per mezzo delle *fingerprint*, quindi, si potrebbero identificare delle porzioni di codice, chiamate asset, che potrebbero contenere delle informazioni ritenute sensibili (password, algoritmi proprietari, dati confidenziali). Tali asset generalmente risultano protetti: trovata una zona di codice protetta quindi molto probabilmente potrebbe contenere un asset.

Nell'articolo "Towards Optimally Hiding Protected Assets in Software Applications" [11], sono state discusse le *fingerprint* ed è stato teorizzato che esse siano legate alle metriche. Lo scopo dei test che verranno effettuati consisterà nel dimostrare la presenza di tali *fingerprint* e porre in risalto la correlazione, nel caso esistesse, con quali metriche di riferimento.

4.1.2 Potency

La potency (cfr. Sezione 2.3) è una metrica che fornisce un valore oggettivo di quanto un dato offuscamento protegga il codice sorgente originale mettendo a confronto il valore di una data metrica post offuscamento rispetto alla stessa metrica calcolata sulla funzione *vanilla*.

A seguito di interventi e di interazioni con gli esperti del progetto ASPIRE[1] è emerso che, dopo numerosi studi da loro effettuati, le potency di maggiore rilevanza risultano essere le potency legate alle metriche che rappresentano la complessità ciclomatica e lunghezza di Halstead (calcolate sui file binari). Si cercherà quindi di dimostrare la presenza di ulteriori metriche che potrebbero ritenersi utili per il calcolo della *potency*.

[1] Progetto che si occupa della protezione del codice: https://aspire-fp7.eu/.

Un ulteriore studio che verrà affrontato in merito alle *potency* è legato alla ricerca di un metodo in grado di identificare quali offuscamenti siano in grado di introdurre un livello di sicurezza maggiore. Si cercherà dunque di dimostrare se tale metrica possa ritenersi utile ad indicare il livello di protezione introdotto dall'offuscamento applicato al codice che si è voluto proteggere.

4.2 Automatizzazione

Per poter ottenere una quantità di dati sufficienti a ricavare dei dati che permettano di esprimere dei risultati significati, veritieri in generale e non legati al singolo caso preso in oggetto, si è progettato ed implementato uno script per automatizzare la raccolta dati. I requisiti principali richiesti per lo sviluppo di tale script, risultano essere:

- estrarre le metriche su file sorgenti e binari;
- compilare i programmi;
- applicare determinati offuscamenti sia su file sorgenti sia su file binari.

Per poter estrarre le metriche ed applicare gli offuscamenti sui file analizzati, lo script interagisce con appositi strumenti in grado di compiere tali azioni. Si è inoltre pensato, per ottenere molti programmi da analizzare, di utilizzare i programmi sorgenti contenuti nelle *benchmark suite* aventi anche a disposizione il *Makefile* per poter procedere alla loro compilazione.

Vincoli

Per la creazione dello script, gli unici vincoli imposti riguardano l'utilizzo degli strumenti atti all'estrazione delle metriche ad all'applicazione degli offuscamenti. Difatti le caratteristiche ricercate risultano essere:

- operare su file con estensione *.c* e file binari;
- operare su ambiente Unix;
- essere open source.

Inoltre per gli strumenti di estrazione metriche, un ulteriore vincolo imposto, risulta essere la capacità di estrarre, tra le metriche a disposizione, la complessità ciclomatica e le metriche di Halstead. Per gli strumenti in grado di applicare gli offuscamenti, invece, è stata richiesta la capacità di applicare offuscamenti complessi come l'*anti branch analysis*, il *control flow flattening*, i *predicati opachi*.

4.3 Architettura

La Figura 4.1 mostra l'architettura dello script *Metrics_analysis.py*.

Tale script si interfaccia con diversi moduli:

- Ctags[2]: strumento che permette di estrarre vari tipi di informazione, utilizzato in questo progetto per rintracciare il nome delle funzioni implementate nei vari file sorgente e per estrarne il numero di riga di inizio e di fine;
- Frama-c: strumento utilizzato per estrarre i valori delle metriche software (cfr. Capitolo 1) su file sorgenti (cfr. Sezione 1.2.1);

[2] https://ctags.io/.

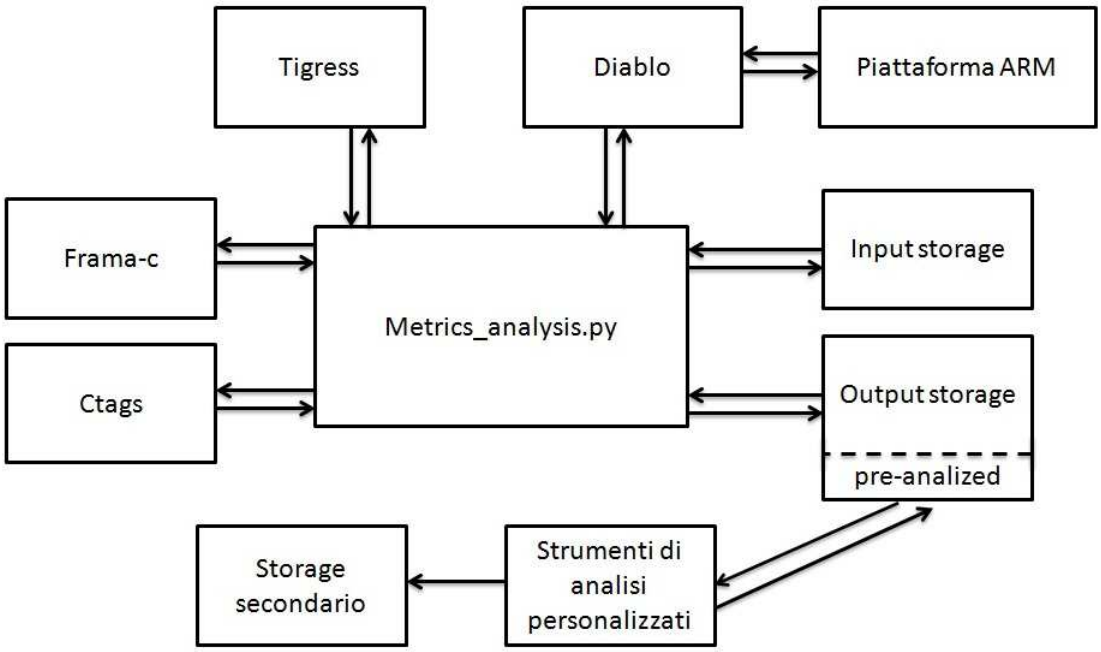


Figura 4.1: Architettura di Metrics_analysis.py.

- Tigress: strumento utilizzato per apportare determinati offuscamenti a livello di codice sorgente (cfr. Sezione 2.5.4);
- Diablo: strumento utilizzato per estrarre i valori delle metriche software (cfr. Capitolo 1) e per applicare offuscamenti su file binari (cfr. Sezione 2.5.8);
- piattaforma ARM: piattaforma necessaria al corretto funzionamento di *Diablo*;
- input storage: directory principale, fornita da linea di comando, contenente le cartelle dei programmi scaricati[3]. Lo script *Metrics_analysis.py* analizzerà ogni cartella presente in questa directory;
- output storage: directory, fornita da linea di comando, in cui lo script *Metrics_analysis.py* depositerà tutti gli output generati.

Nello schema architetturale sono stati riportati altri due moduli non collegati direttamente allo script *Metrics_analysis.py*, ma di fondamentale importanza per l'analisi dei dati estratti dallo script:

- strumenti di analisi personalizzati: questo blocco raccoglie logicamente l'insieme degli script implementati per l'analisi custom, ad esempio calcolo della media e della potency (cfr. Sezione 2.3) o gli script necessari per la creazione dei file CSV (cfr. Sezione B.2.3), dei file di output generati (contenuti nella cartella *pre-analized*);
- storage secondario: directory, fornita da linea di comando agli script citati al punto precedente, in cui gli script personalizzati depositano il proprio output prodotto.

[3]I programmi sono scaricati dal benchmark Phoronix test suite (cfr. Sezione 3.2.2).

4.4 Input, output e log

Lo script *Metrics_analysis.py* si trova all'interno di una cartella chiamata *Metrics_analysis*. In questa cartella sono contenuti inoltre:

- una cartella chiamata *sub* contenente a sua volta:
  - una cartella chiamata *util* in cui depositare ulteriori script se ritenuti necessari per migliorare il funzionamento di *Metrics_analysis.py*, gli script per calcolare la media e la potency (cfr. Capitolo 6), gli script utilizzati per effettuare l'analisi dei dati raccolti;
  - un file di configurazione chiamato *conf.txt*, trattato in Appendice C.3.1, contenente i parametri necessari per richiamare i moduli esterni allo script ed i parametri attraverso i quali è possibile selezionare il comportamento dello script stesso;
  - due ulteriori script chiamati *create_diablo_csv.py* e *create_frama_csv.py* richiamati da *Metrics_analysis.py* ed utilizzati per creare i file *.csv* (cfr. Sezione B.2.3) contenuti nella cartella *pre-analized* (cfr. Listato 4.4.2).
- un file chiamato *LOGFILE.txt*, si veda la Sezione 4.4.3), utilizzato per riportare tutti gli errori avvenuti durante l'esecuzione dello script;
- una cartella chiamata *Log*, si veda Sezione 4.4.3, in cui verranno creati tanti file quanti gli errori generati ed ogni file conterrà il dettaglio dell'errore riscontrato.

Di seguito vengono dettagliati i parametri di input di cui necessita lo script, la struttura dell'output prodotto e del file dei log creato.

4.4.1 Input

Siccome il file di configurazione *conf.txt* (cfr. Appendice C.3.1) permette di impostare tutti i parametri necessari all'esecuzione dello script, si è dunque deciso di implementare tale strumento affinché esso possa essere lanciato da linea di comando utilizzando il minor numero di parametri:

```
./Metrics_analysis.py [source] [destination]
```

- `[source]`: rappresenta la directory principale in cui si trovano le sotto-cartelle dei programmi da voler analizzare scaricati da Phoronix test suite, descritta in Sezione 3.2.2;
- `[destination]`: rappresenta la directory in cui si vuole memorizzare l'intero output prodotto dallo script.

4.4.2 Output

Durante l'esecuzione dello script, vengono eseguite numerose operazioni. Si è deciso di tenere traccia di ogni output prodotto e di creare una gerarchia ordinata di sotto-cartelle che permettono di guidare, in maniera intuitiva, un utente all'output di proprio interesse. La Figura 4.2 mostra le cartelle e sotto-cartelle generate da *Metrics_analysis.py*.

compiled. Cartella contenente i file binari generati. Al suo interno si trovano due cartelle:

- *clean*[4]: contiene i file binari in chiaro (senza alcun offuscamento applicato);

[4] In tale cartella sono presenti tante sotto-cartelle quanti sono i programmi analizzati.

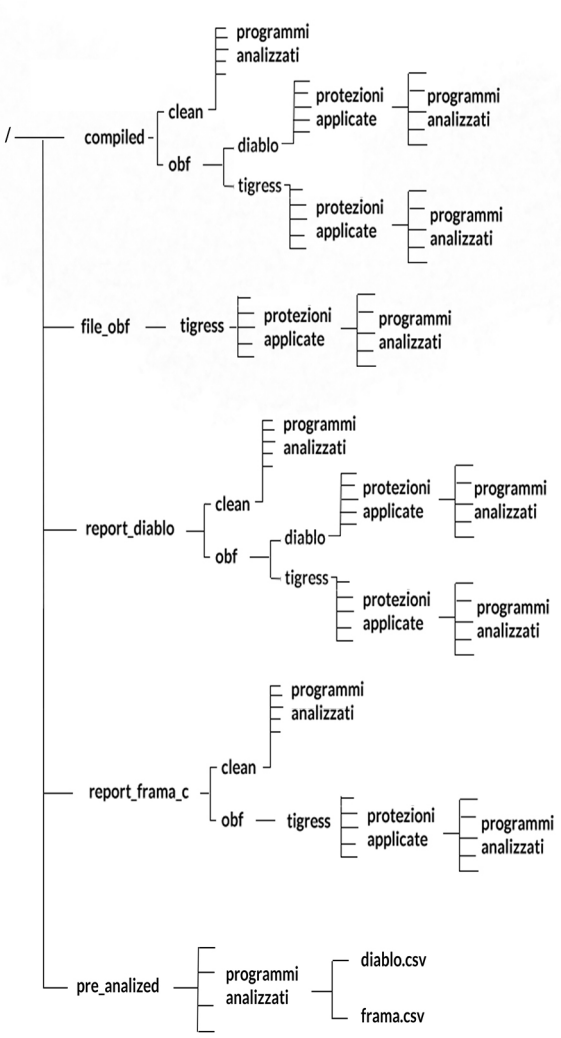


Figura 4.2: Output generato.

- *obf*: contiene ulteriori due sotto-cartelle, *diablo*[5] e *tigress*[6]. La prima raccoglie i file binari offuscati dallo strumento *Diablo*, la seconda i file binari offuscati dallo strumento *Tigress*.

file_obf. Cartella contenente i file sorgenti offuscati dallo strumento Tigress 2.5.4. Al suo interno è presente un'unica cartella chiamata *tigress*;

report_diablo: Cartella contenente i report generati dallo strumento di Diablo 2.5.8. Al suo interno si trovano due cartelle:

- *clean*[4]: contiene i report generati sui file binari in chiaro (senza alcun offuscamento applicato);

[5]In tale cartella sono presenti tante sotto-cartelle quanti sono gli offuscamenti selezionati per lo strumento *Diablo*.

[6]In tale cartella sono presenti tante sotto-cartelle quanti sono gli offuscamenti selezionati per lo strumento *Tigress*.

- *obf*: contiene ulteriori due sotto-cartelle, *diablo*[5] e *tigress*[6]. Tali cartelle raccolgono i report generati sui file binari offuscati dagli strumenti di *Diablo* e *Tigress* rispettivamente.

report_frama_c. Cartella contenente i report generati dallo strumento di Frama-c 1.2.1. Al suo interno si trovano due cartelle:

- *clean*[4]: contiene i report generati sui file sorgenti in chiaro (senza alcun offuscamento applicato);
- *obf*: al suo interno è presente un'unica cartella chiamata *tigress*[6] che raccoglie i report generati sui file sorgenti offuscati dallo strumento *Tigress*.

pre_analized. Cartella[4] contenente i file in cui sono raccolti i valori delle metriche estratte dai report generati dagli strumenti *Frama-c* e *Diablo*. Per ogni sotto-cartella generata saranno presenti due file *csv* (cfr. Sezione B.2.3): uno contenente tutte le metriche estratte dallo strumento di *Frama-c* ed uno contenente tutte le metriche estratte dallo strumento *Diablo*.

Inoltre, per ogni sotto-cartella dei programmi analizzati, si troveranno tante altre sotto-cartelle quanti saranno i *target* del Makefile utilizzati per la compilazione (ad ogni cartella sarà associato il nome del programma concatenato al nome del *target* del Makefile).

4.4.3 Log

Metrics_analysis.py dispone di due meccanismi per memorizzare i log: un file chiamato *LOGFILE.txt* ed una directory chiamata *Log* situati nella cartella *Metrics_analysis*. Sia il file che la cartella sono aggiornati con i log appartenenti all'ultima esecuzione dello script.

Il file *LOGFILE.txt* è un file di testo, suddiviso per programmi[7], che memorizza tutti i malfunzionamenti o comportamenti anomali ottenuti durante l'esecuzione dello script. Ogni errore riscontrato è suddiviso dal seguente mediante una linea vuota. In questo file viene riportato il tipo di errore riscontrato (ad esempio offuscamento non avvenuto o errore nel lancio di uno strumento esterno allo script), il file che ha generato tale errore e, se disponibile, viene anche riportata la conseguenza dovuta al malfunzionamento. Se il comportamento anomalo viene generato da uno strumento esterno richiamato dallo script *Metrics_analysis.py*, l'output generato viene scritto su un file e il percorso a tale file viene mostrato su questo file di log sotto la dicitura `"See file : [nome del file] in [percorso del file]"`.

La cartella *Log* contiene i file di errore generati dagli strumenti esterni richiamati da *Metrics_analysis.py*. Per ogni programma analizzato dallo script viene creata una sotto-cartella corrispondete al programma analizzato e al suo interno vengono create tante cartelle, se vengono riscontrati degli errori, quanti sono i *target* del Makefile utilizzati per la compilazione (ad ogni cartella sarà associato il nome del programma concatenato al nome del *target* del Makefile). In ognuna di queste cartelle vengono depositati tutti i file di errore (gli stessi file il cui path è presente in *LOGFILE.txt*) che si generano durante l'esecuzione dello script.

4.5 Configurazioni

Per poter funzionare, lo script *Metrics_analysis.py* ha bisogno della corretta configurazione di alcuni parametri, azione che deve essere compiuta dall'utente utilizzatore di tale script, prima di essere lanciato da linea di comando. Si è pensato di suddividere tali parametri in due file di configurazione. In Appendice B.1.4 sono riportati e dettagliati tutti i campi presenti in questi file di configurazione.

[7] Programmi analizzati da *Metrics_analysis.py*.

File di configurazione per Metrics_analysis.py

Il file di configurazione di *Metrics_analysis.py* è un file necessario per il corretto funzionamento di tale script poiché contiene le informazioni grazie alle quali è possibile richiamare altri moduli esterni allo script, come mostrato nella sua architettura in Figura 4.1. Inoltre, sempre all'interno di questo file di configurazione, sono settabili dei parametri tramite i quali l'utente può selezionare il comportamento desiderato di tale script. T Questo file di configurazione deve essere posizionato in: *Metrics_analysis/sub/conf.txt*.

Tra i campi più importanti si riportano:

- tigress: riporta il path assoluto della directory in cui è stato installato *Tigress*;
- CC_Diablo: riporta il path assoluto della directory in cui è stato installato il compilatore usato da *Diablo*;
- OBF_Diablo: riporta il path assoluto della directory in cui è stato installato l'offuscatore di *Diablo*.

File di configurazione per i programmi da analizzare

Per ogni programma scaricato dalla suite di benchmark di Phoronix (cfr. Sezione 3.2.2) è necessario creare un file di configurazione affinchè lo script *Metrics_analysis.py* sappia quali file *.c* analizzare ed i parametri corretti da utilizzare in fase di compilazione. Questo file può deve essere posizionato in un una qualsiasi sotto-cartella appartenente al programma al quale fa riferimento.

Tra i campi più importanti si riportano:

- [target_Makefile]: il nome *target* del Makefile che si vuole associare a tali parametri, possono esserci più *target* per ogni file di configurazione;
- _dirmakefile: indica il percorso del Makefile come percorso relativo dalla cartella principale del programma da voler analizzare;
- file: lista dei file .c, separati da una virgola, con path relativo del file stesso dalla cartella principale del programma preso in considerazione;
- cmd: lista dei comandi da eseguire, separati da una virgola, per testare il corretto funzionamento del binario compilato.

Ogni file di configurazione creato dovrà necessariamente essere chiamato con il seguente formato: *confmake_[name].ini* dove *[name]* è una stringa alfanumerica scelta dall'utente.

4.6 Workflow

In questa sezione si discute del diagramma di flusso dello script *Metrics_analysis.py*. Tale progettazione risulta necessaria per una corretta gestione ed implementazione del codice che sarebbe stato sviluppato.

Ogni azione necessaria al conseguimento dell'obiettivo finale, l'estrazione di metriche sia su file sorgenti che binari sia su file protetti che in chiaro, è stata suddivisa in singoli blocchi logici: la gestione di ogni operazione, in tal modo, risulta indipendente dalle precedenti. La Figura 4.3 mostra il workflow. L'intestazione presente in ogni blocco fornisce una descrizione concettualizzata delle azioni che verranno eseguite all'interno dello stesso.

Si riporta una breve descrizione per ogni blocco raffigurato:

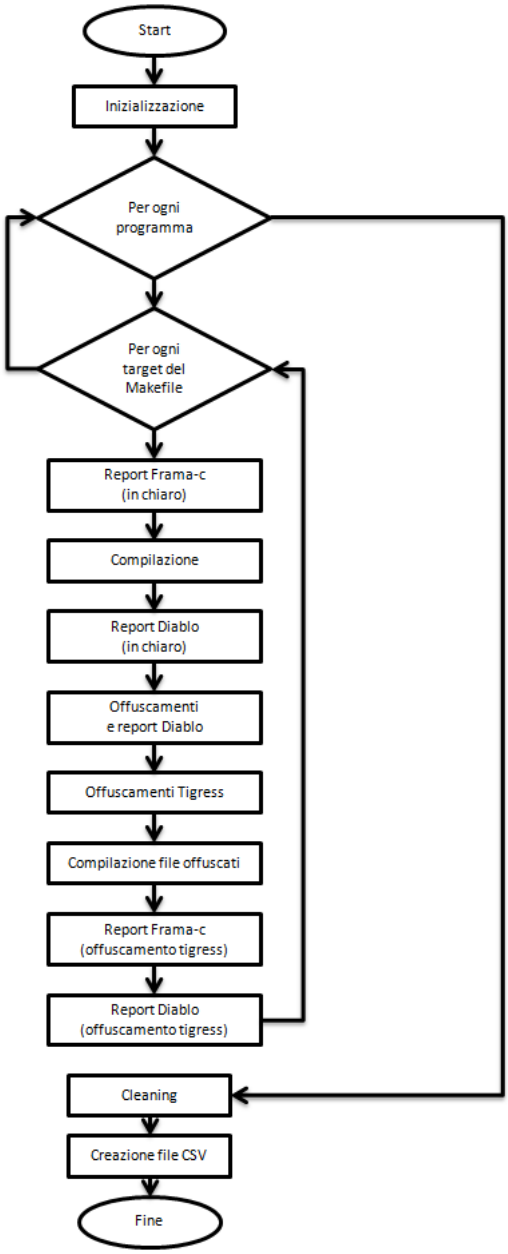


Figura 4.3: Workflow di Metrics_analysis.py.

1. inizializzazione: in questa fase vengono create le cartelle principali in cui memorizzare le informazioni principali (tra cui i report generati dagli strumenti di Frama-c e Diablo, i file in chiaro e quelli offuscati ed i rispettivi sorgenti e binari) e vengono configurati i parametri necessari al corretto funzionamento dello script. Tali parametri vengono estratti dal file di configurazione associato a *Metrics_analysis.py* (cfr. Sezione 4.5);

2. per ogni programma: blocco condizionale *for*. Tale ciclo permette di analizzare, uno alla volta, tutti i file all'interno della cartella *Input storage* (cfr. Listato 4.3). Tale cartella viene passata come primo parametro da linea di comando;

3. per ogni target del Makefile: blocco condizionale *for*. Tale ciclo permette di analizzare, uno alla volta, tutti i comandi *target* per ciascun programma;

4. report Frama-c (in chiaro): si richiama lo strumento *Frama-c* per generare il report contenente il valore delle metriche estratte dai file sorgenti;

5. compilazione: si richiama il comando *make*, passando i target del comando da voler prendere in considerazione, per la compilazione dei file sorgenti ed ottenere in tal modo i rispettivi binari desiderati;

6. report Diablo (in chiaro): si richiama lo strumento *Diablo* per generare il report contenente il valore delle metriche estratte dai file binari;

7. offuscamenti e report Diablo: si richiama lo strumento *Diablo* per apportare gli offuscamenti selezionati ai file binari e per generare il report contenente il valore delle metriche estratte dai file binari appena generati;

8. offuscamenti Tigress: si richiama lo strumento *Tigress* per generare file sorgenti offuscati con gli offuscamenti selezionati partendo dai file sorgenti in chiaro;

9. compilazione file offuscati: si richiama il comando *make*, passando i target del comando da voler prendere in considerazione, per la compilazione dei file sorgenti offuscati ed ottenere in tal modo i rispettivi binari offuscati desiderati;

10. report Frama-c (offuscamento tigress): si richiama lo strumento *Frama-c* per generare il report contenente il valore delle metriche estratte dai file sorgenti offuscati;

11. report Diablo (offuscamento tigress): si richiama lo strumento *Diablo* per generare il report contenente il valore delle metriche estratte dai file binari offuscati;

12. cleaning: vengono eliminati i file che non risultano essere necessari o di cui non si vuole tenere traccia in base ai parametri settati nel file di configurazione;

13. creazione file CSV: si richiamano due script che permettono di organizzare e raccogliere rispettivamente le informazioni sulle metriche estratte dai report generati dagli strumenti di *Frama-c* e *Diablo*. Ognuno dei due script genererà un file *CSV*.

Lo scopo dello script dunque, come si può evincere dal workflow descritto, è raccogliere i dati estratti dai vari strumenti integrati con *Metrics_analysis.py* e organizzarli e raggrupparli logicamente per poterli successivamente analizzare tramite script personalizzabili.

4.7 File CSV generati

Per ogni programma analizzato dallo script *Metrics_analysis.py*, come accennato precedentemente nel corso di questo capitolo, verranno generati due file nel formato *CSV*: uno per le metriche estratte dallo strumento *Frama-c* ed uno per le metriche estratte dallo strumento *Diablo*. In Appendice B.2.3 viene descritto dettagliatamente ogni singolo campo presenti in questi file.

File CSV di Frama-c

In questo file vengono raccolte tutte le metriche estratte dallo strumento *Frama-c* appartenenti ad uno stesso programma. Inoltre, per ottenere una corretta gestione delle metriche evitando di perdere o tralasciare informazioni utili, in aggiunta al valore delle metriche si è pensato di riportare ulteriori dati come ad esempio il tipo di protezione applicata, il target del *Makefile* utilizzato ed il nome della funzione. Il nome del *CSV* creato sarà:

[programma analizzato]_frama.csv

dove *[programma analizzato]* sarà sostituito con il nome del programma a cui fanno riferimento le metriche riportate.

File CSV di Diablo

In questo file vengono raccolte tutte le metriche estratte dallo strumento *Diablo* appartenenti ad uno stesso programma. Anche per questo file si è pensato di riportare ulteriori dati come ad esempio il tipo di protezione applicata, il target del *Makefile* utilizzato ed il nome della funzione. Il nome del *CSV* creato sarà:

[programma analizzato]_diablo.csv

dove *[programma analizzato]* sarà sostituito con il nome del programma a cui fanno riferimento le metriche riportate.

Capitolo 5

Implementazione della soluzione

In questo capitolo verrà mostrato e descritto come si è implementato lo script *Metrics_analysis.py*, script scritto con il linguaggio di programmazione *python*. Anche se non espressamente scritto, tutti gli errori generati vengono gestiti e riportati nei file di log propri di ogni programma analizzato. Nelle Appendici B.3 e C vengono riportati maggiori dettagli tecnici.

5.1 Definizione funzioni

Sono state create delle funzioni con lo scopo di tenere ordinato il codice sorgente e per evitare di ripetere identiche parti di codice in differenti parti dello script.

Le funzioni utilizzate sono:

- *Run_Diablo*: utilizzata per generare i report contenenti i valori delle metriche calcolati sui file binari tramite lo strumento *Diablo* e testare il corretto funzionamento del file binario generato da tale strumento;
- *Delete_output_created*: utilizzata per rimuovere gli elementi presenti nella lista del campo *rm_out* del file di configurazione associato al programma che si vuole analizzare. Si veda la definizione di tale parametro in Appendice B.1.4;
- *Check_main*: utilizzata per cercare se nel file passato come parametro è presente la funzione *main*;
- *Modify_if_main*: utilizzata per modificare la funzione *main* del file passato come parametro aggiungendo una lista di funzioni da richiamare prima di eseguire le istruzioni riportate nel *main* stesso (cfr. Sezione 5.2.1);
- *Modify_if_not_main*: utilizzata per trovare il pattern `void main(void)` all'interno del file che si sta analizzando e, se presente, rinominarlo con la stringa nota `void main_[nome file]` ove *[nome file]* è il nome del file che si sta analizzando (in tal modo tale nome rimane univoco)(cfr. Sezione 5.2.1);
- *Exec_command*: utilizzata per eseguire il comando di shell che viene passato come parametro(cfr. Sezione 5.2.1);
- *Create_report_frama_c*: utilizzata per generare i report contenenti i valori delle metriche calcolate sui file sorgenti tramite lo strumento *Frama-c*. Le metriche verranno estratte a livello di funzione;
- *Create_compiled*: utilizzata per compilare i programmi sorgenti richiamando a sua volta la funzione *Exec_command*.

5.2 Implementazione del diagramma di flusso

Ogni sezione è dedicata ad uno specifico blocco raffigurato in Figura 4.3.

5.2.1 Applicazione degli offuscamenti e generazione report

Vengono di seguito descritti i passaggi principali con cui vengono applicati gli offuscamenti e generati i report contenenti le metriche estratte dai file sorgenti e binari.

Inizializzazione

In questa fase vengono raccolti i parametri di input passati allo script *Metrics_analysis.py* da linea di comando, configurati i parametri e le variabili di ambiente necessari al corretto funzionamento degli strumenti quali *Diablo e Tigress* ed infine configurate le variabili, le liste ed i dizionari contenenti i valori ed i parametri scelti per creare la struttura di file-system per memorizzare l'output prodotto (cfr. Sezione 4.4.2) e per apportare gli offuscamenti selezionati.

"Per ogni programma"

In questa fase, si sviluppa l'intero processo di analisi apportato dallo script *Metrics_analysis.py*. Tramite il ciclo *for* verranno, uno per volta, analizzati tutti i programmi selezionati.

"Per ogni target del Makefile"

Per ogni programma preso in considerazione, si svolge l'analisi di tale programma selezionando uno ad uno i vari *target* del Makefile così da poter analizzare uno stesso programma con diverse configurazioni.

Report Frama-c (in chiaro)

In questa fase, tramite la funzione *Create_report_frama_c*, vengono generati i report dello strumento *Frama-c* contenenti i valori delle metriche calcolati sui file sorgenti e non offuscati per ogni file *.c* appartenente al programma che si sta analizzando. I report generati verranno salvati nella relativa cartella di destinazione (cfr. Sezione 4.4.2).

Compilazione

Dopo avere processato i dati tramite lo strumento *Frama-c*, il programma analizzato viene compilato mediante l'utilizzo della funzione *Create_compiled*. La compilazione risulta necessaria per poter, in fase successiva, controllare la correttezza dei futuri programmi compilati.

Successivamente alla fase di compilazione, se questa viene eseguita correttamente, viene richiamata la funzione *Exec_command* che permette di eseguire sul programma binario appena compilato determinati comandi prestabiliti. In un dizionario apposito vengono quindi memorizzati i comandi eseguiti ed i valori di ritorno da essi generati.

Infine viene chiamata la funzione *Delete_output_created* utilizzata per eliminare eventuali file di output creati dall'esecuzione del comando eseguito.

I programmi compilati verranno salvati nella relativa cartella di destinazione (cfr. Sezione 4.4.2).

Report Diablo (in chiaro)

In questa fase, tramite la funzione *Run_Diablo*, vengono generati i report dello strumento *Diablo* contenenti i valori delle metriche calcolati sui file binari e non offuscati. I report generati verranno salvati nella relativa cartella di destinazione (cfr. Sezione 4.4.2).

Offuscamenti e report Diablo

In questa fase vengono generati i report dello strumento *Diablo* contenenti i valori delle metriche calcolati sul programma binario e offuscato mediante gli offuscamenti apportati dallo stesso strumento *Diablo*.

Viene inoltre effettuato un controllo sulla correttezza del file binario offuscato generato. Se il file generato risulta corretto i report ed il file binario stesso vengono memorizzati nelle rispettive cartelle di destinazione (cfr. Sezione 4.4.2), altrimenti essi vengono cancellati e sui file di log viene riportata l'azione eseguita.

Offuscamenti Tigress

In questa fase, per ogni programma che si sta analizzando, vengono apportati tutti gli offuscamenti scelti a livello di file sorgente per ogni file *.c* per lo strumento *Tigress*. In particolare:

1. prima di eseguire l'offuscamento viene richiamata la funzione *Check_main* che permette di selezionare in quale file *.c* è presente la funzione *main*;
2. tramite la funzione *Exec_command*, passandogli come parametro la stringa per eseguire lo strumento *Tigress*, vengono apportati gli offuscamenti desiderati;
3. viene richiamata la funzione *Modify_if_not_main* per ogni file *.c* analizzato;
4. viene richiamata la funzione *Modify_if_main* terminato di analizzare tutti i file *.c* presenti.

Lo strumento *Tigress*, di default, aggiunge ad ogni file analizzato in cui non è presente la funzione *main* una funzione sempre chiamata *void main(void)*, in cui si richiama una lista di funzioni create e definite dallo strumento stesso che devono essere eseguite prima di ogni altra istruzione. Pertanto le funzioni *Check_main*, *Modify_if_not_main* e *Modify_if_main* risultano necessarie per emulare lo stesso comportamento, ovvero, per tutti i file analizzati appartenenti ad uno stesso programma, cercare le funzioni *void main(void)* aggiunte di default dallo strumento *Tigress*, rinominarle, ed inserire il richiamo di queste funzioni all'interno dell'unica necessaria funzione *main* presente prima che venga eseguita qualsiasi altra istruzione. La rinomina è necessaria poiché rende univoche le funzioni create da *Tigress* e permette di richiamarle senza creare conflitti.

Nel caso in cui l'offuscamento di alcuni file non abbia esito positivo, oltre ad essere riportato nei file di log, il nome del file che non è stato offuscato viene memorizzato in un file apposito. In tal modo, in fase di creazione dei file CSV contenenti il valore delle metriche estratte (cfr. Sezione 5.2.2), si potrà riportare che non è stato possibile estrarre le metriche da tale file poiché non è stato offuscato correttamente mediante lo strumento *Tigress*.

Compilazione file offuscati

Dopo aver offuscato il programma che si sta analizzando con lo strumento *Tigress*, si passa alla fase di compilazione del programma offuscato tramite il richiamo alla funzione *Create_compiled*. La compilazione risulta necessaria per controllare se l'offuscamento è avvenuto correttamente o meno.

Dopo che il programma è stato compilato, si controlla che il valore di ritorno del programma offuscato sia uguale al valore di ritorno del programma non offuscato (i valori di ritorno vengono confrontati con lo stesso comando eseguito):

- in caso positivo, il file binario viene salvato nella relativa cartella di destinazione (cfr. Sezione 4.4.2);
- in caso negativo, il file binario viene eliminato, viene riportato nei file di log tale azione, ed il nome del file viene memorizzato in un file apposito in modo che, in fase di creazione dei file CSV contenenti il valore delle metriche estratte (cfr. Sezione 5.2.2), si potrà riportare che non è stato possibile estrarre le metriche da tale file poiché non è stato offuscato correttamente mediante lo strumento *Tigress*;
- infine verrà chiamata la funzione *Delete_output_created* utilizzata per eliminare eventuali file di output creati dall'esecuzione del comando eseguito.

Report Frama-c (offuscamento Tigress)

Nel caso in cui la fase di compilazione dei file offuscati (cfr. Sezione 5.2.1) abbia avuto esito positivo, tramite la funzione *Create_report_frama_c*, vengono generati i report dello strumento *Frama-c* contenenti i valori delle metriche calcolati sui file sorgenti e offuscati (tramite strumento *Tigress*) per ogni file *.c* appartenente al programma che si sta analizzando. I report generati verranno salvati nella relativa cartella di destinazione (cfr. Sezione 4.4.2).

Report Diablo (offuscamento Tigress)

Nel caso in cui la fase di compilazione dei file offuscati (cfr. Sezione 5.2.1) abbia avuto esito positivo tramite la funzione *Run_Diablo*, vengono generati i report contenenti i valori delle metriche calcolati sui file binari e offuscati (tramite strumento *Tigress*). I report generati verranno salvati nella relativa cartella di destinazione (cfr. Sezione 4.4.2).

Successivamente, viene chiamata la funzione *Delete_output_created* utilizzata per eliminare eventuali file di output creati dall'esecuzione del comando eseguito.

Cleaning

Una volta terminata l'analisi di tutti i programmi, si procede alla cancellazione dei dati non ritenuti necessari. In base ai parametri settati nel file di configurazione dello script *Metrics_analysis.py* (cfr. Appendice B.1.4), si decide quali cartelle del file-system di output generato (cfr. Sezione 4.4.2) devono essere rimosse o meno.

5.2.2 Creazione file CSV

Prima di terminare l'esecuzione dello script *Metrics_analysis.py* vengono lanciati due ulteriori script per raccogliere i dati raccolti, in modo ordinato, all'interno di file con estensione *.csv*.

Script create_frama_csv.py

Lo scopo di questo script è creare un file con estensione *.csv*, per ogni programma analizzato, contenente tutte le metriche estratte dallo strumento *Frama-c*. Il file *.csv* creato conterrà i parametri descritti in Sezione B.2.3.

L'esecuzione dello script risulta essere la seguente:

1. analizzare tutti i report dei programmi sorgenti non offuscati e raccogliere i dati in essi contenuti all'interno del file *.csv* che si sta creando;
2. analizzare i report dei programmi sorgenti offuscati (offuscati dallo strumento *Tigress*) e raccogliere i dati in essi contenuti all'interno del file *.csv* che si sta creando;

In questo script viene anche utilizzato il file descritto in Sezione 5.2.1 ed in Sezione 5.2.1 per riportare su quali funzioni non è stato possibile calcolare il valore delle metriche estratte dallo strumento *Frama-c* poiché l'offuscamento non ha avuto esito positivo. Nel file *.csv* generato, il campo chiamato *flag_obf_tigress(0=made_1=not)* (cfr. Sezione B.2.3) potrà contenere i seguenti valori: 0 se l'offuscamento è avvenuto o 1 in caso contrario.

Script create_diablo_csv.py

Lo scopo di questo script è creare un file con estensione *.csv*, per ogni programma analizzato, contenente tutte le metriche estratte dallo strumento *Diablo*. Il file *.csv* creato conterrà i parametri descritti in Sezione B.2.3.

L'esecuzione dello script risulta essere la seguente:

1. analizzare i report dei file binari non offuscati e raccogliere i dati in essi contenuti all'interno del file *.csv* che si sta creando;
2. analizzare i report dei file binari offuscati (offuscati dagli strumenti *Tigress* e *Diablo*) e raccogliere i dati in essi contenuti all'interno del file *.csv* che si sta creando;

In questo script viene anche utilizzato il file descritto in Sezione 5.2.1 ed in Sezione 5.2.1 per riportare su quali funzioni non è stato possibile calcolare il valore delle metriche estratte dallo strumento *Diablo* poiché l'offuscamento non ha avuto esito positivo. Nel file *.csv* generato, il campo chiamato *flag_obf_tigress(0=made_1=not)* (cfr. Sezione B.2.3) potrà contenere i seguenti valori: 0 se l'offuscamento è avvenuto o 1 in caso contrario.

Capitolo 6

Analisi dei dati

In questo capitolo si descrivono le metodologie con le quali si è deciso di analizzare i dati estratti e raccolti nei due file *csv* generati da *Metrics_analysis.py*. Nella Sezione B.2 vengono riportati maggiori dettagli tecnici.

6.1 Media

Lo script *media_dev_std.py*, situato nella cartella *Metrics_analysis/sub/utils/media_devstd/*, permette di calcolare la media dai file *csv*, contenuti nella cartella *pre-analized*, generati dallo script *Metrics_analysis.py*.

Lo script *media_dev_std.py* accetta esattamente due parametri da linea di comando:

```
./media_dev_std.py [source] [destination]
```

- `[source]`: rappresenta il path alla cartella che contiene i file che si vogliono analizzare (la cartella *pre-analized* generata da *Metrics_analysis.py*);
- `[destination]`: rappresenta la cartella in cui si vuole memorizzare l'output prodotto.

All'interno della cartella `[destination]` verrà creata la sotto-cartella *result_media_dev* (Figura 6.1). Questa cartella conterrà tante sotto-cartelle quanti saranno i programma analizzati da *Metrics_analysis.py* e per ognuna di queste sotto cartelle saranno presenti due file *csv*: uno contenente i valori della media riferiti all'intero programma analizzato ed uno contenente i valori della media raggruppati e suddivisi per comando *target* del Makefile.

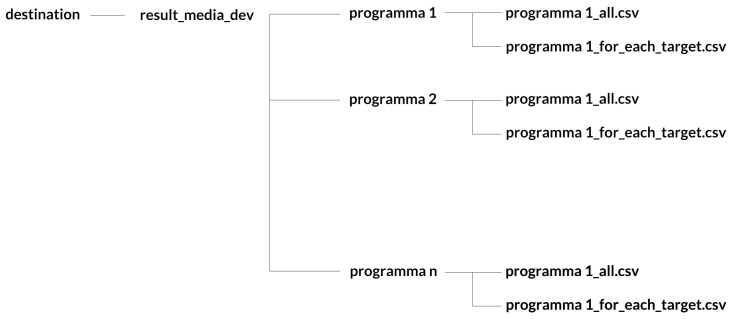


Figura 6.1: Esempio output generato dallo script *media_dev_std.py*.

L'header di riferimento per la creazione di questi file CSV risulta essere composto dai seguenti campi:

1. protection: l'offuscamento sul quale sono stati calcolati i valori, *vanilla* se i valori sono stati calcolati sul valore delle metriche in chiaro;
2. tool: lo strumento con il quale si sono calcolate le metriche utilizzate per il calcolo dei valori (*Tigress* o *Diablo*);
3. target: comando target del Makefile, altrimenti *all* se i valori sono stati calcolati su tutti i comandi target del Makefile senza alcuna distinzione;
4. metric: il nome della metrica su cui si sono calcolati i valori;
5. media: il valore della media aritmetica [27] [28];
6. num_samples_m: il numero di campioni utilizzati per calcolare la media aritmetica.

6.2 Calcolo potency

Calcolando le *potency* (cfr. Sezione 2.3) sul valore delle metriche estratte si può effettuare uno studio sulla possibile presenza di ulteriori metriche (oltre alla complessità ciclomatica ed alla lunghezza di Halstead) da poter utilizzare nel calcolo della *potency combinata* (cfr. Sezione 2.4). Gli studi effettuati vengono riportati in Sezione 7.3.1.

Lo script *Potency.py*, situato nella cartella *Metrics_analysis/sub/utils/potency*, permette di calcolare la potency sulle metriche estratte dagli strumenti *Frama-c* e *Diablo*, la potency combinata sulle metriche estratte dallo strumento *Diablo* e per ognuna di queste metriche calcola a sua volta la media, la varianza, la deviazione standard e l'errore standard.

Lo script *Potency.py* accetta esattamente due parametri da linea di comando:

```
./Potency.py [source] [destination]
```

- `[source]`: rappresenta il path alla cartella che contiene i file che si vogliono analizzare (la cartella *pre-analized* generata da *Metrics_analysis.py*);
- `[destination]`: rappresenta la cartella in cui si vuole memorizzare l'output prodotto.

All'interno della cartella `[destination]` verrà creata la sotto-cartella *potency* (Figura 6.2). Questa cartella conterrà tante sotto-cartelle quanti saranno i programma analizzati dallo script *Metrics_analysis.py* e per ognuna di queste sotto cartelle saranno presenti sei file *csv*:

- un file contenente la potency calcolata sulle metriche estratte dallo strumento *Frama-c*;
- un file contenente la media, la varianza, la deviazione standard e l'errore standard relativi alla potency calcolata sulle metriche estratte dallo strumento *Frama-c*;
- un file contenente la potency calcolata sulle metriche estratte dallo strumento *Diablo*;
- un file contenente la media, la varianza, la deviazione standard e l'errore standard relativi alla potency calcolata sulle metriche estratte dallo strumento *Diablo*;
- un file contenente la potency combinata;
- un file contenente la media, la varianza, la deviazione standard e l'errore standard relativi alla potency combinata calcolata.

L'header di riferimento utilizzato per la creazione dei file

```
[programma analizzato]_potency_function_protection_frama.csv
```

risulta essere composto dai seguenti campi:

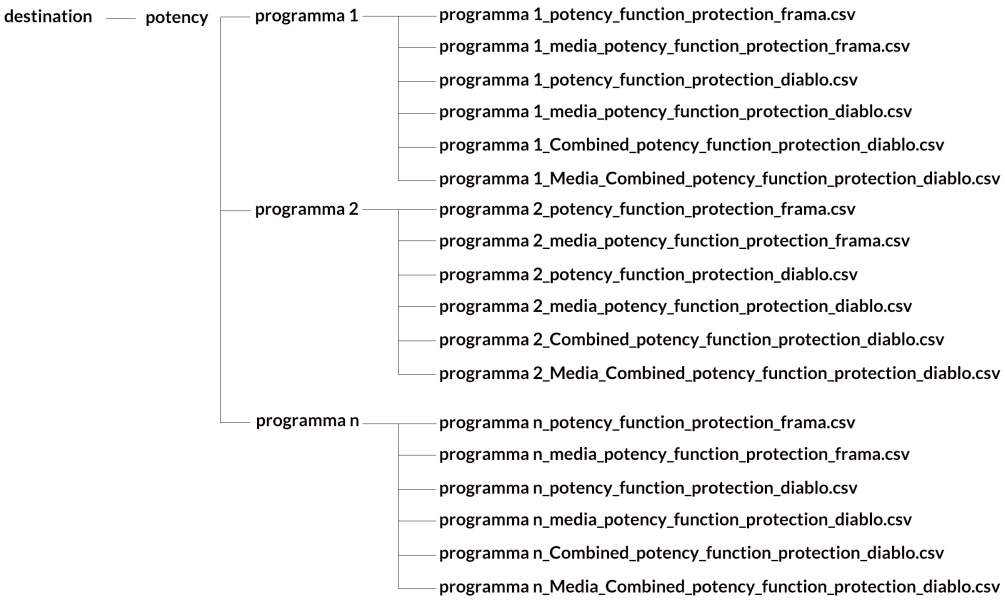


Figura 6.2: Esempio output generato dallo script *Potency.py*.

1. protection: l'offuscamento sul quale sono stati calcolati i valori, *vanilla* se i valori sono stati calcolati sul valore delle metriche in chiaro;
2. target: comando target del Makefile, altrimenti *all* se i valori sono stati calcolati su tutti i comandi target del Makefile senza alcuna distinzione;
3. path_file_from_main_directory_program: path relativo dalla cartella principale del programma analizzato al file *.c* preso in considerazione;
4. function: il nome della funzione analizzata;
5. flag_obf_tigress(0=made_1=not): flag per il controllo dell'avvenuto offuscamento tramite lo strumento *Diablo*. Se posto a 0 l'offuscamento è avvenuto, se posto a 1 no (cfr. Sezione 5.2.1, Sezione 5.2.1, Sezione 5.2.2);
6. Sloc: valore metrica estratta da *Frama-c*, indica il numero di linee di codice;
7. Decision point: valore metrica estratta da *Frama-c*, indica il numero di punti decisionali come per esempio *if*, *for*, *while*;
8. Global variables: valore metrica estratta da *Frama-c*, indica il numero di varibili globali;
9. If: valore metrica estratta da *Frama-c*, indica il numero di *if*;
10. Loop: valore metrica estratta da *Frama-c*, indica il numero di *loop*;
11. Goto: valore metrica estratta da *Frama-c*, indica il numero di *goto*;
12. Assignment: valore metrica estratta da *Frama-c*, indica il numero di assegnazioni;
13. Exit point: valore metrica estratta da *Frama-c*, indica il numero di *exit point*;
14. Function: valore metrica estratta da *Frama-c*, indica il numero di funzioni contenute nel programma[1];
15. Function call: valore metrica estratta da *Frama-c*, indica il numero di funzioni chiamate;

[1]Nel caso si analizzasse il programma a livello di funzione, tale parametro ritorna il valore 1

16. Pointer dereferencing: valore metrica estratta da *Frama-c*, indica numero di *pointer dereferencing*[2];
17. Cyclomatic complexity: valore metrica estratta da *Frama-c*, indica il valore della complessità ciclomatica.

L'header di riferimento dei file

`[programma analizzato]_potency_function_protection_diablo.csv`

risulta essere composto dai seguenti campi:

1. protection: l'offuscamento sul quale sono stati calcolati i valori, *vanilla* se i valori sono stati calcolati sul valore delle metriche in chiaro;
2. target: comando target del Makefile, altrimenti *all* se i valori sono stati calcolati su tutti i comandi target del Makefile senza alcuna distinzione;
3. path_file_from_main_directory_program: path relativo dalla cartella principale del programma analizzato al file *.c* preso in considerazione;
4. function: il nome della funzione analizzata;
5. flag_obf_tigress(0=made_1=not): flag per il controllo dell'avvenuto offuscamento tramite lo strumento *Diablo*. Se posto a 0 l'offuscamento è avvenuto, se posto a 1 no (cfr. Sezione 5.2.1, Sezione 5.2.1, Sezione 5.2.2);
6. nr_ins_static: valore metrica estratta da *Diablo*, indica il numero di istruzioni assembly;
7. nr_src_oper_static: valore metrica estratta da *Diablo*, indica il numero di operandi sorgente[3];
8. nr_dst_oper_static: valore metrica estratta da *Diablo*, indica il numero di operandi destinazione[4];
9. halstead_program_size_static: valore metrica estratta da *Diablo* , indica la lunghezza di Halstead;
10. nr_edges_static: valore metrica estratta da *Diablo*, indica il numero ramificazioni del Control Flow Graph (CFG);
11. nr_indirect_edges_CFIM_static: valore metrica estratta da *Diablo*, indica il numero di ramificazioni indirette (cfr. Sezione 1.1.1);
12. cyclomatic_complexity_static: valore metrica estratta da *Diablo*, indica il valore della complessità ciclomatica.

Per entrambi i file appena descritti i campi *target* e *path_file_from_main_directory_program* sono stati riportati per mantenere la distinzione tra funzioni aventi lo stesso nome e compilate con *target* del Makefile differenti.

L'header di riferimento utilizzato per la creazione dei file chiamati *[programma analizzato]_Combined_potency_function_protection_diablo.csv* risulta essere composto dai seguenti campi:

1. protection: l'offuscamento sul quale sono stati calcolati i valori, *vanilla* se i valori sono stati calcolati sul valore delle metriche in chiaro;

[2]Il numero di puntatori utilizzati all'intero della funzione.

[3]Gli operandi presi come input da un'istruzione.

[4]Gli operandi presi come output da un'istruzione.

2. target: comando target del Makefile, altrimenti *all* se i valori sono stati calcolati su tutti i comandi target del Makefile senza alcuna distinzione;
3. path_file_from_main_directory_program: path relativo dalla cartella principale del programma analizzato al file *.c* preso in considerazione;
4. function: il nome della funzione analizzata;
5. combined_Potency: valore della potency combinata (cfr. Sezione 2.4).

L'header di riferimento utilizzato per la creazione dei file chiamati *[programma analizzato]_Media_Combined_potency_protection_diablo.csv*, *[programma analizzato]_media_potency_protection_diablo.csv*, *[programma analizzato]_media_potency_protection_frama.csv* risulta essere composto dai seguenti campi:

1. protection: l'offuscamento sul quale sono stati calcolati i valori, *vanilla* se i valori sono stati calcolati sul valore delle metriche in chiaro;
2. tool: lo strumento con il quale si sono calcolate le metriche utilizzate per il calcolo dei valori (*Tigress* o *Diablo*);
3. target: comando target del Makefile, altrimenti *all* se i valori sono stati calcolati su tutti i comandi target del Makefile senza alcuna distinzione;
4. metric: il nome della metrica su cui si sono calcolati i valori;
5. media: il valore della media aritmetica [27] [28];
6. num_samples_m: il numero di campioni utilizzati per calcolare la media aritmetica.

Per entrambi i file appena descritti il campo *path_file_from_main_directory_program* è stato riportato per mantenere la distinzione tra funzioni aventi lo stesso nome.

Capitolo 7

Risultati sperimentali

In questo capitolo si descrivono i test eseguiti per mezzo dei valori estratti dallo script *Metrics_analysis.py* (cfr. Capitoli 4 e 5). Nello specifico, si tratterà degli studi effettuati sulle *fingerprint* e sulla potency (cfr. Sezione 6.2) riportando le analisi emerse di maggiore rilievo.

7.1 Dataset

Per mezzo dello script *Metrics_analysis.py* si è potuto estrarre ed analizzare una grande quantità di dati, in particolare:

- programmi analizzati: 15;
- file *.c* analizzati: 51;
- funzioni analizzate: 587.

Inoltre ogni programma analizzato:

- è stato offuscato con 4 protezioni differenti a livello di file sorgente: *control flow flattening*, *anti branch analysis*, *predicati opachi* ed *encoding*;
- per ogni funzione a livello di file sorgente, protetta e non protetta, sono state estratte 12 metriche;
- è stato offuscato con 9 protezioni differenti a livello di file binario: *control flow flattening*, *anti branch analysis* e *predicati opachi*. Ognuna delle quali applicata con tre diverse percentuali di applicazione: 10%, 30% e 50%;
- per ogni funzione a livello di file binario, protetta e non protetta, sono state estratte 7 metriche.

7.2 Fingerprint

Il termine *fingerprint* (cfr. Sezione 4.1.1) viene utilizzato per indicare un pattern di codice riconoscibile o un comportamento particolare assunto da un programma in esecuzione [11].

Nei test di seguito riportati, si è cercato di dimostrare empiricamente la presenza delle *fingerprint* analizzando le metriche di complessità estratte dai programmi protetti.

Dagli studi effettuati è emerso che ogni offuscamento riporta dei cambiamenti specifici su determinate metriche di interesse. Tuttavia è anche emerso che alcune protezioni risultano più

difficili da individuare rispetto ad altre: alcune protezioni contengono delle *fingerprint* identificabili per mezzo di determinate metriche di complessità, al contrario altre protezioni non riportano *fingerprint* molto evidenti e pertanto risultano particolarmente complicate da rilevare ed identificare. Inoltre si è dimostrato come non tutte le metriche rappresentano delle *fingerprint* di riferimento: ad esempio il numero di linee di codice non può essere utilizzato come metrica per risalire all'offuscamento apportato in quanto non rappresenta un valore di riferimento costante.

7.2.1 Fingerprint analizzate

Per poter verificare la presenza delle *fingerprint* sono state calcolate, raggruppate per protezione, le medie delle metriche estratte dallo script *Metrics_analysis.py*. Alcune di queste medie calcolate sono state riportate nelle Tabelle 7.1 e 7.2, suddivise in base agli strumenti di estrazione utilizzati: *Frama-c* e *Diablo* rispettivamente. Analizzando i valori contenuti nelle tabelle, si può notare come alcuni valori di specifiche metriche possano rivelare il potenziale utilizzo di una determinata protezione: nella Tabella 7.1 la media del numero di `loop` associati alla protezione *control flow flattening* risulta molto vicina al valore 1 e risulta essere la media più bassa rispetto alle medie delle stessa metrica calcolate su differenti protezioni; nella Tabella 7.2 invece si può notare che il maggior valore della media calcolata sulla metrica che rappresenta il numero di istruzioni assembly risulta essere associata alla protezione che utilizza i predicati opachi al 50%. Successivamente verrà riportato un studio effettuato su questi valori per determinare se e quali di questi possano effettivamente essere utilizzati per identificare delle *fingerprint*. Inoltre dai valori riportati nelle

	Control flow flattening	Predicati opachi	Anti branch analysis	Encoding
Complessità ciclomatica	21,46	35,01	57,38	13,62
Numero di `loop`	1,35	1,49	1,38	1,46
Linee di codice	91,37	162,76	263,74	66,34
Numero di istruzioni `if`	12,10	33,62	55,89	12,10
Numero di assegnazioni	27,69	70,39	129,46	18,46

Tabella 7.1: Media del valore di alcune metriche estratte tramite lo strumento *Frama-c*.

	Control flow flattening al 50%	Predicati opachi al 50%	Anti branch analysis al 50%
Complessità ciclomatica	43,45	34,77	49,21
Numero di istruzioni assembly	353,99	431,86	328,90
Numero di ramificazioni del control flow graph	33,3	96,42	74,23
Halstead	1183,52	1474,50	1076,11

Tabella 7.2: Media del valore di alcune metriche estratte tramite lo strumento *Diablo*.

Tabelle 7.1 e 7.2 si evince anche che non tutte le protezioni analizzate possono essere rilevate per mezzo delle *fingerprint*. Ad esempio per la protezione *encoding*, applicata a livello di file sorgente, nessuna metrica tra quelle estratte dallo strumento *Frama-c* risulta essere altamente significativa per poter risalire a tale protezione. Allo stesso modo anche la protezione *control flow flattening al 50%* risulta difficile da identificare per mezzo delle metriche estratte dallo strumento *Diablo* poiché nessuna tra quelle riportate fornisce un valore altamente rilevante da potersi definire *fingerprint*.

Per effettuare i test sull'identificazione di pattern ricorrenti sulle metriche calcolate si è creato un algoritmo in grado di:

1. analizzare tutti i file generati dallo script *Metrics_analysis.py* (cfr. Sezioni B.2.3 e B.2.3);

2. controllare se ogni funzione *vanilla* analizzata rispetta dei vincoli[1] sulle metriche ad essa associate: nel caso i vincoli non siano rispettati, la funzione non protetta e la stessa funzione protetta con i vari offuscamenti apportati dagli strumenti *Tigress* e *Diablo* non viene considerata per il calcolo delle *fingerprint*. Questo passaggio viene effettuato per eliminare la presenza di possibili outliers;
3. generare, per ogni offuscamento apportato dagli strumenti *Tigress* e *Diablo*, tanti grafici quante sono le metriche calcolate su tale offuscamento.

Per le metriche calcolate da *Frama-c* si è deciso di applicare i vincoli solamente sul numero di linee di codice: funzioni troppo piccole potrebbero risultare prive di significato, funzioni troppo grandi potrebbero essere generate da strumenti automatici per la generazione di codice. Per lo stesso motivo appena riportato, i vincoli imposti per le metriche calcolate dallo strumento *Diablo* risultano essere sia sul numero di istruzioni assembler sia sul numero di ramificazioni del CFG.

Per ogni grafico generato, la rappresentazione degli assi risulta essere:

- asse x: sull'asse x vengono rappresentati i valori assunti dalla metrica presa come riferimento;
- asse y: sull'asse y viene rappresentato il numero di occorrenze del valore della metrica.

Tutti i grafici generati sono stati riportati in Appendice D. Di seguito vengono riportate le casistiche più interessanti emerse durante gli studi effettuati sull'identificazione delle *fingerprint*.

Control flow flattening

La Figura 7.1 mostra il grafico generato per rappresentare il numero di occorrenze legato al numero di cicli (cfr. Sezione 1.2.1), calcolata dallo strumento *Frama-c*, in funzione del valore della stessa metrica apportata dalla protezione *control flow flattening*, applicata dallo strumento *Tigress*. Le curve rappresentanti la protezione *vanilla* e la protezione *control flow flattening* risultano molto simili nel loro andamento. Tuttavia nella parte sinistra del grafico si può notare come il picco della curva *control flow flattening* superi il picco della curva *vanilla*. Entrambe le curve raggiungono il loro picco in corrispondenza del valore 1 per la metrica presa in considerazione, ed in particolare si può osservare che:

- per la protezione *vanilla* al valore della metrica 1 corrispondono 129 occorrenze;
- per la protezione *control flow flattening* al valore della metrica 1 corrispondono 189 occorrenze.

Essendo la protezione *control flow flattening* (cfr. Sezione 2.2.2) una protezione che raggruppa i singoli basic block (cfr. Sezione 1.1.1) all'interno di un unico ciclo, come ci si sarebbe potuto aspettare, dopo avere applicato tale protezione è presente un significativo aumento del numero di funzioni (occorrenze) in funzione del valore 1 della metrica rappresentante il numero di cicli. Si può quindi affermare che la metrica `loop` rappresenta una *fingerprint* per la protezione *control flow flattening*.

Anti branch analysis

Lo scopo della protezione *anti branch analysis* (cfr. Sezione 2.2.3) è rendere più complicato per un deoffuscatore determinare quale sarà la prossima istruzione da eseguire. La Figura 7.2 mostra il grafico generato per rappresentare il numero di occorrenze dell'istruzione `if` (cfr. Sezione 1.2.1), calcolata dallo strumento *Frama-c*, in funzione del valore della stessa metrica apportata dalla protezione *anti branch analysis*, applicata dallo strumento *Tigress*.

[1] I vincoli sono dei parametri di massimo e di minimo decidibili dall'utente.

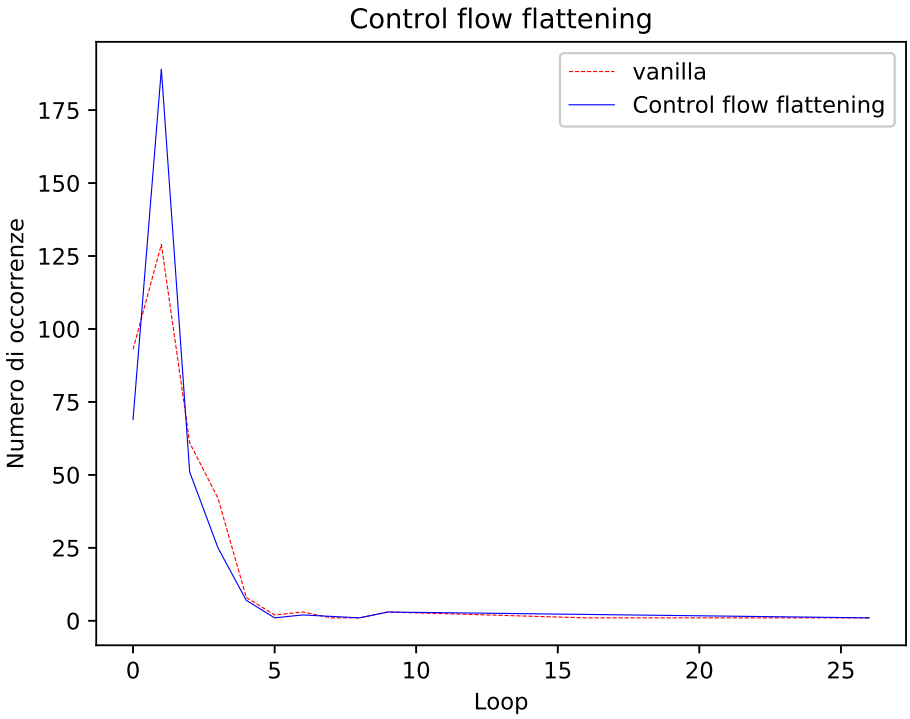


Figura 7.1: Grafico del numero di occorrenze per la metrica `loop` calcolata sulla protezione *control flow flattening*.

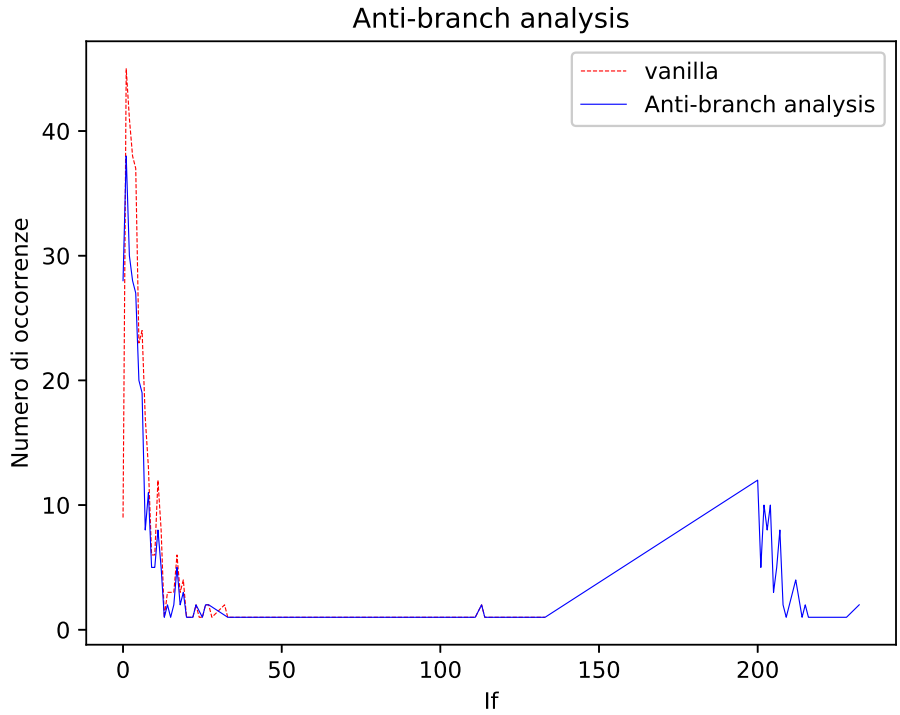


Figura 7.2: Grafico del numero di occorrenze per la metrica `if` calcolata sulla protezione *anti branch analysis*.

La curva che rappresenta la funzione *vanilla* risulta schiacciata nella parte sinistra di tale grafico: ciò significa che ci sono molto funzioni che contengono al loro interno nessuna o poche istruzioni condizionali `if`. La curva che rappresenta la funzione *anti branch analysis* nella parte sinistra del grafico segue lo stesso andamento della curva *vanilla*, ma si può notare nella parte destra del grafico la presenza di un secondo picco, raggiunto nel punto in cui il numero di istruzioni `if` assume il valore 200 con 19 occorrenze. In prossimità di questo secondo picco si può osservare un elevato numero di funzioni aventi un numero di istruzioni `if` maggiore uguale a 200 che nella curva *vanilla* sono del tutto assenti. Tale variazione indica che l'applicazione dell'offuscamento analizzato ha introdotto ulteriori istruzioni `if` e pertanto tale metrica rappresenta una *fingerprint* utile per il riconoscimento della protezione *anti branch analysis*.

Predicati opachi al 50%

La Figura 7.3 mostra il grafico generato per rappresentare il numero di occorrenze della metrica che rappresenta il numero di istruzioni assembly (cfr. Sezione 1.2.8), calcolata dallo strumento *Diablo*, in funzione del valore della stessa metrica apportata dalla protezione che utilizza i *predicati opachi al 50%*, applicata anch'essa dallo strumento *Diablo*.

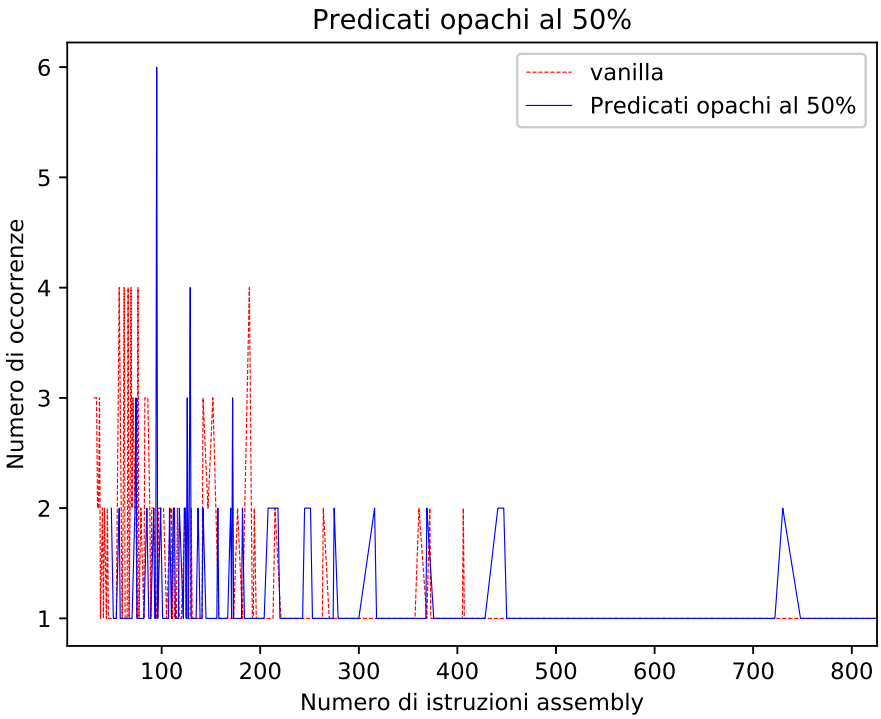


Figura 7.3: Grafico del numero di occorrenze per la metrica *nr_ins_static* calcolata sull'utilizzo dei *predicati opachi al 50%*.

Si può osservare come la curva che identifica la protezione associata all'utilizzo dei *predicati opachi al 50%* presenti un picco non presente nella curva che rappresenta la funzione *vanilla* in corrispondenza del valore 95 e come la stessa curva associata alla protezione che utilizza i *predicati opachi al 50%* risulti spostata a destra rispetto alla curva *vanilla*. Questo spostamento indica appunto che c'è stato un incremento delle istruzioni assembly su tutte le funzioni analizzate.

Lo scopo della protezione che utilizza i *predicati opachi al 50%* (cfr. Sezione 2.2.1) è rendere più complicato per un deoffuscatore calcolare il valore di una determinata variabile o di un predicato invece noti all'offuscatore. La metrica che rappresenta il numero di istruzioni assembly risulta quindi essere una buona metrica di riferimento per identificare questa specifica protezione poiché, una volta applicata, il numero di istruzioni assembly aumenta proprio in funzione di rendere più

complicato risalire al valore iniziale delle variabili o dei predicati noti a priori all'offuscatore, ma non ne rappresenta una *fingerprint* data l'assenza di un pattern significativo in grado di identificare univocamente tale protezione.

Anti branch analysis al 50%

Come precedentemente spiegato nella Sezione 7.2.1, lo scopo della protezione *anti branch analysis* (cfr. Sezione 2.2.3) è rendere più complicato per un deoffuscatore determinare quale sarà la prossima istruzione da eseguire. La Figura 7.4 mostra il grafico generato per rappresentare il numero di occorrenze della complessità ciclomatica (cfr. Sezione 2.2.1), calcolata dallo strumento *Diablo*, in funzione del valore della stessa metrica apportata dalla protezione *anti branch analysis al 50%*, applicata anch'essa dallo strumento *Diablo*.

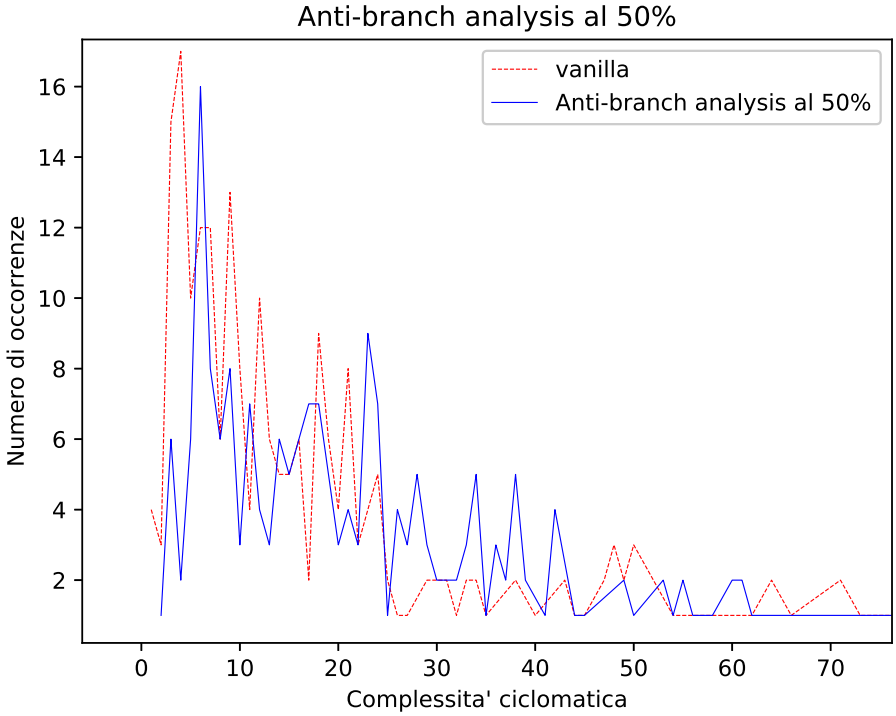


Figura 7.4: Grafico del numero di occorrenze per la complessità ciclomatica calcolata sulla protezione *anti branch analysis al 50%*.

Nella parte sinistra del grafico, compresa tra i valori 0 e 25, si può notare come la curva che rappresenta la protezione *anti branch analysis al 50%* sia leggermente traslata a destra rispetto alla curva rappresentante funzione *vanilla*. Nella parte destra del grafico invece, compresa tra i valori 25 e 70, si può notare che la curva associata alla protezione *anti branch analysis al 50%* sia traslata verso l'alto rispetto alla funzione *vanilla*. Le traslazioni verso destra e verso l'alto della curva *anti branch analysis al 50%* indicano un aumento della complessità ciclomatica dopo l'applicazione di tale protezione e pertanto tale metrica rappresenta una buona metrica di riferimento per poter identificare la protezione *anti branch analysis al 50%*. Tuttavia tale metrica non rappresenta una *fingerprint* data la mancanza di un pattern significativo che possa permettere di identificare univocamente la protezione analizzata.

7.3 Potency

La potency (cfr. Sezione 2.3) è una metrica che fornisce un valore oggettivo di quanto un dato offuscamento protegga il codice sorgente originale mettendo a confronto il valore di una data

metrica post offuscamento rispetto alla stessa metrica calcolata sulla funzione *vanilla*. In merito all'utilizzo di questa metrica sono stati eseguiti due casi di studio (cfr. Sezione 4.1.2):

- `metriche utili al calcolo della potency`: dimostrare la presenza di ulteriori metriche utili al calcolo della *potency* partendo dal presupposto che le uniche metriche ad ora ritenute di interesse risultano essere la complessità ciclomatica e le metriche di Halstead (cfr. Sezione 7.3.1);

- `livello di protezione legato alle potency`: dimostrare l'efficacia delle *potency* come strumento in grado di indicare quale offuscamento, tra quelli disponibili, risulta applicare un livello di sicurezza maggiore (cfr. Sezione 7.3.2).

Dal primo caso di studio effettuato, non è emersa alcuna nuova metrica come possibile metrica di riferimento per il calcolo della *potency*. Tuttavia ha però posto l'attenzione sulla possibilitá di poter utilizzare altre metriche al posto della complessità ciclomatica e di Halstead: le metriche in questione risultano essere direttamente collegate a quelle appena citate e viene richiesto un *effort* minore per il loro calcolo.

Dal secondo caso di studio invece, si è potuto dimostrare come la *potency* risulti essere un ottimo valore per stimare il livello di protezione associato alle protezioni applicate, a patto che si utilizzino delle metriche di riferimento coerenti al tipo di sicurezza che si vuole misurare.

7.3.1 Metriche utili al calcolo della *potency*

Per verificare l'eventuale presenza di ulteriori metriche utili al calcolo della *potency* (cfr. Sezione 4.1.2), si è pensato di confrontare i diversi andamenti delle potency calcolate sulle metriche estratte dagli strumenti di *Frama-c* e *Diablo* in funzione delle potency legate appunto alla complessità ciclomatica e lunghezza di Halstead (calcolate sui file binari), queste ultime ordinate per valori crescenti.

Per effettuare questo studio è stato implementato un algoritmo in grado, per ogni protezione utilizzata, di:

1. calcolare le potency su tutte le metriche estratte dallo script *Metrics_analysis.py* per ogni funzione analizzata;

2. ordinare le potency calcolate su complessità ciclomatica e lunghezza di Halstead (calcolate sui file binari) con valore crescente rispetto alle funzioni analizzate;

3. generare grafici per mettere a confronto l'andamento della potency legato alla complessità ciclomatica o alla lunghezza di Halstead con le altre metriche estratte, mantenendo inalterato l'ordine delle funzioni spiegato nel punto precedente.

Si riporta una precisazione: durante il calcolo delle potency, per ogni funzione analizzata, se il valore della metrica presa in considerazione risulta uguale a 0 prima e dopo l'applicazione di un determinata protezione, di default è stato settato a 0 anche il valore della potency ad essa associato. Questo perché non risulta essere avvenuto né un aumento né un decremento del grado di sicurezza. Nel caso in cui, invece, la metrica calcolata sul programma offuscato risulti diversa da 0 ma la stessa metrica calcolata sul programma senza alcuna protezione applicata sia uguale a 0 è stato riportato il valore "NaN"[2].

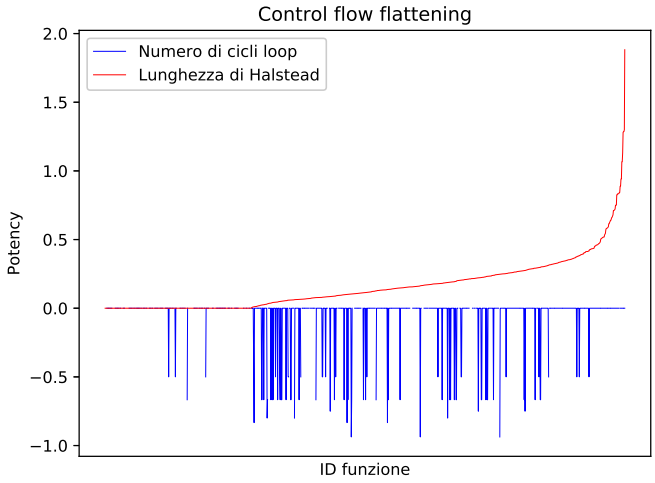
[2]Not a Number.

Potency di riferimento calcolate su metriche estratte dallo strumento *Frama-c*

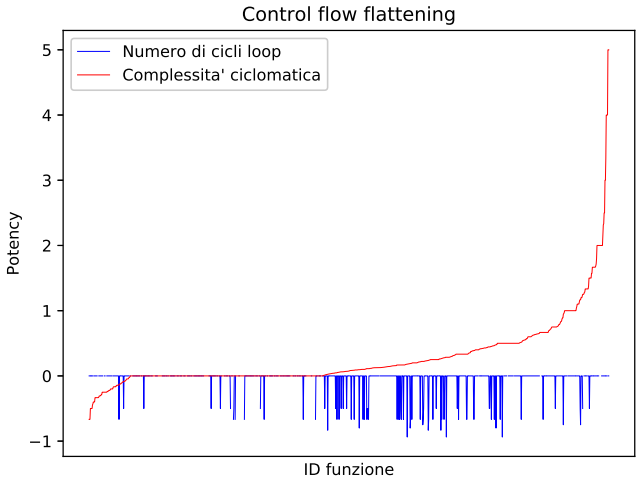
Per questo caso di studio non è stato possibile mettere a confronto le *potency* calcolate sulle metriche estratte dai file protetti con l'offuscamento encoding data l'assenza della stessa protezione per lo strumento *Diablo*. Pertanto, il confronto delle *potency* risulta essere avvenuto sui seguenti offuscamenti: *control flow flattening*, *predicati opachi* e *anti branch analysis*.

Dai grafici generati non è emersa alcuna corrispondenza dal riscontro degli andamenti legati alle *potency* né per quelle confrontate sulla complessità ciclomatica né per quelle confrontate sulla lunghezza di Halstead. Tutti i grafici generati vengono riportati in Appendice E.1.1.

Tuttavia alcuni di questi grafici hanno riportato un comportamento particolare. La Figura 7.5 riporta i grafici generati per la *potency* calcolata sulla metrica che rappresenta il numero di `loop` posta a confronto con le *potency* calcolate sia per la complessità ciclomatica (cfr. Figura 7.5a) sia per la lunghezza di Halstead (cfr. Figura 7.5b). Come mostrato dai grafici, la *potency* legata



(a) Confronto tra *potency* in riferimento alla complessità ciclomatica.



(b) Confronto tra *potency* in riferimento alla lunghezza di Halstead.

Figura 7.5: Confronto della *potency* calcolata sul numero di `loop` per la protezione *control flow flattening*.

al numero di cicli `loop` risulta negativa. L'obiettivo della protezione del *control flow flattening* è

raggruppare i singoli basic block (cfr. Sezione 1.1.1) all'interno di un unico ciclo: per via di questa modifica la *potency* calcolata su questa metrica per questa protezione risulta negativa poiché indica la presenza di una diminuzione dei `loop`. È stata riscontrata la presenza di *potency* negative anche per la metrica che rappresenta il numero di istruzioni `goto` sia per la protezione *predicati opachi* sia per la protezione *anti branch analysis*: anche per queste casistiche risulta corretta la presenza di valori negativi perché sono correlate alle modifiche applicate dal tipo di protezione che si è voluto utilizzare. Non necessariamente, quindi, una potency negativa rappresenta una perdita di protezione: analizzando infatti le altre *potency* legate alle stesse protezioni prese in considerazione, ma calcolate su differenti metriche, si può osservare che esse risultano positive.

Successivamente, si è pensato di modificare leggermente l'algoritmo utilizzato per poter mettere a confronto le *potency* legate alle metriche estratte dallo strumento *Frama-c* con la *potency* legata alla complessità ciclomatica calcolata sui file sorgenti e non più sui file binari (lo strumento *Frama-c* non permette di estrarre la lunghezza di Halstead). In questa analisi si è potuto utilizzare anche l'offuscamento *encodig* in aggiunta agli altri offuscamenti. I grafici studiati sono stati riportati in Appendice E.1.2.

In Figura 7.6 viene riportato il grafico generato per la *potency* calcolata sul numero di punti decisionali per la protezione *predicati opachi*. Il grafico mette in risalto l'andamento molto simile

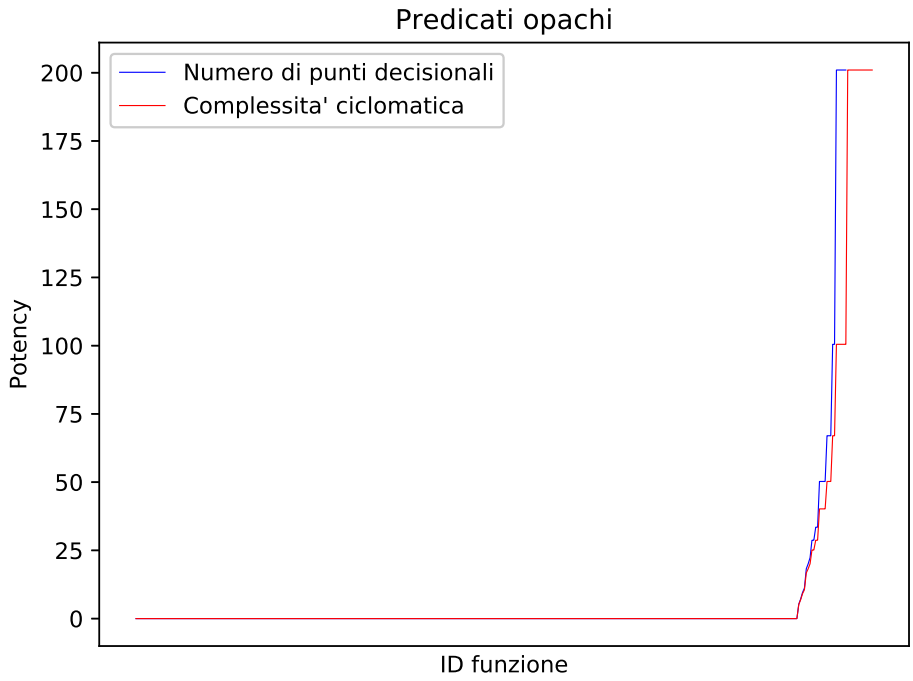


Figura 7.6: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di punti decisionali per la protezione *predicati opachi*.

delle due curve rappresentanti le *potency* e lo stesso comportamento si può osservare anche per tutti gli altri offuscamenti analizzati. La complessità ciclomatica rappresenta il numero di percorsi linearmente indipendenti di un programma. Il numero di punti decisionali rappresenta il numero di punti in cui il programma prende una decisione: a fronte di una decisione presa, il flusso del programma eseguirà uno specifico percorso. Le due curve risultano essere molto simili proprio perché esiste una correlazione tra queste due metriche. Essendo più semplice e veloce calcolare il valore della metrica che rappresenta il numero di punti decisionali si potrebbe dunque utilizzare tale valore come stima approssimativa della complessità ciclomatica calcolata a livello di file sorgente.

Potency di riferimento calcolate su metriche estratte dallo strumento *Diablo*

Durante l'implementazione dell'algoritmo utilizzato per analizzare questo caso di studio si è deciso, per ogni offuscamento applicato, di non tenere conto della diversa percentuale di protezione utilizzata. Gli offuscamenti analizzati risultano quindi essere: *control flow flattening*, *predicati opachi* e *anti branch analysis*. Tutti i grafici generati vengono riportati in Appendice E.2.

Dai grafici generati usando come metrica di riferimento la complessità ciclomatica non è emersa nessuna caratteristica particolare in quanto gli andamenti delle metriche sottoposte a confronto risultano completamente scorrelati dall'andamento della metrica utilizzato per il riferimento.

Utilizzando come metrica di riferimento la lunghezza di Halstead, invece, alcuni grafici hanno mostrato curve con andamenti molto simili tra loro. La Figura 7.7 mostra il grafico generato avendo come metrica di confronto il numero di istruzioni assembly per la protezione *anti branch analysis*. Come si può osservare dal grafico, le due curve mostrano andamenti analoghi. Essendo

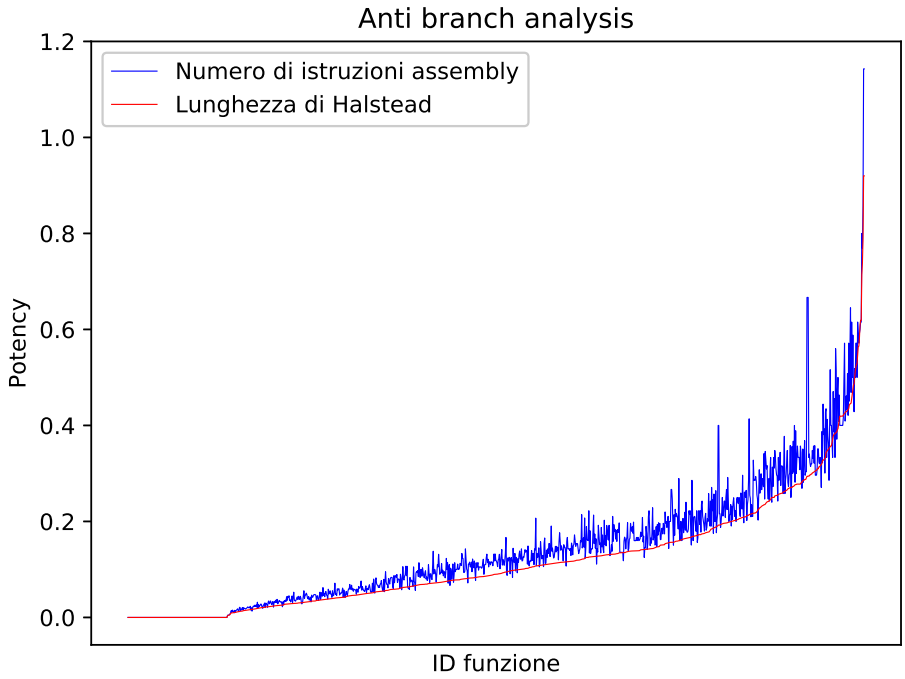
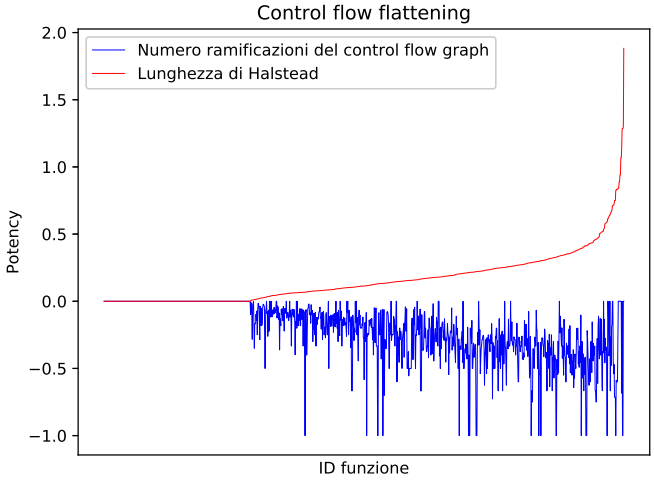


Figura 7.7: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di istruzioni assembly per la protezione *anti branch analysis*.

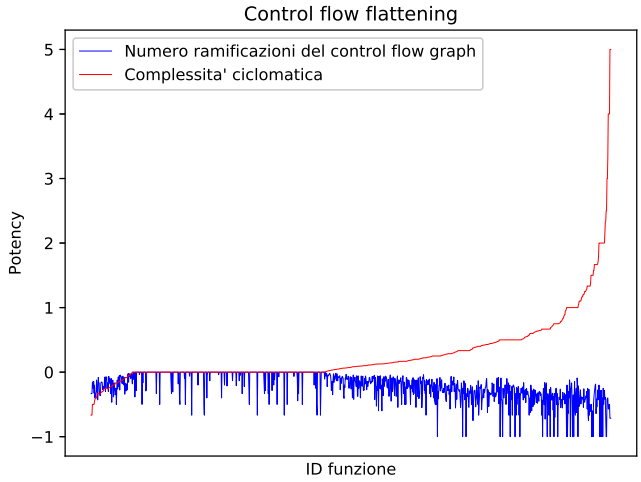
la metrica di Halstead calcolata come la somma tra il numero di istruzioni assembly ed il numero di operandi risulta dunque corretto che le curve assumano lo stesso andamento. Inoltre, lo stesso comportamento ripetuto per tutti gli offuscamenti analizzati, lo si può osservare anche nei grafici generati utilizzando come metrica di confronto il numero di operandi sorgente ed il numero di operandi destinazione. Anche in questi grafici, il simile andamento delle curve mostra semplicemente la correlazione tra queste metriche: il numero di operandi (utilizzato nel calcolo della lunghezza di Halstead) è ottenuto come somma del numero di operandi sorgenti e del numero di operandi destinazione. Tali grafici pongono l'attenzione sul fatto che, avendo le curve andamenti affini, si potrebbe pensare di utilizzare tali metriche (il numero di istruzioni assembly, il numero di operandi sorgente o destinazione) come stime approssimative per il calcolo della lunghezza di Halstead: risulta infatti molto più semplice e veloce calcolare il valore delle prime tre metriche citate piuttosto che sommarle per ottenere quest'ultima.

Un caso particolare è emerso studiando i grafici generati per la protezione *control flow flattening* utilizzando come metrica di confronto il numero di ramificazione del control flow graph ponendo

come metrica di riferimento sia la lunghezza di Halstead (cfr. Figura 7.8a) sia la complessità ciclomatica (cfr. Figura 7.8b). In questi grafici, gli andamenti delle curve risultano completamente



(a) Confronto tra *potency* in riferimento alla lunghezza di Halstead.



(b) Confronto tra *potency* in riferimento alla complessità ciclomatica.

Figura 7.8: Confronto della *potency* calcolata sul numero di ramificazioni del control flow graph per la protezione *control flow flattening*.

differenti tra di loro non riflettendo alcuna correlazione particolare. Tuttavia le *potency* calcolate sul numero di ramificazioni del control flow graph risultano negative. La protezione del *control flow flattening* è una protezione che raggruppa i singoli basic block (cfr. Sezione 1.1.1) all'interno di un unico ciclo: per via di questa modifica i basic block soggetti a tale offuscamento risulteranno avere un singolo arco in entrata ed uno in uscita al contrario di quanto possa essere in assenza di protezione (più ramificazioni in entrata e più ramificazioni in uscita). Risulta quindi corretto che la *potency* calcolata sulla metrica che rappresenta il numero di ramificazioni del control flow graph, per la protezione *control flow flattening*, sia negativa. Analizzando tuttavia le altre *potency* calcolate sulle ulteriori metriche estratte a seguito dell'utilizzo dello stesso offuscamento si nota che risultano tutte positive: anche in questo caso la *potency* negativa non rappresenta una perdita di protezione.

Dai grafici analizzati, quindi, si può dedurre che nessuna metrica risulta al pari della complessità ciclomatica e della lunghezza di Halstead. Tuttavia sono però emerse alcune relazioni

tra differenti tipi di metriche: le potency calcolate sulle metriche che rappresentano il numero di operandi sorgente, il numero di operandi destinazione e il numero di istruzioni assembly risultano essere ottime approssimazioni per la *potency* calcolata sulla lunghezza di Halstead.

7.3.2 Livello di protezione legato alle *potency*

Per poter decidere quale protezione risulta in grado di applicare il maggior livello di sicurezza, si è pensato di ricorrere all'utilizzo delle *potency*.

È stato creato un algoritmo in grado di calcolare, per ogni offuscamento e per ogni percentuale di utilizzo ad esso associato, la media della potency. In particolare tale algoritmo permette:

1. di analizzare tutti i file *[programma analizzato]_potency_function_protection_frama.csv* e *[programma analizzato]_potency_function_protection_diablo.csv* generati dallo script *Potency.py* (cfr. Sezione 6.2);
2. di calcolare la media della potency per ogni offuscamento apportato dagli strumenti di *Tigress* e *Diablo*;
3. di generare i grafici corrispondenti.

Un caso interessante è emerso osservando il grafico generato per la protezione del *control flow flattening* per la metrica che rappresenta il numero ramificazioni del control flow graph. Tale grafico è mostrato in Figura 7.9. Infatti in questo grafico le *potency* risultano essere negative e

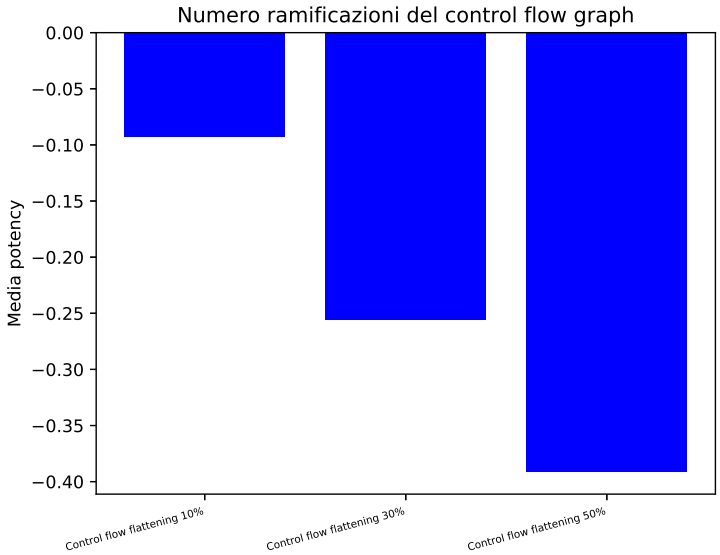


Figura 7.9: Grafico del numero di occorrenze per il numero di istruzioni assembly calcolato sulla protezione *control flow flattening*.

decrescenti al crescere della percentuale di applicazione di tale protezione. Il ragionamento risulta essere lo stesso già descritto nella Sezione 7.3.1 in quanto la protezione del *control flow flattening* è una protezione che raggruppa i singoli basic block (cfr. Sezione 1.1.1) all'interno di un unico ciclo: per via di questa modifica i basic block soggetti a tale offuscamento risulteranno avere un singolo arco in entrata ed uno in uscita al contrario di quanto possa essere in assenza di protezione (più ramificazioni in entrata e più ramificazioni in uscita). Risulta quindi corretto che la *potency* calcolata su questa metrica, per questa protezione, sia negativa ed è corretto che all'aumentare della percentuale di applicazione decresca il valore della *potency* poiché aumenta il numero di basic block oggetto di tale trasformazione.

Le Figure 7.10 e 7.11, invece, mostrano i grafici generati riferiti alle potency calcolate sulle metriche che rappresentano la complessità ciclomatica e la lunghezza di Halstead. Ogni grafico riporta una determinata protezione applicata dallo strumento *Diablo*:

- sull'asse x è presente la stessa protezione applicata con percentuali diverse e crescenti: $10\%, 30\%$ e 50% (da sinistra verso destra);
- sull'asse y viene indicato il valore della media della potency in funzione delle protezioni presenti sull'asse x.

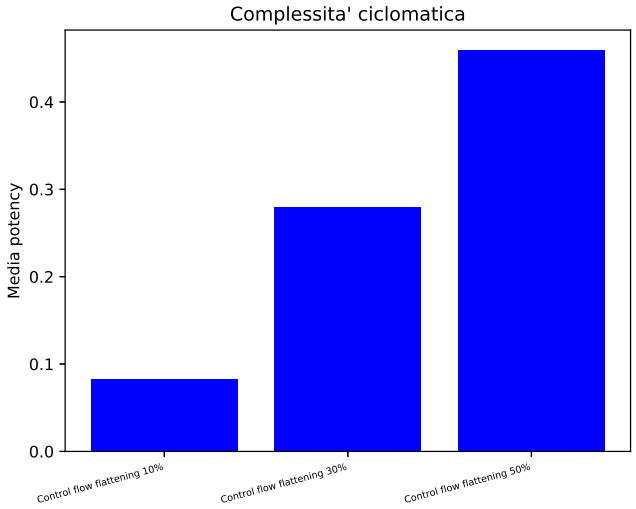
La Figura 7.10 mostra la media della potency calcolata sulla complessità ciclomatica per:

- la protezione *control flow flattening* (cfr. Figura 7.10a);
- la protezione *anti branch analysis* (cfr. Figura 7.10b);
- la protezione *predicati opachi* (cfr. Figura 7.10c).

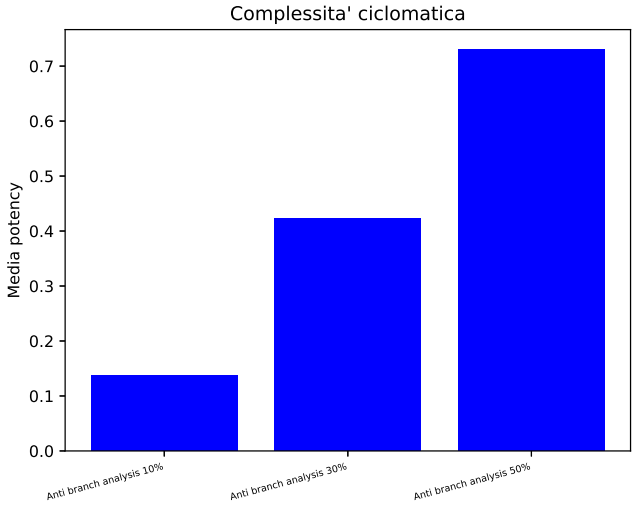
La Figura 7.11 mostra la media della potency calcolata sulla lunghezza di Halstead per:

- la protezione *control flow flattening* (cfr. Figura 7.11a);
- la protezione *anti branch analysis* (cfr. Figura 7.11b);
- la protezione *predicati opachi* (cfr. Figura 7.11c).

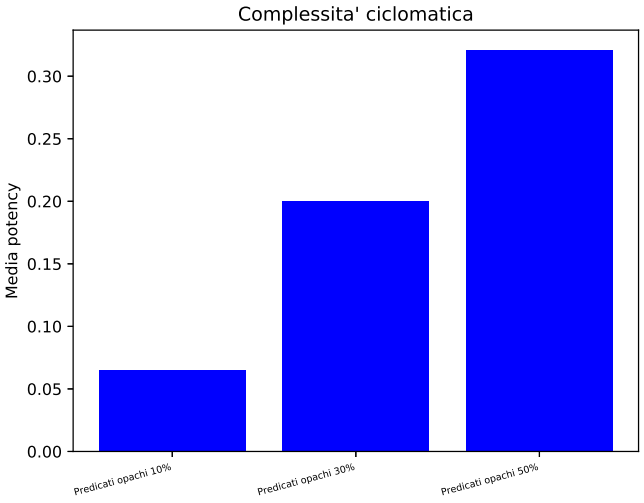
È stato scelto di riportare solo le *potency* estratte dallo strumento di *Diablo* perché esso permette di applicare uno stesso offuscamento scegliendo il grado di protezione, in percentuale, da dover applicare. Mettendo a confronto i grafici generati, come si può anche osservare dai grafici appena descritti, si ottiene un veloce ed intuitivo riscontro: all'aumentare della percentuale di applicazione di una data protezione, aumenta il valore della *potency*.



(a) Media potency per la protezione *control flow flattening*.

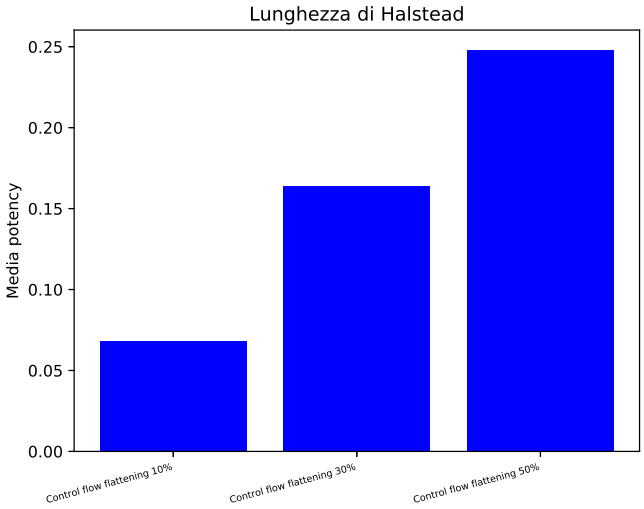


(b) Media potency per la protezione *anti branch analysis*.

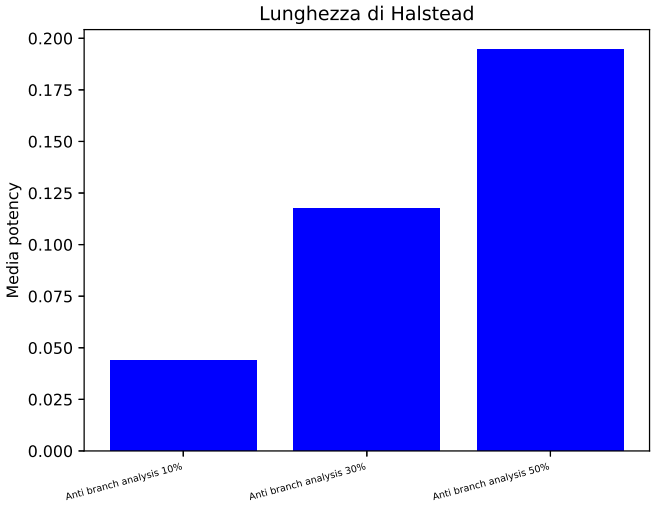


(c) Media potency per la protezione *predicati opachi*.

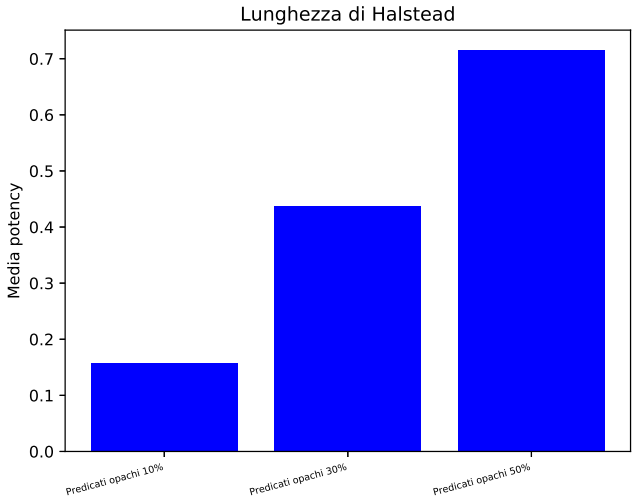
Figura 7.10: Media potency per la complessità ciclomatica.



(a) Media potency per la protezione *control flow flattening*.



(b) Media potency per la protezione *anti branch analysis*.



(c) Media potency per la protezione *predicati opachi*.

Figura 7.11: Media potency per la lunghezza di Halstead.

Capitolo 8

Conclusioni

Nello studio affrontato abbiamo analizzato le metriche di complessità, utilizzate per rappresentare numericamente quanto un codice risulti complesso in termini di interazioni tra diverse entità. Abbiamo successivamente analizzato diversi tipi di offuscamento ad oggi utilizzati e si è posto l'attenzione su come ognuno di essi risulti in grado di apportare delle modifiche software, sia a livello di codice sia a livello di controllo di flusso, in base alla propria funzione di utilizzo ed al tipo di protezione da introdurre. Infine è stato progettato ed implementato lo script *Metrics_analisis.py* che ha permesso di automatizzare l'applicazione di specifiche protezioni e l'estrazione di determinate metriche (sia su file sorgenti sia su binari) tramite gli strumenti di *Tigress*, di *Frama-c*, e di *Diablo*, potendo così ricavare e sottoporre ad analisi una notevole quantità di dati.

Gli studi effettuati e le analisi condotte sui dati raccolti si sono concentrati principalmente salla dimostrazione della presenza delle *fingerprint* e sull'utilizzo delle *potency*.

Grazie all'analisi delle metriche di complessità estratte dai programmi protetti siamo stati in grado di dimostrare empiricamente la presenza delle *fingerprint*: ogni modifica introdotta dagli offuscamenti utilizzati influisce direttamente sul valore di determinate metriche permettendo, in questo modo, di identificare la protezione applicata. Abbiamo anche potuto notare che non tutti gli offuscamenti risultano facilmente identificabili per mezzo delle *fingerprint* poiché i valori delle metriche estratte non risultano particolarmente significativi per quella data protezione e non evidenziano alcun pattern particolarmente rilevante ai fini del riconoscimento. Inoltre, si è anche potuto studiare come alcune metriche siano più coinvolte di altre in funzione del tipo di offuscamento applicato. Ragionando ed essendo a conoscenza delle proprietà e delle modifiche che un determinato offuscamento è in grado di apportare al codice da proteggere, si può ristringere lo studio delle metriche da esaminare alle metriche direttamente interessate: ad esempio il numero di linee di codice non può essere utilizzato come metrica di riferimento per risalire all'offuscamento apportato in quanto non rappresenta un valore di riferimento ideale; al contrario, il numero di cicli `loop` si è rivelato essere un ottimo indicatore per rilevare la presenza della protezione *control flow flattening*.

I test effettuati in merito all'utilizzo della *potency* sono stati eseguiti sul calcolo del valore della media delle metriche esaminate riportando in questo modo valori di massima che, proprio a causa della media, potrebbero non rappresentare il comportamento atteso in assoluto. A seguito di interventi con gli esperti del progetto ASPIRE[1] è stato riportato che, ad oggi, le uniche metriche di rilevante importanza per il calcolo della *potency* risultano essere la lunghezza di Halstead e la complessità ciclomatica, calcolate su file binari. È stata quindi eseguita una ricerca, basandosi sul metodo del confronto, per cercare potenziali altre metriche utili a tale calcolo. Sono stati quindi generati dei grafici su cui sono state rappresentate le *potency* calcolate sulla lunghezza di Halstead e sulla complessità ciclomatica, ordinate per valori crescenti per ogni funzione analizzata. Successivamente, mantenendo inalterato l'ordine delle funzioni, è stato rappresentato sugli stessi grafici anche il valore delle *potency* calcolato sulle altre metriche da porre a confronto. I grafici

[1] https://aspire-fp7.eu/.

risultanti non hanno evidenziato nessuna metrica che potesse risultare al pari delle due metriche usate come riferimento: hanno però permesso di riporre l'attenzione su alcune relazioni presenti tra differenti tipi di metriche poiché gli andamenti delle loro curve risultano essere molto simili. In particolare è emersa la correlazione tra:

- la complessità ciclomatica con la metrica rappresentante il numero di punti decisionali;
- la lunghezza di Halstead con le metriche che rappresentano il numero di operandi sorgenti, il numero di operandi destinazione ed il numero di istruzioni assembly.

Si è quindi ipotizzato di poter utilizzare tali metriche come stima approssimativa per la complessità ciclomatica e per la lunghezza di Halstead: risulta infatti più oneroso calcolare queste ultime piuttosto che le altre.

Infine è stato effettuato uno studio sull'utilizzo della *potency* per valutarne la sua capacità di indicare quale offuscamento, tra quelli disponibili, risulti applicare un livello di sicurezza maggiore. Sfruttano la proprietà dello strumento *Diablo* che permette di applicare uno stesso offuscamento con diverse percentuali di utilizzo, si è potuto dimostrare che per una stessa protezione, all'aumentare della percentuale applicata aumenta anche il valore della *potency* calcolata in riferimento ad una stessa metrica. Risulta dunque importante la scelta delle corrette metriche per il calcolo della *potency*.

In conclusione, si può quindi affermare che ogni offuscamento, in certi casi in maniera più visibile, in altri meno, applica dei cambiamenti ben precisi identificabili dalle *fingerprint* . La *potency*, invece, si è dimostrata essere un buon indicatore del livello di sicurezza applicato. In comune, *fingerprint* e *potency*, risultano avere la giusta scelta delle metrica da utilizzare per giungere allo scopo finale: ogni metrica, infatti, pone in risalto diverse caratteristiche del codice analizzato e quindi diverse caratteristiche legate all'aspetto di sicurezza che si vuole analizzare ed osservare.

Sviluppi futuri

Durante l'implementazione dello script creato per automatizzare la raccolta dati e durante gli studi ed esperimenti effettuati su di essi, sono emersi e si è ragionato anche su possibili sviluppi che potrebbero essere trattati in futuro:

- migliorare lo script implementato con l'integrazione di ulteriori strumenti che permettano di estrarre metriche diverse da quelle già estratte, con strumenti in grado di analizzare file con estensione diversa dal *.c* e con strumenti in gradi di applicare differenti tipi di protezioni da quelli già utilizzati;
- calcolare i pesi, che attualmente risultano essere a peso unitario, utilizzati per il calcolo della *potency combinata* (cfr. Sezione 2.4);
- confrontare le metriche estratte dal programma protetto sorgente con le stesse metriche estratte dal medesimo programma offuscato compilato per studiare se le ottimizzazioni dei compilatori impattano la protezione applicata.

Appendice A

File delle annotazioni

Il file delle annotazioni passato a *Diablo* è un file in formato *.json* in cui si strutturano in modo ordinato i dati che lo strumento dovrà eseguire. Si riporta un esempio di un file delle annotazioni per capirne meglio la struttura nella Figura A.1: In questo file:

```
[
        {
                "annotation content": "protection(none)",
                "annotation type": "code",
                "file name": <path_to_file>,
                "function name": "foo",
                "line number": [
                        3,
                        8
                ]
        }
]
```

Figura A.1: Esempio file di annotazioni.

- ‘‘`annotation content`’’: rappresenta il tipo di operazione che deve essere eseguita su questo oggetto ed in questo caso ‘‘`protection(none)`’’ indica che non deve essere applicata nessuna protezione. Anziché `none`, altri valori attribuibili a tale proprietà sono:
  - `obfuscations,enable_obfuscation(flatten_function: percent_apply=<n>)`[1]: applica la protezione *control flow flattening*, descritta in Sezione 2.2.2;
  - `obfuscations,enable_obfuscation(branch_function: percent_apply=<n>)`[1]: applica la protezione *anti branch analysis*, descritta in Sezione 2.2.3;
  - `obfuscations,enable_obfuscation(opaque_predicate: percent_apply=<n>)`[1]: applica la protezione *predicati opachi*, descritta in Sezione 2.2.1.
- `"annotation type"`: il tipo di annotazione riportata, in questo esempio è `"code"`;
- `"file name"`: path al file *.c* che si vuole analizzare;
- `"function name"`: il nome della funzione a cui si riferisce questa sezione;

[1] <n> è un valore decidibile dall'utente nel *range[0,100]* ed indica con quale valore, in percentuale, deve venire applicata la protezione adottata.

- `"line number"`: il numero di linea di inizio e fine della parte di codice sulla quale applicare l'operazione riportata in `"annotation content"`.

Ogni oggetto aggiunto nel file *.json* viene identificato da un codice numerico crescente partendo dal valore 1. Questo valore viene poi riutilizzato nel file di output generato da *Diablo* nel campo denominato `region\_idx` come riportato nella Figura 1.5 per riconoscere a quale oggetto del file delle annotazioni fanno riferimento le metriche estratte.

Appendice B

Manuali

Vengono di seguito riportati i manuali utente e programmatore degli script: Metrics_analysis.py, media_dev_std.py e Potency.py. Tali manuali riportano le specifiche per utilizzare gli script implementati ed una spiegazione dettagliata del loro funzionamento.

B.1 Manuale utente per lo script Metrics_analysis.py

In questa Sezione vengono riportate le specifiche del manuale utente relative allo script Metrics_analysis.py ed in particolare verrà spiegato il suo funzionamento ad alto livello, verrà fornita una spiegazione di come configurare i paramenti necessari al corretto funzionamento ed infine verrà spiegato come eseguire lo script e dove ritrovare l'output generato.

B.1.1 Introduzione

Metrics_analysis.py è uno script implementato in python che permette di estrarre metriche calcolate sia su file sorgenti sia su file binari, prima e dopo una data offuscazione. All'interno dello script vengono richiamati vari strumenti:

- `Frama-c`: usato per estrarre metriche su file sorgenti;
- `Diablo`: usato per offuscare ed estrarre metriche su file binari;
- `Tigress`: usato per offuscare i file sorgenti.

Metrics_analysis.py supporta i seguenti offuscamenti per lo strumento di *Tigress*:

- `Flatten`;
- `Opaque`;
- `EncodeArithmetic`;
- `AntiBranchAnalysis`.

I seguenti per lo strumento di *Diablo*:

- `flatten_function`;
- `branch_functions`;
- `opaque_predicate`.

Le metriche estratte da entrambi gli strumenti di estrazione vengono raccolte e riscritte in file in formato *.csv*.

B.1.2 Requisiti

Metrics_analysis.py richiede Python 2.7.12 o superiore e le seguenti dipendenze:

- `Universal Ctags v5.9` https://ctags.io/;
- `Frama-c Magnesium-20151002` https://frama-c.com/;
- `Tigress v2.2` https://http://tigress.cs.arizona.edu/;
- `Diablo v2.9.0` https://diablo.elis.ugent.be/;
- `GCC 4.6+ per ARM` https://gcc.gnu.org/.

Le precedenti dipendenze dovranno essere installate manualmente dall'utente.

B.1.3 Installazione

Bisogna copiare la cartella denominata *Metrics_analysis* sul proprio PC.

B.1.4 Configurazione

Prima di lanciare Metrics_analysis.py sarà necessario modificare correttamente il file di configurazione dello script presente all'interno della cartella *Metrics_analysis/sub/conf.txt*. Un esempio è mostrato nel Listato B.1.

```
tigress=/home/stefano/Scrivania/tigress/tigress-2.2
CC_Diablo=/home/stefano/Scrivania/diablo/gcc/bin/arm-diablo-linux-gnueabi-gcc
OBF_Diablo=/home/stefano/Scrivania/diablo/diablo/build/diablo-obfuscator
Board_ARM=stefano@[indirizzoIP]
max_dim_file_obf=8384000
see_output=false
keep_compiled=true
keep_obf=true
keep_diablo=true
keep_report_frama=true
```

Listing B.1: Esempio file di configurazione.

La definizione dei campi è la seguente:

- `tigress`: path assoluto della directory in cui è stato installato Tigress;
- `CC_Diablo`: path assoluto della directory in cui è stato installato il compilatore usato da Diablo;
- `OBF_Diablo`: path assoluto della directory in cui è stato installato l'offuscatore di Diablo;
- `Board_ARM`: username più indirizzo IP al quale è raggiungibile la board ARM per testare il corretto funzionamento dei programmi processati da Diablo;
- `max_dim_file_obf`: la dimensione massima in byte dei file **.c* che si vogliono processare dopo aver applicato una determinata protezione;
- `see_output`: se impostato a *true* permette la visualizzazione di tutto l'output generato da Metrics_analysis.py, altrimenti se impostato a *false* permette solamente la visualizzazione dei passaggi intermedi che lo script sta eseguendo;

- `keep_compiled`: se impostato a *true*, al termine dello script, la cartella creata *compiled*, contenente tutte le directory dei programmi processati da *Tigress* e *Diablo*, sarà mantenuta, altrimenti se impostato a *false* la cartella, al termine dell'esecuzione dello script, verrà eliminata;
- `keep_obf`: se impostato a *true*, al termine dello script, la cartella creata *file_obf*, contenente i file sorgenti protetti o non protetti dei programmi processati da *Tigress*, sarà mantenuta, altrimenti se impostato a *false* la cartella, al termine dell'esecuzione dello script, verrà eliminata;
- `keep_diablo`: se impostato a *true*, al termine dello script, la cartella creata *report_diablo*, contenente gli eseguibili ed i report dei programmi processati da *Diablo*, sarà mantenuta, altrimenti se impostato a *false* la cartella, al termine dell'esecuzione dello script, verrà eliminata;
- `keep_report_frama`: se impostato a *true*, al termine dello script, la cartella creata *report_frama_c*, contenente i report dei file sorgenti processati da *Frama-c*, sarà mantenuta, altrimenti se settato a *false*, la cartella al termine dell'esecuzione dello script, verrà eliminata.

I programmi che vorranno essere analizzati dovranno essere raccolti in un'unica cartella e per ogni programma dovrà essere creato un file di configurazione denominato *confmake_[name].ini* dove *[name]* è una stringa alfanumerica scelta dall'utente. Un esempio è mostrato nel Listato B.2. Tale configurazione dovrà essere inserita all'interno della directory principale del programma di appartenenza.

```
[pipe]
cc =
cppflags = -DTIME
lflags =
cflags =
ldflags =
_dirmakefile = ./bm
_namemakefile = make
file = ./bm/src/pipe.c
dir_obj_file = ./bm/
binary = ./bm/pgms/pipe
cmd = ./bm/pgms/pipe 2
rm_out =
```

Listing B.2: Esempio file di configurazione per ogni programma.

I campi all'interno di questo file di configurazione, Listato B.2, sono:

- `[pipe]`: nome del target del Makefile a cui si vuole fare riferimento[1];
- `cc`: tipo di compilatore da voler usare;
- `cppflags`: flags di pre-processing separati da almeno uno spazio;
- `lflags`: flags da passare al generatore di scanner;
- `cflags`: flags da passare al compilatore separati da almeno uno spazio;
- `ldflags`: flags da passare al linker separati da almeno uno spazio;
- `_dirmakefile`: path relativo dalla directory principale del programma alla cartella in cui è presente il Makefile;

[1] In un file di configurazione possono esserci più target riferiti anche a Makefile diversi purchè siano compilati i rispettivi sottocampi.

- `_namemakefile`: il nome dato al Makefile;
- `file`: lista dei file *.c* con path relativo dalla directory principale del programma da volere analizzare;
- `dir_obj_file`: path relativo dalla main directory del programma alla sotto-cartella in cui verrà generato il file oggetto;
- `binary`: path relativo dalla directory principale del programma alla cartella in cui è presente il file binario;
- `cmd`: lista di comandi da eseguire[2], separati da una virgola, per testare il corretto funzionamento del binario compilato;
- `rm_out`: lista di file di output generati dalla lista di comandi *cmd* con path relativo dalla directory principale del programma che si desidera rimuovere al termine della compilazione, separati da una virgola.

B.1.5 Utilizzo

Lo script Metrics_analysis.py è presente nella cartella *Metrics_analysis* e lo si lancia tramite il comando da terminale:

```
./Metrics_analysis.py Source Destination
```

dove il parametro `Source` rappresenta il path alla cartella contenente i programmi da analizzare mentre il parametro `Destination` il path alla cartella in cui generare gli output.

In particolare sarà possibile trovare le metriche estratte organizzate in file *.csv* nella cartella chiamata *pre-analized* suddivise per programma.

Di seguito, nella Figura B.2, viene riportata la struttura di come sarà organizzata la cartella di destinazione contenente gli output generati da Metrics_analysis.py

Al termine di Metrics_analysis.py, nella cartella *Metrics_analysis*, sarà prodotto un file di log denominato *LOGFILE.txt* contenente un resoconto di tutti gli errori generati durante l'esecuzione dello script. Per gli errori più specifici sarà generato un file di testo riportante l'output prodotto dall'errore e tale file verrà inserito nella cartella *Metrics_analysis/Log/*. In *LOGFILE.txt* verrà inoltre visualizzato il percorso all'interno di *Metrics_analysis/Log/* relativo all'errore riportato.

B.2 Manuale programmatore per Metrics_analysis.py

In questa Sezione vengono riportate le specifiche del manuale programmatore relative allo script Metrics_analysis.py ed in particolare verranno spiegati dettagliatamente l'architettura dello script e la struttura dell'output generato. Infine verrà riportato come modificare le protezioni attualmente apportate ai programmi analizzati.

B.2.1 Architettura dello script

La Figura B.1 mostra l'architettura dello script *Metrics_analysis.py*:

- Ctags[3]: strumento che permette di estrarre vari tipi di informazione, utilizzato in questo progetto per rintracciare il nome delle funzioni implementate nei vari file sorgente e per estrarne il numero di riga di inizio e di fine;

[2] I comandi saranno lanciati dalla directory principale del programma, pertanto occorrerà specificare il path relativo dalla directory principale all'eseguibile.

[3] https://ctags.io/.

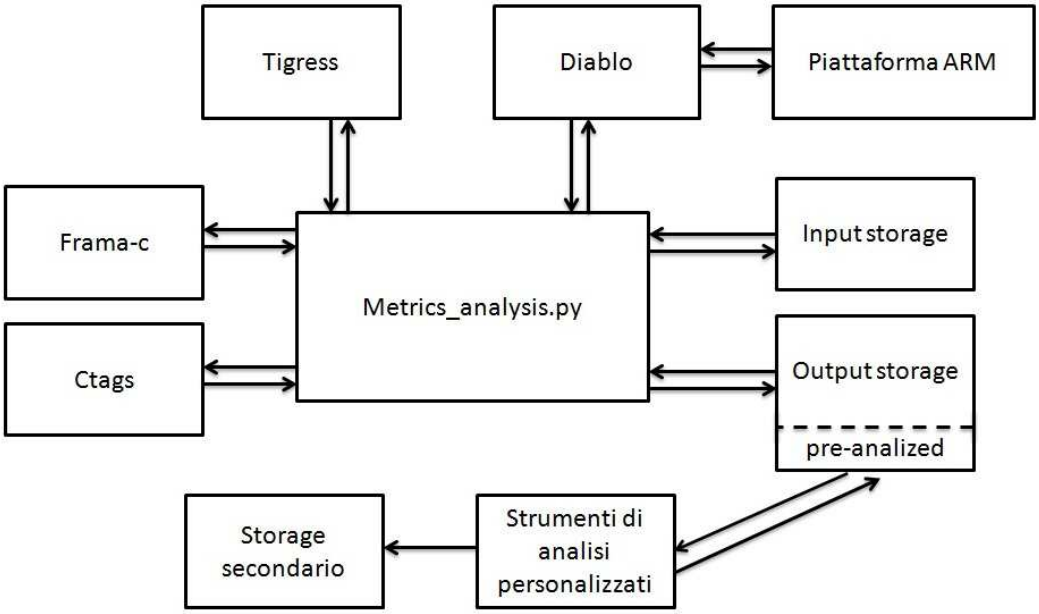


Figura B.1: Architettura di Metrics_analysis.py.

- Frama-c (cfr. Sezione 1.2.1): strumento utilizzato per estrarre i valori delle metriche software (cfr. Capitolo 1) su file sorgenti;
- Tigress (cfr. Sezione 2.5.4): strumento utilizzato per apportare determinati offuscamenti a livello di codice sorgente;
- Diablo (cfr. Sezione 2.5.8): strumento utilizzato per estrarre i valori delle metriche software (cfr. Capitolo 1) su file binari;
- piattaforma ARM: piattaforma necessaria al corretto funzionamento di *Diablo* poiché questo strumento è in grado di lavorare solo su questo tipo di architettura;
- input storage: directory principale, fornita da linea di comando, contenente le cartelle dei programmi scaricati[4]. Lo script *Metrics_analysis.py* analizzerà ogni cartella presente in questa directory;
- output storage: directory, fornita da linea di comando, in cui lo script *Metrics_analysis.py* depositerà tutti gli output generati;
- strumenti di analisi personalizzati: questo blocco raccoglie logicamente l'insieme degli script implementati per l'analisi custom, ad esempio calcolo della media e della potency (cfr. Sezione 2.3) o gli script necessari per la creazione dei file CSV (cfr. Sezione B.2.3), dei file di output generati (contenuti nella cartella *pre-analized*);
- storage secondario: directory, fornita da linea di comando agli script citati al punto precedente, in cui gli script personalizzati depositano il proprio output prodotto.

B.2.2 Struttura dell'output generato

La Figura B.2 mostra le cartelle e sotto-cartelle generate da *Metrics_analysis.py*.

[4]I programmi sono scaricati dal benchmark Phoronix test suite (cfr. Sezione 3.2.2).

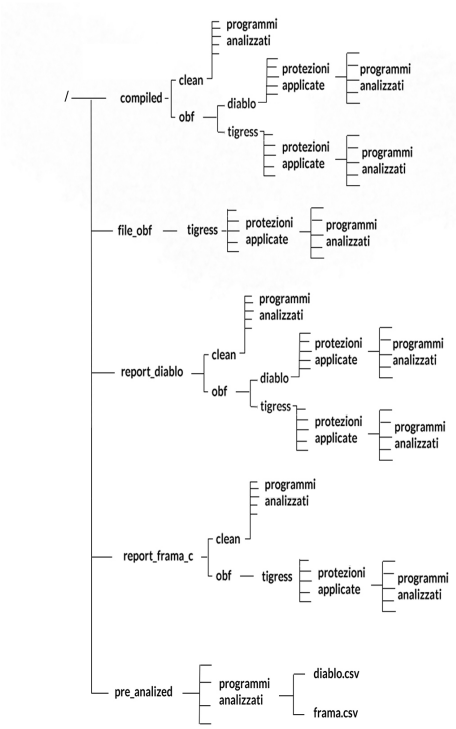


Figura B.2: Struttura organizzativa dell'output generato.

compiled. Cartella contenente i file binari generati. Al suo interno si trovano due cartelle:

- *clean*[5]: contiene i file binari in chiaro (senza alcun offuscamento applicato);
- *obf*: contiene ulteriori due sotto-cartelle, *diablo*[6] e *tigress*[7]. La prima raccoglie i file binari offuscati dallo strumento *Diablo*, la seconda i file binari offuscati dallo strumento *Tigress*.

file_obf. Cartella contenente i file sorgenti offuscati dallo strumento Tigress 2.5.4. Al suo interno è presente un'unica cartella chiamata *tigress*;

report_diablo: Cartella contenente i report generati dallo strumento di Diablo 2.5.8. Al suo interno si trovano due cartelle:

- *clean*[5]: contiene i report generati sui file binari in chiaro (senza alcun offuscamento applicato);
- *obf*: contiene ulteriori due sotto-cartelle, *diablo*[6] e *tigress*[7]. Tali cartelle raccolgono i report generati sui file binari offuscati dagli strumenti di *Diablo* e *Tigress* rispettivamente.

[5]In tale cartella sono presenti tante sotto-cartelle quanti sono i programmi analizzati.

[6]In tale cartella sono presenti tante sotto-cartelle quanti sono gli offuscamenti selezionati per lo strumento *Diablo*.

[7]In tale cartella sono presenti tante sotto-cartelle quanti sono gli offuscamenti selezionati per lo strumento *Tigress*.

report_frama_c. Cartella contenente i report generati dallo strumento di Frama-c 1.2.1. Al suo interno si trovano due cartelle:

- *clean*[5]: contiene i report generati sui file sorgenti in chiaro (senza alcun offuscamento applicato);
- *obf*: al suo interno è presente un'unica cartella chiamata *tigress*[7] che raccoglie i report generati sui file sorgenti offuscati dallo strumento *Tigress*.

pre_analized. Cartella[5] contenente i file in cui sono raccolti i valori delle metriche estratte dai report generati dagli strumenti *Frama-c* e *Diablo*. Per ogni sotto-cartella generata saranno presenti due file *csv* (cfr. Sezione B.2.3): uno contenente tutte le metriche estratte dallo strumento di *Frama-c* ed uno contenente tutte le metriche estratte dallo strumento *Diablo*.

Inoltre, per ogni sotto-cartella dei programmi analizzati, si troveranno tante altre sotto-cartelle quanti saranno i *target* del Makefile utilizzati per la compilazione (ad ogni cartella sarà associato il nome del programma concatenato al nome del *target* del Makefile).

B.2.3 File CSV generati

Per ogni programma analizzato dallo script *Metrics_analysis.py*, come accennato precedentemente nel corso di questo capitolo, verranno generati due file nel formato *CSV*: uno per le metriche estratte dallo strumento *Frama-c* ed uno per le metriche estratte dallo strumento *Diablo*.

File CSV di Frama-c

L'header di riferimento per la creazione di questo file *CSV* risulta essere composto dai seguenti campi:

1. program: il nome del programma a cui si riferisce, ovvero il programma di cui sono state estratte le metriche;
2. protection: il nome dell'offuscamento apportato (se le metriche sono calcolate sul programma in chiaro senza alcun offuscamento apportato, il campo sarà compilato con la dicitura *vanilla*);
3. tool_protection: lo strumento con il quale è stato apportato l'offuscamento, quindi verrà scritto *Tigress* o *Diablo* (se le metriche sono calcolate sul programma in chiaro senza alcun offuscamento apportato, il campo sarà compilato con la dicitura *null*);
4. target_makefile: il comando *target* del Makefile con il quale è stato compilato il programma;
5. path_file_from_main_directory_program: path relativo dalla cartella principale del programma analizzato al file *.c* preso in considerazione;
6. function: il nome della funzione analizzata;
7. flag_obf_tigress(0=made_1=not): flag per il controllo dell'avvenuto offuscamento tramite lo strumento *Diablo*. Se posto a 0 l'offuscamento è avvenuto, se posto a 1 no (cfr. Sezione 5.2.1, Sezione 5.2.1, Sezione 5.2.2);
8. Sloc: valore metrica estratta da *Frama-c*, indica il numero di linee di codice;
9. Decision point: valore metrica estratta da *Frama-c*, indica il numero di punti decisionali come per esempio *if*, *for*, *while*;
10. Global variables: valore metrica estratta da *Frama-c*, indica il numero di varibili globali;
11. If: valore metrica estratta da *Frama-c*, indica il numero di *if*;

12. Loop: valore metrica estratta da *Frama-c*, indica il numero di *loop*;
13. Goto: valore metrica estratta da *Frama-c*, indica il numero di *goto*;
14. Assignment: valore metrica estratta da *Frama-c*, indica il numero di assegnazioni;
15. Exit point: valore metrica estratta da *Frama-c*, indica il numero di *exit point*;
16. Function: valore metrica estratta da *Frama-c*, indica il numero di funzioni contenute nel programma[8];
17. Function call: valore metrica estratta da *Frama-c*, indica il numero di funzioni chiamate;
18. Pointer dereferencing: valore metrica estratta da *Frama-c*, indica numero di *pointer dereferencing*[9];
19. Cyclomatic complexity: valore metrica estratta da *Frama-c*, indica il valore della complessità ciclomatica.

Il nome del *CSV* creato sarà:

[programma analizzato]_frama.csv

dove *[programma analizzato]* sarà sostituito con il nome del programma a cui fanno riferimento le metriche riportate.

File CSV di Diablo

L'header di riferimento per la creazione di questo file *CSV* risulta essere composto dai seguenti campi:

1. program: il nome del programma a cui si riferisce, ovvero il programma da cui sono state estratte le successive metriche;
2. protection: il nome dell'offuscamento apportato (se le metriche sono calcolate sul programma in chiaro senza alcun offuscamento apportato, il campo sarà compilato con la dicitura *vanilla*);
3. tool_protection: lo strumento con il quale è stato apportato l'offuscamento (se le metriche sono calcolate sul programma in chiaro senza alcun offuscamento apportato, il campo sarà compilato con la dicitura *null*);
4. target_makefile: il comando *target* del Makefile con il quale è stato compilato il programma;
5. path_file_from_main_directory_program: path relativo dalla cartella principale del programma analizzato al file *.c* preso in considerazione;
6. function: il nome della funzione analizzata;
7. flag_obf_tigress(0=made_1=not): flag per il controllo dell'avvenuto offuscamento tramite lo strumento di *Diablo* (se posto a 0 l'offuscamento è avvenuto, se posto a 1 no);
8. nr_ins_static: valore metrica estratta da *Diablo*, indica il numero di istruzioni assembly;
9. nr_src_oper_static: valore metrica estratta da *Diablo*, indica il numero di operandi sorgente[10];

[8] Nel caso si analizzasse il programma a livello di funzione, tale parametro ritorna il valore 1

[9] Il numero di puntatori utilizzati all'intero della funzione.

[10] Gli operandi presi come input da un'istruzione.

10. nr_dst_oper_static: valore metrica estratta da *Diablo*, indica il numero di operandi destinazione[11];

11. halstead_program_size_static: valore metrica estratta da *Diablo* , indica la lunghezza di Halstead;

12. nr_edges_static: valore metrica estratta da *Diablo*, indica il numero ramificazioni del Control Flow Graph (CFG);

13. nr_indirect_edges_CFIM_static: valore metrica estratta da *Diablo*, indica il numero di ramificazioni indirette (cfr. Sezione 1.1.1);

14. cyclomatic_complexity_static: valore metrica estratta da *Diablo*, indica il valore della complessità ciclomatica.

Il nome del *CSV* creato sarà:

[programma analizzato]_diablo.csv

dove *[programma analizzato]* sarà sostituito con il nome del programma a cui fanno riferimento le metriche riportate.

B.2.4 Modifica protezioni

Per le protezioni da applicare sono stati creati due dizionari `obf_tigress` e `obf_diablo`, per gli strumenti di Tigress e di Diablo rispettivamente. In entrambi i casi la *chiave* rappresenta il nome con il quale riconoscere la protezione assegnata ed il *valore* contiene i parametri da passare al rispettivo strumento per applicare la protezione.

Per inserire una nuova protezione da applicare dunque bisogna aggiungere una nuova coppia *chiave-valore* nel relativo dizionario assegnato allo strumento sul quale si vuole effettuare la modifica; per rimuovere una protezione invece cancellare la coppia *chiave-valore* esistente dal dizionario.

Se si desiderasse invece modificare alcuni parametri di una protezione esistente, basterà modificare il *valore* della *chiave* associata alla protezione prescelta.

B.3 Manuale programmatore per gli script atti all'analisi dei dati

Di seguito vengono riportati i workflow ed una loro descrizione dei due script utilizzati per effettuare l'analisi dei dati estratti dallo script *Metrics_analysis*.

B.3.1 Script media_dev_std.py

Lo script *media_dev_std.py*, situato nella cartella *Metrics_analysis/sub/utils/media_devstd/*, permette di calcolare la media, la varianza, la deviazione standard e l'errore standard dai file *csv*, contenuti nella cartella *pre-analized*, generati dallo script *Metrics_analysis.py*.

La Figura B.3 mostra il workflow di esecuzione dello script *media_dev_std.py*. Si riporta una breve descrizione per ogni blocco del diagramma di flusso:

[11] Gli operandi presi come output da un'istruzione.

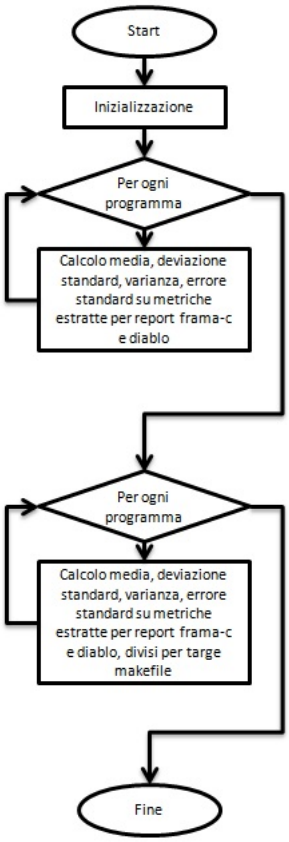


Figura B.3: Workflow media, varianza, deviazione standard ed errore standard.

1. Inizializzazione: vengono importate le librerie, definite le funzioni, preparate le variabili e le liste contenenti il nome delle metriche su cui eseguire i calcoli e gli offuscamenti apportati, infine vengono controllati i parametri passati come input allo script *media_dev_std.py*;

2. per ogni programma: ciclo *for* per mezzo del quale si scorrono tutte le sotto-cartelle contenute nella cartella `[source]` passata come parametro;

3. Calcolo media, deviazione standard, varianza, errore standard su metriche estratte per report Frama-c e Diablo: per tutte le metriche estratte da entrambi gli strumenti *Frama-c* e *Diablo* viene calcolata la media. I valori calcolati saranno riportati in un file *csv* chiamato *[programma analizzato]_all.csv* contenuto nella cartella `[destination]`, dove *[programma analizzato]* è il nome del programma che si sta analizzando;

4. per ogni programma: terminato il ciclo *for* precedente, per mezzo di un ulteriore ciclo *for* si scorrono nuovamente tutte le sotto-cartelle contenute nella cartella `[source]`;

5. Calcolo media, deviazione standard, varianza, errore standard su metriche estratte per report Frama-c e Diablo, divisi per target makefile: per tutte le metriche estratte da entrambi gli strumenti *Frama-c* e *Diablo*, raggruppate per target makefile, viene calcolata la media. I valori calcolati saranno riportati in un file *csv* chiamato *[programma analizzato]_for_each_target.csv* contenuto nella cartella `[destination]`, dove *[programma analizzato]* è il nome del programma che si sta analizzando.

B.3.2 Script Potency.py

Lo script *Potency.py*, situato nella cartella *Metrics_analysis/sub/utils/potency*, permette di calcolare la potency (cfr. Sezione 2.3) sulle metriche estratte dagli strumenti *Frama-c* e *Diablo*, la potency combinata (cfr. Sezione 2.4) sulle metriche estratte dallo strumento *Diablo* e per ognuna di queste metriche calcola a sua volta la media, la varianza, la deviazione standard e l'errore standard.

La Figura B.4 mostra il workflow di esecuzione dello script *Potency.py*. Si riporta una breve descrizione per ogni blocco del diagramma di flusso:

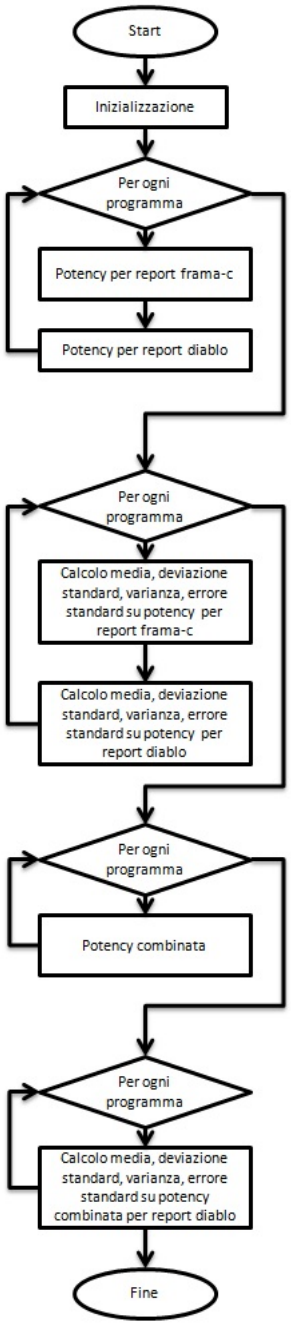


Figura B.4: Workflow script *Potency.py*.

1. Inizializzazione: vengono importate le librerie, definite le funzioni, preparate le variabili e le

liste contenenti il nome delle metriche su cui eseguire i calcoli e gli offuscamenti apportati, infine vengono controllati i parametri passati come input allo script *Potency.py*;

2. per ogni programma: ciclo *for* per mezzo del quale si scorrono tutte le sotto-cartelle contenute nella cartella `[source]` passata come parametro;

3. potency per report Frama-c: viene calcolata la potency (cfr. Sezione 2.3) per le metriche estratte dallo strumento di *Frama-c*. Tali valori saranno raccolti in un file chiamato *[programma analizzato]_potency_function_protection_frama.csv* contenuto nella cartella `[destination]`, dove *[programma analizzato]* è il nome del programma che si sta analizzando;

4. potency per report Diablo: viene calcolata la potency (cfr. Sezione 2.3) per le metriche estratte dallo strumento di *Diablo*. Tali valori saranno raccolti in un file chiamato *[programma analizzato]_potency_function_protection_diablo.csv* contenuto nella cartella `[destination]`, dove *[programma analizzato]* è il nome del programma che si sta analizzando;

5. per ogni programma: terminato il precedente ciclo *for*, un ulteriore ciclo *for* permette di scorrere tutte le sotto-cartelle appena create contenute nella cartella `[destination]` passata come parametro;

6. calcolo media, varianza, deviazione standard ed errore standard su potency per report Frama-c: per ogni file *[programma analizzato]_potency_function_protection_frama.csv* precedentemente creato, ne viene calcolata la media. I valori calcolati saranno riportati in un file *csv* chiamato *[programma analizzato]_media_potency_function_protection _frama.csv* contenuto nella cartella `[destination]`, dove *[programma analizzato]* è il nome del programma che si sta analizzando;

7. calcolo media, deviazione standard, varianza, errore standard su potency per report Diablo: per ogni file *[programma analizzato]_potency_function_protection_diablo.csv* precedentemente creato, ne viene calcola la media. I valori calcolati saranno riportati in un file *csv* chiamato *[programma analizzato]_media_potency_function_protection_diablo.csv* contenuto nella cartella `[destination]`, dove *[programma analizzato]* è il nome del programma che si sta analizzando;

8. per ogni programma: terminato il precedente ciclo *for*, un ulteriore ciclo *for* permette di scorrere nuovamente tutte le sotto-cartelle contenute nella cartella `[destination]` passata come parametro;

9. potency combinata: per ogni file chiamato *[programma analizzato]_potency_function_protection_diablo.csv* precedentemente creato viene calcolata la potency combinata (cfr. Sezione 2.4). Tali valori saranno raccolti in un file chiamato *[programma analizzato]_Combined_potency_function_protection_diablo.csv* contenuto nella cartella `[destination]`, dove *[programma analizzato]* è il nome del programma che si sta analizzando;

10. per ogni programma: terminato il precedente ciclo *for*, un ulteriore ciclo *for* permette di scorrere nuovamente tutte le sotto-cartelle contenute nella cartella `[destination]` passata come parametro;

11. calcolo media, varianza, deviazione standard ed errore standard su potency combinata per report Diablo: per ogni file chiamato *[programma analizzato]_Combined_potency_function_protection_diablo.csv* precedentemente creato, ne viene calcolata la media. I valori calcolati saranno riportati in un file *csv* chiamato *[programma analizzato]_Media_Com-bined_potency_function_protection_diablo.csv* contenuto nella cartella `[destination]`, dove *[programma analizzato]* è il nome del programma che si sta analizzando.

Appendice C

Implementazione Metrics_analysis.py

In questa Appendice verrà mostrato e descritto come si è deciso di implementare lo script *Metrics_analysis.py*, script scritto con il linguaggio di programmazione *python*. Anche se non espressamente scritto, tutti gli errori generati vengono gestiti e riportati nei file di log propri di ogni programma analizzato.

C.1 Import

Sono stati importati i seguenti moduli:

- *os*: tale modulo fornisce le funzioni necessarie per interagire con il sistema operativo;
- *sys*: tale modulo fornisce le funzioni necessarie per processare gli argomenti passati da linea di comando allo script *Metrics_analysis.py*;
- *string*: tale modulo fornisce le funzioni necessarie per processare le stringhe;
- *re*: tale modulo fornisce le funzioni necessarie per poter utilizzare le espressioni regolari;
- *ConfigParser*: tale modulo fornisce le funzioni necessarie per analizzare i file di configurazione;
- *pdb*: tale modulo fornisce le funzioni necessarie per poter eseguire il debug, se necessario;
- *json*: tale modulo fornisce le funzioni necessarie per leggere e scrivere file in formato json.

C.2 Definizione funzioni

Sono state create delle funzioni con lo scopo di tenere ordinato il codice sorgente e per evitare di ripetere identiche parti di codice in differenti parti dello script.

C.2.1 Run_Diablo

Lo scopo di questa funzione è eseguire lo strumento *Diablo* ed estrarre le metriche sui file binari. Per fare ciò, i passaggi eseguiti risultano essere:

1. creazione di una stringa contenente parametri estratti dal file di configurazione associato al programma che si vuole analizzare, da passare allo strumento di *Diablo* per la corretta compilazione del programma[1]:

   - cc: tipo di compilatore che si vuole utilizzare;
   - cppflags: flag di pre-processing;
   - lflags: le librerie che il compilatore dovrà utilizzare;
   - cflags: flag da passare al compilatore;
   - ldflags: flag da passare al linker.

   A questo elenco devono essere aggiunti alcuni paramatri necessari allo strumento di *Diablo*, quali:

   - CFLAGS_Diablo: flag necessari al compilatore di *Diablo*;
   - LDFLAGS_Diablo: flag da passare al linker di *Diablo*;
   - file.map: un file chiamato con il nome del file binario che verrà generato[2], con estensione *.map*.

   I parametri CFLAGS_Diablo e LDFLAGS_Diablo sono definiti in fase di inizializzazione (cfr. Appendice C.3.1) come variabili globali.

2. preparazione e creazione del file delle annotazioni, il file *annotations.json*, da passare allo strumento di *Diablo*. Per maggiori dettagli sul file delle annotazioni si rimanda alla Appendice A;

3. esecuzione del *Makefile*, richiamando la funzione `Exec_command` (cfr. Appendice C.2.6), per la creazione del file binario che verrà utilizzato dallo strumento di *Diablo* per generare il report contenente il valore delle metriche;

4. lanciare lo strumento *Diablo*, con i parametri appena creati, ovvero la stringa `flag` e il file delle annotazioni, sul file che si sta analizzando, richiamando la funzione `Exec_command` (cfr. Appendice C.2.6);

5. copiare sulla *Board_ARM*, variabile globale contenente l'indirizzo ip al quale la board ARM è raggiungibile, il file binario appena generato e creare una cartella per contenere i file che verranno generati dall'esecuzione di tale binario;

6. testare il corretto funzionamento del file binario generato. Per fare ciò bisogna comparare i valori di ritorno generati dal file binario generato da *Diablo* con i valori di ritorno generati dal binario generato tramite il *Makefile*, avendo a disposizione gli stessi comandi da eseguire[3]. Prima che questa funzione venga chiamata sarà generato un dizionario chiamato `clean_makecmd_ret` contenete come *chiavi* i comandi da eseguire e come *valori* il valore ritornato dall'esecuzione del comando eseguito sul binario creato tramite *Makefile*. Nel caso in cui i valori di ritorno tra file in chiaro e file offuscati siano diversi, il file binario generato da *Diablo* verrà cancellato;

7. cleaning: cancellazione del binario e della cartella precedentemente creati sulla board ARM.

Tale funzione accetta cinque parametri:

1. `key_diablo`: stringa associata al nome dell'offuscamento che si vuole applicare, se non deve essere applicato alcun offuscamento sarà riportata la stringa `protection[none]`;

[1] Parametri `[CC]`, `[CPPFLAGS]`, `[LFLAGS]`, `[CFLAGS]`, `[LDFLAGS]` presenti nel file di configurazione appartenente al programma che si sta analizzando.

[2] Parametro `[binary]` presente nel file di configurazione appartenente al programma che si sta analizzando.

[3] Parametro `[cmd]` presente nel file di configurazione appartenente al programma che si sta analizzando

2. `diablo_obf`: stringa contenente i parametri necessari per poter applicare la protezione selezionata, se non deve essere applicato alcun offuscamento sarà riportata la stringa `protection (none)`;

3. `work_dir`: stringa contenente il path in cui si trovano i file che si vogliono sottoporre all'analisi dello strumento di *Diablo*;

4. `report_output`: stringa contenente il path in cui si vuole memorizzare il report contenente il valore delle metriche generato dallo strumento di *Diablo*;

5. `tigress_obf`: stringa contenente il nome dell'offuscamento apportato dallo strumento *Tigress*, se nessun offuscamento è stato apportato la stringa conterrà il valore `none`.

Questa funzione ritorna la stringa `Error` in caso di errore, altrimenti non ha alcun valore di ritorno.

C.2.2 Delete_output_created

Lo scopo di questa funzione è rimuovere gli elementi presenti nella lista del campo `rm_out` del file di configurazione associato al programma che si vuole analizzare. Si veda la definizione di tale parametro in Appendice B.1.4.

Questa funzione non riceve alcun parametro e non ritorna alcun valore.

C.2.3 Check_main

Lo scopo di questa funzione è cercare se nel file `_file` passato a questa funzione è presente la funzione `main`. Per far ciò si invoca lo strumento *ctags* (cfr. Appendice B.2.1) e si analizza il risultato ottenuto.

Questa funzione accetta un unico parametro `_file` ovvero percorso ed il nome file, in formato *.c*, che si vuole analizzare. La funzione ritorna il valore 1 se è presente la funzione chiamata `main`, ritorna 0 altrimenti.

C.2.4 Modify_if_main

Lo scopo di questa funzione è modificare la funzione chiamata `main` aggiungendo una lista di funzioni da richiamare prima di eseguire le istruzioni riportate nel `main` stesso.

Per fare ciò, partendo dal file che si sta analizzando, prima si ricava il numero di linea alla quale è stata implementata la funzione `main` tramite il comando `grep` e successivamente, nella linea successiva alla funzione cercata, si inseriscono le funzioni da inserire, una per linea tramite il comando `sed`.

Questa funzione accetta due parametri:

1. `file_with_main`: il percorso ed il nome del file in cui è presente la funzione `main` del programma che si sta analizzando;

2. `list_main_mod`: una lista contenete le funzioni che si vogliono inserire.

Questa funzione non ritorna alcun valore.

C.2.5 Modify_if_not_main

Lo scopo di questa funzione è trovare il pattern `void main(void)` all'interno del file che si sta analizzando e, se presente, rinominarlo con la stringa nota `void main_[nome file]` ove `[nome file]` è il nome del file che si sta analizzando (in tal modo tale nome rimane univoco), il tutto eseguito tramite il comando `sed`.

Questa funzione accetta quattro parametri:

1. mod_file: il percorso ed il nome del file al quale si vuole apportare tale modifica;
2. name_file: il nome del file al quale si vuole apportare tale modifica;
3. location: non utilizzato;
4. cmd: il nome del target *Makefile*[4].

Questa funzione ritorna la stringa `append` se è avvenuta la sostituzione cercata, 0 altrimenti.

C.2.6 Exec_command

Lo scopo di questa funzione è eseguire il comando che viene passato come parametro. Per mezzo del parametro `see_output`, settato nel file di configurazione dello script *Metrics_analysis.py* (cfr. Appendice B.1.4), si decide se stampare o non stampare a video l'output generato da tale comando. In caso di errore l'output viene rediretto su un file di testo per tenere traccia dell'errore generato.

Questa funzione accetta due parametri:

1. `command_to_run`: stringa contenete il comando da voler eseguire;
2. `file_err`: il percorso ed il nome del file su cui si vuole redirigere il flusso di informazioni in caso di errori del comando eseguito, ovvero lo *standar error*.

La funzione ritorna, a sua volta, il valore ritornato dal comando `os.system`.

C.2.7 Create_report_frama_c

Lo scopo di questa funzione è eseguire il comando per estrarre le metriche sui file sorgenti tramite lo strumento *Frama-c*.

Questa funzione accetta quattro parametri:

1. name_file: il nome del file *.c* che si vuole analizzare;
2. _input: il nome del file pre-elaborato *.i*, compreso di path, da passare allo strumento *Frama-c* sul quale si vogliono estrarre le metriche;
3. _output: il percorso ed il nome del file su cui lo strumento *Frama-c* deve stampare il report generato;
4. file_err: il nome del file su cui si vuole redirigere il flusso di informazioni in caso di errori del comando eseguito, ovvero lo *standar error*.

Questa funzione non ritorna alcun valore.

[4]Parametro `[target_Makefile]` presente nel file di configurazione appartenente al programma che si sta analizzando.

C.2.8 Create_compiled

Lo scopo di questa funzione è creare i programmi compilati, richiamando a sua volta la funzione `Exec_command` (cfr. Appendice C.2.6), eseguendo il comando *make* sul *Makefile* relativo al programma che si vuole analizzare.

Questa funzione accetta i seguenti parametri:

1. `name_prog`: il nome del file programma che si sta analizzando;
2. `_input`: il percorso al quale si trova la directory principale del programma che si sta analizzando;
3. `_output`: il percorso nel quale si vuole copiare la cartella del programma analizzato contenente il file binario, una volta generato dal *Makefile*;
4. `make_dir_rel`: il percorso relativo dalla directory principale del programma che si sta analizzando alla cartella in cui è presente il Makefile, parametro `_dirmakefile` presente nel file di configurazione appartenente al programma che si sta analizzando;
5. `_nameMakefile`: il nome dato al *Makefile*, parametro `_nameMakefile` presente nel file di configurazione appartenente al programma che si sta analizzando;
6. `make_cmd`: il nome del target *Makefile* che si vuole eseguire[4];
7. `flag`: stringa contenente la concatenazione dei flag da passare al compilatore, comprendente:
   - `cc`: tipo di compilatore che si vuole utilizzare;
   - `cppflags`: flag di pre-processing;
   - `lflags`: le librerie che il compilatore dovrà utilizzare;
   - `cflags`: flag da passare al compilatore;
   - `ldflags`: flag da passare al linker.

   Tali parametri vengono estratti dal file di configurazione associato al programma che si vuole analizzare[1].
8. `file_err`: il nome del file su cui si vuole redirigere il flusso di informazioni in caso di errori del comando eseguito, ovvero lo *standar error*.
9. `where`: parametro che può assumere due valori:
   - `Compiled/Clean` se si tratta di file compilati senza offuscamenti ad essi associati;
   - `Compiled/obf/key` se si tratta di file compilati con offuscamenti ad essi associati, ove *key* indica il nome dell'offuscamento apportato;

La funzione ritorna il valore ritornato a sua volta dalla funzione `Exec_command` (cfr. Appendice C.2.6).

C.3 Implementazione del diagramma di flusso

Ogni sezione è dedicata ad uno specifico blocco raffigurato in Figura 4.3.

C.3.1 Inizializzazione

In questa fase vengono raccolti i parametri di input passati allo script *Metrics_analysis.py* da linea di comando, configurati i parametri e le variabili di ambiente necessari al corretto funzionamento degli strumenti quali *Diablo e Tigress* ed infine configurate le variabili, le liste ed i dizionari contenenti i valori ed i parametri scelti per creare la struttura di file-system per memorizzare l'output prodotto (cfr. Sezione 4.4.2) e per apportare gli offuscamenti selezionati.

obf_tigress

`obf_tigress` è un dizionario contenete i parametri per applicare gli offuscamenti selezionati offerti dallo strumento di *Tigress*. Contiene come *chiave* il nome dell'offuscamento da applicare e come *valore* i parametri necessari per ottenere l'offuscamento selezionato. La Figura C.1 mostra le configurazioni scelte.

```
obf_tigress={
"Flatten" : "--Transform=Flatten --Functions=* --FlattenDispatch=switch --
    FlattenSplitBasicBlocks=true --FlattenRandomizeBlocks=true",
"Opaque" : "--Transform=InitOpaque --InitOpaqueCount=1 --Functions=* --Transform=
    AddOpaque --Functions=* --AddOpaqueKinds=call --AddOpaqueStructs=list",
"AntiBranchAnalysis" : "--Transform=InitOpaque --InitOpaqueCount=1 --Functions=* --
    Transform=InitBranchFuns --Functions=*",
"EncodeArithmetic" : "--Transform=EncodeArithmetic --Functions=*"
}
```

Figura C.1: Dizionario `obf_tigress`.

I parametri più importanti risultano essere[5]:

- `--Transform`: l'offuscamento da applicare;
- `--Function`: a quali funzioni applicare l'offuscamento, il simbolo * indica tutte le funzioni;
- `--InitOpaqueCount`: il numero di operatori opachi da utilizzare;

obf_diablo e relativi parametri

`obf_diablo` è un dizionario contenete i parametri per applicare gli offuscamenti selezionati offerti dallo strumento di *Diablo*. Contiene come *chiave* il nome dell'offuscamento da applicare e come *valore* i parametri necessari per ottenere l'offuscamento selezionato. La Figura C.2 mostra le configurazioni scelte.

```
obf_diablo={
"function_flattening_10%" : "protection(obfuscations,enable_obfuscation(flatten_function:percent_apply
    =10))",
"branch_functions_10%": "protection(obfuscations,enable_obfuscation(branch_function:percent_apply=10))
    ",
"opaque_predicates_10%": "protection(obfuscations,enable_obfuscation(opaque_predicate:percent_apply
    =10))",
"function_flattening_30%" : "protection(obfuscations,enable_obfuscation(flatten_function:percent_apply
    =30))",
"branch_functions_30%": "protection(obfuscations,enable_obfuscation(branch_function:percent_apply=30))
    ",
"opaque_predicates_30%": "protection(obfuscations,enable_obfuscation(opaque_predicate:percent_apply
    =30))",
"function_flattening_50%" : "protection(obfuscations,enable_obfuscation(flatten_function:percent_apply
    =50))",
"branch_functions_50%": "protection(obfuscations,enable_obfuscation(branch_function:percent_apply=50))
    ",
"opaque_predicates_50%": "protection(obfuscations,enable_obfuscation(opaque_predicate:percent_apply
    =50))"
}
```

Figura C.2: Dizionario `obf_tigress`.

[5] All'url http://tigress.cs.arizona.edu/transformPage/index.html sono disponibili le informazioni per tutti i parametri utilizzati.

Per mezzo del parametro `percent\_apply=` si può decidere la percentuale con cui si è intenzionati ad applicare tale protezione. L'Appendice A riporta maggiori dettagli in merito.

Dopo aver definito il dizionario `obf_diablo`, vengono create due variabili globali (Figura C.3) contenenti i parametri necessari allo strumento di *Diablo* in fase di compilazione.

```
CFLAGS_Diablo = "-g -marm -mcpu=cortex-a8 -Wl,--hash-style=sysv -Wl,--no-demangle -
    Wl,--no-merge-exidx-entries"
LDFLAGS_Diablo= "-Bdynamic -Wl,-Map,"
```

Figura C.3: Variabili globali per lo strumento *Diablo*.

Input

Lo script *Metrics_analysis.py* accetta due parametri da linea di comando:

```
./Metrics_analysis.py [source] [destination]
```

- `[source]`: rappresenta la directory principale in cui si trovano le sotto-cartelle dei programmi da voler analizzare;
- `[destination]`: rappresenta la directory in cui si vuole memorizzare output prodotto dallo script.

Viene eseguito un controllo sui parametri passati come input e nel caso non fossero esattamente due, lo script blocca la sua esecuzione. In caso contrario, il parametro `source` viene memorizzato all'interno di una variabile globale chiamata `source_file` mentre il parametro `destination` viene utilizzato per cominciare a creare il file system di output come mostrato in Figura 4.2, senza la creazione delle cartelle denominate con il nome dell'offuscamento riportato, ed il percorso a tali cartelle generate viene memorizzato in variabili globali.

Variabili file di configurazione *Metrics_analysis.py*

Viene, per prima, generata una variabile globale denominata `here` contenente il percorso alla cartella all'interno della quale è presente lo script *Metrics_analysis.py*.

In seguito, tramite il comando `grep`, vengono create le variabili globali contenenti i valori dei parametri contenuti nel file di configurazione *conf.txt* (cfr. Appendice B.1.4). La Figura C.4 mostra le variabili create.

Variabili per il file dei log

Partendo dalla variabile globale `here` precedentemente creata, vengono, prima rimossi nel caso esistessero e poi ricreati, la cartella *Log* ed il file *LOGFILE.txt* (cfr. Sezione 4.4.3). Successivamente vengono create le variabili globali contenenti i percorsi alla cartella ed al file appena creati.

La Figura C.5 mostra i passaggi descritti.

Sotto-cartelle per l'output di *Tigress*

Questo passaggio permette la completa creazione del file system di output progettato. La Figura C.6 mostra il ciclo *for* che, tramite selezione di tutte le *chiavi* presenti nel dizionario `obf\_tigress`, permette la creazione di tutte le sotto-cartelle di output denominate con il nome dell'offuscamento da apportare.

```
here=sys.argv[0].replace("Metrics_analysis.py","")
see_output=os.popen("grep␣'see_output'␣"+here+"/sub/conf.txt␣|␣cut␣-f␣2␣-d=").read().rstrip().replace
    ("␣","")
dir_tigress=os.popen("grep␣'tigress'␣"+here+"/sub/conf.txt␣|␣cut␣-f␣2␣-d=").read().rstrip().replace("␣",
    "")
max_dim_file_obf=os.popen("grep␣'max_dim_file_obf'␣"+here+"/sub/conf.txt␣|␣cut␣-f␣2␣-d=").read().
    rstrip().replace("␣","")
CC_Diablo=os.popen("grep␣'CC_Diablo'␣"+here+"/sub/conf.txt␣|␣cut␣-f␣2␣-d=").read().rstrip().
    replace("␣","")
OBF_Diablo=os.popen("grep␣'OBF_Diablo'␣"+here+"/sub/conf.txt␣|␣cut␣-f␣2␣-d=").read().rstrip().
    replace("␣","")
Board_ARM=os.popen("grep␣'Board_ARM'␣"+here+"/sub/conf.txt␣|␣cut␣-f␣2␣-d=").read().rstrip().
    replace("␣","")
keep_obf=os.popen('grep␣"keep_obf"␣'+here+'/sub/conf.txt␣|␣cut␣-f␣2␣-d=').read().rstrip().replace("␣",
    "")
keep_report_frama=os.popen('grep␣"keep_report_frama"␣'+here+'/sub/conf.txt␣|␣cut␣-f␣2␣-d=').read().
    rstrip().replace("␣","")
keep_compiled=os.popen('grep␣"keep_compiled"␣'+here+'/sub/conf.txt␣|␣cut␣-f␣2␣-d=').read().rstrip().
    replace("␣","")
keep_diablo=os.popen('grep␣"keep_diablo"␣'+here+'/sub/conf.txt␣|␣cut␣-f␣2␣-d=').read().rstrip().
    replace("␣","")
```

Figura C.4: Variabili globali del file *conf.txt*.

```
os.system("rm␣-r␣"+here+"/LOGFILE.txt")
os.system("rm␣-r␣"+here+"/Log")
os.system("mkdir␣"+here+"/Log")
LOGFILE=open(here+"/LOGFILE.txt",'a')
LOG_base=here+"Log/"
```

Figura C.5: Variabili globali per i log.

Sotto-cartelle per l'output di *Diablo*

Questo passaggio permette la completa creazione del file system di output progettato. La Figura C.7 mostra il ciclo *for* che, tramite selezione di tutte le *chiavi* presenti nel dizionario `obf\_diablo`, permette la creazione di tutte le sotto-cartelle di output denominate con il nome dell'offuscamento da apportare.

Variabili d'ambiente

La Figura C.8 mostra i passaggi eseguiti per configurare le variabili di ambiente prima di concludere la parte relativa all'inizializzazione.

Per il corretto funzionamento dello strumento *Tigress*, infatti, bisogna:

- creare e configurare la variabile d'ambiente `TIGRESS\_HOME` che dovrà contenere il path assoluto in cui lo strumento *Tigress* è stato installato;
- aggiungere alla lista di path contenuti nella variabile d'ambiente `PATH` il path assoluto in cui lo strumento *Tigress* è stato installato;

Il path assoluto in cui lo strumento *Tigress* è stato installato è contenuto nella variabile globale chiamata `dir_tigress`, configurata come descritto nella Sezione C.3.1.

C.3.2 "Per ogni programma"

In questa fase, si sviluppa l'intero processo di analisi apportato dallo script *Metrics_analysis.py*. Per mezzo del comando:

```
for key in obf_tigress.keys():
    os.system("mkdir -p "+file_obf_tigress+"/"+key)
    os.system("mkdir -p "+report_frama_obf_tigress+"/"+key)
    os.system("mkdir -p "+report_diablo_obf_tigress+"/"+key)
    os.system("mkdir -p "+compiled_obf_tigress+"/"+key)
```

Figura C.6: Creazione cartelle offuscamenti *Tigress*.

```
for key in obf_diablo.keys():
    os.system("mkdir -p "+report_diablo_obf_diablo+"/"+key)
    os.system("mkdir -p "+compiled_obf_diablo+"/"+key)
```

Figura C.7: Creazione cartelle offuscamenti *Diablo*.

```
list_prog = os.popen("ls -d "+source_file+"/*/").readlines()
```

nella variabile `list_prog` vengono memorizzati tutti i nomi delle cartelle dei programmi presenti nella variabile `source_file`, ovvero tutti i programmi che si desidera siano analizzati.

Tramite il ciclo *for*:

```
for _dir in list_prog:
```

sarà preso in considerazione un unico programma per volta (`_dir`), fino al termine dei programmi contenuti nella variabile `list_prog`.

Si riportano di seguito i passaggi principali eseguiti successivamente:

1. nella variabile `name_prog` viene memorizzato il nome del programma analizzato;
2. nella variabile `parser` viene memorizzato il file *confmake_[name].ini* (cfr. Appendice B.1.4);
3. vengono create le sotto-cartelle, chiamate con il nome del programma che si sta analizzando, per la cartella *pre_analized*[6] e per le sotto-cartelle *clean*[6] che dovranno contenere i report del programma non offuscato;
4. nella cartella *pre_analized* vengono creati i file *.csv* sia per lo strumento *Frama-c* sia per lo strumento *Diablo* contenenti i rispettivi header (cfr. Appendice B.2.3);
5. per le sotto cartelle *obf*[6] vengono create le sotto-cartelle chiamate con i nomi degli offuscamenti selezionati sia per lo strumento di *Tigress* sia per lo strumento di *Diablo* e per ognuna di queste sotto-cartelle viene creata un'ulteriore sotto cartella chamata con il nome del programma che si sta analizzando.

C.3.3 "Per ogni target del Makefile"

In questa fase si selezionano uno ad uno i *target* presenti nella variabile `parser` selezionandoli tramite il comando:

```
for make_cmd in parser.sections():
```

così da poter analizzare uno stesso programma con diverse configurazioni.

Per ogni *target* presente, quindi, vengono estratti tutti i parametri ad esso associati e memorizzati nella propria variabile corrispondente come mostrato dalla Figura C.9.

Successivamente viene creata una variabile chiamata `flag`, i passaggi eseguiti sono riportati in Figura C.10, come concatenazione delle variabili `cc`, `cflag`, `lflag`, `ldflag` contenente rispettivamente i parametri `CC`, `CFLAGS`, `LFLAGS` e `LDFLAGS`[7]

[6] Si veda Sezione 4.2.

[7] Si veda Appendice B.1.4.

```
os.environ["TIGRESS_HOME"] = dir_tigress
os.environ["PATH"] = os.path.expandvars("${PATH}")+":"+dir_tigress
```

Figura C.8: Configurazione variabili d'ambiente.

```
make_dir_rel=parser.get(make_cmd, '_dirMakefile').replace("␣","")
_nameMakefile=parser.get(make_cmd, '_nameMakefile').replace("␣","")
cc=parser.get(make_cmd, 'CC')
cflag=parser.get(make_cmd, 'CFLAGS')
lflag=parser.get(make_cmd, 'LFLAGS')
ldflag=parser.get(make_cmd, 'LDFLAGS')
cppflag=parser.get(make_cmd, 'CPPFLAGS')
command=parser.get(make_cmd, 'cmd')
list_file_used=parser.get(make_cmd, 'file').replace("␣","")
rm_out_created=parser.get(make_cmd, 'rm_out').replace("␣","")
binary=parser.get(make_cmd, 'binary').replace("␣","")
dir_obj_file=parser.get(make_cmd, 'dir_obj_file').replace("␣","")
```

Figura C.9: Configurazione variabili per comando *target*.

C.3.4 Report frama-c (in chiaro)

In questa fase vengono preparate le variabili che devono essere passate come parametri alla funzione `Create_report_frama_c`, si veda l'Appendice C.2.7, per ottenere i valore delle metriche calcolate sul programma sorgente e non offuscato. Inoltre tramite il comando:

```
for _file in list_file_used.split(","):
```

i file *.c* contenuti nella variabile `list_file_used` vengono selezionati uno per volta. Per ogni file selezionato, all'interno del ciclo *for*, viene appunto chiamata la funzione `Create_report_frama_c` con i relativi parametri appositamente configurati.

La Figura C.11 mostra il codice utilizzato per la creazione dei report dello strumento *Frama-c*. La variabile chiamata `tmp` viene utilizzata per specificare il percorso di base utilizzato per memorizzare il report creato e la variabile `path` indica il percorso di base in cui si trovano i file che si vogliono sottoporre all'analisi dello strumento *Frama-c*. Prima di passare i file con estensione *.c* allo strumento *Frama-c* bisogna pre-compilarli, operazione effettuata tramite il comando `os.system("cc -E "+cppflag+" -o "+path.rstrip()+".i "+path.rstrip())`.

C.3.5 Compilazione

Dopo avere processato i dati tramite lo strumento *Frama-c*, il programma analizzato viene compilato. Per far ciò si utilizza la funzione `Create_compiled` descritta in Appendice C.2.8.

La Figura C.12 mostra i passaggi eseguiti per la compilazione del programma. La variabile chiamata `tmp` viene utilizzata per specificare il percorso di base utilizzato per memorizzare il file binario in chiaro generato.

Successivamente alla fase di compilazione, se questa viene eseguita correttamente (il valore di ritorno della funzione `Create_compiled` è pari a zero), viene creato un dizionario chiamato `clean_makecmd_ret`. In tale dizionario viene memorizzata come *chiave* il comando eseguito, estratto dalla variabile `command`, e come *valore* il valore di ritorno dell'esecuzione del comando stesso. Per eseguire il comando viene richiamata la funzione `Exec_command`, descritta in Appendice C.2.6.

Infine viene chiamata la funzione `Delete_output_created` (cfr. Appendice C.2.2), utilizzata per eliminare eventuali file creati dall'esecuzione del comando eseguito.

La Figura C.13 mostra i passaggi eseguiti per la creazione del dizionario `clean_makecmd_ret`. La variabile chiamata `tmp` viene utilizzata per specificare il percorso di base utilizzato per lanciare il comando che si vuole eseguire.

```
flag=""
if(cc!=""):
    flag=" CC="'"+cc+"'""
if(cppflag!="" or cflag!=""):
    flag=flag+" CFLAGS="'"+cflag+"'""
if(lflag!=""):
    flag=flag+" LFLAGS="'"+lflag+"'""
if(ldflag!=""):
    flag=flag+" LDFLAGS="'"+ldflag+"'""
```

Figura C.10: Creazione variabile *flag*.

```
print "\033[32;1m-----(Create_report_frama-c_source_file_for_command_:_"+make_cmd+"_)
    -----\033[0m\n"
tmp=report_frama_clean+"/"+name_prog+"/"+name_prog+"_"+make_cmd
os.system("mkdir_"+tmp)
for _file in list_file_used.split(","):
    _file=re.sub("^(_)*","",_file) #delete space* if exist
    name_file=re.sub("^.*/","",_file).rstrip()
    path=source_file+"/"+name_prog+"/"+_file
    #pre-compiled verion with cflag
    os.system("cc_-E_"+cppflag+"_-o_"+path.rstrip()+".i_"+path.rstrip())
    #print "cc -E "+cppflag+" -o "+path.rstrip()+".i "+path.rstrip()
    Create_report_frama_c(name_file,path.rstrip()+".i",tmp+"/"+name_file+".txt",name_prog+"_"+
        make_cmd+"_"+name_file+".txt")
    os.system("rm_"+path.rstrip()+".i_")
```

Figura C.11: Report frama-c (in chiaro).

C.3.6 Report diablo (in chiaro)

In questa fase vengono preparate le variabili da passare come parametri alla funzione `Run_Diablo`, si veda Appendice C.2.1, per ottenere il valore delle metriche calcolate sul programma binario e non offuscato. Viene creata una cartella temporanea chiamata `dir_tmp` in cui copiare momentaneamente la cartella contenente il programma che si sta analizzando, la variabile `_output` contiene il percorso al programma che si sta analizzando (nella cartella temporanea appena creata) e la variabile `report_output` contiene il percorso in cui si vuole memorizzare il report generato dallo strumento *Diablo*.

La Figura C.14 mostra il codice utilizzato per la creazione dei report dello strumento *Diablo*. Poiché si vuole richiamare lo strumento *Diablo* senza applicare nessuna protezione, si passano manualmente come stringhe i parametri `protection[none]` e `protection(none)` alla funzione `Run_Diablo`. Alla fine la cartella temporanea `dir_tmp` viene rimossa.

C.3.7 Offuscamenti e report diablo

In questa fase vengono preparate le variabili da passare come parametri alla funzione `Run_Diablo`, si veda Appendice C.2.1, per ottenere il valore delle metriche calcolate sul programma binario e offuscato con gli offuscamenti riportati nel dizionario `obf_diablo`. Viene creata una cartella temporanea chiamata `dir_tmp` in cui copiare momentaneamente la cartella contenente il programma che si sta analizzando, la variabile `_output` contiene il percorso al programma che si sta analizzando (nella cartella temporanea appena creata) e la variabile `report_output` contiene il percorso in cui si vuole memorizzare il report generato dallo strumento *Diablo*.

La Figura C.15 mostra il codice utilizzato per la creazione dei report dello strumento *Diablo*. Tramite il comando:

```
for obf_Diablo in obf_diablo.keys():
```

nella variabile `obf_Diablo` viene memorizzata la *chiave* del dizionario `obf_diablo`. In questo modo, quando viene richiamata la funzione `Run_Diablo` si possono passare come parametri la

```
print "\033[32;1m-----(Execute_make_for_source:_"+make_cmd+")-----\033[0m"
tmp=compiled_clean+"/"+name_prog+"/"+name_prog+"_"+make_cmd
res_compiler_clean=Create_compiled(name_prog,_dir,tmp,make_dir_rel,_nameMakefile,make_cmd,flag,"
    error_compiled_"+name_prog+"_"+make_cmd,"Compiled/Clean")
```

Figura C.12: Compilazione.

```
tmp=compiled_clean+"/"+name_prog+"/"+name_prog+"_"+make_cmd
clean_makecmd_ret={}
if(res_compiler_clean==0):
for cmd in command.split(","):
cmd=re.sub("^(_)*","",cmd) #delete space* if exist
print "\n\033[32;1m-----(Run_command_:_"+cmd+"_and_capture_return_value)-----\033[0m"
clean_makecmd_ret.update({cmd : Exec_command(tmp+"/"+cmd,"tmp.txt")})
os.system("rm_-f_"+LOG+"tmp.txt")
#delete outputs created
Delete_output_created()
```

Figura C.13: Creazione dizionario *clean_makecmd_ret*.

chiave ed il *valore* di tale dizionario per apportare al programma analizzato tutte gli offuscamenti selezionati.

Infine se la funzione `obf_Diablo` ritorna un valore diverso dal codice di errore (la stringa `Error`), il file binario generato (nella cartella `dir_tmp`) da *Diablo* viene memorizzato nella corretta cartella dedicata del file-system di output precedentemente illustrato e descritto (cfr. Sezione 4.4.2), ovvero nell'apposita sotto cartella relativa al percorso *compiled/obf/diablo/*. La cartella temporanea `dir_tmp` viene rimossa.

C.3.8 Offuscamenti tigress

Tramite il comando:

```
for key in obf_tigress.keys()
```

nella variabile `key` viene memorizzata la *chiave* del dizionario `obf_tigress` utilizzata in seguito per apportare il corretto offuscamento al programma analizzato. Il ciclo *for* viene utilizzato per scorrere tutti gli elementi presenti nel dizionario e quindi apportare tutti gli offuscamenti selezionati. Vengono inoltre create:

- una directory chiamata `file_notobf` contenente un file, memorizzato nella variabile `f`, in cui verranno scritti tutti i file che non verranno offuscati dallo strumento *Tigress*;
- una directory temporanea chiamata `tmp_tigress` in cui copiare momentaneamente la cartella contenente il programma che si sta analizzando;
- una variabile chiamata `tmp_tigress` contenente il percorso al programma che si sta analizzando (nella cartella temporanea appena creata);
- una variabile chiamata `file_with_main` che conterrà il nome del file contenente la funzione `main`;
- una lista chiamata `list_file_mod_main` che conterrà il nome dei file che conterranno la funzione `main` che lo strumento *Tigress* inserisce in automatico quando non trova la funzione `main` propria del programma stesso.

Successivamente tramite il comando:

```
for _file_to_obf in list_file_used.split(","):
```

```
os.system("mkdir -p "+dir_tmp)
_output=dir_tmp+"/"+name_prog+"_"+make_cmd
report_output=report_diablo_clean+"/"+name_prog+"/"+name_prog+"_"+make_cmd
os.system("mkdir "+report_output)
os.system("cp -r "+_dir+" "+_output)
print "\n\033[32;1m-----(Run Diablo with no protection on clean program to extract metrics )
    -----\033[0m"
Run_Diablo("protection[none]","protection(none)",_output,report_output,"nope")
os.system("rm -f -r "+dir_tmp)
```

Figura C.14: Report diablo (in chiaro).

```
name_binary=re.sub("^.*/","",binary)
os.system("mkdir -p "+dir_tmp)
for obf_Diablo in obf_diablo.keys():
    ret=""
    _output=dir_tmp+"/"+name_prog+"_"+make_cmd
    report_output=report_diablo_obf_diablo+"/"+obf_Diablo+"/"+name_prog+"/"+name_prog+"_"+
        make_cmd
    os.system("mkdir "+report_output)
    os.system("cp -r "+_dir+" "+_output)
    print "\n\033[32;1m-----(Run Diablo : "+obf_Diablo+" to obf program and extract metrics )
        -----\033[0m"
    ret=Run_Diablo(obf_Diablo,obf_diablo[obf_Diablo.rstrip()],_output,report_output,"nope")
    #copy binary for diablo_obf in compiled/obf/diablo
    if(ret!="Error"):
        os.system("mkdir "+compiled_obf_diablo+"/"+obf_Diablo+"/"+name_prog+"/"+name_prog+"_
            "+make_cmd)
        os.system("cp -r "+report_output+"/Diablo_"+name_binary+" "+compiled_obf_diablo+"/"+
            obf_Diablo+"/"+name_prog+"/"+name_prog+"_"+make_cmd)
os.system("rm -f -r "+dir_tmp)
```

Figura C.15: Offuscamenti e report diablo.

si esegue il ciclo *for* per scorrere tutti, i file con estensione *.c*, presenti nella variabile `list_file_used` e memorizzati nella variabile `_file_to_obf`. All'interno di questo ciclo vengono eseguite numerose operazioni:

1. viene chiamata la funzione `Check_main` descritta in Appendice C.2.3 e nel caso la funzione ritorni il valore 1 il nome del file analizzato viene memorizzato nella variabile `file_with_main` (Figura C.16);
2. viene eseguito il controllo sulla grandezza del file `_file_to_obf` che si sta analizzando. Se la grandezza del file è maggiore della soglia imposta dalla variabile `max_dim_file_obf` passa al file successivo altrimenti si continua con l'analisi (Figura C.17);
3. si crea una variabile chiamata `name_file` in cui si memorizza il nome del file che si sta analizzando, si crea una variabile chiamata `location` contenente il percorso al file `_file_to_obf` che si sta analizzando, si rinomina il file pre-ponendo la particella `tmp_` al nome del file e si crea la variabile `after_obf` contenente il nome del file prima che venisse rinominato, compreso di percorso (tale variabile viene utilizzata per reindirizzare l'output prodotto dallo strumento *Tigress*, ovvero conterrà il codice sorgente del programma offuscato).

A questo punto per mezzo della funzione `Exec_command`, descritta in Appendice C.2.6, si esegue lo strumento *Tigress* per apportare l'offuscamento selezionato (Figura C.18) sul file sorgente. In funzione del valore di ritorno della funzione `Exec_command`:

- se la funzione ritorna un valore diverso da 0 significa che l'offuscamento non è terminato correttamente e pertanto si riportano i vari errori riscontrati nei corrispettivi file di log, viene eliminato il file `after_obf` contenente l'output prodotto dallo strumento *Tigress*, si ripristina il corretto nome del file `_file_to_obf` cancellando dal nome la particella `tmp_` precedentemente inserita e si inserisce il nome del file `name_file` nel file contenuto dalla variabile `f` (Figura C.19);

```
if(Check_main(_file_to_obf)==1):
    file_with_main=_file_to_obf
```

Figura C.16: Check_main.

```
if int(os.path.getsize(_file_to_obf))>int(max_dim_file_obf):
    #case too bigger
    f.write(name_file+"\n")
    print "\n\033[31;1m----------(File_:_'"+_file_to_obf+"'_will_be_too_bigger)
          ----------\nThis_file_will_not_protected\033[0m\n"
    LOGFILE.write("\nFile_:_'"+_file_to_obf+"'_too_bigger_and_will_not_protected\n\n")
    continue
```

Figura C.17: Controllo grandezza file.

- altrimenti se la funzione ritorna un valore uguale a 0 possiamo dividere tre casistiche:
  - se la grandezza del file offuscato `after_obf` è maggiore della soglia imposta dalla variabile `max_dim_file_obf` allora si inserisce il nome del file `name_file` nel file contenuto dalla variabile `f`, si elimina il file `after_obf` contente l'output prodotto dallo strumento *Tigress*, si ripristina il corretto nome del file `_file_to_obf` cancellando dal nome la particella `tmp_` precedentemente inserita e si riportano i vari errori riscontrati nei corrispettivi file di log (Figura C.20);
  - se la grandezza del file offuscato `after_obf` è esattamente pari a 0 allora si inserisce il nome del file `name_file` nel file contenuto dalla variabile `f`, si elimina il file `after_obf` contente l'output prodotto dallo strumento *Tigress*, si ripristina il corretto nome del file `_file_to_obf` cancellando dal nome la particella `tmp_` precedentemente inserita e si riportano i vari errori riscontrati nei corrispettivi file di log (Figura C.20);
  - se l'offuscameto è avvenuto correttemente, il file è minore della soglia imposta dalla variabile `max_dim_file_obf` ed il file non è vuoto allora si rimuove il file contenente la particella `tmp_` all'interno del suo nome e se il file che si sta analizzando, `_file_to_obf`, è diverso dal nome del file contenuto nella variabile `file_with_main` allora si richiama la funzione `Modify_if_not_main` descritta in Appendice C.2.5 e si aggiunge alla lista `list_file_mod_main` la stringa definita come `"main_"+name_file+"()"` (cfr. Figura C.22)

Una volta terminato di analizzare tutti i file presenti nella variabile `list_file_used` si richiama la funzione `Modify_if_main` (cfr. Appendice C.2.4) e si elimina il file *a.out*, file generato dall'esecuzione dello strumento *Tigress*.

C.3.9 Compilazione file offuscati

Dopo aver offuscato il programma che si sta analizzando con lo strumento *Tigress*, si passa alla fase di compilazione del programma offuscato e si crea una variabile `dest` contente il percorso nel quale si vuole salvare il file binario che verrà generato. Successivamente si applica un controllo sul contenuto della variabile `res_compiler_clean` contenente il valore di ritorno della funzione `Create_compiled` utilizzata per creare il file binario del programma senza alcun offuscamento riportato:

- se tale valore risulta essere diverso da 0 significa che il file sorgente in chiaro non è stato compilato correttamente per cui non verrà compilato il file sorgente offuscato, si riportano i dettagli nei corrispondenti file di log, si richiama la funzione `Delete_output_created()` (cfr. Appendice C.2.2) e si passa al successivo step del ciclo *for* `for key in obf_tigress.keys()`, descritto in Appendice C.3.8;
- se tale valore risulta essere pari a 0 tramite l'istruzione

```
if(Exec_command("tigress "+obf_tigress[key]+" "+cppflag+" "+cflag+" "+lflag+" "+ldflag+" --out=
    "+after_obf+" "+location+"tmp_"+name_file,file_err)!=0):
```

Figura C.18: Tigress - Exec_command.

```
print "\n\033[31;1m----------(ERROR during obfuscation)----------\nThe file : "+
    file_to_obf+"' not protected\033[0m\n"
LOGFILE.write("\nError during obf : "+file_to_obf+"\n")
LOGFILE.write("See file : "+file_err+"' in "+LOG+"\n\n")
os.system("rm "+after_obf)
os.system("mv "+location+"tmp_"+name_file+" "+file_to_obf)
f.write(name_file+"\n")
```

Figura C.19: Tigress - file non offuscato.

```
if(Create_compiled(name_prog,tmp_tigress,dest,make_dir_rel,nameMakefile,make_cmd,flag,"
    error_compiled_"+name_prog+"_"+make_cmd+"_"+key,"Compiled/obf/"+key)==0):
```

si richiama la funzione `Create_compiled` (cfr. Appendice C.2.8) ed in base al valore di ritorno:

- se pari a 0 si effettua un controllo sulla correttezza della compilazione eseguita, si creano i report sul programma sorgente offuscato (cfr. Appendice C.3.10 e i report sul file binario offuscato (cfr. Appendice C.3.11);
- se diverso da 0 si riportano i dettagli nei corrispondenti file di log, si richiama la funzione `Delete_output_created()` (cfr. Appendice C.2.2) e si passa al successivo step del ciclo *for* `for key in obf_tigress.keys()`, descritto in Appendice 5.2.1;

Il controllo sulla correttezza della compilazione (Figura C.23) eseguita viene eseguito tramite l'utilizzo del dizionario `clean_makecmd_ret` (descritto in Appendice C.3.5). Viene prima configurato un flag tramite la variabile `ok` e successivamente per mezzo della *chiave* del dizionario, che contiene il comando da eseguire, si esegue il comando riportato e si controlla che il valore di ritorno del file binario offuscato sia uguale al valore di ritorno del file binario non offuscato (*valore* del dizionario). Nel caso in cui i valori di ritorno siano differenti si riportano i dettagli nei corrispondenti file di log ed il flag `ok` viene settato a *false*. Il flag configurato con il valore *false* permette di saltare le fasi di creazione dei report per mezzo degli strumenti *Frama-c* e *Diablo*, per cui verrà in seguito chiamata la funzione `Delete_output_created()` (cfr. Appendice C.2.2) e si passerà al successivo step del ciclo *for* `for key in obf_tigress.keys()`, descritto in Sezione 5.2.1;

C.3.10 Report frama-c (offuscamento tigress)

Se la variabile `ok` è configurata con il valore *ok*, vengono preparate le variabili da passare come parametri alla funzione `Create_report_frama_c`, si veda appendice C.2.7, per ottenere i valore delle metriche calcolate sul programma sorgente offuscato dallo strumento *Tigress*. Inoltre tramite il comando:

```
for _file in list_file_used.split(","):
```

i file *.c* contenuti nella variabile `list_file_used` vengono selezionati uno per volta. Per ogni file selezionato, all'interno del ciclo *for*, viene appunto chiamata la funzione `Create_report_frama_c` con i relativi parametri appositamente configurati.

La Figura C.24 mostra il codice utilizzato per la creazione dei report dello strumento *Frama-c*. La variabile chiamata `dest` viene utilizzata per specificare il percorso di base utilizzato per memorizzare il report creato (in questo percorso viene anche salvato il file presente al percorso contenuto dalla variabile `file_notobf`) e la variabile `path` indica il percorso di base in cui si trovano i file che si vogliono sottoporre all'analisi dello strumento *Frama-c*. Prima di passare i file con estensione *.c* allo strumento *Frama-c* bisogna pre-compilarli, operazione effettuata tramite il comando `os.system("cc -E "+cppflag+" -o "+path.rstrip()+".i "+path.rstrip())`.

```
if int(os.path.getsize(after_obf)) > int(max_dim_file_obf):
    #case too bigger
    f.write(name_file+"\n")
    os.system("rm "+after_obf)
    os.system("mv "+location+"tmp_"+name_file+" "+_file_to_obf)
    print "\n\033[31;1m----------(File: '"+_file_to_obf+"' too_bigger)
        ----------\nThis_file_will_not_protected\033[0m\n"
    LOGFILE.write("\nFile: '"+_file_to_obf+"' too_bigger_and_will_not_protected\n\n")
```

Figura C.20: Tigress - file offuscato maggiore della soglia.

```
elif int(os.path.getsize(after_obf))==0:
    #case file=0
    f.write(name_file+"\n")
    os.system("rm "+after_obf)
    os.system("mv "+location+"tmp_"+name_file+" "+_file_to_obf)
    print "\n\033[31;1m----------(File: '"+_file_to_obf+"' empty)----------\
        nThis_file_will_not_protected\033[0m\n"
    LOGFILE.write("\nFile: '"+_file_to_obf+"' empty_and_will_not_protected\n\n")
```

Figura C.21: Tigress - file offuscato ma vuoto.

C.3.11 Report diablo (offuscamento tigress)

Se la variabile `ok` è configurata con il valore *ok*, vengono preparate le variabili da passare come parametri alla funzione `Run_Diablo`, si veda Appendice C.2.1, per ottenere il valore delle metriche calcolate sul programma binario offuscato dallo strumento *Tigress*. Viene creata una cartella temporanea chiamata `dir_tmp` in cui copiare momentaneamente la cartella contenente il programma che si sta analizzando, la variabile `_output` contiene il percorso al programma che si sta analizzando (nella cartella temporanea appena creata) e la variabile `report_output` contiene il percorso in cui si vuole memorizzare il report generato dallo strumento *Diablo*.

La Figura C.25 mostra il codice utilizzato per la creazione dei report dello strumento *Diablo*. Poiché si vuole richiamare lo strumento *Diablo* senza applicare nessuna protezione, si passano manualmente come stringhe i parametri `protection[none]` e `protection(none)` alla funzione `Run_Diablo`. Alla fine la cartella temporanea `dir_tmp` viene rimossa.

Infine, prima di proseguire al successivo step del ciclo *for* `for key in obf_tigress.keys()`, descritto in Sezione 5.2.1, viene richiamata la funzione `Delete_output_created` (cfr. Appendice C.2.2).

C.3.12 Cleaning

Una volta terminata l'analisi di tutti i programmi, si procede alla cancellazione dei dati non ritenuti necessari. In base ai parametri settati nel file di configurazione dello script *Metrics_analysis.py* (cfr. Appendice B.1.4), si decide quali cartelle del file-system di output generato (cfr. Sezione 4.4.2) devono essere rimosse o meno (cfr. Figura C.26).

C.3.13 Creazione file cvs

Prima di terminare l'esecuzione dello script *Metrics_analysis.py* vengono lanciati ulteriori due script per raccogliere i dati raccolti, in modo ordinato, all'interno di file con estensione *.csv*, come mostrato in Figura C.27.

create_frama_csv.py

Lo scopo di questo script è creare un file con estensione *.csv*, per ogni programma analizzato, contenente tutte le metriche estratte dallo strumento *Frama-c*. Il file *.csv* verrà chiamato con

```
        else:
            #case correct
            #print "tigress "+obf_tigress[key]+" "+cppflag+" "+cflag+" "+lflag+" "+ldflag+" --
                out="+after_obf+" "+location+"tmp_"+name_file,file_err
            os.system("rm "+location+"tmp_"+name_file)
            if (file_with_main!=_file_to_obf):
                #modify main add by tigress
                Modify_if_not_main(after_obf,name_file,location,make_cmd)
                list_file_mod_main.append("main_"+name_file+"()")
```

Figura C.22: Tigress - file offuscato.

```
ok="true"
for test in clean_makecmd_ret:
    if(clean_makecmd_ret[test]!=Exec_command(dest+"/"+test,"Error_run_command_["+test.replace("/",
        "-").replace(" ","-").replace(".","-")+"]_"+name_prog+"_"+make_cmd+"_"+key)):
        LOGFILE.write("Prog_'"+name_prog+"',protected_with_'"+key+"'_has_return_value_different_
            from_source_->_no_report_Frama-c_and_Diablo\n")
        LOGFILE.write("This_program_will_be_deleted_from_compiled/obf_\n\n_")
        print "\033[31;1mProg_'"+name_prog+"',protected_with_'"+key+"'_has_return_value_different_
            from_source_->_no_report_Frama-c_and_Diablo\n\033[0m\n\n"
        os.system("rm_-f_-r_"+dest)
        ok="false"
        break
    else:
        os.system("rm_-f_-r_"+LOG+"Error_run_command_["+test.replace("/","-").replace(" ","-").
            replace(".","-")+"]_"+name_prog+"_"+make_cmd+"_"+key)
```

Figura C.23: Controllo sulla correttezza della compilazione.

la seguente nomenclatura *[programma analizzato]_frama.csv* e conterrà i parametri descritti in Appendice B.2.3.

Tale script accetta due parametri in ingresso:

1. il percorso alla cartella in cui sono stati raccolti i dati estratti dallo strumento *Frama-c*, ovvero la cartella *report_frama* (cfr. Sezione 4.4.2);
2. il percorso alla cartella in cui memorizzare i file *.csv* che verranno generati, ovvero la cartella *pre_analized* (cfr. Sezione 4.4.2).

L'esecuzione dello script risulta essere la seguente:

1. analizzare i report dei programmi non offuscati. Per ogni programma presente nella cartella *pre_analized* viene chiamata la funzione `Create_cvs_frama` che ha il compito di parsare il file di report generato dallo strumento *Frama-c*, rispettivo del programma preso in considerazione, e scrivere i dati raccolti all'interno del file *.csv* che si sta creando;
2. analizzare i report dei programmi offuscati, per ogni programma presente nella cartella *report_frama_c/obf*, eseguendo le stesse procedure appena descritte, per i report dei file offuscati generati sia dallo strumento *Tigress* sia dallo strumento *Diablo*.

In questo script viene anche utilizzato il file */filenotobf/file.txt* citato in Appendice C.3.10 per inserire il flag chiamato *flag_obf_tigress(0=made_1=not)* (cfr. Appendice B.2.3) con i seguenti valori: 0 se l'offuscamento è avvenuto o 1 in caso contrario.

create_diablo_csv.py

Lo scopo di questo script è creare un file con estensione *.csv*, per ogni programma analizzato, contenente tutte le metriche estratte dallo strumento *Diablo*. Il file *.csv* verrà chiamato con

```
print "\n\033[32;1m-----(Create_report_frama-c_file_obf_:_"+key+"'_for_command_:_"+
    make_cmd+"_)-----\033[0m\n"
dest=report_frama_obf_tigress+"/"+key+"/"+name_prog+"/"+name_prog+"_"+make_cmd
os.system("mkdir_"+dest)
os.system("mv_"+file_notobf+"_"+dest)
for _file in list_file_used.split(","):
    _file=re.sub("^(_)*","",_file) #delete space* if exist
    name_file=re.sub("^.*/","",_file).rstrip()
    path=file_obf_tigress+"/"+key+"/"+name_prog+"/"+name_prog+"_"+make_cmd+"/"+_file
    #pre-compiled verion with cflag
    os.system("cc_-E_"+cppflag+"_-o_"+path.rstrip()+".i_"+path.rstrip())
    Create_report_frama_c(name_file,path.rstrip()+".i",dest+"/"+name_file+".txt",name_prog+"_"+
        make_cmd+"_"+name_file+"_"+key+".txt")
    os.system("rm_"+path.rstrip()+".i_")
```

Figura C.24: Report frama-c (offuscamento tigress).

```
os.system("mkdir_-p_"+dir_tmp)
_output=dir_tmp+"/"+name_prog+"_"+make_cmd
report_output=report_diablo_obf_tigress+"/"+key+"/"+name_prog+"/"+name_prog+"_"+make_cmd
os.system("mkdir_"+report_output)
os.system("cp_-r_"+report_frama_obf_tigress+"/"+key+"/"+name_prog+"/"+name_prog+"_"+
    make_cmd+"/filenotobf"+"_"+report_output)
os.system("cp_-r_"+file_obf_tigress+"/"+key+"/"+name_prog+"/"+name_prog+"_"+make_cmd+"_"+
    _output)
print "\n\033[32;1m-----(Run_Diablo_with_no_protection_on_program_obf_with_tigress_:_"+key+"_
    to_extract_metrics_)-----\033[0m"
Run_Diablo("protection[none]","protection(none)",_output,report_output,key)
os.system("rm_-f_-r_"+dir_tmp)
```

Figura C.25: Report diablo (offuscamento tigress).

la seguente nomenclatura *[programma analizzato]_diablo.csv* e conterrà i parametri descritti in Appendice B.2.3. L'intero script viene riportato in Appendice B.

Tale script accetta due parametri in ingresso:

1. il percorso alla cartella in cui sono stati raccolti i dati estratti dallo strumento *FDiablo*, ovvero la cartella *report_diablo* (cfr. Sezione 4.4.2);
2. il percorso alla cartella in cui memorizzare i file *.csv* che verranno generati, ovvero la cartella *pre_analized* (cfr. Sezione 4.4.2).

L'esecuzione dello script risulta essere la seguente:

1. analizzare i report dei programmi non offuscati. Per ogni programma presente nella cartella *pre_analized* viene chiamata la funzione `Create_cvs_frama` che ha il compito di parsare il file di report generato dallo strumento *Frama-c*, rispettivo del programma preso in considerazione, e scrivere i dati raccolti all'interno del file *.csv* che si sta creando;
2. analizzare i report dei programmi offuscati, per ogni programma presente nella cartella *report_diablo/obf*, eseguendo le stesse procedure appena descritte, per i report dei file offuscati generati sia dallo strumento *Tigress* sia dallo strumento *Diablo*.

In questo script viene anche utilizzato il file */filenotobf/file.txt* citato in Appendice C.3.10 per inserire il flag chiamato *flag_obf_tigress(0=made_1=not)* (cfr. Appendice B.2.3) con i seguenti valori: 0 se l'offuscamento è avvenuto o 1 in caso contrario.

```
if keep_compiled=="false":
    os.system("rm␣-r␣-f␣"+compiled)
if keep_obf=="false":
    os.system("rm␣-r␣-f␣"+file_obf)
if keep_diablo=="false":
    os.system("rm␣-r␣-f␣"+report_diablo)
if keep_report_frama=="false":
    os.system("rm␣-f␣"+report_frama_c)
```

Figura C.26: Cleaning.

```
os.system(here+"sub/create_frama_csv.py␣"+report_frama+"␣"+pre_analized)
os.system(here+"sub/create_diablo_csv.py␣"+report_diablo+"␣"+pre_analized)
```

Figura C.27: Esecuzione script per raccolta dati.

Appendice D

Grafici fingerprint

Vengono di seguito riportati tutti i grafici più rappresentativi generati per lo studio effettuato sulle *fingerprint*.

D.1 Frama-c

Vengono di seguito riportati tutti i grafici generati sulle metriche estratte dallo strumento *Frama-c* per l'identificazione delle *fingerprint*.

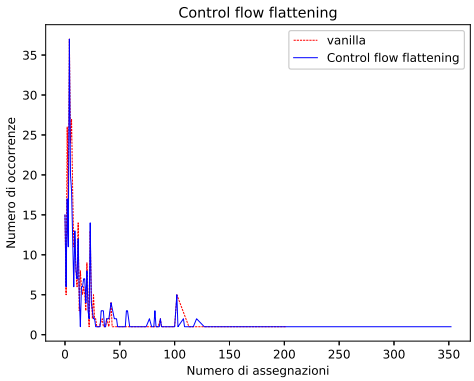


Figura D.1: Grafico del numero di occorrenze per il numero di assegnazioni calcolato sulla protezione *control flow flattening*.

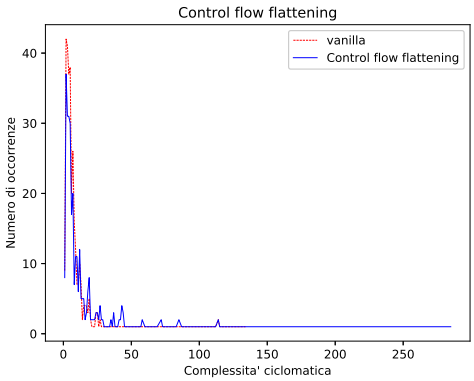


Figura D.2: Grafico del numero di occorrenze per la complessità ciclomatica calcolata sulla protezione *control flow flattening*.

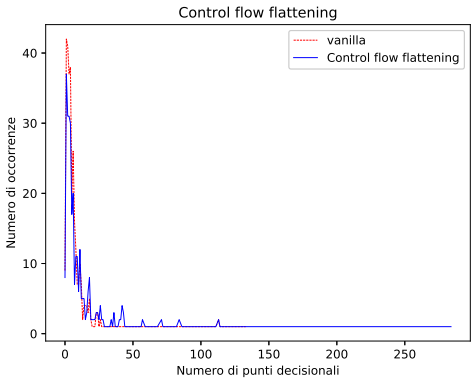


Figura D.3: Grafico del numero di occorrenze per il numero di punti decisionali calcolato sulla protezione *control flow flattening*.

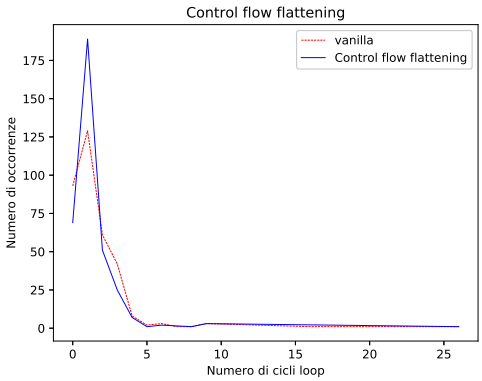


Figura D.4: Grafico del numero di occorrenze per il numero di loop calcolato sulla protezione *control flow flattening*.

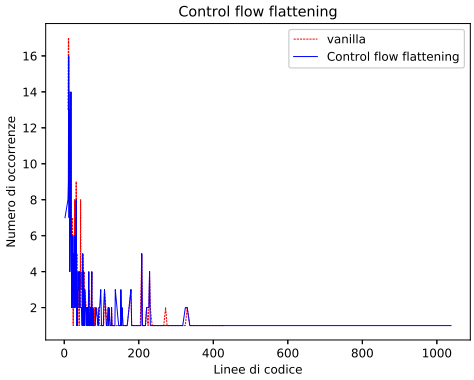


Figura D.5: Grafico del numero di occorrenze per il numero di linee di codice calcolato sulla protezione *control flow flattening*.

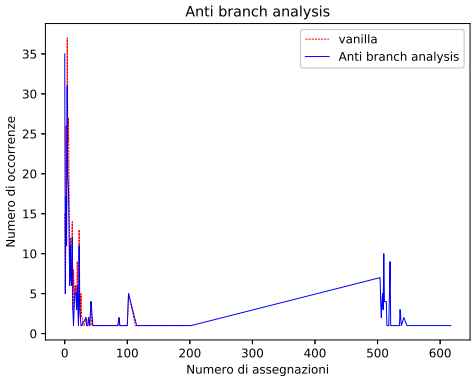


Figura D.6: Grafico del numero di occorrenze per il numero di assegnazioni calcolato sulla protezione *anti branch analysis*.

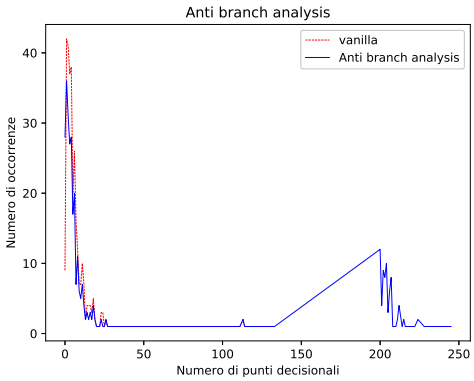


Figura D.7: Grafico del numero di occorrenze per il numero di punti decisionali calcolato sulla protezione *anti branch analysis*.

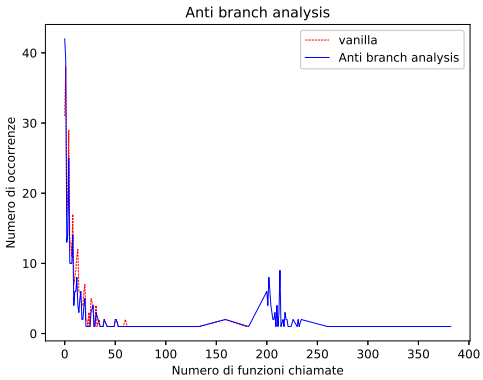


Figura D.8: Grafico del numero di occorrenze per il numero di funzioni chiamate calcolato sulla protezione *anti branch analysis*.

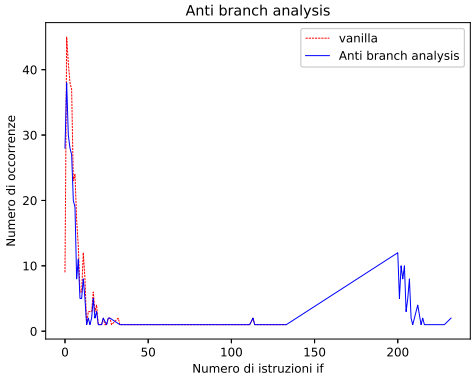


Figura D.9: Grafico del numero di occorrenze per il numero di istruzioni if calcolato sulla protezione *anti branch analysis*.

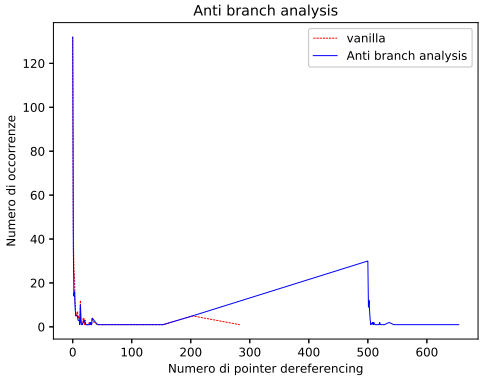


Figura D.10: Grafico del numero di occorrenze per il numero di pointer dereferencing calcolato sulla protezione *anti branch analysis*.

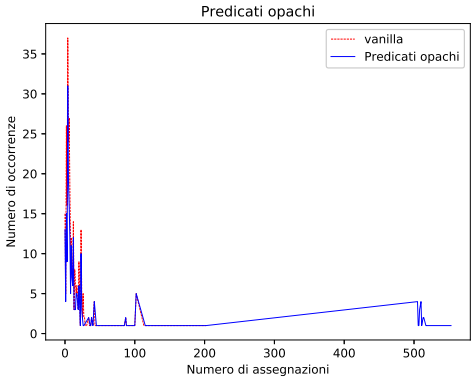


Figura D.11: Grafico del numero di occorrenze per il numero di assegnazioni calcolato sulla protezione *predicati opachi*.

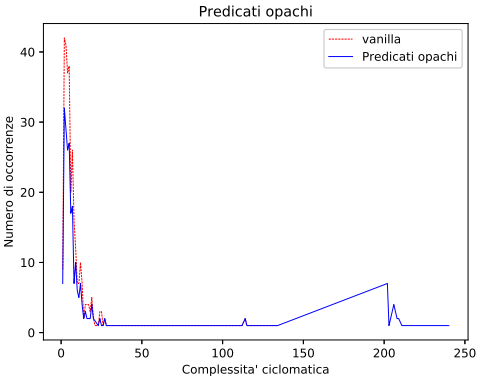


Figura D.12: Grafico del numero di occorrenze per la complessità ciclomatica calcolata sulla protezione *predicati opachi*.

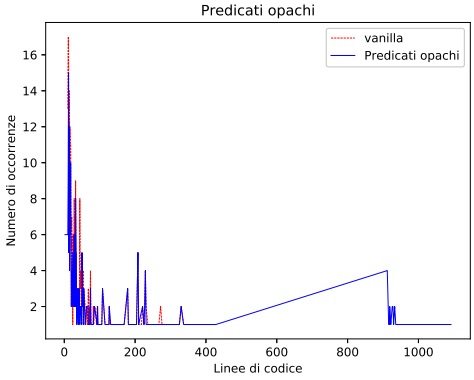


Figura D.13: Grafico del numero di occorrenze per il numero di linee di codice calcolato sulla protezione *predicati opachi*.

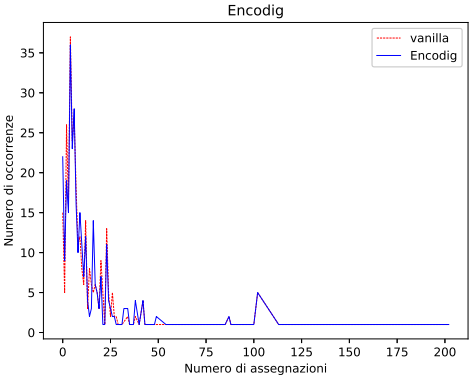


Figura D.14: Grafico del numero di occorrenze per il numero di assegnazioni calcolato sulla protezione *Encoding*.

D.2 Diablo

Vengono di seguito riportati tutti i grafici generati sulle metriche estratte dallo strumento *Diablo* per l'identificazione delle *fingerprint*, per le protezioni applicate al 50%.

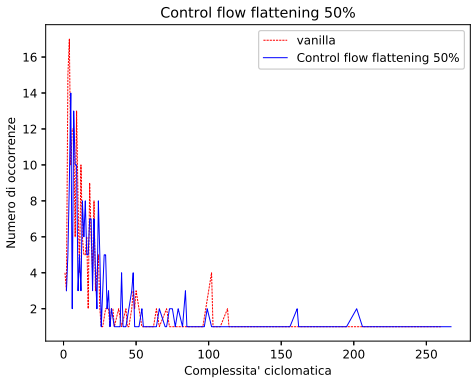


Figura D.15: Grafico del numero di occorrenze per la complessità ciclomatica calcolata sulla protezione *control flow flattening 50%*.

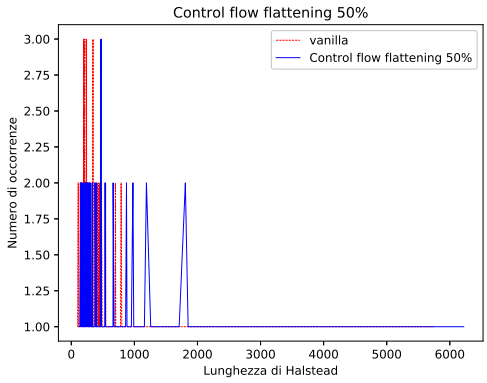


Figura D.16: Grafico del numero di occorrenze per il Lunghezza di Halstead calcolato sulla protezione *control flow flattening 50%*.

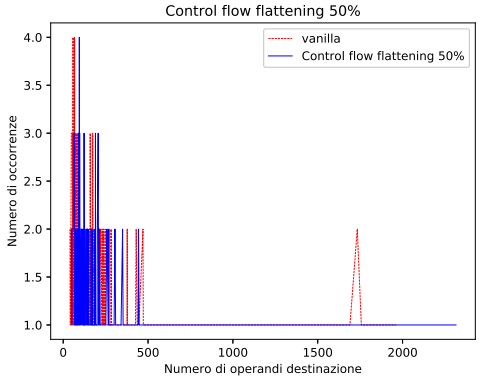


Figura D.17: Grafico del numero di occorrenze per il numero di operandi destinazione calcolato sulla protezione *control flow flattening 50%*.

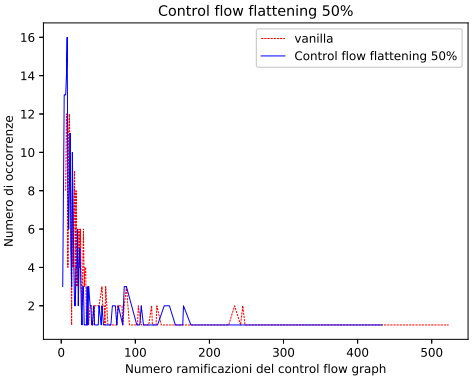


Figura D.18: Grafico del numero di occorrenze per il numero ramificazioni del control flow graph calcolato sulla protezione *control flow flattening 50%*.

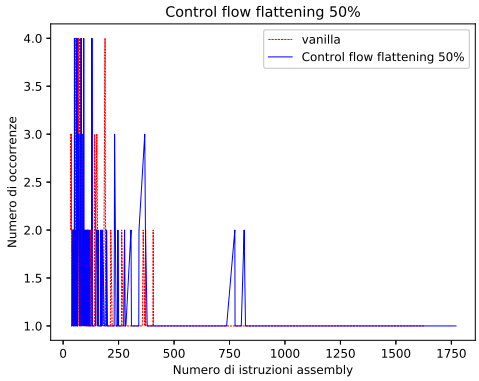


Figura D.19: Grafico del numero di occorrenze per il numero di istruzioni assembly calcolato sulla protezione *control flow flattening 50%*.

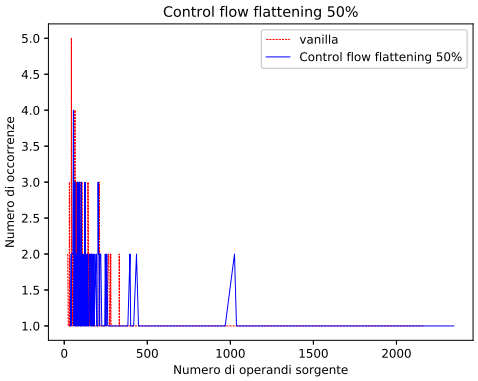


Figura D.20: Grafico del numero di occorrenze per il numero di operandi sorgente calcolato sulla protezione *control flow flattening 50%*.

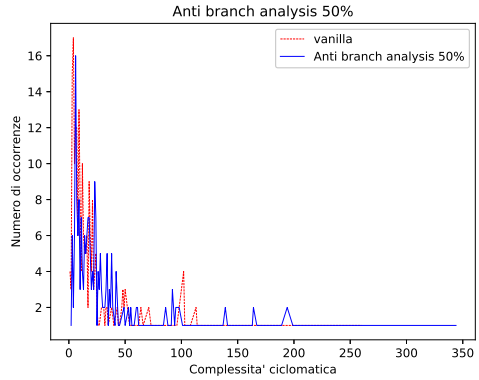


Figura D.21: Grafico del numero di occorrenze per la complessità ciclomatica calcolata sulla protezione *anti branch analysis 50%*.

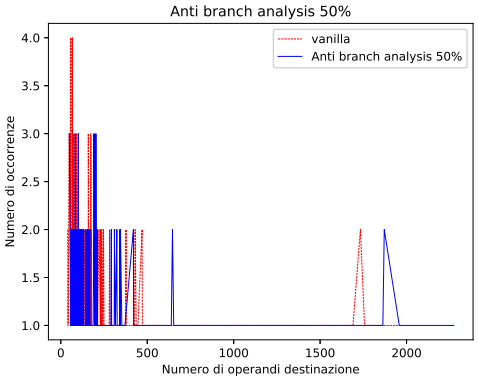


Figura D.22: Grafico del numero di occorrenze per il numero di operandi destinazione calcolato sulla protezione *anti branch analysis 50%*.

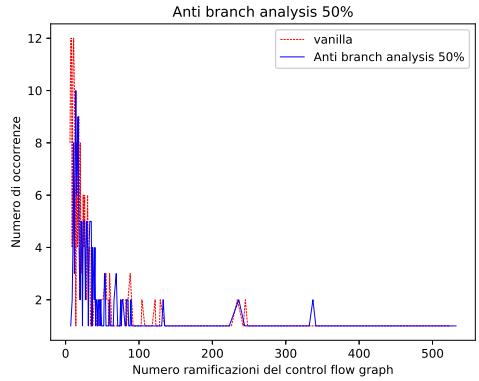


Figura D.23: Grafico del numero di occorrenze per il numero ramificazioni del control flow graph calcolato sulla protezione *anti branch analysis 50%*.

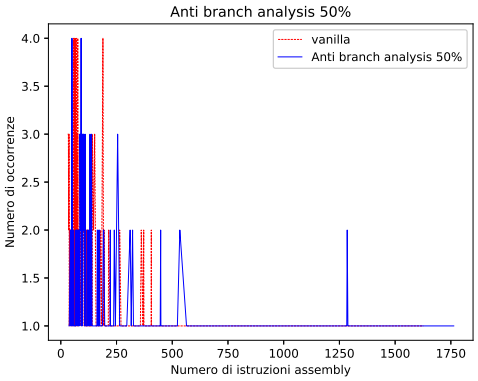


Figura D.24: Grafico del numero di occorrenze per il numero di istruzioni assembly calcolato sulla protezione *anti branch analysis 50%*.

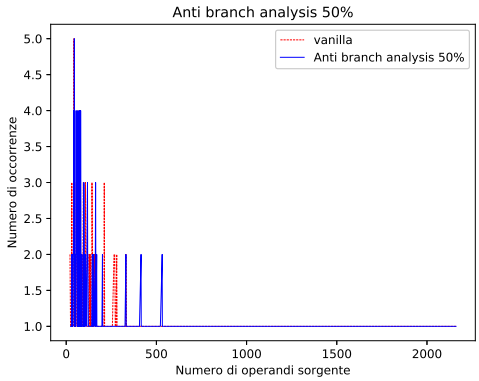


Figura D.25: Grafico del numero di occorrenze per il numero di operandi sorgente calcolato sulla protezione *anti branch analysis 50%*.

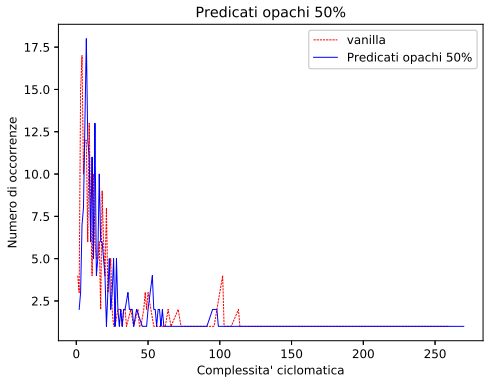


Figura D.26: Grafico del numero di occorrenze per la complessità ciclomatica calcolata sulla protezione *predicati opachi 50%*.

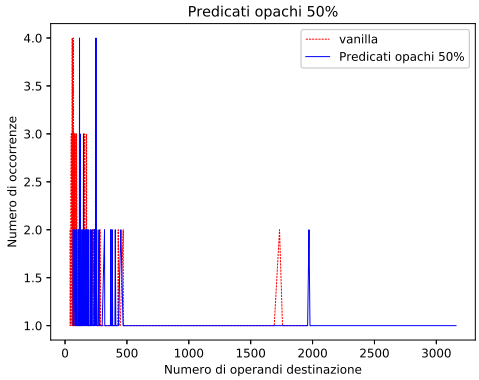


Figura D.27: Grafico del numero di occorrenze per il numero di operandi destinazione calcolato sulla protezione *predicati opachi 50%*.

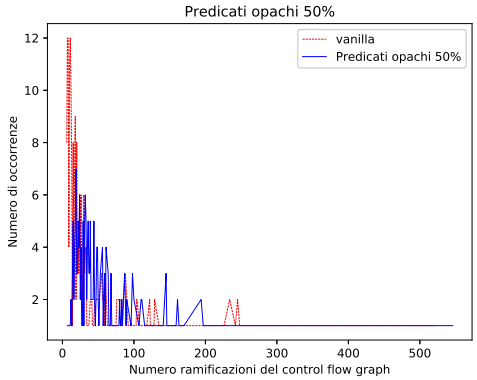


Figura D.28: Grafico del numero di occorrenze per il numero ramificazioni del control flow graph calcolato sulla protezione *predicati opachi 50%*.

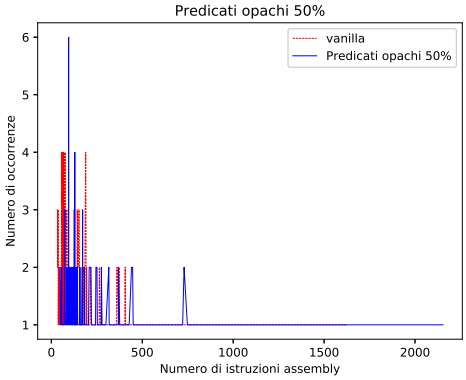


Figura D.29: Grafico del numero di occorrenze per il numero di istruzioni assembly calcolato sulla protezione *predicati opachi 50%*.

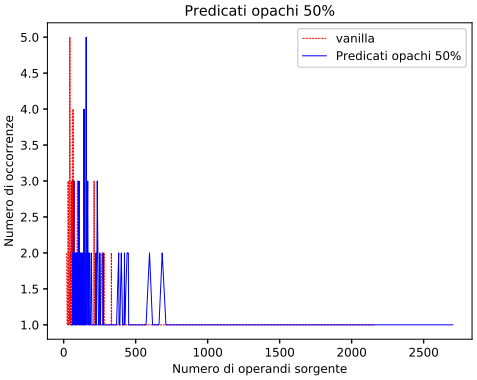


Figura D.30: Grafico del numero di occorrenze per il numero di operandi sorgente calcolato sulla protezione *predicati opachi 50%*.

Appendice E

Grafici potency

In questa Appendice verranno riportati tutti i grafici più significativi generati ed analizzati per lo studi effettuati sulle *potency*.

E.1 Grafici potency di riferimento calcolate su metriche estratte dallo strumento *Frama-c*

Di seguito i grafici generati confrontando le metriche estratte su file sorgenti.

E.1.1 Complessità ciclomatica e lunghezza di Halstead calcolate su file binari

I grafici mostrati mostrano gli andamenti delle curve delle *potency* calcolate sulle metriche estratte dai file sorgenti messe a confronto con gli andamenti delle curve che rappresentano la *potency* calcolata su complessità ciclomatica e lunghezza di Halstead estratte da file binari.

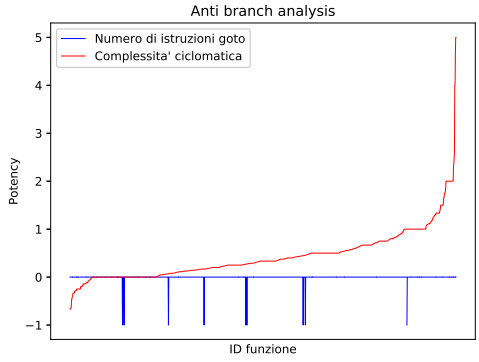


Figura E.1: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di istruzioni goto per la protezione *anti branch analysis*.

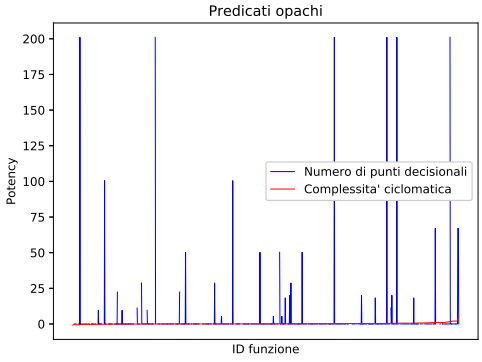


Figura E.2: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di punti decisionali per la protezione *predicati opachi*.

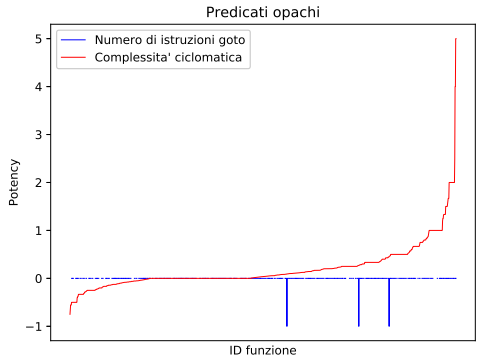


Figura E.3: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di istruzioni goto per la protezione *predicati opachi*.

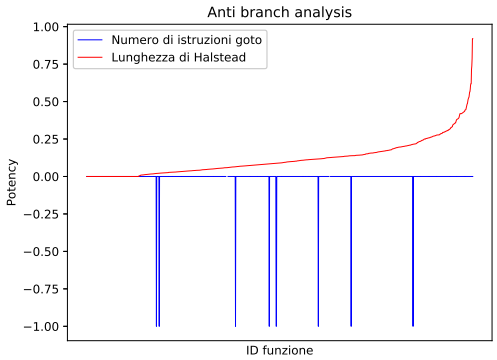


Figura E.4: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di istruzioni goto per la protezione *anti branch analysis*.

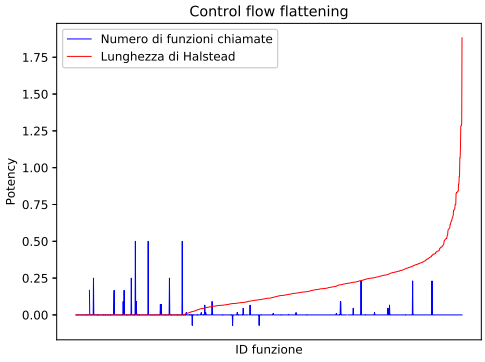


Figura E.5: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di funzioni chiamate per la protezione *control flow flattening*.

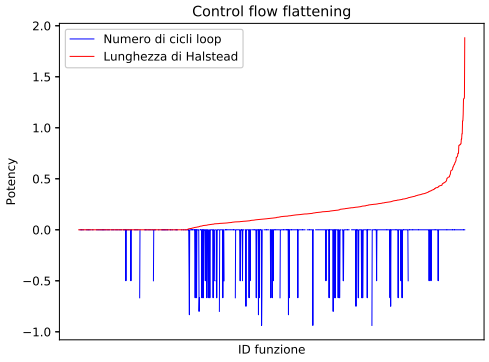


Figura E.6: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di loop per la protezione *control flow flattening*.

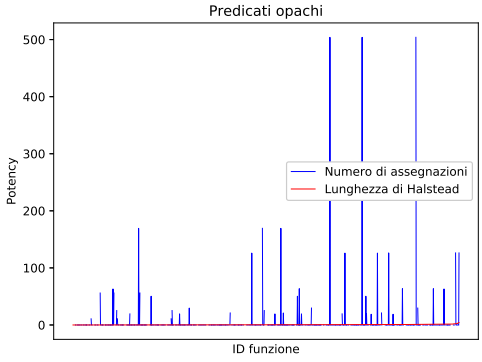


Figura E.7: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di assegnazioni per la protezione *predicati opachi*.

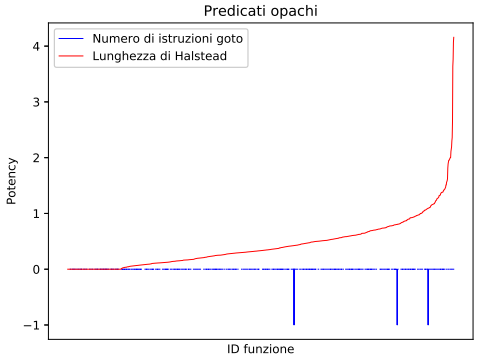


Figura E.8: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di istruzioni goto per la protezione *predicati opachi*.

E.1.2 Complessità ciclomatica calcolata su file sorgenti

I grafici mostrati mostrano gli andamenti delle curve delle *potency* calcolate sulle metriche estratte dai file sorgenti messe a confronto con l'andamento della curva che rappresenta la *potency* calcolata su complessità ciclomatica estratta su file sorgenti.

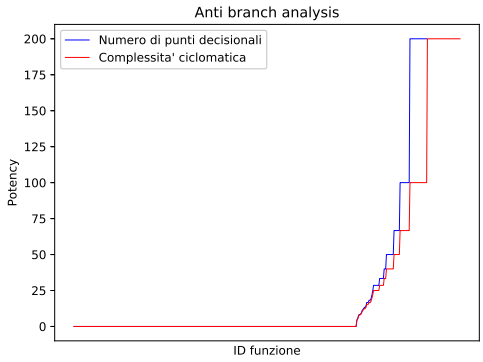


Figura E.9: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di punti decisionali per la protezione *anti branch analysis*.

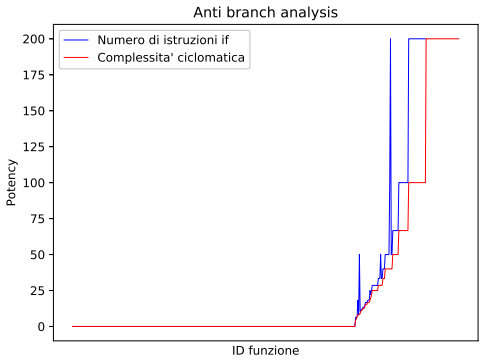


Figura E.10: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di istruzioni if per la protezione *anti branch analysis*.

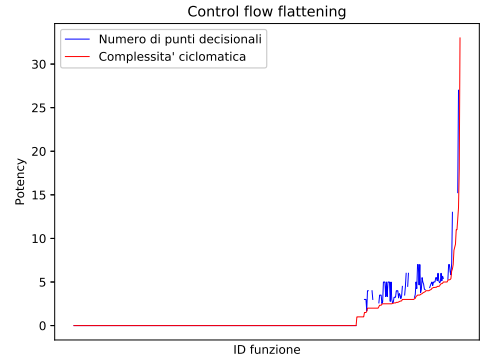


Figura E.11: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di punti decisionali per la protezione *control flow flattening*.

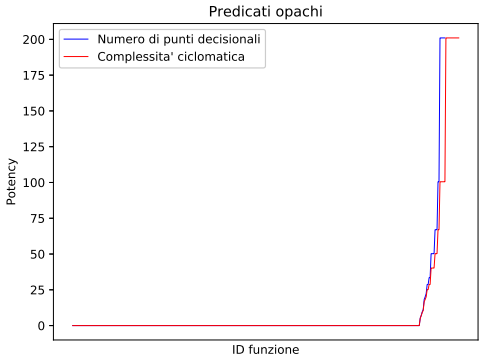


Figura E.12: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di punti decisionali per la protezione *predicati opachi*.

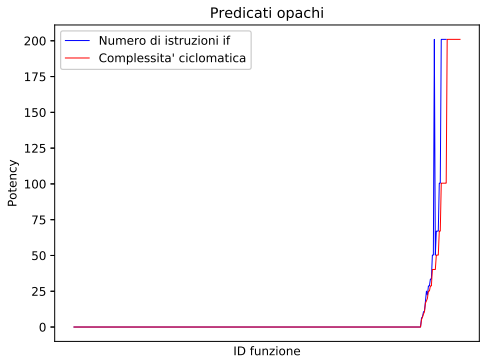


Figura E.13: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di istruzioni if per la protezione *predicati opachi*.

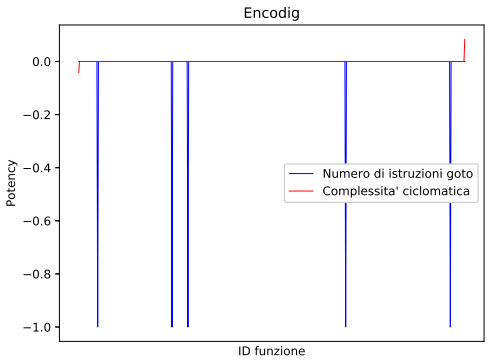


Figura E.14: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di istruzioni goto per la protezione *Encoding*.

E.2 Grafici potency di riferimento calcolate su metriche estratte dallo strumento *Diablo*

I grafici mostrati mostrano gli andamenti delle curve delle *potency* calcolate sulle metriche estratte dai file binari messe a confronto con gli andamenti delle curve che rappresentano la *potency* calcolata su complessità ciclomatica e Halstead anch'esse estratte da file binari.

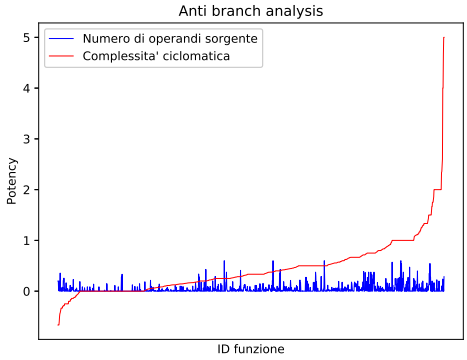


Figura E.15: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di operandi sorgente per la protezione *anti branch analysis*.

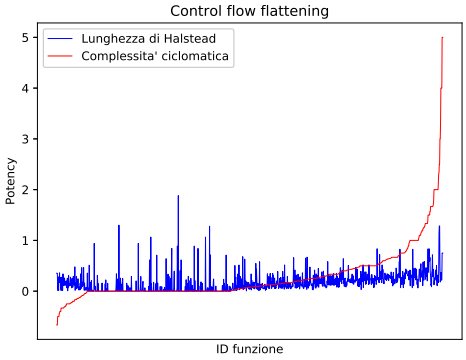


Figura E.16: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il Lunghezza di Halstead per la protezione *control flow flattening*.

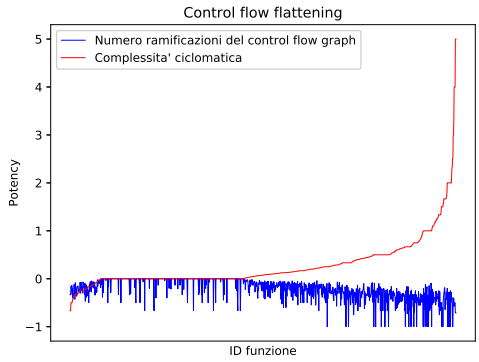


Figura E.17: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero ramificazioni del control flow graph per la protezione *control flow flattening*.

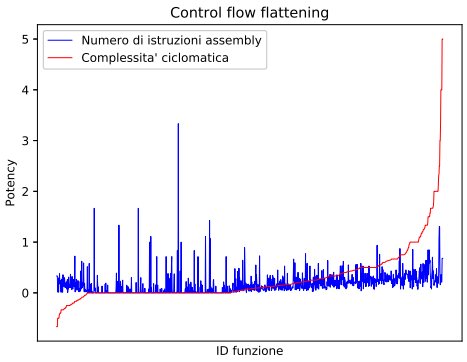


Figura E.18: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di istruzioni assembly per la protezione *control flow flattening*.

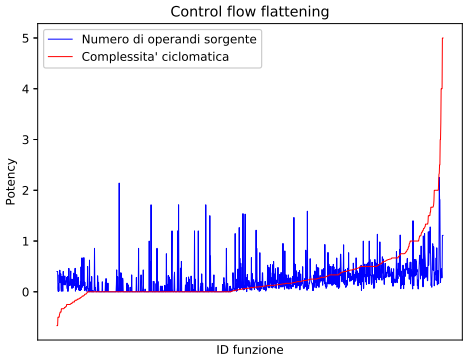


Figura E.19: Confronto tra la *potency* calcolata sulla complessità ciclomatica ed il numero di operandi sorgente per la protezione *control flow flattening*.

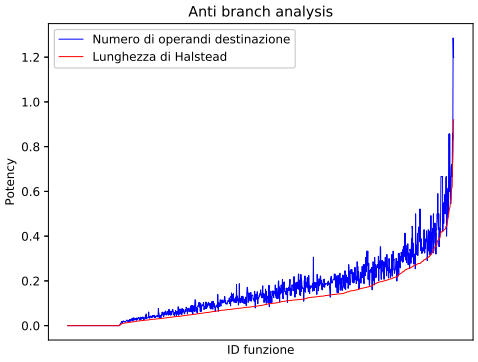


Figura E.20: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di operandi destinazione per la protezione *anti branch analysis*.

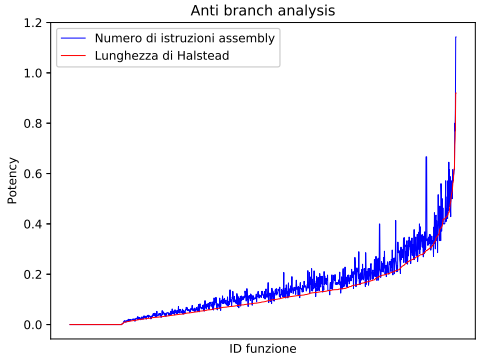


Figura E.21: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di istruzioni assembly per la protezione *anti branch analysis*.

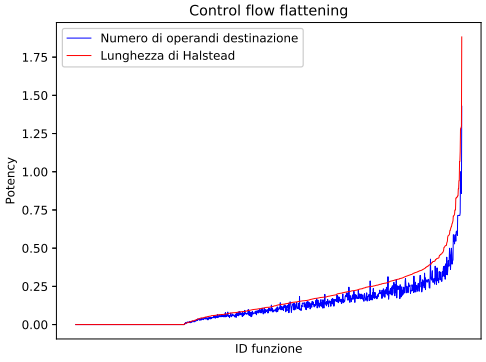


Figura E.22: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di operandi destinazione per la protezione *control flow flattening*.

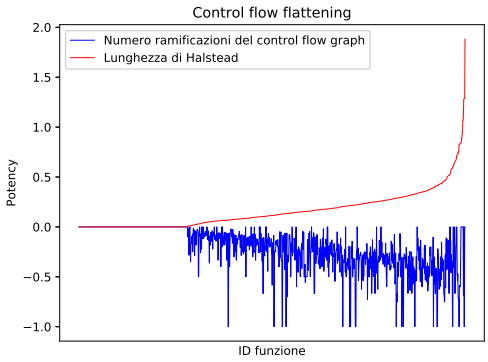


Figura E.23: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero ramificazioni del control flow graph per la protezione *control flow flattening*.

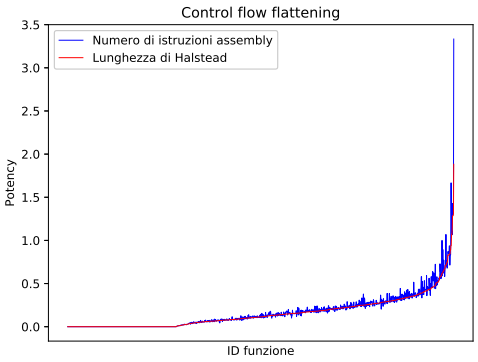


Figura E.24: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di istruzioni assembly per la protezione *control flow flattening*.

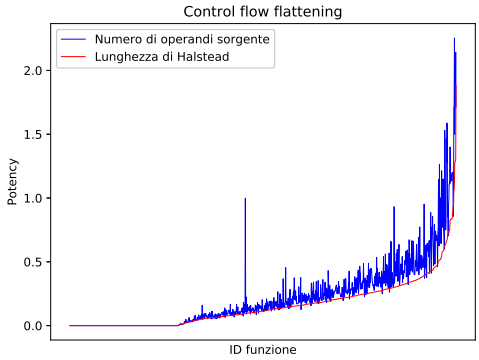


Figura E.25: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di operandi sorgente per la protezione *control flow flattening*.

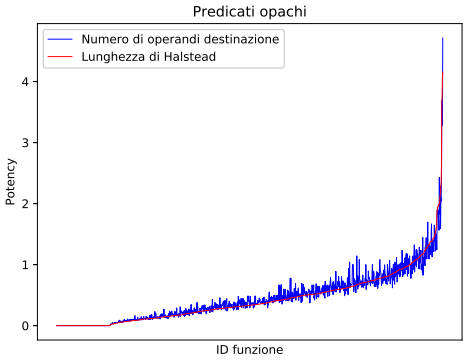


Figura E.26: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di operandi destinazione per la protezione *predicati opachi*.

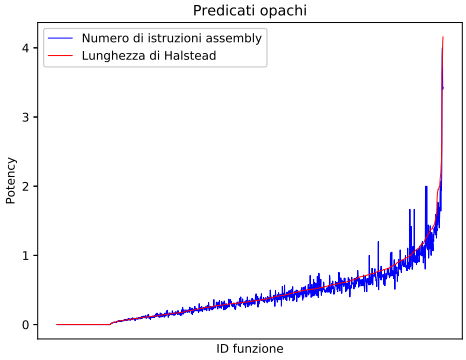


Figura E.27: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di istruzioni assembly per la protezione *predicati opachi*.

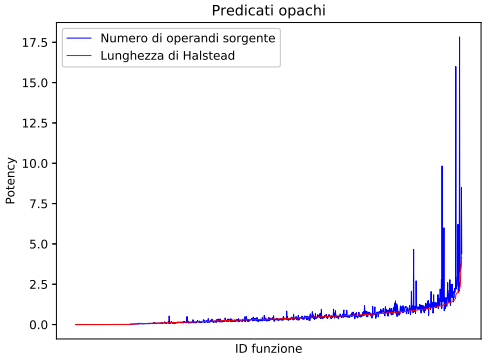


Figura E.28: Confronto tra la *potency* calcolata sulla lunghezza di Halstead ed il numero di operandi sorgente per la protezione *predicati opachi*.

Bibliografia

[1] S. Widup, M. Spitler, D. Hylender, and G. Bassett, "2018 verizon data breach investigations report", 04 2018

[2] A. Antonielli, L. Bechelli, F. Bosco, G. Butti, N. Ciardi, D. D. Vecchio, L. Dinardo, L. Dozio, G. Dragoni, F. Faenzi, G. Faggioli, S. Fumagalli, M. T. Giordano, P. Giudice, C. Giustozzi, A. Granata, M. Onorato, S. Orciari, M. Pacchiardo, P. Pace, A. L. Pennasilico, A. Piazza, A. Piva, D. Raguseo, M. Raimondi, P. L. Rotondo, R. Saccani, L. Sangalli, F. Santi, S. Scozzari, C. Telmon, M. Terranova, G. Tesoriere, E. Tonello, G. Tonello, G. Vaciago, A. Vallavanti, A. Vallega, G. Vercellino, and A. Z. Manzoni, "Rapporto clusit 2018 sulla sicurezza ict in italia", 2018

[3] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski, "Software protection", 03 2011

[4] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?", ACM Comput. Surv., vol. 49, April 2016, DOI 10.1145/2886012

[5] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code", 2015 IEEE Symposium on Security and Privacy, May 2015, pp. 674–691, DOI 10.1109/SP.2015.47

[6] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: reverse engineering obfuscated code", 12th Working Conference on Reverse Engineering (WCRE'05), Nov 2005, pp. 10 pp.–54, DOI 10.1109/WCRE.2005.13

[7] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations", Tech. Rep. 148, Department of Computer Sciences, The University of Auckland, Jul 1997

[8] B. Curtis, S. B. Sheppard, and P. Milliman, "Third time charm: Stronger prediction of programmer performance by software complexity metrics", Proceedings of the 4th International Conference on Software Engineering, Piscataway, NJ, USA, 1979, pp. 356–360

[9] F. E. Allen, "Control flow analysis", Proceedings of a Symposium on Compiler Optimization, New York, NY, USA, 1970, pp. 1–19, DOI 10.1145/800028.808479

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Compilers: Principles, techniques, and tools (2nd edition)", Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811

[11] L. Regano, D. Canavese, C. Basile, and A. Lioy, "Towards optimally hiding protected assets in software applications", 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), July 2017, pp. 374–385, DOI 10.1109/QRS.2017.47

[12] A. Aho, M. Lam, J. Ullman, and R. Sethi, "Compilers: Principles, techniques, and tools", Pearson Education, 2011, ISBN: 9780133002140

[13] R. Diestel, "Graph theory: 5th edition", Springer Graduate Texts in Mathematics, Springer-Verlag, © Reinhard Diestel, 2017, ISBN: 9783961340057

[14] T. J. McCabe, "A complexity measure", IEEE Transactions on Software Engineering, vol. SE-2, Dec 1976, pp. 308–320, DOI 10.1109/TSE.1976.233837

[15] M. H. Halstead, "Elements of software science (operating and programming systems series)", Elsevier Science Inc., 1977, ISBN: 0444002057

[16] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs", Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, 1998, pp. 184–196, DOI 10.1145/268946.268962

[17] S. Blazy and A. Trieu, "Formal Verification of Control-flow Graph Flattening", Certified Proofs and Programs (CPP 2016) (ACM, ed.), Saint-Petersburg, United States, January 2016, p. 12, DOI 10.1145/2854065.2854082

[18] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly", Proceedings of the 10th ACM Conference on Computer and Communications Security, New York, NY, USA, 2003, pp. 290–299, DOI 10.1145/948109.948149

[19] D. Canavese, L. Regano, C. Basile, and A. Viticchié, "Estimating software obfuscation potency with artificial neural networks", Security and Trust Management (G. Livraga and C. Mitchell, eds.), Cham, 2017, pp. 193–202, DOI 10.1007/978-3-319-68063-7_13

[20] L. M. Laird and M. C. Brennan, "Software measurement and estimation: A practical approach", Wiley-IEEE Computer Society Pr, 1st ed., 2006, ISBN: 0471676225, 9780471676225

[21] J. Waller, "Performance benchmarking of application monitoring frameworks", Books on Demand, 2015, ISBN: 9783738689310

[22] B. Ainapure, "Software testing and quality assurance", Technical Publications, 2008, ISBN: 9788184315011

[23] D. Graham, E. Van Veenendaal, and I. Evans, "Foundations of software testing: Istqb certification", Cengage Learning, 2008, ISBN: 9781844809899

[24] R.Rajani, "Software testing", McGraw-Hill Education (India) Pvt Limited, 2004, ISBN: 9780070583528

[25] E. Dustin, T. Garrett, and B. Gauf, "Implementing automated software testing: How to save time and lower costs while raising quality", Pearson Education, 2009, ISBN: 9780321619594

[26] J. Mitchell and R. Black, "Advanced software testing - vol. 3, 2nd edition: Guide to the istqb advanced certification as an advanced technical test analyst", Rocky Nook, 2015, ISBN: 9781457189104

[27] S. Ross, "Introduzione alla statistica", Apogeo education, Apogeo Education, 2014, ISBN: 9788891602671

[28] G. Leti, "Statistica descrittiva", Strumenti (Società editrice il Mulino).: Scienze sociali, Il Mulino, 1983, ISBN: 9788815002785