

Offuscamento del bytecode Java

Corso di Sicurezza su reti
Prof.: Alfredo De Santis
Anno Accademico 2003-2004

A cura di :
Fiorillo Antonio
Lo Prete Giovanni
Miliana Antonella
Santillo Pasquale

1

Sommario

- Protezione del software ←
- Design di un offuscatore Java
- Classificazione delle trasformazioni offuscanti
- Valutazione delle trasformazioni offuscanti
- Trasformazioni del controllo
- Trasformazioni dei dati
- Valori e predicati opachi
- Deoffuscamento e trasformazioni preventive
- Algoritmi di offuscamento

2

Protezione del software

- Il software contiene
 - Informazioni riservate, chiavi segrete
 - Codice (sorgente, oggetto, intermedio, ...)
- Il software è una proprietà intellettuale di chi lo produce
- Il reverse engineering consiste nel ricavare il codice sorgente da una data applicazione
- Un malintenzionato può così accedere alle informazioni segrete di un'applicazione
- Occorre preservare questa proprietà dalla pirateria

3

Protezione del software

- Una volta creata, un'applicazione deve essere distribuita in una qualche forma: codice nativo, codice indipendente dalla macchina (bytecode).
- Il codice nativo è difficile da decompilare ma se un programmatore ha abbastanza risorse in tempo e sforzo, è sempre capace di fare il reverse engineering di un'applicazione.
- Il bytecode ha un formato indipendente dall'architettura.
- Dal bytecode è possibile recuperare in modo estremamente facile la struttura del sorgente.

4

Protezione del software

- Consideriamo il seguente scenario
- Alice è una sviluppatrice software
 - Bob è uno sviluppatore concorrente (malizioso) che vuole trarre vantaggi commerciali dall'accesso ai segreti (algoritmi, strutture dati) delle applicazioni create da Alice

5

Protezione del software

- Alice vuole proteggere i segreti delle proprie applicazioni
- Mezzi legali
 - le leggi sul copyright non coprono adeguatamente gli artefatti software, quindi poco utilizzabili
 - Mezzi tecnici
 - Rendere il reverse engineering impossibile o almeno economicamente proibitivo

6

Protezione del software : strategie

- Evitare che Bob possa avere accesso fisico all'applicazione o almeno alle parti più importanti di essa.
- Esistono varie strategie che Alice può adottare
 - 1) vendere i suoi servizi
 - 2) cifratura del codice
 - 3) compilatori just-in-time
 - 4) offuscamento del codice

7

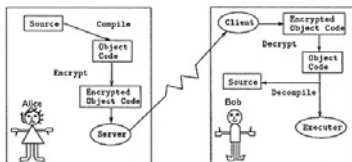
Strategie : vendere i servizi

- Chi vuole usare l'applicazione creata da Alice deve collegarsi al suo sito e pagare una certa quota
- Vantaggi : Bob non avrà mai accesso fisico all'applicazione
- Svantaggi : applicazione più lenta
 - Inerente lentezza della comunicazione in rete
 - Soluzione parziale : dividere l'applicazione in due parti
 - Una pubblica eseguita sul sito dell'utente
 - Una privata eseguita in remoto sul sito di Alice.

8

Strategie : Cifratura

- Cifrare il codice prima di inviarlo agli utenti
- Svantaggi : è efficace solo se l'intero processo di decifratura/esecuzione è effettuato in hardware
 - Bob potrebbe intercettare il codice che passa attraverso la VM



9

Strategie : Compilatori just-in-time

- I compilatori just-in-time traducono al volo il bytecode Java al codice nativo.
- Alice crea una versione in codice nativo della sua applicazione per tutte le piattaforme più popolari.
- Vantaggi : Bob ha accesso solo al codice nativo
- Svantaggi : Il verificatore della VM non può controllare il codice ricevuto
 - può contenere codice malizioso
- Soluzione : Alice firma il codice dimostrando che è affidabile

10

Strategie : Offuscamento del codice

- Alice dà in input la sua applicazione ad un offuscatore
- L'offuscatore è un programma che trasforma l'applicazione in una che sia funzionalmente identica all'originale ma che è molto più difficile per Bob da capire
- Vantaggi
 - Mantiene l'indipendenza dalla piattaforma
 - Non serve la firma del codice
 - Non ci sono ritardi di rete
 - Non c'è bisogno di hardware specifico (es. cifratura)

11

Strategie : Offuscamento del codice

- Svantaggi
 - accesso diretto all'applicazione
 - la probabilità di reverse engineering non è nulla.
 - Bob può usare deoffuscatori automatici
- Il livello di sicurezza aggiunto da un offuscatore ad un'applicazione dipende da
 - le sofisticazioni delle trasformazioni impiegate dall'offuscatore,
 - la potenza degli algoritmi di offuscamento disponibili
 - la quantità di risorse (tempo e spazio) disponibili al deoffuscatore.

12

Sommario

- Protezione del software ←
- Design di un offuscatore Java ←
- Classificazione delle trasformazioni offuscanti ←
- Valutazione delle trasformazioni offuscanti
- Trasformazioni del controllo
- Trasformazioni dei dati
- Valori e predicati opachi
- Deoffuscamento e trasformazioni preventive
- Algoritmi di offuscamento

13

Design di un Offuscatore Java

- L'input del tool è un'applicazione Java
 - insieme delle classi Java.
- Contiene una serie di trasformazioni offuscanti
- Si può selezionare il livello di offuscamento (la potenza) e il massimo overhead (tempo/spazio) che l'offuscatore aggiunge all'applicazione
- occorre effettuare una fase di preprocessing per raccogliere vari tipi di informazioni sull'applicazione per determinare le trasformazioni più adatte da applicare.

14

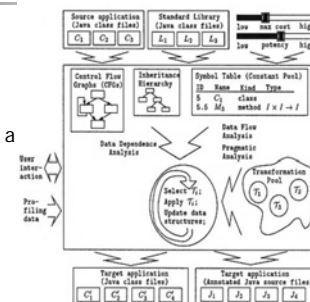
Design di un Offuscatore Java

- Molti dati sono raccolti usando strumenti della teoria dei compilatori
 - l'analisi del flusso di controllo interprocedurale
 - l'analisi della dipendenza dei dati
 - ...
- **L'output del tool è una nuova applicazione funzionalmente equivalente a quella originale ...**
 - ...ma quasi irriconoscibile nella struttura.

15

Design di un Offuscatore Java

- Permette di inserire dei profili per evitare che trasformazioni troppo costose siano applicate a parti del codice usate molto frequentemente.
- Contiene un gran numero di trasformazioni



16

Sommario

- Protezione del software
- Design di un offuscatore Java ←
- Classificazione delle trasformazioni offuscanti ←
- Valutazione delle trasformazioni offuscanti
- Trasformazioni del controllo
- Trasformazioni dei dati
- Valori e predicati opachi
- Deoffuscamento e trasformazioni preventive
- Algoritmi di offuscamento

17

Classificazione delle trasformazioni offuscanti

- Le trasformazioni si classificano e si valutano rispetto alla loro
 - **potenza** - il grado di confusione causato nell'utente malizioso
 - **resilienza** - quanto il codice offuscato è resistente ad un attacco deoffuscante
 - **costo** - quanto overhead è aggiunto all'applicazione originale.
- Il **comportamento osservabile** è definito debolmente come "comportamento sperimentato dall'utente"
 - possibilità di side effect (creare file, inviare messaggi sulla rete)

18

Definizione di trasformazione offuscante

Definizione 1 (Trasformazione offuscante)_ Sia $P \xrightarrow{\tau} P'$ una trasformazione di un programma sorgente P in un altro programma P'.

$P \xrightarrow{\tau} P'$ è una trasformazione offuscante se P e P' hanno lo stesso comportamento osservabile e più precisamente devono essere mantenute le seguenti condizioni :
se P fallisce nel terminare o termina con una condizione di errore, allora P' potrebbe terminare oppure no.
Altrimenti, P' deve terminare e produrre lo stesso output di P.

19

Sommario

- Protezione del software
- Design di un offuscatore Java
- Classificazione delle trasformazioni offuscanti ←
- Valutazione delle trasformazioni offuscanti ←
- Trasformazioni del controllo
- Trasformazioni dei dati
- Valori e predicati opachi
- Deoffuscamento e trasformazioni preventive
- Algoritmi di offuscamento

20

Valutazione delle trasformazioni offuscanti

- Misura della Potenza
- Misura della resilienza
- Misure del costo di esecuzione
- Misura di qualità

21

Metriche della complessità di una trasformazione offuscante (1)

$E(X)$ è la complessità di un componente software X. F è una funzione o metodo, C è una classe, e P è un programma.

μ_1 **Lunghezza del programma** : $E(P)$ aumenta col numero di operatori e di operandi in P.

μ_2 **Cyclomatic Complexity** : $E(F)$ aumenta col numero di predicati in F.

μ_3 **Complessità annidata** : $E(F)$ aumenta con il numero di livelli annidati di predicate condizionali in F .

μ_4 **Complessità del flusso di dati** : $E(F)$ aumenta col numero di riferimenti a basic block in F .

22

Metriche della complessità di una trasformazione offuscante (2)

μ_5 **Complessità fan-in/out** : $E(F)$ aumenta col numero di parametri formali di F, e con il numero di strutture dati globali lette o aggiornate da F.

μ_6 **Complessità delle strutture dati** : $E(P)$ aumenta con la complessità delle strutture dati statiche dichiarate in P .

- La complessità di una variabile scalare è costante.
- La complessità di un array aumenta col numero di dimensioni e con la complessità del tipo degli elementi.
- La complessità di un record aumenta con il numero e la complessità dei suoi campi.

23

Metriche della complessità di una trasformazione offuscante (2)

Metrica 00 : $E(C)$ aumenta con

- μ_{7a} , il numero di metodi in C,
- μ_{7b} , la profondità (la distanza dalla radice) di C nell'albero di ereditarietà
- μ_{7c} , il numero di sottoclassi dirette di C
- μ_{7d} , il numero di altre classi alle quali C è associata
- μ_{7e} , il numero di metodi che possono essere eseguiti in risposta all'invio di un messaggio ad un oggetto di C
- μ_{7f} , il grado di quali metodi di C non fanno riferimenti allo stesso insieme di variabili d'istanza.

24

Potenza della trasformazione

Definizione 2 (Potenza della trasformazione) _ Sia T una trasformazione che conserva il comportamento, tale che $P \xrightarrow{T} P'$, cioè che trasforma un programma sorgente P in un programma P' . Sia $E(P)$ la complessità di P

$T_{pot}(P)$, la potenza di T rispetto al programma P , è la misura di quanto T cambia la complessità di P . Essa è definita nel seguente modo :

$$T_{pot}(P) = E(P')/E(P)-1.$$

T è una trasformazione offuscante potente se $T_{pot}(P) > 0$.

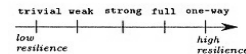
• **una trasformazione è potente** se essa fa un buon lavoro nel confondere Bob

• si misura su una scala a tre punti (low, medium, high).

25

Misura della resilienza

- T è una trasformazione
 - **locale** se ha effetto su un singolo basic block di un **grafo del flusso di controllo** (CFG)
 - **globale** se ha effetto sull'intero CFG
 - **interprocedurale** se ha effetto sul flusso di informazioni tra le procedure
 - **interprocesso** se ha effetto sulle interazioni tra thread di controllo in esecuzione indipendente.
- Una trasformazione è potente se riesce a confondere un lettore umano, ma è resiliente se riesce a confondere un deoffuscatore automatico.
- Si misura su una scala che va da "trivial" a "one-way"



26

Misura della resilienza

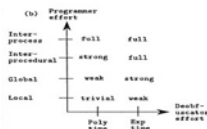
Definizione 3 (Resilienza della trasformazione) _ Sia T una trasformazione che conserva il comportamento, tale che $P \xrightarrow{T} P'$, cioè che trasforma un programma sorgente P in un programma P' .

$T_{res}(P)$ è la resilienza di T rispetto al programma P .

$T_{res}(P)$ = one-way se l'informazione è rimossa da P in modo che P non possa più essere ricostruito da P' .

Altrimenti, $T_{res} = \text{Resilienza}(T_{\text{deoffuscatore}}, T_{\text{programmatore}})$,

dove **Resilienza** è la funzione definita nella matrice



27

Misura del costo di esecuzione

- è l'overhead di tempo/spazio che una trasformazione aggiunge ad un'applicazione.
- è classificato su una scala a 4 punti (free, cheap, costly, dear)

Definizione 4 (Costo della trasformazione) _ Sia T una trasformazione che conserva il comportamento,

▪ tale che $P \xrightarrow{T} P'$, cioè che trasforma un programma sorgente P in un programma P' . $T_{cost}(P)$ è l'overhead in termini di tempo/spazio di P' rispetto a P e può essere :

- **dear**, se P' richiede una quantità esponenziale di risorse rispetto a P .
- **costly**, se P' richiede più di $O(n^p)$, $p > 1$, risorse rispetto a P .
- **cheap**, se P' richiede più di $O(n)$ risorse rispetto a P .
- **free**, se P' richiede più di $O(1)$ risorse rispetto a P .

28

Misura di qualità

Definizione 5 (Trasformazione della qualità) _ $T_{qual}(P)$, la qualità di una trasformazione T , è definita come la combinazione della potenza, della resilienza e del costo di T :

$$T_{qual}(P) = (T_{pot}(P), T_{res}(P), T_{cost}(P)).$$

29

Trasformazioni del layout

- Sono trasformazioni tipiche dei correnti offuscatori Java come CREMA
- Si rimuovono le informazioni di formattazione del codice nei .class
 - è one-way
 - la formattazione originale non può più essere recuperata
 - è poco potente
 - c'è pochissima semantica contenuta nella formattazione
 - è free
 - la complessità di tempo/spazio dell'applicazione non ne risente
- si mischiano i nomi degli identificatori
 - è one-way
 - è free
 - è più potente della precedente
 - Gli identificatori contengono molte informazioni

30

Sommario

- Protezione del software
- Design di un offuscatore Java
- Classificazione delle trasformazioni offuscanti
- Valutazione delle trasformazioni offuscanti ←
- Trasformazioni del controllo ←
- Trasformazioni dei dati
- Valori e predicati opachi
- Deoffuscamento e trasformazioni preventive
- Algoritmi di offuscamento

31

Trasformazioni del controllo

- Predicati opachi
- Trasformazioni computazionali
- Trasformazioni per aggregazione
- Trasformazioni dell' Ordine

32

Trasformazioni del controllo

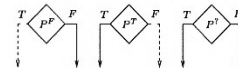
- Le trasformazioni del controllo tentano di oscurare il flusso di controllo dell'applicazione sorgente.
 - **Aggregazione** : spezzano le computazioni logicamente affini oppure fonde computazioni non logicamente affini
 - **Ordine di controllo** : randomizza l'ordine con cui le computazioni sono eseguite
 - **Computazioni del flusso di controllo** : inseriscono nuovo codice
 - Creano molto overhead

33

Predicati opachi

Definizione 6 (Costrutti opachi) Una variabile V è opaca nel punto p di un programma, se V ha una proprietà q in p che è conosciuta al momento dell'offuscamento. Indicheremo ciò con V_p^q o con V^q se p è libero dal contesto.

Un **predicato** P è opaco nel punto p di un programma, se il suo esito è conosciuto al momento dell'offuscamento. Indicheremo con P_p^T (P_p^F) se P è sempre valutato con False(True) nel punto p , e $P_p^?$ se P a volte è valutato True e altre volte è valutato False. Ancora, p è omesso se è libero dal contesto.



- misura della resilienza di una variabile o di un predicato : trivial, ..., one-way.
- misura del costo aggiunto di un costrutto opaco : free, ..., dear.

34

Costrutti opachi trivial e weak

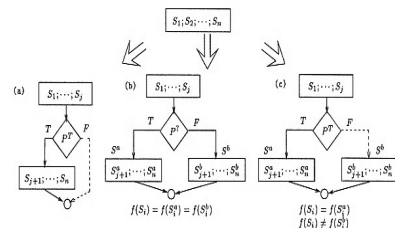
- (a) **trivial**
 - Rotto solo attraverso un'analisi locale statica.
 - Un'analisi locale è ristretta ad un singolo basic block del CFG
- (b) **weak**
 - rotto solo attraverso un'analisi globale statica.
 - Un'analisi globale è ristretta ad un singolo CFG

<pre> { int v, a=5; b=6; v=11 = a + b; if (b > 5) ... if (random(1,5) < 0) ... } </pre> <p>(a)</p>	<pre> { int v, a=5; b=6; if (...) ... : (b is unchanged) if (b < 7) a++; v=36 = (a > 5)?v=b*b:v=b; } </pre> <p>(b)</p>
--	--

35

Trasformazioni computazionali : Inserire codice morto o irrilevante

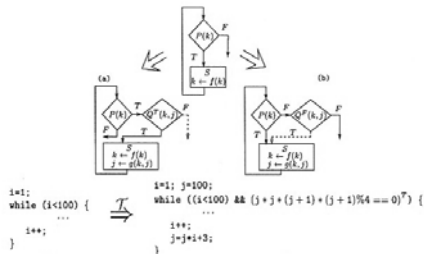
- abbiamo il basic block $S_1; S_2; \dots; S_n$;
- aumentano le metriche μ_2 e μ_3



36

Trasformazioni computazionali : Estendere le condizioni dei cicli

- Il predicato usato $x^2(x+1)^2 \equiv 0 \pmod{4}$ è sempre valutato True



37

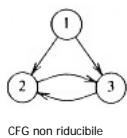
Trasformazioni computazionali : Convertire un grafo di flusso riducibile ad uno non riducibile (1)

- Trasformazioni *language-breaking*
 - il bytecode Java ha un'istruzione goto mentre il linguaggio Java no
 - il CFG prodotto dai programmi Java sarà sempre riducibile
 - il bytecode Java può esprimere CFG non riducibili
 - i CFG non riducibili rendono inutilizzabili molte tecniche di ottimizzazione utilizzate dai compilatori
 - sono rari
- Un CFG G è *riducibile* se e solo se possiamo partizionare gli archi in due gruppi disgiunti, spesso chiamati archi in avanti e archi all'indietro, aventi le seguenti proprietà :
 - Gli *archi in avanti* formano un grafo aciclico in cui ogni nodo può essere raggiunto dal nodo iniziale di G .
 - Gli *archi all'indietro* consistono solo di archi le cui teste dominano le loro code

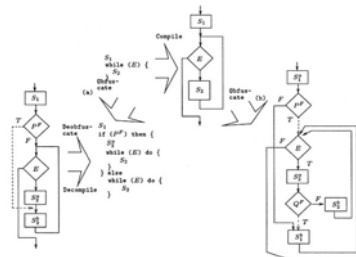
38

Trasformazioni computazionali : Convertire un grafo di flusso riducibile ad uno non riducibile (2)

- Un ciclo con più di un header non è riducibile
- Idea : convertiamo un ciclo in un ciclo con header multipli



CFG non riducibile



39

Trasformazioni computazionali :

Rimuovere le chiamate a libreria e gli idiomi della programmazione

Molti programmi scritti in Java si basano fortemente

- sulle chiamate alle librerie standard.
 - tali chiamate forniscono utili tracce per il reverse engineer.
- Soluzione : trasformarle in chiamate a classi aventi lo stesso comportamento, ma implementate diversamente
- sui cliché (o pattern)
 - Problema : un reverse engineer cercherà tali pattern per iniziare a capire un programma
 - Soluzione : Le tecniche basate sul "*riconoscimento automatico dei programmi*"

40

Trasformazioni computazionali : Interpretazione delle tabelle

- È tra le più efficaci
 - ma è molto costosa
- Si converte una sezione di codice in un differente codice per macchine virtuali.
- Il nuovo codice è eseguito da una VM inclusa nell'applicazione ofuscata
- Consigliabile solo
 - per sezioni di codice eseguite non frequentemente
 - a chi ha bisogno di un livello di protezione molto alto.

41

Trasformazioni computazionali : Aggiungere operatori ridondanti

- Uso delle leggi dell'algebra sulle variabili opache contenute in espressioni aritmetiche
- Aumenta la metrica μ_1

$$\begin{array}{l} (1) X=X+V; \\ (2) Z=L+1; \end{array} \xrightarrow{T} \begin{array}{l} (1') X=X+V * P^{-1}; \\ (2') Z=L+(P=2q/q=P/2)/2; \end{array}$$

P e Q possono assumere tutti i valori il cui quoziente sia 2 ogni volta che l'espressione (2') è raggiunta.

42

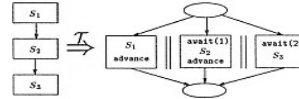
Trasformazioni computazionali : Parallelizzazione del codice

- 1. Possiamo creare dei processi fantoccio che non eseguono alcun compito utile
- 2. possiamo spezzare una sezione sequenziale dell'applicazione in sezioni multiple eseguite in parallelo.
- Vantaggi
 - alta resilienza e potenza
 - analisi statica molto difficile
 - numero di possibili percorsi di esecuzione di un programma cresce esponenzialmente

43

Trasformazioni computazionali : Parallelizzazione del codice

- Svantaggi
 - su macchine uniprocessore l'applicazione gira più lentamente
 - la parallelizzazione risulta difficile in presenza di dipendenza dei dati
 - Occorrono costrutti di sincronizzazione



44

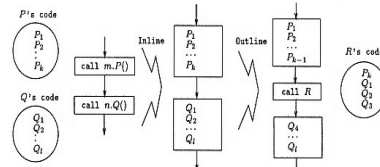
Trasformazioni per aggregazione

- I programmatori vincono l'inerente difficoltà del programmare usando varie astrazioni tra cui l'astrazione procedurale
- Queste astrazioni sono rimosse usando
 - inlining
 - outlining
 - interleaving
 - clonazione
- idea comune
 - (1) il codice che il programmatore ha aggregato insieme in un metodo deve essere rotto e sparpagliato nel programma
 - (2) il codice che sembra non essere logicamente connesso deve essere aggregato in un metodo.

45

Trasformazioni per aggregazione : Metodi Inline e Outline (1)

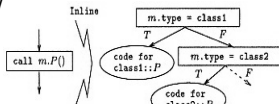
- Inlining _ si rimpiazza una chiamata a procedura con il corpo della procedura chiamata e si rimuove la procedura
 - altamente resiliente (generalmente one-way)
 - Non c'è più traccia dell'astrazione
- outlining _ si trasforma una sequenza di espressioni in una subroutine



46

Trasformazioni per aggregazione : Metodi Inline e Outline (2)

- Nei linguaggi object oriented come Java, l'inlining può non sempre essere una trasformazione totalmente one-way



- L'attuale procedura chiamata dipenderà dal tipo di m determinato a run-time.
- occorre applicare l'inlining a tutti i possibili metodi e ramificare su m

47

Trasformazioni per aggregazione : Metodi Interleave

- La scoperta del codice interfogliato è un difficile compito di reverse engineering
- idea
 - fondere i corpi e la lista dei parametri dei due metodi
 - aggiungere un parametro extra (o parametro globale) per discriminare tra le chiamate ai metodi individuali.

```

class C {
  method M1 (T1 a) {
    S1^1; ... S1^m;
  }
  method M2 (T1 b; T2 c) {
    S2^1; ... S2^m;
  }
}

class C' {
  method M (T1 a; T2 c; int V) {
    if (V == p) {
      S1^1; ... S1^m;
    }
    else {
      S2^1; ... S2^m;
    }
  }
}

{ C x=new C;
  x.M1(a); x.M2(b, c); }

{ C' x=new C';
  x.M(a, c, V==p);
  x.M(b, c, V!=p); }
    
```

48

Trasformazioni per aggregazione : Metodi di clonazione

- per capire il comportamento di una subroutine sono importanti anche i differenti ambienti in cui è stata chiamata
- Idea : offuscare il punto di una chiamata ad un metodo per far sembrare che differenti routine sono state chiamate

```

class C {
  method m (int x)
  { S1...Sk }
}
{ C x = new C;
  x.m(5); ... x.m(7);
}

class C1 {
  method m (int x)
  { S11...Sk1 }
  method m1 (int x)
  { S12...Sk2 }
}
class C2 inherits C1 {
  method m (int x)
  { S11...Sk1 }
}
{ C1 x ;
  if (P1) x=new C1 else x=new C2;
  x.m(5); ...; x.m1(7);
}

```

49

Trasformazioni per aggregazione : Trasformazioni dei cicli

- progettate con l'intento di aumentare le prestazioni) delle applicazioni numeriche.
- Alcune di queste trasformazioni sono molto utili in quanto accrescono la complessità delle metriche
 - loop blocking
 - loop unrolling
 - loop fission
- aumentano la dimensione del codice totale e il numero di condizioni dell'applicazione sorgente
- Singolarmente la resilienza è bassa
- L'uso combinato innalza drammaticamente la resilienza

50

Trasformazioni per aggregazione : Trasformazioni dei cicli – loop blocking

- Il loop blocking è usato per migliorare il comportamento della cache di un ciclo

```

for(i=1,i<=n,i++)
  for(j=1,j<=n,j++)
    a[i,j]=b[j,i]

for(I=1,I<=n,I+=64)
  for(J=1,J<=n,J+=64)
    for(i=I,i<=min(I+63,n),i++)
      for(j=J,j<=min(J+63,n),j++)
        a[i,j]=b[j,i]

```

51

Trasformazioni per aggregazione : Trasformazioni dei cicli – loop unrolling

- Il loop unrolling replica il corpo di un ciclo una o più volte.

```

for(i=2,i<=(n-1),i++)
  a[i] += a[i-1]*a[i+1]

for(i=2,i<=(n-2),i+=2) {
  a[i] += a[i-1]*a[i+1]
  a[i+1] += a[i]*a[i+2]
};
if (((n-2) % 2) == 1)
  a[n-1] += a[n-2]*a[n]

```

52

Trasformazioni per aggregazione : Trasformazioni dei cicli – loop fission

- //loop fission converte un ciclo avente un corpo composto in diversi cicli aventi lo stesso spazio di iterazione

```

for(i=1,i<=n,i++) {
  a[i] += c;
  x[i+i]=d+x[i+1]*a[i]
}

for(i=1,i<=n,i++)
  a[i] += c;
for(i=1,i<=n,i++)
  x[i+i]=d+x[i+1]*a[i]

```

53

Sommario

- Protezione del software
- Design di un offuscatore Java
- Classificazione delle trasformazioni offuscanti
- Valutazione delle trasformazioni offuscanti
- Trasformazioni del controllo \leftarrow
- Trasformazioni dei dati \leftarrow
- Valori e predicati opachi
- Deoffuscamento e trasformazioni preventive
- Algoritmi di offuscamento

54

Trasformazioni dei dati

- oscurano le strutture dati usate nell'applicazione sorgente.
- sono classificate in
 - Storage
 - Encoding
 - Aggregation
 - Ordering .

55

Trasformazioni Storage e Encoding

- In molti casi c'è
 - un naturale modo di immagazzinare un particolare oggetto in un programma
 - Es. `int i; a[i]=3;`
 - una naturale interpretazione del pattern di bit che una particolare variabile può possedere
 - Se i contiene `000000000001100`, questo sarà interpretato come 12.
- Le trasformazioni storage tentano di scegliere una classe di immagazzinamento non naturale sia per i dati dinamici che per quelli statici
- le trasformazioni encoding tentano di scegliere codifiche non naturali per i comuni tipi di dati.

56

Trasformazioni Storage e Encoding : Cambiare la codifica

Come semplice esempio rimpiazzeremo una variabile intera i da con $i' = c1*i + c2$, dove c1 e c2 sono costanti.

```

int i=1;
while (i < 1000) {
  ... A[i] ...;
  i++;
}
    
```

 \Rightarrow

```

int i=1;
while (i < 8003) {
  ... A[(i-3)/8] ...;
  i+=8;
}
    
```

Può essere facilmente deoffuscato usando le comuni tecniche di analisi dei compilatori

57

Trasformazioni Storage e Encoding : Promozione di variabili

- ❖ Esistono trasformazioni del tipo di memorizzazione dei dati che promuovono le variabili da una classe di memorizzazione ad una più generale
- ❖ in Java, una variabile integer può essere promossa ad un oggetto integer

```

int i=1;
while (i < 9) {
  ... A[i] ...;
  i++;
}
    
```

 \Rightarrow

```

Int i = new Int(1);
while (i.value < 9) {
  ... A[i.value] ...;
  i.value++;
}
    
```

- ❖ E' anche possibile cambiare la durata del ciclo di vita di una variabile

```

void P() {
  int i; ... i ...;
}
void Q() {
  int k; ... k ...;
}
    
```

 \Rightarrow

```

int C;
void P() {
  ... C ...;
}
void Q() {
  ... C ...;
}
    
```

58

Trasformazioni Storage e Encoding : Splitting delle Variabili (1)

- ❖ Le variabili aventi raggio d'azione ridotto possono essere spezzate in due o più variabili.
 - ❖ una variabile V che è spezzata in k variabili p1,p2,...pk si indica con $V = [p1,p2,...,pk]$
- ❖ La potenza, la resilienza, e il costo di questa trasformazione crescono con k
 - ❖ Purtroppo i valori di k utilizzabili sono 2 e 3.
- ❖ Per permettere ad una variabile V di tipo T di essere spezzata in due variabili p e q di tipo U, è necessario fornire 3 informazioni:
 - ❖ una funzione $f(p, q) = V$
 - ❖ una funzione $g(V)$
 - ❖ nuove operazioni che possono essere effettuate su p e q.

59

Trasformazioni Storage e Encoding : Splitting delle Variabili (2)

- ❖ ci sono diversi possibili modi di computare la stessa espressione booleana
- ❖ La resilienza può essere ulteriormente migliorata selezionando la codifica a run-time.

$g(V)$	$f(p,q)$	$2p+q$	$VAL(p,q)$	P	$AND(A,B)$	A	$OR(A,B)$	A
0 0	False	0	0 0	0 1	0	0 1 2 3	0	0 1 2 3
0 1	True	1	0 1	1 0	B	1 3 1 2 3	B	1 1 1 2 2
1 0	True	2	1 0	0 1	2	2 0 2 1 3	2	2 2 2 1 1
1 1	False	3	1 1	1 0	3	3 0 0 0 3	3	0 1 1 2 0
	(a)		(b)		(c)		(d)	

```

(1) bool A,B,C;
(2) A = True;
(3) B = False;
(4) C = False;
(5) C = A & B;
(6) C = A & B;
(7) C = A | B;
(8) if (A) ...;
(9) if (B) ...;
(10) if (C) ...;
    
```

 \Rightarrow

```

(1') short a1,a2,b1,b2,c1,c2;
(2') a1=0; a2=1;
(3') b1=0; b2=0;
(4') c1=1; c2=1;
(5') x=AND[2*a1+a2,2*b1+b2]; c1=x/2; c2=x/2;
(6') c1=(a1 & a2) & (b1 & b2); c2=0;
(7') x=OR[2*a1+a2,2*b1+b2]; c1=x/2; c2=x/2;
(8') x=2*a1+a2; if ((x==1) || (x==2)) ...;
(9') if (b1 & b2) ...;
(10') if (VAL[c1,c2]) ...;
    
```

60

Trasformazioni Storage e Encoding :

Convertire dati statici in dati procedurali

- ♦ I dati statici, particolarmente stringhe di caratteri, contengono molte informazioni utili per il reverse engineer
- ♦ Idea : convertire una stringa statica in un programma che produce le stringhe (DFA)

```
String G (int n) {
  int i=0,k;
  String S;
  while (1) {
    L1: if (n==1) {S[i++]="A";k=0;goto L6;}
    L2: if (n==2) {S[i++]="B";k=2;goto L6;}
    L3: if (n==3) {S[i++]="C";goto L9;}
    L4: if (n==4) {S[i++]="X";goto L9;}
    L5: if (n==5) {S[i++]="C";goto L11;}
        if (n>12) goto L1;
    L6: if (k++<2) {S[i++]="A";goto L6} else goto L8;
    L8: return S;
    L9: S[i++]="C"; goto L10;
    L10: S[i++]="B"; goto L8;
    L11: S[i++]="C"; goto L12;
    L12: goto L10;
  }
}
```

*G offusca le stringhe "AAA", "BAAAA", e "CCB".

*G(1)="AAA"

*G(2)="BAAAA"

*G(3)=G(5)="CCB"

*G(4)="XCB"

Trasformazioni per Aggregazione

- ♦ in un programma object oriented, il controllo è organizzato attorno alle strutture dati
 - ♦ array , oggetti.
- ♦ un offuscatore deve provare a nascondere queste strutture dati
 - ♦ Fusioni di variabili scalari
 - ♦ Ristrutturazione di Array
 - ♦ Modificare le relazioni di ereditarietà

Trasformazioni per Aggregazione :

Fusioni di variabili scalari (1)

- ♦ Due o più variabili scalari V_1, \dots, V_k possono essere fuse in una variabile V_n
 - ♦ L'aritmetica sulle variabili individuali deve essere trasformata in una aritmetica su V_n
- ♦ Es. fusione di due variabili integer a 32 bit X e Y in una variabile Z a 64 bit usando la funzione $Z(X,Y) = 2^{32} * Y + X$

$$Z(X+r, Y) = 2^{32} \cdot Y + (r + X) = Z(X, Y) + r$$

$$Z(X, Y+r) = 2^{32} \cdot (Y+r) + X = Z(X, Y) + r \cdot 2^{32}$$

$$Z(X \cdot r, Y) = 2^{32} \cdot Y + X \cdot r = Z(X, Y) + (r-1) \cdot X$$

$$Z(X, Y \cdot r) = 2^{32} \cdot Y \cdot r + X = Z(X, Y) + (r-1) \cdot 2^{32} \cdot Y$$

Trasformazioni per Aggregazione :

Fusioni di variabili scalari (2)

- ♦ La resilienza è veramente bassa
 - ♦ Basta solo esaminare l'insieme di operazioni aritmetiche applicate ad una particolare variabile
 - ♦ Può aumentata introducendo finte operazioni
 - ♦ Es. $\text{if}(P^F) Z = \text{rotate}(Z,5)$

```
(1) int X=45, Y=95;
(2) X += 5;
(3) Y += 11;
(4) X *= c;
(5) Y += d;

(1') long Z=167759086119551045;
(2') Z += 5;
(3') Z += 47244640256;
(4') Z += (c-1)*(Z & 4294967295);
(5') Z += (d-1)*(Z & 18446744069414584320);
```

Trasformazioni per Aggregazione :

Ristrutturare Array

Varie trasformazioni possono essere create per oscurare operazioni eseguite su array : splitting(1-2), merging(3-5), folding(6-7), flattening(8-9)



Trasformazioni per Aggregazione :

Modificare le relazioni di ereditarietà

- Si dice che la classe C_1 eredita dalla classe C_2 e indichiamo ciò con $(C_2 = C_1 \oplus \Delta C_2)$
- la complessità di una classe cresce con la sua ampiezza (gerarchia di ereditarietà).
- Idea : aumentiamo l'ampiezza della gerarchia
 - Fattorizzazione
 - Inserimento di classi fantoccio
 - Variante : falsa rifattorizzazione
- Problema : bassa resilienza
- Soluzione : usate in modo combinato

Trasformazioni dell'ordine

- randomizzare l'ordine con cui le computazioni sono svolte e l'ordine delle dichiarazioni delle variabili è un utile offuscamento
- La potenza è bassa ma la resilienza è one-way.
- possibile riordinare gli elementi all'interno di un array una funzione $f(i)$

```

int i=1, A[1000];
while (i < 1000) {
  ... A[i] ...;
  i++;
}
    ⇨
int i=1, A[1000];
while (i < 1000) {
  ... A[f(i)] ...;
  i++;
}
    
```

67

Sommario

- Protezione del software
- Design di un offuscatore Java
- Classificazione delle trasformazioni offuscanti
- Valutazione delle trasformazioni offuscanti
- Trasformazioni del controllo
- Trasformazioni dei dati ←
- Valori e predicati opachi ←
- Deoffuscamento e trasformazioni preventive
- Algoritmi di offuscamento

68

Valori e predicati opachi

- Costrutti opachi che usano oggetti e alias
- Costrutti opachi che usano i thread

69

Valori e predicati opachi

- vorremmo essere capaci di costruire predicati opachi che richiedano nel caso peggiore un tempo esponenziale (nella dimensione del programma) per romperli e tempo polinomiale per costruirli
- Ci sono due di tali tecniche.
 - Una basata sugli alias
 - L'altra sui thread.

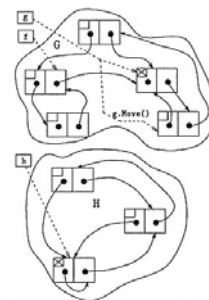
70

Valori e predicati opachi : Uso di alias (1)

- L' aliasing complica significativamente l'analisi statica interprocedurale
 - L'analisi diventa un problema NP-hard o addirittura indecidibile
 - idea : costruire una struttura dinamica complessa e mantenere un insieme di puntatori in questa struttura

71

Valori e predicati opachi : Uso di alias (2)



```

Node g, h;
method P(..., Node f) {
  /* 1 */ g = g.Move();
  h = h.Move();
  /* 2 */ h = h.Insert(new Node);
  ...
  /* 3 */ x.R(..., f.Move());
  ...
  /* 4 */ if (f == g) ...
  /* 5 */ if (g == h) ...
  ...
  /* 6 */ f.Token=False;
  g.Token=True;
  /* 7 */ if (f.Token) ...
  ...
  /* 8 */ f.Token=True;
  h.Token=False;
  /* 9 */ if (f.Token) ...
}
    
```

72

Valori e predicati opachi :

Uso dei thread (1)

- I programmi paralleli sono molto più difficili da analizzare rispetto alla loro controparte sequenziale.
 - semantica interfogliante : n espressioni in una regione parallela (thread)
 - Molte analisi devono considerare n! interfogliamenti
- Idea base analoga a quella che usa gli alias
 - struttura dati globale V aggiornato da thread
- Vantaggi
 - i predicati opachi richiedono nel caso peggiore tempo esponenziale per essere rotti
 - grado molto alto di resilienza
 - Si combinano il data race con gli effetti di interfogliamento e aliasing,

73

Valori e predicati opachi :

Uso dei thread (2)

```

thread S {
  int R;
  while (1) {
    R = random(1,C);
    X = R*R;
    sleep(3);
  }
}

thread T {
  int R;
  while (1) {
    R = random(1,C);
    Y = 7*R*R;
    sleep(2);
    X *= X;
    sleep(5);
  }
}

int X, Y;
const C = sqrt(maxint)/10;
main () {
  S.run(); T.run();
  ...
  if ((Y - 1) == X) P
  ...
}
    
```

74

Sommario

- Protezione del software
- Design di un offuscatore Java
- Classificazione delle trasformazioni offuscanti
- Valutazione delle trasformazioni offuscanti
- Trasformazioni del controllo
- Trasformazioni dei dati
- Valori e predicati opachi ←
- Deoffuscamento e trasformazioni preventive ←
- Algoritmi di offuscamento

75

Deoffuscamento e trasformazioni preventive

- Trasformazioni preventive
- Identificare e valutare i costrutti opachi
- Identificazione tramite Pattern Matching
- Identificazione tramite decomposizione (slicing) del programma
- Analisi Statistica
- Valutazione attraverso il flusso dei dati
- Valutazione attraverso verifica di teoremi
- Deoffuscamento e valutazione parziale

76

Deoffuscamento e trasformazioni preventive

- Un deoffuscatore deve esaminare l'applicazione offuscata e automaticamente identificare e rimuovere i falsi programmi in esso
 - il deoffuscatore deve prima identificare e valutare i costrutti opachi
- trasformazioni preventive sono contromisure che un offuscatore può impiegare per rendere il deoffuscamento più complicato.

```

{
  if (P1) { S1; }
  else S1;
}

{
  if (Q1) { S1; }
  else S1;
}

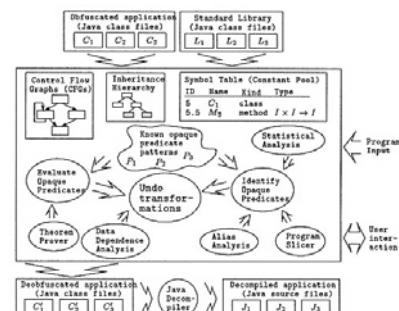
{
  if (P1) { S1; }
  else {
    if (True) { S1; }
    else S1;
  }
}

{
  if (R1) { S1; }
  else S1;
}
    
```

77

Deoffuscamento e trasformazioni preventive

preventive : anatomia di un tool di deoffuscamento



78

Trasformazioni preventive

- Sono progettate per rendere le tecniche di deoffuscamento conosciute più difficili
 - *trasformazioni preventive inerenti*
 - *trasformazioni preventive mirate*

79

Trasformazioni preventive inerenti

- Generalmente hanno poca potenza e molta resilienza.
 - esse hanno la capacità di aumentare la resilienza di altre trasformazioni
- Es. riordiniamo un ciclo for per farlo girare all'indietro
- Non c'è dipendenza dai dati
- Problema : Il deoffuscatore può comunque invertire la trasformazione
- Soluzione : aggiungere una falsa dipendenza dei dati

```
for(i=1;i<=10;i++)  
A[i]=i  
└──┬──  
for(i=10;i>=1;i--){  
A[i]=i;  
for(i=1;i<=10;i++)  
A[i]=i  
└──┬──  
int B[50];  
for(i=10;i>=1;i--){  
A[i]=i;  
B[i]*=B[i+1/2]
```

80

Trasformazioni preventive mirate

- HoseMocha progettato specificatamente per indagare sulle debolezze del decompilatore Mocha
- HoseMocha inserisce istruzioni extra dopo ogni espressione return in ogni metodo del programma sorgente
- Mocha va in crush

81

Identificare e valutare i costrutti opachi

- E' la parte più difficile del deoffuscamento
- Un costrutto opaco può essere :
 - *Locale* (contenuto in un singolo basic block)
 - *globale* (contenuto in una singola procedura)
 - *interprocedurale* (distribuito lungo tutto il programma)

82

Identificazione tramite Pattern Matching

- Si esamina un offuscatore e si costruiscono regole di *pattern-matching* che possano identificare i predicati opachi comunemente usati
- l'offuscatore dovrebbe evitare di usare predicati opachi delimitati

83

Identificazione tramite decomposizione del programma (1)

- Un tool decompone il programma in pezzi maneggevoli chiamati *slice*
- Una slice di un programma P rispetto ad un punto p e ad una variabile v consiste di tutte le espressioni di P che possono aver contribuito al valore di v nel punto p.
- Il tool deve estrarre dal programma offuscato le espressioni degli algoritmi che computano una variabile opaca v

84

Identificazione tramite decomposizione del programma (1)

- L'offuscatore deve rendere la vita difficile allo slicer
 - Aggiungere parametri alias**
 - due parametri formali che si riferiscono sempre a qualche locazione di memoria
 - Lo slicer è rallentato o è reso impreciso
 - Aggiungere dipendenza delle variabili**
 - I popolari slicer funzionano bene per piccoli slice, ma qualche volta richiedono tempo eccessivo per computare quelli grandi.

```

main() {
  int x=1;
  x = x * 3;
}

 $\xrightarrow{T}$ 

main() {
  int x=1;
  if (P^P) x++;
  x = x + V^0;
  x = x * 3;
}
    
```

85

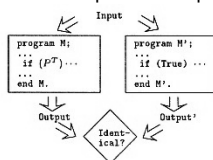
Analisi Statistica (1)

- qualunque metodo di deoffuscamento che esamina le caratteristiche run-time di una applicazione offuscata
 - deve segnalare qualunque predicato che restituisce sempre lo stesso valore di verità durante un gran numero di test eseguiti
 - non può rimpiazzare alla cieca tali predicati con True o False
 - predicati che si comportano come predicati opachi

86

Analisi Statistica (2)

- Si ipotizza il valore del predicato opaco potenziale identificato in M
- Si crea M' in cui il predicato opaco potenziale è rimpiazzato dal valore ipotizzato
- Si testano M e M' con gli stessi input
- Gli input devono coprire tutti i percorsi



87

Contromisure contro l'analisi statistica

- Preferire le trasformazioni che inseriscono i predicati $P^?$
- progettare predicati opachi in modo che diversi predicati devono essere rotti allo stesso tempo.

```

int k=0;
bool Q1(x) {
  k+=2^31; return (P1^?)
}
bool Q2(x) {
  k-=2^31; return (P2^?)
}

{ S1: ...
  S2: ...
}

 $\xrightarrow{T}$ 

{
  if (Q1(j)^?) S1;
  ...
  if (Q2(k)^?) S2;
}
    
```

Se si rimpiazza un predicato (ma non entrambi) con True, k andrà in overflow

88

Valutazione attraverso verifica di teoremi

- Problema : Un predicato opaco può essere rotto attraverso la dimostrazione di un teorema.
- Soluzione : usare i teoremi che sappiamo essere difficili da dimostrare o che sappiamo che nessuna prova esiste.
- problema di Collatz.** Una ipotesi dice che il ciclo terminerà sempre. Sebbene non esista alcuna prova di questa ipotesi, è risaputo che il codice termina per tutti i numeri fino a $7 * 10^{11}$

```

{
  S1;
  S2;
}

 $\xrightarrow{T}$ 

{
  S1;
  n = random(1, 2^32);
  do
    n = ((n%2)!=0)?3*n+1:n/2;
  while (n>1);
  S2;
}
    
```

89

Sommario

- Protezione del software
- Design di un offuscatore Java
- Classificazione delle trasformazioni offuscanti
- Valutazione delle trasformazioni offuscanti
- Trasformazioni del controllo
- Trasformazioni dei dati
- Valori e predicati opachi
- Deoffuscamento e trasformazioni preventive \leftarrow
- Algoritmi di offuscamento \leftarrow

90

Algoritmi di offuscamento

- ALGORITMO 1 (Offuscamento del codice)
- ALGORITMO 2 (SelectCode)
- ALGORITMO 3 (SelectTransform)
- ALGORITMO 4 (Done)
- ALGORITMO 5 (Informazioni pragmatiche)
- ALGORITMO 6 (Priorità dell'offuscamento)
- ALGORITMO 7 (Appropriatezza dell'offuscamento)

91

Algoritmi di offuscamento : ciclo principale

```
WHILE NOT Done(A) DO
  S := SelectCode(A);
  T := SelectTransform(S);
  A := Apply(T,S);
END;
```

- **SelectCode** restituisce il prossimo codice oggetto sorgente che deve essere offuscato.
- **SelectTransform** restituisce la trasformazione che dovrebbe essere usata per offuscare il particolare codice oggetto sorgente.
- **Apply** applica le trasformazioni al codice oggetto sorgente
- **Done** determina quando il richiesto livello di offuscamento è stato ottenuto.

92

Algoritmi di offuscamento : strutture dati

- Per ogni codice oggetto sorgente S e per ogni routine M
 - $P_s(S)$ è l'insieme dei costrutti del linguaggio che il programmatore ha usato in S.
 - usato per trovare le trasformazioni offuscanti appropriate per S
 - $A(S) = \{T_1 \mapsto V_1, \dots, T_n \mapsto V_n\}$ è un mapping tra le trasformazioni T_i e i valori V_i
 - $R(M)$ è il rango del tempo di esecuzione di M
 - $R(M) = 1$ se è speso più tempo per eseguire M che le altre routine

93

Algoritmi di offuscamento : funzioni di qualità

- Restituiscono informazioni di tipo numerico riguardanti ogni trasformazione
 - $T_{res}(S)$ restituisce una misura della resilienza della trasformazione T quando è applicata al codice oggetto sorgente S.
 - $T_{pot}(S)$ restituisce una misura della potenza della trasformazione T quando è applicata al codice oggetto sorgente S
 - $T_{cost}(S)$ restituisce una misura del tempo di esecuzione e dell'overhead di spazio aggiunto da T a S.
 - P_t mappa ogni trasformazione T nell'insieme dei costrutti del linguaggi che T aggiungerà all'applicazione

94

Algoritmi di offuscamento : ALGORITMO 1 (OFFUSCAMENTO DEL CODICE)

- Input
 - Un'applicazione A costituita dai file C_1, C_2, \dots
 - Le librerie standard L_1, L_2, \dots
 - $\{T_1, T_2, \dots\}$
 - $P_t, T_{pot}(S), T_{res}(S), T_{cost}(S)$
 - Un insieme di dati di input $I = \{I_1, I_2, \dots\}$ ad A
 - $AcceptCost > 0$
 - Max overhead accettabile
 - $ReqObf > 0$
 - Quantità di offuscamento richiesta
- Output : Un'applicazione A' offuscata

95

Algoritmi di offuscamento : ALGORITMO 1 (OFFUSCAMENTO DEL CODICE)

1. Caricare l'applicazione C_1, C_2, \dots da offuscare
 - a) caricare file contenenti codice sorgente, oppure
 - b) caricare file contenenti codice oggetto
2. Caricare il codice contenuto nei file delle librerie L_1, L_2, \dots
3. Costruire una rappresentazione interna dell'applicazione
 - (a) un grafo di controllo del flusso (CFG) per ogni routine A.
 - (b) un call-graph per le routine in A
 - (c) un grafo di ereditarietà per le classi in A.
4. Costruire $R(M)$ e $P_s(S)$ usando l'Algoritmo 5, $I(S)$ usando l'Algoritmo 6, e $A(S)$ usando l'Algoritmo 7.

96

Algoritmi di offuscamento :

ALGORITMO 1 (OFFUSCAMENTO DEL CODICE)

5. Applicare le Trasformazioni offuscanti all'applicazione
REPEAT
 $T \leftarrow \text{SelectTransform}(S,A);$
 Applica T ad S ed aggiorna le strutture dati rilevanti del punto 3;
UNTIL Done(ReqObf, AcceptCost, S, T, I)
6. Ricostituire il codice oggetto sorgente offuscato in una nuova applicazione offuscata X

97

Algoritmi di offuscamento :

ALGORITMO 2 (SelectCode)

- Input
 - Il mapping della priorità di offuscamento I come computato dall'Algoritmo 6
- Output : Un codice oggetto sorgente S
- I mappa ogni oggetto sorgente S in I(S).
 - trattiamo I come una coda a priorità
 - selezioniamo S in modo da massimizzare I(S).

98

Algoritmi di offuscamento :

ALGORITMO 2 (SelectTransform)

- Input
 - a) Un codice oggetto sorgente S.
 - b) La mappa di appropriatezza computata dall'Algoritmo 7
- Output : Una trasformazione T
- Due aspetti importanti da considerare
 - T deve essere inglobata in modo naturale con il resto del codice in S.
 - Deve avere un alto valore di appropriatezza in A(S)
 - T deve rendere alti livelli di offuscamento con bassi costi di overhead

99

Algoritmi di offuscamento :

ALGORITMO 2 (SelectTransform)

- Restituisci una trasformazione T tale che
 $T \mapsto V \in A(S)$ e
$$\frac{\omega_1 T_{pot}(S) + \omega_2 T_{res}(S) + \omega_3 V}{T_{cost}(S)}$$
 è massimizzata
dove $\omega_1, \omega_2, \omega_3$ sono costanti definite dall'implementazione

100

Algoritmi di offuscamento :

ALGORITMO 4 (Done)

- Input
 - ReqObf, AcceptCost, S, T, I
- Output
 - a. un ReqObf aggiornato.
 - b. un AcceptCost aggiornato.
 - c. una mappa delle priorità di offuscamento I aggiornata.
 - d. un valore di ritorno booleano che è TRUE se la condizione di terminazione è stata raggiunta.

101

Algoritmi di offuscamento :

ALGORITMO 4 (Done)

- Svolge vari compiti
 - Aggiorna la coda a priorità I
 - La riduzione è basata su una combinazione resilienza/potenza
 - aggiorna anche ReqObf e AcceptCost
 - determina se la condizione di terminazione è stata raggiunta
 $I(S) \leftarrow I(S) - (\omega_1 T_{pot}(S) + \omega_2 T_{res}(S));$
 $\text{ReqObf} \leftarrow \text{ReqObf} - (\omega_3 T_{pot}(S) + \omega_4 T_{res}(S));$
 $\text{AcceptCost} \leftarrow \text{AcceptCost} - T_{cost}(S);$
RETURN AcceptCost ≤ 0 OR ReqObf ≤ 0;

102

Algoritmi di offuscamento :

ALGORITMO 5 (INFORMAZIONI PRAGMATICHE)

- Input
 - Un'applicazione A
 - $I = \{I_1, I_2, \dots\}$
- Output : $R(M)$, $P_s(S)$
- Si computano le informazioni pragmatiche
 1. dinamiche
 - Uso di profiler su I
 - Calcolare $R(M)$ per ogni routine/basic block
 2. statiche
 - Calcolare $P_s(S)$

103

Algoritmi di offuscamento :

ALGORITMO 5 (INFORMAZIONI PRAGMATICHE)

- **FOR** S ← ogni codice oggetto sorgente in A **DO**
 - O ← l'insieme di operatori che S usa;
 - C ← l'insieme dei costrutti del linguaggio ad alto livello (WHILE , eccezioni, threads, etc.) che S usa;
 - L ← l'insieme di classi/routine di libreria che S referencia;
 - $P_s(S) \leftarrow O \cup C \cup L$;
- **END FOR**

104

Algoritmi di offuscamento :

ALGORITMO 6 (PRIORITA' DELL'OFFUSCAMENTO)

- Input
 - Un'applicazione A
 - $R(M)$
- Output : $I(S)$
- Possibili euristiche per $I(S)$ possono essere
 - se molto tempo è speso ad eseguire una routine M, allora M è probabilmente una procedura importante che dovrebbe essere pesantemente offuscata
 - il codice complesso è più probabile che contenga importanti segreti commerciali che semplice codice

105

Algoritmi di offuscamento :

ALGORITMO 7 (APPROPRIATEZZA DELL'OFFUSCAMENTO)

- Input
 - Un'applicazione A costituita dai file C_1, C_2, \dots
 - $P_i, P_s(S), A(S)$
- Output : $A(S)$
- **FOR** S ← ogni codice oggetto sorgente in A **DO**
 - FOR** T ← ogni trasformazione **DO**
 - V ← grado di somiglianza tra e ;
 - $A(S) \leftarrow A(S) \cup \{T \rightarrow V\}$;
- **END FOR**
- **END FOR**

106

Altri usi dell'offuscamento

- Controllo dei pirati del software
 - Si crea una nuova versione offuscata dell'applicazione per ogni nuovo acquirente e si registrano i dati di chi ha acquistato ogni versione
 - Se si scopre una versione piratata la si confronta con il proprio database
- Uso illecito
 - un pirata potrebbe oscurare un programma Java legalmente acquistato
 - La versione offuscata potrebbe essere rivenduta
 - È un problema giuridicamente complesso
 - Il codice rivenduto è completamente differente dall'originale

107