Progetto di ingegneria informatica:

Software obfuscation techniques evaluation

Luca Molteni: 10664900
Vincenzo Converso: 10625358

Anno accademico: 2021/2022

1. Introduzione

L’obiettivo di questo progetto è valutare l’efficacia di un software di offuscamento del codice in relazione ad un possibile attaccante con accesso ai registri del processore. Inoltre, una volta calcolata l’efficacia, l’obiettivo è trovare, se presente, una correlazione tra essa e una metrica prodotta dal software di offuscamento.

1.1 Nota sull’offuscatore

Il software di offuscamento che andremo ad analizzare, a cui ci andremo a riferire con il nome di “DETON”, prende come input un file assembly RISC-V e opera su di esso allo scopo di renderlo più sicuro da un attacco di un Hardware Trojan con accesso ai registri. L’offuscatore funziona diversamente al variare di due parametri che sono uno relativo all’inserimento di istruzioni garbage ed uno relativo all’offuscamento dei valori immediati presenti nel codice. A seguito dell’offuscamento crea inoltre una metrica che traccia il calore medio dei registri durante l’esecuzione alla quale da ora in avanti ci riferiremo con Heat.

1.2 Approccio adottato

Si è valutata l’analisi statica del software ma si è ritenuto che questo tipo di approccio potesse fornire risultati poco fedeli alla realtà in quanto non si sarebbe tenuto in considerazione il control flow del programma.
Si è deciso quindi di costruire uno strumento che eseguisse in maniera dinamica un programma in linguaggio C e che potesse tracciare nei registri le variabili dichiarate all’interno del codice sorgente. Per il tracciamento delle variabili si è deciso di procedere analizzando le istruzioni macchina eseguite ad ogni ciclo di clock ed in base ad esse inferire la posizione delle variabili nei registri.

1.3 Strumenti utilizzati

Si è deciso di optare, per via della sua versatilità, del linguaggio di programmazione Python versione 3. La scelta dell’IDE è ricaduta di conseguenza su PyCharm.
Il progetto opera con un offuscatore di codice per un’architettura di tipo RISC-V, per poter quindi eseguire i programmi d’interesse è stato utilizzato l’emulatore QEMU. L’ambiente in cui tutto questo viene eseguito è una distribuzione Linux: Ubuntu 20.04.
Per poter estrarre le informazioni rilevanti relative all’esecuzione si è optato per un'operazione di debug. La scelta in questo caso è ricaduta su GDB che è stato automatizzato con uno script Python sfruttando le API già disponibili all’interno del debugger.
Un altro strumento utilizzato è stata la libreria PyElfTools che permette la lettura strutturata dei dati presenti all’interno di un eseguibile, nello specifico il suo utilizzo è stato relativo all’estrazione dei simboli per elencare variabili e rispettive posizioni in memoria.
Per il parsing delle istruzioni macchina in dati strutturati è stato utilizzato un parser esterno già esistente.
Altri strumenti utilizzati sono stati il linguaggio JSON per la memorizzazione dei dati in maniera strutturata e la libreria MatPlotLib di Python per la visualizzazione dei risultati.

2. Architettura software

Il software si compone di tre componenti principali:

- EDG: Generatore di dump di esecuzione
- Tracer: Esegue il tracciamento delle variabili
- Evaluator: Emula un trojan hardware con accesso ai registri.

Di seguito viene riportato un link al repository GitHub contenente il software:
Link al repository

2.1 EDG

Questo componente è stato progettato con l'intento di fornire uno strumento per poter eseguire un programma e poter estrarre informazioni rilevanti riguardanti l'esecuzione. Nello specifico le informazioni d'interesse riguardano le istruzioni macchina eseguite e il contenuto dei registri in ogni momento dell'esecuzione.
Uno scopo secondario assunto dal componente è l'analisi del file eseguibile per poter estrarre i simboli e poter così collegare un indirizzo al nome della variabile nel codice sorgente.

L'implementazione è relativa ad un'architettura di tipo RISC-V ma la struttura del componente è facilmente adattabile ad altre piattaforme.

2.1.1 Funzionamento

L'approccio utilizzato è stato quello di utilizzare l'interfaccia Python di GDB (GNU debugger) per poter automatizzare il processo di debug di un'applicazione ed estrarre i dati attraverso un'esecuzione step-by-step.
L'approccio ha degli svantaggi, primo su tutti la velocità, il fatto che l'esecuzione venga emulata attraverso QEMU e che questo comunichi via socket TCP con GDB che esegue uno script Python rende l'esecuzione di programmi con una complessità temporale troppo elevata impraticabile.
Uno svantaggio secondario consiste nel fatto che è GDB stesso ad eseguire lo script python e non il contrario, cosa che rende più scomoda l'integrazione del componente in un software più grande.
L'output dell'esecuzione verrà quindi salvato su file e letto dal progetto che fa da chiamante per lo script su GDB.
Al fine di rendere più veloce l'esecuzione è stato creato un debugger scritto in C che rimuovesse la necessità di GDB e Python e che comunicasse direttamente con il kernel per eseguire il debug. L'idea è stata abbandonata per la mancanza della flag lato hardware per consentire l'esecuzione step-by-step nell'architettura di riferimento, la quantità di codice da dover implementare per supplire alla mancanza ci ha fatto desistere dal cambiare un componente già completato.
Un terzo approccio è stato l'uso della libreria GDBmi ma questa aveva problemi di velocità ancora più marcati ed è stata abbandonata velocemente.

2.1.2 Composizione del modulo

Il modulo è composto dalle classi rappresentanti le informazioni di un dump e da dei metodi che ne consentono l'esecuzione.
La creazione di un dump comporta l'avvio di QEMU e di GDB, quest'ultimo eseguirà uno script che raccoglierà le informazioni a partire dalla prima istruzione di un metodo indicato fino all'ultima passando per le funzioni chiamate da esso. Il programma raccoglie informazioni solamente da metodi contenuti nel file di interesse, escludendo eventuali file di libreria secondari (ad es. libc). È anche possibile escludere il binario principale del programma ed eseguire il dump solamente su un solo file di libreria.
L'esecuzione ritorna poi un oggetto dump con tutti i dati raccolti, le istruzioni macchina vengono elaborate da un parser esterno per poter ottenere dei dati strutturati.

2.1.3 Offset finder

L'ultimo componente del modulo permette la ricerca e l'estrazione del nome e dell'indirizzo dei simboli di debug.
Questo modulo analizza il file ELF estraendone i simboli, fa uso di una libreria esterna chiamata PyElfTools e ritorna una serie di coppie nome variabile e offset relativo allo stack pointer/frame pointer (variabile locale) oppure relativo al global pointer (variabile globale).

2.1.4 Input al programma

Non è stata implementata l'interazione con standard input, per variare gli input al programma eseguito si è usato il vettore argv.

2.1.5 Nota sulle prestazioni

Vista la quantità di tempo necessaria ad eseguire questa parte del progetto si è deciso di implementare un sistema di caching dei risultati delle esecuzioni, questo ha permesso di diminuire di molto il tempo di test anche se la cache è arrivata ad avere dimensioni abbastanza importanti (~100GB). Questo componente è stato poi adattato per essere utilizzato parallelamente per velocizzare test che avrebbero richiesto giorni solamente alcune ore.

2.2 Tracer

Questo componente ha lo scopo di effettuare un tracciamento dei valori assunti dalle variabili e di ogni copia di essi all'interno dei vari registri.

2.2.1 Funzionamento

Il programma itera su ogni istruzione eseguita controllando i registri di destinazione o sorgente di un'istruzione che salva o carica una variabile di interesse da o alla memoria. Dopodiché viene processata la propagazione del valore all'interno di altri registri passando in rassegna le istruzioni prima e/o dopo l'operazione di memoria. Questo controllo viene propagato ricorsivamente su tutti i registri interessati dal passaggio della variabile.
Per l'implementazione si è scelto di utilizzare un adapter pattern, ogni istruzione ha un'implementazione di default relativa al suo comportamento (scrittura o sola lettura), per

alcune istruzioni come ad esempio l’istruzione “move” viene definita un'implementazione più specifica.
Questo approccio ha consentito di poter differenziare con facilità le varie casistiche.

2.2.2 Prestazioni

Le prestazioni offerte da questo componente in termini di tempo e spazio sono accettabili ma bisogna fare una precisazione: la grossa criticità risiede nel fatto che non è sempre possibile creare un tracciamento completo in cui viene preso in considerazione ogni singolo valore derivato da un'operazione su una variabile. Se nel caso di alcune operazioni aritmetiche estremamente lunghe questo tracciamento non avvenisse il valore non verrebbe tracciato durante la sua modifica ma solo prima e dopo di essa creando un quadro di tracciamento incompleto, comprendente solamente i valori effettivamente scritti in memoria. Si è deciso quindi di aggiungere un tempo di vita massimo per le variabili modificate. In questo modo la variabile viene tracciata correttamente e il quadro di tracciamento non esplode in casi particolari come nel caso di un algoritmo di hashing in cui il valore finale dipende dal valore iniziale creando una propagazione eccessiva ed inutile.

2.2.3 Valori non aggiornati

Un altra problematica riguarda il tracciamento di valori non aggiornati della variabile che vengono catalogati tramite l’ordine in cui vengono eseguite le operazioni di load e di store. I valori vecchi sono comunque tracciati ma al fine dei nostri calcoli non vengono utilizzati, questi valori come detto sopra dopo un tempo di vita specificato non vengono più tracciati per evitare un quadro di tracing troppo grande e ingestibile.

2.2.4 Verifica

Grazie alla lettura di tutti i valori effettivi contenuti nei registri si è potuto testare il funzionamento del tracer andando a verificare che il valore reale contenuto nel registro coincidesse con quanto riportato nel quadro di tracciamento.

2.2.5 Score

Utilizza la struttura risultante dal tracer e per ogni variabile e per ogni esecuzione effettuata trova in quali registri la variabile è presente quantificando questo valore in cicli di istruzione e avendo così una struttura che mappi in modo completo la presenza di variabili nei registri in maniera aggregata.
Essendo questo un processo abbastanza lungo e tedioso per la macchina è stato deciso di rendere il processo multicore e di salvare i risultati su un file JSON in modo da poterli consultare in seguito senza dover ripetere il processo.

2.3 Calcolo risultati (Componente Evaluate)

In questa fase il componente Evaluate riceve i dati dal componente Score e da DETON e li si grafica in modo da renderli confrontabili.
Da DETON si riceve la metrica (il calore medio dei registri) composto da 32 valori (il numero di registri dell'architettura di riferimento) per ogni esecuzione dell'offuscatore.
Da Score si riceve il numero di cicli di istruzione per cui una variabile è stata presente in un registro ovvero 32 valori per ogni variabile. Score fornisce questi valori per ogni combinazione di input e di programma offuscato.

2.3.1 Media

Il processo di calcolo dei valori d'interesse viene eseguito come illustrato di seguito:
si inizia definendo la percentuale di presenza di una variabile *varx* nel registro *rx* per un'esecuzione *ex*:

$$Presence_{ex,\,rx,\,varx} = \frac{N°\ di\ cicli\ in\ cui\ varx\ \text{è}\ nel\ registro\ rx}{Cicli\ totali\ esecuzione\ ex} \times 100$$

Si procede mediando tutti i risultati ottenuti per un set di parametri dell'offuscatore *prx*:

$$Score_{prx,\,rx,\,varx} = \frac{\sum_{ex \in prx} Presence_{ex,\,rx,\,varx}}{m}$$

Dove *m* è il numero di esecuzioni con parametri *prx*.

Si definisce $Heat_{runx,\,rx}$ come heat del registro *rx* sulla run dell'offuscatore *runx.*
Si procede poi eseguendo la media sui valori dello heat per ogni set di parametri all'offuscatore *prx*:

$$HeatAggregated_{prx,\,rx} = \frac{\sum_{runx \in prx} (Heat_{runx,\,rx})}{n}$$

Dove n è il numero di run dell'offuscatore con parametri *prx.*

2.3.2 Simulazione di Hardware Trojan

In questa fase si simula la presenza di Hardware Trojan nelle modalità seguenti:

Si definisce l'insieme contente tutti i possibili sottoinsiemi di dimensione regnum composti dai registri dell'architettura. Questo insieme verrà chiamato set_{regNum}.

Dato l'insieme set_{regNum} e $collectionRatio \in (0, 1]$ definiamo $testset_{regNum}$ con la seguente regola:

Per $regNum != 1$: è definito come un insieme formato da $num = collectionRatio \times |set_{regnum}|$ elementi presi in maniera casuale da set_{regNum}.

Per $regNum = 1$: vale la seguente: $testset_1 = set_1$.

Si definisce poi il valore aggregato degli score per una variabile *varx,* parametri a DETON *prx* e *regNum* registri osservati dal Trojan:

$$AVG_{regNum,\ prx, varx} = \frac{\sum_{regs \in testset_{regNum}} \sum_{rx \in regs} Score_{prx, rx, varx}}{|testset_{regNum}|}$$

Definiamo inoltre il corrispettivo per la metrica Heat:

$$AVG\ Heat_{regNum, prx} = \frac{\sum_{regs \in testset_{regNum}} \sum_{rx \in regs} HeatAggregated_{rx, prx}}{|testset_{regNum}|\ x\ regNum}$$

Questi ultimi due valori sono quelli che verranno messi a confronto poter fare delle osservazioni.

Note:
Per uno stesso esperimento la parte randomica di *testset* non cambia.
Per semplicità ci riferiremo ad AVG_x come l'esperimento avente *regNum=x.*

3. Esperimenti e Conclusioni

Si è tracciata l'esecuzione di un algoritmo in particolare: sha256, tenendo come variabili di interesse i dati di input al programma (valore da "hash-are") come primo caso e la struttura ctx di sha256 come secondo caso.

3.1 Parametri dell'offuscatore

Si è iniziato a valutare l'efficacia del garbage inserter (inserimento di istruzioni inutili) variando il parametro relativo. I parametri di garbage utilizzati sono stati: 1,10,20,30,40,50.
Si è poi passati a valutare il parametro di obfuscate (offuscamento dei valori immediati nel codice), i parametri utilizzati sono: 1, 2, 3, 4 per poi concludere con un set di parametri ibrido su rispettivamente garbage ed obfuscate ovvero: (1, 10), (2, 20) e (3, 30).
DETON è stato eseguito 10 volte per ogni set di parametri avendo riscontrato una variazione dei risultati tra le singole run.

3.2 Input al programma

Sono state create multiple esecuzioni cambiando gli input al programma (i dati da hashare) varandone dimensione e contenuto.
Nel caso specifico di sha si è pensato che fosse più probabile che il comportamento variasse sulla base della dimensione dei dati in input che con l'effettivo contenuto.
I test effettuati sono stati i seguenti:

- 5 input di 10 byte
- 1 input di 100 byte
- 1 input di 200 byte
- 1 input di 300 byte
- 1 input di 400 byte
- 1 input di 500 byte
- 1 input di 1000 byte

Tutti questi sono stati generati a partire da dati provenienti da /dev/urandom per garantire la randomicità.

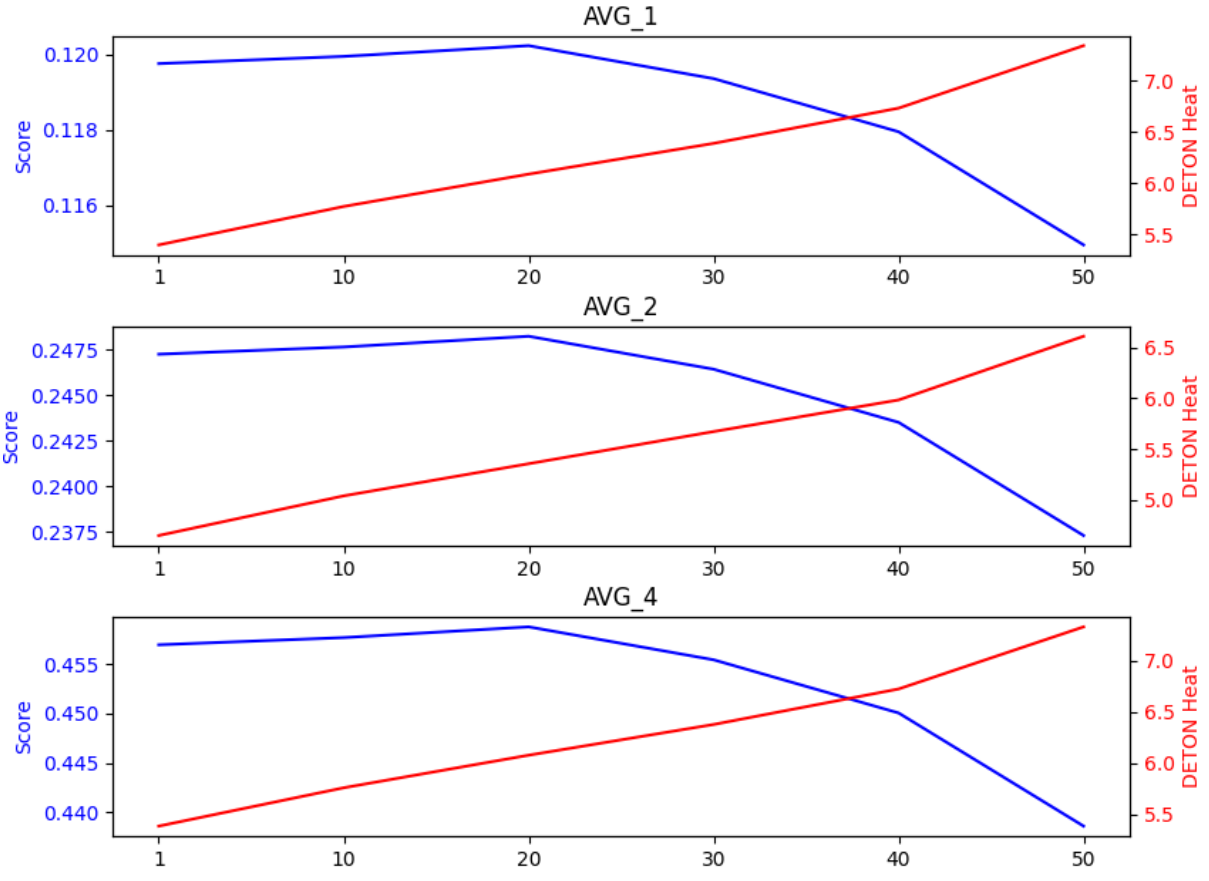
3.3 L'emulazione di un trojan

Per quanto riguarda la creazione dei Trojan si è optato per un $collectionRatio = 3/4$ e si è scelto di eseguire esperimenti con *regNum* uguale a 1, 2 e 4.

3.4 Risultati

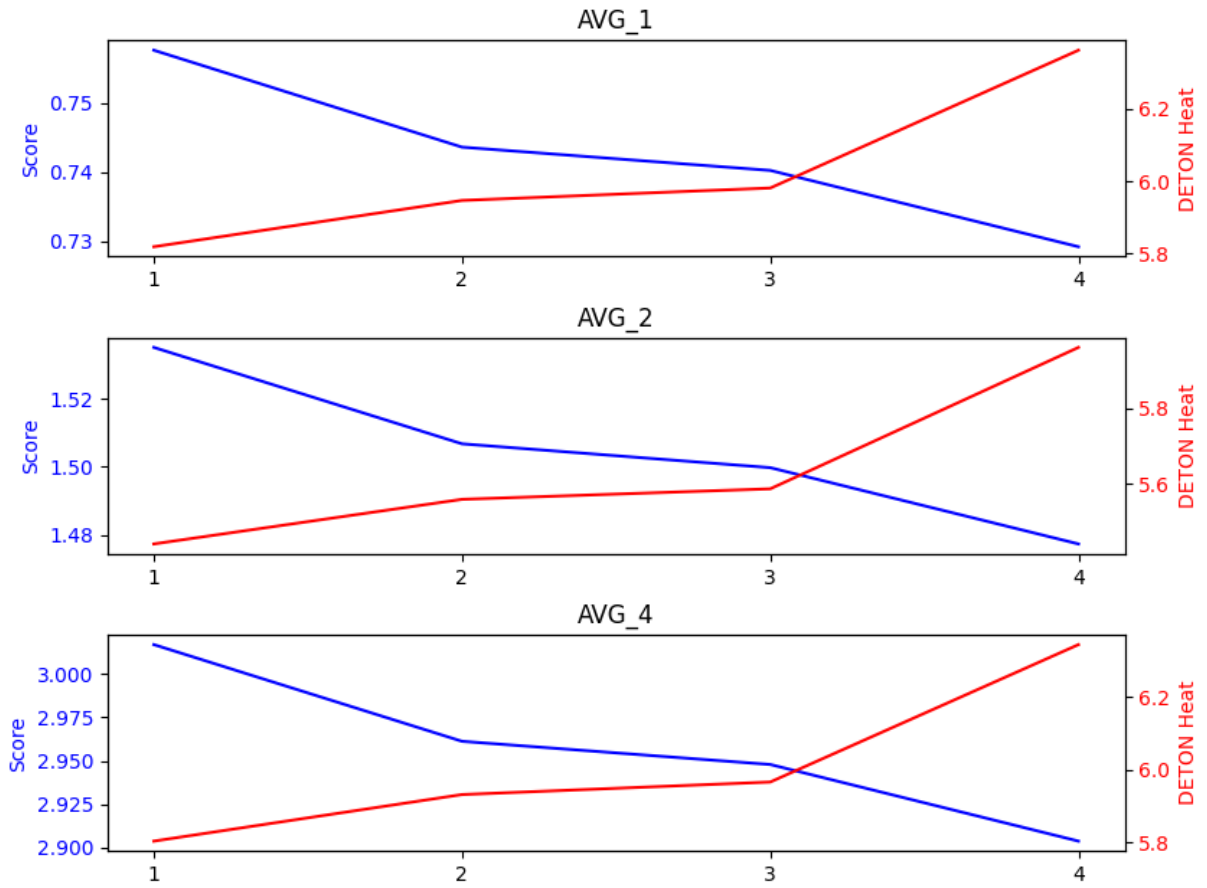
Qui vengono riportati alcuni risultati significativi, verranno discussi nella loro interezza nella conclusione.
I grafici riportati sono ottenuti sfruttando la libreria MatplotLib di Python.

Di seguito si riporta il grafico relativo al tracciamento delle variabili contenenti i dati in input al programma. Sull'asse delle x troviamo la variazione del parametro garbage inserter passato a DETON. Sull'asse delle y abbiamo sia lo score calcolato da noi sia il valore fornitoci dalla metrica di DETON:



Dati versione non offuscata (Valore da hashare):
AVG$Heat_1 = 5.437$ $AVGHeat_2 = 4.724$ $AVGHeat_4 = 5.417$
$AVG1 = 0.09\%$ $AVG2 = 0.18\%$ $AVG3 = 0.33\%$

Grafico simile sulla variazione del parametro rep obfuscate presa come variabile di osservazione la struttura ctx:



Dati versione non offuscata (Struttura ctx):

$AVGHeat_1 = 5.437 \quad AVGHeat_2 = 5.108 \quad AVGHeat_4 = 5.427$

$AVG1 = 0.63\% \quad AVG2 = 1.29\% \quad AVG3 = 2.54\%$

3.5 Conclusioni

Sulla base di quanto ottenuto si nota come la metrica di heat fornitaci da DETON segua un andamento correlabile ai valori ottenuti con il software appena costruito.
In particolare si è notato che a un andamento crescente dello heat corrisponde un andamento decrescente della presenza della variabile.
Notiamo che l'efficienza della versione offuscata migliora con l'aumentare del Heat anche se i valori della versione non offuscata risultano, in tutti gli esperimenti effettuati, sempre superiori alle versioni offuscate.