Tecniche automatiche di offuscamento in java

Corso di sicurezza su reti
Prof. Alfredo De Santis

Anno accademico 2005-2006

A cura di:

Boccardo Alberto	056/101841
Camodeca Nicola	056/101801
Caronna Alessandro	056/101791
Santoro Vittorio	056/101971

Sommario

- Introduzione al problema
- Offuscamento del control-flow
- Riorganizzazione delle strutture dati
- Offuscamento parametrizzato delle variabili
- Offuscamento di variabili temporanee

Introduzione al problema

- Ogni programma scritto in java funziona su qualsiasi hardware dotato di JVM.
- La JVM interpreta il bytecode eseguendo le istruzioni codificate.
  - Il bytecode è:
    - Un file binario java.
    - Una forma eseguibile in linguaggio java.

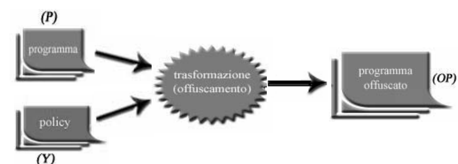
Motivi dell'offuscamento

- Il codice java generato dalla compilazione può sempre essere de-compilato.
- Questo perché i file .class conservano un numero maggiore di informazioni sul codice sorgente rispetto ad un eseguibile tradizionale.

Motivi dell'offuscamento(2)

- Questo facilita la realizzazione di programmi per effettuare un reverse engineering molto accurato.
- *Per arginare la piaga della decompilazione si ricorre all'offuscamento del programma!*

Schema generale di una trasformazione offuscante



Proprietà del programma offuscato

1) Il comportamento del programma offuscato (OP) deve essere semanticamente identico al programma originario (P).

2) Rendere necessaria la conoscenza della policy (Y) per poter capire il programma offuscato (OP).

Proprietà del programma offuscato(2)

- Queste due proprietà garantiscono che il programma offuscato ***OP*** abbia lo stesso comportamento del programma originario ***P***.
- La necessità di conoscere la policy ***Y*** per determinare gli stati del programma, ne limita gli attacchi.

Tecniche di offuscamento

- Di seguito verranno presentate alcune tecniche per offuscare programmi scritti in Java.
  - Definite tecniche automatiche di offuscamento.

Sommario

- Introduzione al problema
- Offuscamento del control-flow
- Riorganizzazione delle strutture dati
- Offuscamento parametrizzato delle variabili
- Offuscamento di variabili temporanee

Offuscamento del control-flow

- Analizzando il control-flow del programma è possibile costruire il ***control-flow graph***.
- Con questa analisi possiamo infrangere la seconda proprietà dell'offuscamento *(robustezza della policy).*

Che cos'è il control-flow graph

Il ***Control Flow Graph*** è un grafo diretto in cui:

- ogni nodo rappresenta un *basic block*;
- ogni arco rappresenta il control-flow tra i basic blocks.

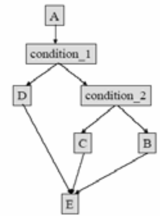
Che cos’è il basic block

Un ***basic block*** è una sequenza di istruzioni consecutive in cui:

- il control-flow entra ed esce:
  - senza diramazioni (if statements)
  - senza terminazione (eccetto che per il blocco finale).

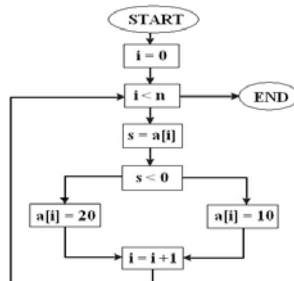
Esempio: costruzione del CFG

```
basic block A
if (condition_1) {
      if (condition_2) {
            basic block B
      }
      else {
            basic block C
      }
}
else {
      basic block D
}
basic block E
```



Esempio: costruzione del CFG(2)

```
for (i=0; i<n; i++) {
     s=a[i];
     if (s<0)
        a[i]=10;
     else
        a[i]=20;
   }
```



Analisi del control-flow

Un malintenzionato può individuare gli archi tra i basic block in due modi:

- derivazione degli archi tra i basic block;
- pattern matching di basic block.

Derivazione degli archi tra i basic block

Il malintenzionato può trovare le istruzioni che trasferiscono il controllo tra i vari blocchi.

- Effettuando un’ ***analisi statica*** (senza eseguire il programma) sul programma offuscato ***OP***.

Pattern matching di basic block

Un malintenzionato può ottenere facilmente gli archi del CFG:

- cercando le corrispondenze tra i blocchi del programma originario ***P*** e quelli del programma offuscato ***OP***.

Offuscamento del control-flow

1. Offuscamento del controllo delle transizioni tra i basic block.
2. Offuscamento dei basic block.

1) Offuscamento del controllo delle transizioni tra i basic block

- I programmi java mantengono molte informazioni il che favorisce l'analisi del controllo delle transizioni.
- Bisogna proteggere le informazioni riguardanti il controllo delle transizioni.

1) Offuscamento del controllo delle transizioni tra i basic block(2)

- Rimpiazzare meccanismi di alto livello con meccanismi di basso livello:
  - meno dipendenze;
  - non impone grosso overhead.
- Rimuovere entità importanti (chiamate di metodi e eccezioni):
  - Complica l'analisi dinamica.

1) Offuscamento del controllo delle transizioni tra i basic block(3)

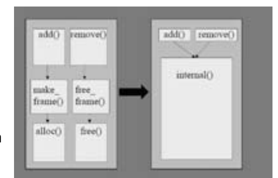
- Fusione di tutti i metodi interni.
- Flattern delle ramificazioni.
- Uso ambiguo del controllo delle ramificazioni.
- Ordinamento arbitrario dei blocchi.

Fusione di tutti i metodi interni

- Fusione di tutti i metodi di una classe in un unico metodo, che gestisce le chiamate ai metodi contenuti in esso.
- I metodi che erano visibili e quindi utilizzabili all'esterno devono continuare ad esserlo.
  - Sono tradotti in interfacce che ricevono le chiamate e passano il controllo ai metodi interni.

Fusione di tutti i metodi interni(2)

- *add* e *remove* sono rimpiazzati con interfacce che si occupano solo di passare il controllo ai metodi interni.
- Gli altri metodi della classe di partenza sono fusi per formare l'unico metodo ***internal()***.



Flattern delle ramificazioni

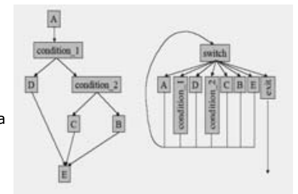
In java:

- le istruzioni condizionali sono facilmente individuabili;
- il malintenzionato può facilmente costruire il CFG.

- Utilizzando tecniche che *appiattiscono* il control-flow del programma si nascondono più informazioni che riguardano le istruzioni.

Flattern delle ramificazioni(2)

- Il control-flow del programma è riorganizzato in:
  - un'istruzione ***switch*** racchiusa in un ciclo che ha come casi i basic block.



Uso ambiguo del controllo delle ramificazioni

- Nella tecnica precedente il flusso di controllo del programma è guidato dalla singola istruzione switch.
  - i valori delle costanti si trovano in cima allo stack e sono passati all'istruzione *tableswitch;*
  - i dati utilizzati sotto forma di costanti sono semplici da analizzare.

Uso ambiguo del controllo delle ramificazioni(2)

- Per eliminare questa vulnerabilità, si possono utilizzare tecniche che oscurano i valori delle variabili usate dall'istruzione *tableswitch*.
- Queste tecniche codificano i valori in ***strutture dati*** migliorandone la sicurezza.

Ordinamento arbitrario dei blocchi

- Non c'è nessun costo nel disporre in maniera *arbitraria* i blocchi utilizzati dall'istruzione *tableswitch:*
  - questo ci permette di offuscare ulteriormente gli archi di un CFG.

2) Offuscamento dei basic block

- Il pattern matching sui basic block può aiutare il malintenzionato a collegare i blocchi offuscati con quelli del programma originale.
- Contando la frequenza dei blocchi si riesce ad effettuare una analisi statica del programma offuscato.

2) Offuscamento dei basic block(2)

- Divisione dei basic block
- Aggiungere istruzioni inutili all’interno dei basic block
- Duplicare i basic block
- Introdurre threading inutile

2) Offuscamento dei basic block(3)

- ***Divisione dei basic block:***
  - Si complicano gli attacchi di tipo pattern matching.
- ***Aggiungere istruzioni inutili all’interno dei basic block:***
  - Rende ancora più confuso il blocco preso singolarmente.

2) Offuscamento dei basic block(4)

- ***Duplicare i basic block***
- ***Introdurre threading inutile***
  - Entrambe le tecniche aumentano la resistenza del programma all’analisi statica.

Sommario

- Introduzione al problema
- Offuscamento del control-flow
- Riorganizzazione delle strutture dati
- Offuscamento parametrizzato delle variabili
- Offuscamento di variabili temporanee

Riorganizzazione delle strutture dati

Problema:
il valore delle variabili di stato permette di determinare il comportamento del programma.

Soluzione:
rendere dispendiosa l’analisi delle variabili di stato.

Esempio:

Abbiamo un programma per gestire le gare di appalto:

- ogni partecipante non deve conoscere le offerte presentate dagli altri.

Problema:

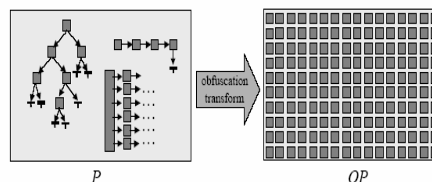
- un malintenzionato, accedendo alle locazioni in cui sono memorizzate le offerte potrebbe:
  1) modificare le offerte degli altri;
  2) fare un’offerta migliore.

Riorganizzazione delle strutture dati(2)

- I dati del programma sono organizzati in strutture dati.
- Per un malintenzionato è semplice analizzare le strutture dati seguendo le relazioni tra i puntatori e gli oggetti.

Obiettivo dell'offuscamento delle strutture dati

Lo scopo dell'offuscatore quindi è quello di memorizzare i dati di OP come nella parte destra della figura:



Proprietà dei dati riorganizzati

- le variabili di stato appaiono tutte allo stesso modo;
- non è semplice trovare le corrispondenze tra le variabili di stato di P e quelle di OP;

Proprietà dei dati riorganizzati(2)

- non è facile trovare le relazioni tra le variabili di OP;
- vedendo i punti di accesso alle strutture si avrà una rappresentazione caotica dei dati.

Controlli di accesso alla memoria utilizzati dal linguaggio java

Java utilizza molti controlli sull'accesso alla memoria:

- Gestione del tipo di memoria
- Garbage Collection
- Limitazioni dei puntatori
- Tipi di classi
- Tipi di casting
- Supporto dei metodi virtuali

Gestione del tipo di memoria

- Java utilizza locazioni di memoria tipizzate.
- Le istruzioni della JVM indicano i tipi delle locazioni di memoria a cui si fa riferimento.
- Analizzando queste istruzioni si può capire il tipo di dato a cui si fa riferimento.

Garbage Collection

- Il compilatore Java utilizza due tipi di strutture per allocare memoria:
  - stack
  - heap
- Utilizzando l'heap, la JVM gestisce la memoria allocata tramite Garbage Collection (GC).

Garbage Collection(2)

- Il GC tiene traccia delle locazioni utilizzate.
- Il malintenzionato attraverso un'analisi dinamica del GC può determinare:
  - dove occorre l'allocazione;
  - i riferimenti tra gli oggetti.

Limitazioni dei puntatori

- In java si utilizzano i riferimenti, simili ai puntatori ma tipizzati.
- Questo non favorisce l'utilizzo di tecniche basate sull'aritmetica dei puntatori e sui riferimenti nulli.
- Tramite i riferimenti è possibile individuare il tipo degli oggetti.

Tipi di classi

- Ogni oggetto usato in un programma java è associato ad un file .class, che specifica:
  - i metodi dell'oggetto;
  - le variabili locali.
- Le informazioni sulle strutture dati potrebbero essere facilmente reperibili per un malintenzionato.

Tipi di casting

- In presenza di operazioni di casting, la trasformazione offuscante deve garantire che:
  - la modifica a tali operazioni non comporti alterazioni al comportamento del programma.

Supporto di metodo virtuale.

- In presenza di metodi virtuali, l'offuscamento deve garantire che:
  - le chiamate all' oggetto ottenuto dalla trasformazione abbiano un comportamento identico a quello originale.

Esistono due approcci per l'offuscamento della memoria:

1) traduzioni di classi
2) memoria non strutturata

Tradurre le classi originali in classi di difficile comprensione ma con comportamento identico.

- Vantaggi:
  semplice da implementare e potrebbe avere prestazioni migliori rispetto alla memoria non strutturata.
- Svantaggi:
  è impossibile gestire i puntatori o i cast regolari di dati ad un puntatore.

Implementa un modello non strutturato di memoria e rappresenta gli oggetti java come regioni di questo spazio.

- Vantaggi:
  meno vulnerabile ad alcuni tipi di analisi dinamica.
- Svantaggi:
  il codice per implementare tale tecnica fornisce maggiori informazioni di quella che si vuol offuscare.

- Utilizzare entrambi gli approcci attraverso un processo in due fasi.
  - La prima fase implementa la traduzione di classe.
  - Il suo output è l'input della seconda fase, la quale implementa la memoria non strutturata.

Obiettivo:

- Rendere insignificanti le informazioni sulla semantica contenute dall'allocatore e dal type system.

Strategia:

- Rappresentare oggetti e puntatori nel maggior numero di modi possibili.
- Rendere eterogenee le operazioni di acceso ai campi.
- Mantenere la consistenza del programma originario.

Fase 1: Traduzioni di classi(3)

Diverse tecniche possono essere applicate in questa fase:

1) Riorganizzazione delle classi
2) Interruzione della corrispondenza uno a uno
3) Accesso parametrizzato ai campi
4) Control-flow per l'accesso ai campi
5) Inserimento di riferimenti spuri

1) Riorganizzazione delle classi

- Date due classi distinte A e B, viene creata una superclasse **AB** che le contiene entrambe.
- Vengono implementati alcuni metodi di A e B scelti arbitrariamente.

1) Riorganizzazione delle classi(2)

- Se viene usato un metodo di A sui dati in comune nella superclasse.
  - Se questo metodo usa tali dati su un oggetto creato da B, l'oggetto viene **invalidato**.

2) Interruzione della corrispondenza uno a uno

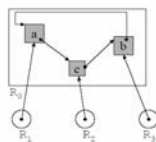
- Un singolo ***user object*** *(oggetto di P)* **non deve** corrispondere ad un ***synthetic object*** *(oggetto di OP).*
  - ***User object*** di tipi differenti devono essere rappresentati dallo stesso ***synthetic type*** *(tipo di oggetto in OP).*
  - Viceversa oggetti dello stesso tipo devono essere divisi in oggetti differenti che sono collegati tra loro.

3) Accesso parametrizzato ai campi:

- Viene interrotta la corrispondenza uno ad uno:

```
class user {
  int a, b c;
}

class syn {
      int x;
      syn p;
}
```



- R1 , R2 ed R3 sono rappresentazioni di un puntatore alla classe user.

Svantaggio dell'accesso parametrizzato ai campi

- La ripetizione del codice di accesso ai campi rende individuabili i campi utilizzati.
- **Soluzione:**
  - identificare i blocchi che usano un campo dell'user-type;
  - mantenere un puntatore all'area di memoria che contiene quel campo.

4) Control-flow per l'accesso ai campi

- Per nascondere il codice di accesso ai campi:
  - si identificano i basic-block;
  - si applica l'offuscamento del control-flow a tali blocchi.
- Miscelando il codice di accesso ai campi con il control-flow si ottiene un codice con un significato meno ovvio.

5) Inserimento di riferimenti spuri

- Aumenta la complessità apparente del programma.
- Per collezionare correttamente i *riferimenti legittimi*, si può:
  1) copiare periodicamente tutti i puntatori spuri;
  2) trasformare dei puntatori spuri in puntatori effettivi, introducendo un flag per rendere note tali modifiche.

5) Inserimento di riferimenti spuri(2)

- Svantaggi:
  - tecnica difficile da implementare;
  - vulnerabile ad analisi statica.
- Vantaggi:
  - usata insieme ad altre tecniche rende più costoso il deoffuscamento.

Fase 2: Memoria non strutturata

- Consiste nell'implementare un semplice array di byte e generare le istruzioni JVM per accedervi.
- Si implementano variabili di stato con all'interno un array di byte.

Fase 2: Memoria non strutturata(2)

- Si realizza la memorizzazione dei dati senza utilizzare:
  - le istruzioni di allocazione della JVM;
  - le opzioni del Garbage Collection.

Fase 2: Memoria non strutturata(3)

Questa fase si focalizza sull'offuscamento delle quattro caratteristiche chiave di java:

1) Mascherare l'allocazione
2) Smart Garbage collection
3) Complex-pointers e Type Queries
4) Chiamata a metodi virtuali

1) Mascherare l’allocazione

- Implementando array pre-allocati, non si possono rilevare i meccanismi di allocazione delle JVM.
- Variando la quantità di memoria da allocare per l’array, non è facile capire il tipo di oggetto allocato.

1) Mascherare l’allocazione(2)

- Allocare memoria in blocchi di dimensioni uguali per ciascun user-object.
- I meccanismi di allocazione non sono correlabili allo stato attuale del programma se gli oggetti sono pre-assegnati.

2) Smart Garbage Collection

- Il codice del GC generato deve gestire l’allocazione e la de-allocazione dei basic-block offuscati.
- Si utilizza un flag ***in-use*** associato ad ogni blocco.
- Periodicamente si de-allocano tutti i blocchi non in uso.
- Offuscamento del codice per gestire i flag.

2) Smart Garbage Collection(2)

- Implementare una serie di metodi virtuali per effettuare la marcatura dei blocchi.
- Tali metodi utilizzano i flag in-use e ripetono ricorsivamente la procedura.
- Il codice che utilizza i flag in-use è distinguibile da un normale frammento di codice.

3) Complex-Pointer e Type Queries

- **Obiettivo:**
  rendere diversa ciascuna chiamata a funzione.
- Le rappresentazioni degli user-object devono essere le stesse per ogni tipo di dato per l’intero programma.
- Si avrebbe una incompatibilità con i metodi di accesso ai campi tra due sezioni del programma.

3) Complex-Pointer e Type Queries(2)

- Si devono supportare le Type Queries su un puntatore a una classe.
- Molti linguaggi implementano un link dagli oggetti alle loro strutture della classe.
- Le Type Queries possono essere facilmente risolte utilizzando i link.

3) Complex-Pointer e Type Queries(3)

- **Problema:**
  i link rivelano le strutture dell'oggetto e informazioni sui tipi globali a run-time
- **Soluzione:**
  utilizzare i complex-pointer al posto dei simple pointer.

3) Complex-Pointer e Type Queries(4)

- I complex-pointer:
  - rappresentano differenti gerarchie di classi in modi differenti;
  - memorizzano informazioni dell'oggetto nel puntatore;
  - possono variare il codice di accesso ai campi e le Type Queries.
- Combinando i complex-pointer con l'offuscamento del control-flow si possono confondere le Type Queries e l'accesso ai campi.

4) Chiamata a metodi virtuali

- **Problema**:
  Riferendosi ad informazioni rappresentate globalmente si rivelano:
  - i tipi specifici degli oggetti;
  - le informazioni sui tipi globali.

4) Chiamata a metodi virtuali(2)

- **Soluzione:**
  Memorizzare localmente le informazioni sui metodi virtuali:
  - verrà determinato localmente il metodo virtuale da chiamare;
  - verrà resa più costosa possibile la determinazione locale del metodo.

4) Chiamata a metodi virtuali(3)

- Si riesce a determinare staticamente il tipo di un oggetto a tempo di offuscamento.
- Quando il metodo è determinato dinamicamente, si utilizza l'offuscamento del control-flow.

4) Chiamata a metodi virtuali(4)

- Evitare di generare un eccessivo codice di dispatch riutilizzando il codice per i normali puntatori.
- Il codice dei puntatori viene inserito all'interno dei complex-pointer.
- Evitare l'uso di codice che utilizza puntatori perché facilmente individuabile.

4) Chiamata a metodi virtuali(5)

- Per derivare una sottoclasse da una classe base non offuscata:
  - si mantiene un puntatore ad un'interfaccia offuscata che ha gli stessi metodi della classe base.
- Per chiamare un metodo nell'interfaccia:
  - si esegue la chiamata al metodo esterno o al metodo offuscato.

Conclusioni del metodo a due fasi

- L'offuscamento della memoria è generalmente un problema difficile.
- Le tecniche analizzate possono riorganizzare in maniera consistente le strutture dati del programma.
- Combinando tra loro tali tecniche si può ottenere un buon livello di offuscamento.

Sommario

- Introduzione al problema
- Offuscamento del control-flow
- Riorganizzazione delle strutture dati
- Offuscamento parametrizzato delle variabili
- Offuscamento di variabili temporanee

Offuscamento parametrizzato di variabili: Obiettivi

- produrre tecniche di offuscamento che rallentino il reverse engineering;
- rendere i dettagli dell'offuscamento dipendenti da dati *random.*

Offuscamento parametrizzato di variabili: Obiettivi(2)

- Supportare operazioni aritmetiche:
  - addizione
  - moltiplicazione
  - operatori logici

direttamente sulla versione offuscata del software.

Offuscamento parametrizzato di variabili: Obiettivi(3)

- Nascondere un parametro necessario ***p,*** dove la variabile parametrizzata è una tripla ***[A,p,e]*** dove:
  - ***e:*** è un valore/variabile non offuscato;
  - ***A:*** è il tipo di algoritmo della trasformazione offuscante;
  - ***p:*** è il parametro a run-time che modifica il comportamento della trasformazione.

- Nascondere le relazioni tra le variabili.
  - Il malintenzionato non deve essere in grado di capire lo schema di offuscamento.
    - Anche se si presentano blocchi simili offuscati con lo stesso schema.

Il modo in cui una variabile viene usata, è un fattore chiave per il successo di una trasformazione offuscante.

- Vediamo come comportarci con variabili diverse:

- Il limite per un contatore
- Accumulatore limitato
- Aritmetica semplice usando interi segreti
- Calcoli complessi usando interi segreti
- Variabili e operazioni booleane
- Variabili di ogni tipo

Dato il ciclo:

```
for (int i = 0; i < secret_int; ++i)
{
     bar(foo(a),b);
}
```

- Se ***secret_int*** è non negativo allora il test può essere rimpiazzato con:

  i != secret_int.

- inoltre è possibile usare il contatore all'interno del ciclo:

  bar(foo(a),b); → **bar(foo(i), b);**

- **Nella versione offuscata:**

  Se il *contatore* può essere confrontato con *secret_int*

- un malintenzionato può vedere le seguenti cose:
  - offuscamento di ***zero*** confrontabile con l'offuscamento di ***secret_int;***

Il limite per un contatore(4)

- l’operazione offuscata ***++i*** confrontabile con l’offuscamento di ***secret_int***;
- il ***test di uguaglianza***, che alcuni sistemi eseguono con un semplice test sui bit.

Accumulatore limitato

Dato:

```
secret_total = 0;
while (secret_total < secret_int)
{
      secret_total = secret_total + foo();
}
```

Accumulatore limitato(2)

- **Nella versione offuscata:**

  se *secret_total* può essere confrontato con *secret_int* e sommato con *foo()*

- un malintenzionato può vedere le seguenti cose:
  - il test relazionale offuscato “<” tra ***secret_total*** e ***secret_int***;

Accumulatore limitato(3)

- L’operazione aritmetica offuscata di ***foo()+secret_total;***
- Entrambi i valori di ritorno offuscati di ***foo()*** in un modo confrontabile con ***secret_total***, prima o dopo il return.

Aritmetica semplice usando interi segreti

Dato:

```
secret_int_A = (secret_int_B + secret_int_C)*
                   secret_int_A;
```

Aritmetica semplice usando interi segreti(2)

- **Nella versione offuscata:**

  Se “+” e “*” hanno la stessa forma offuscata

- un malintenzionato può vedere:
  - l’aritmetica offuscata di entrambi gli operatori.

Calcoli complessi usando interi segreti

Dato:

```
secret_total = 1;
secret_partial = encrypted_message;
while (secret_exponent_temp != zero )
{
    if (lsb(secret_exponent_temp) == 1 )
    secret_total = secret_total * secret_partial;
    secret_exponent_temp == secret_exponent_temp
                                   >> 1;
    secret_partial = secret_partial * secret_partial;
}
```


Calcoli complessi usando interi segreti(2)

- **Nella versione offuscata:**

  Se *"secret_total * secret_partial"* non è offuscato.

- un malintenzionato può vedere:
  - L'operazione "*" offuscata con bit shiftati di *secret_exponent_temp.*
  - Il test di uguaglianza con il bit meno significativo del valore *secret_exponent_temp* non offuscato.

Variabili e operazioni booleane

Dato:

```
if (foo AND bar )
{
 total = total + 1;
}
```

- **Nella versione offuscata:**
- un malintenzionato può vedere:
  - entrambi i deoffuscamenti dei valori booleani o l'equivalente offuscato dell'and logico.

Variabili di ogni tipo

Dato il seguente frammento di codice:

```
# obfus_tran_A() è implementato con l'inline
  bytecodes
secret_integer = obfus_tran_A(param_foo,
                         sensitive_info);
# sovrascrittura
sensitive_info = <unused data>;
```


Variabili di ogni tipo(2)

- un malintenzionato può:
  - esaminare obfus_tran_A(p, arg) determinando p e deoffuscando ogni variabile offuscata in modo simile;
  - effettuare delle corrette operazioni aritmetiche e logiche, sulle variabili offuscate.

Criteri per generare parametri di offuscamento

- Il malintenzionato per capire il funzionamento completo del programma offuscato deve determinare i parametri di offuscamento delle variabili.
- L'assegnamento di un parametro ad una variabile non deve aiutare un malintenzionato nella comprensione del programma offuscato.

Criteri per generare parametri di offuscamento(3)

- L'insieme dei parametri possibili per una variabile dovrebbe essere abbastanza grande ed essere distribuito uniformemente e in modo ***random***.
- I parametri di offuscamento dovrebbero essere esposti quanto meno possibile e per un periodo di tempo più breve possibile.

Criteri per proteggere la rappresentazione di dati

- Se possibile deve essere tenuta segreta quale tipo di trasformazione di offuscamento è stata usata per proteggere una particolare variabile.
- Il malintenzionato che non conosce il parametro di offuscamento di una variabile non dovrebbe poterla deoffuscare tranne che con la ricerca esaustiva.

Criteri per proteggere la rappresentazione di dati(2)

La trasformazione di offuscamento dovrebbe servirsi di semplici operazioni di base:

- per le ***variabili intere*** le operazioni di base sono: addizione, sottrazione, moltiplicazione, divisione, resto, AND, OR, NOT, XOR e bit shift;
- per le ***variabili booleane*** le operazioni di base sono: test, AND, OR, NOT.

Tecniche per generare parametri di offuscamento

- Utilizzo del control-flow offuscato
- Environment testing (test d'ambiente)

1) Utilizzo del control-flow offuscato

In questo approccio i parametri usati per (de)offuscare variabili sono costruiti come parti di una serie di blocchi di codice.

- Questo implica che un parametro di offuscamento può essere determinato solo conoscendo la corretta esecuzione delle sottosequenze del programma.

1) Utilizzo del control-flow offuscato(2)

Esempio:

- il programma consiste di un insieme di blocchi di codice:
  - **A,…, B,…, C,…, D,…**

Il blocco A contiene:

```
## T(i) si riferisce al parametro variabile pv1
## pv1 è conosciuto a priori ed è uguale a w
T(i) = T(i) + x
```

Il blocco B contiene:

```
## S(j) si riferisce al parametro variabile pv1
S(j) = S(j) * y
## in alternativa S(j) = S(j) OR y
```

Il blocco C contiene:

```
## P(m) si riferisce al parametro variabile pv1
## Funct() è un'operazione binaria
W(K) = W(k) + a
## P(m) è uguale a ( ( (W + x) * y) + a)
Z = Funct(P, m, V, X)
```

Il blocco D contiene:

```
## R(K) si riferisce al parametro variabile pv1
R(k) = R(k) + z
```

Il blocco E contiene:

```
## Q(l) si riferisce al parametro variabile pv1
L = Q(l) + zz
```

1) Utilizzo del control-flow offuscato(3)

- La variabile offuscata ***v*** è adoperata da **Funct()** usando il parametro variabile ***pv1***.
- Le etichette ***E*** e ***F*** si riferiscono ad altri blocchi di codice del programma offuscato che non fanno parte dell'esecuzione della sottosequenza.
- In questo esempio le variabili ***i***, ***j***, ***k*** e ***l*** non sono uguali quando sono utilizzate.

1) Utilizzo del control-flow offuscato(4)

- Se i blocchi sono eseguiti in ordine differente allora ***pv***, diventa scorretto quando viene eseguito ***Funct*()**.
- L'uso di un ***array di elementi*** per realizzare *assegnazioni ingannevoli* ai *parametri di offuscamento* incrementa il costo dell'analisi del control-flow per determinare i parametri di offuscamento!

2) Environment testing

Con questo metodo i parametri sono derivati, durante l'esecuzione, dalle informazioni del programma offuscato:

- tipicamente queste informazioni sono fornite dal sistema che fa funzionare il programma offuscato, compreso il suo stato.

2) Environment testing(2)

Esempio:

- Il valore di **X** è nascosto creando **hash(X)**.
- Nel programma offuscato viene memorizzato hash(X) e non X.
- Il programma opera ed interroga il sistema operativo o esamina lo stato del programma.

2) Environment testing(3)

- I risultati Y1, Y2…. vengono sottoposti alla funzione hash creando hash(Y1) e hash(Y2) e confrontati con hash(X).
- Se hash(Yi) è uguale ad hash(X), allora Yi, o qualche altra funzione di Yi, è il deoffuscamento di hash(X).
  - Senza l'appropriato input della funzione hash, il malintenzionato non è in grado di recuperare X.

Tecniche per proteggere la rappresentazione dei dati

1. Cifrari a blocchi a chiave simmetrica e cifrari di flusso
2. Tecniche di cifratura a chiave asimmetrica
3. Interi memorizzati come alberi di Parser
4. Interi offuscati attraverso la mappatura in strutture matematiche basate su crittosistemi
5. Impacchettamento
6. Offuscamento di booleani

1) Cifrari a blocchi a chiave simmetrica e cifrari di flusso

Questo metodo sostituisce i numeri interi con numero interi cifrati:

- usando un cifrario a blocchi a chiave simmetrica quali il **DES** oppure **AES** oppure un cifrario di flusso quale **SEAL**.

1) Cifrari a blocchi a chiave simmetrica e cifrari di flusso(2)

Un'eccezione è il crittosistema a chiave simmetrica **Pohlig-Hellman** che supporta anche la moltiplicazione offuscata.

- **$C = M^e$ mod p** ## p è primo
  $M = C^d$ mod p ## e * d = 1 mod (p-1)

Crittosistema Pohlig-Hellman

2) Tecniche di cifratura a chiave asimmetrica

Vengono usati sistemi di cifratura a chiave asimmetrica come **RSA**, **EL-Gamal** o **XTR**:

- sono supportate offuscamento dell' incremento e le moltiplicazioni.

2) Tecniche di cifratura a chiave asimmetrica(2)

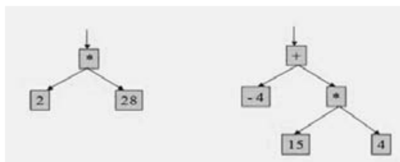
Può inoltre essere usata una variante di RSA detta RSA-bidirezionale.

- **$C = M^e$ mod n** ## n = pq,(p e q sono primi)
  $M = C^d$ mod n ## e * d = 1 mod (p-1)(q-1)

Crittosistema RSA

3) Interi memorizzati come alberi di Parser

- Questo metodo rimpiazza gli interi con un albero di Parser.
- L'operazione di offuscamento delle variabili, come l'addizione, crea nuovi alberi di Parser che vengono combinati.
  - Per esempio il valore 56 può essere rappresentato da entrambi gli alberi:



4) Mappatura in strutture matematiche

- Mappare col teorema cinese del resto
- Permutazioni e altre corrispondenze di ordinamento interno alle liste
- Altre manipolazioni della teoria dei numeri

Teorema cinese del resto

Un importante proprietà nella teoria dei numeri ci dice che:

- presi degli interi a1, a2,…. aj, e un insieme di primi positivi m1, m2, …. mj, esiste un unico X tale che 0 <= X <= (N = mj) tale che X = aj mod mj.
- dove i parametri di offuscamento saranno i valori di N e la sua parziale fattorizzazione m1, m2, …. mj.

Permutazioni e altre corrispondenze di ordinamento interno alle liste

Possiamo offuscare gli interi in permutazioni:

- definiamo 0 come l'operatore incremento;
- 1 definito come l'operatore incremento applicato due volte, e così via.

Permutazioni e altre corrispondenze di ordinamento interno alle liste(2)

Esempio:

- presa la permutazione (1 2) (7 6 5 4 3 8):
  - 0 verrà offuscato come (1 2) (7 6 5 4 3 8)
  - 1 verrà offuscato come (1) (2) (7 5 3) (8 6 4)
  - 2 verrà offuscato come (1 2) (7 4) (8 5) (6 3)
  - etc…

Altre manipolazioni della teoria dei numeri

Un'altra tecnica di offuscamento mappa la variabile intera Y nel seguente valore:

(Y = y * p)

- dove p è un intero primo più grande del valore assoluto di tutti gli interi non offuscati che necessitano di essere rappresentati usando p.

5) Impacchettamento

Questo metodo tenta di proteggere le variabili, da un'analisi dinamica dei riferimenti dei valori.

- Invece di usare parole separate per memorizzare variabili diverse, più variabili sono impacchettate nella stessa parola:
  - impacchettamento di bit;
  - impacchettamento basato sui numeri primi.

Impacchettamento di bit

Ogni insieme di bit è usato per memorizzare una variabile separata come una variabile booleana o un piccolo intero.

I parametri per le variabili controllano quale bit della parola rappresenta la variabile.

Impacchettamento basato sui numeri primi

Il valore di una parola (trattato come un intero senza segno) *mod* un numero primo o un numero composto, è il valore di una variabile separata.

Esempio:

- se una parola contiene 25 può essere usata per rappresentare due variabili, una contiene 7 (mod 9) e l'altra contiene 12 (mod 13).
- 9 e 13 sono i parametri.

Questa tecnica è il CRT su campi piccoli!!

6) Offuscamento di booleani

- Le variabili booleane vengono rappresentate da variabili intere.
  - Esempio:
    - si può dividere il valore della variabile per qualche valore intero per determinare il corrispondente booleano.

Sommario

- Introduzione al problema
- Offuscamento del control-flow
- Riorganizzazione delle strutture dati
- Offuscamento parametrizzato delle variabili
- Offuscamento di variabili temporanee

Offuscamento di variabili temporanee

La manipolazione dei basic block, discussa precedentemente, riorganizza i blocchi di codice e maschera i cicli.

- Tuttavia, il malintenzionato può ancora trovare delle corrispondenze tra il codice offuscato e il codice originale.

Offuscamento di variabili temporanee(2)

L'offuscamento parametrizzato invece va bene solo per i dati persistenti, avendo un costo computazionale inaccettabile per le variabili temporanee.

- Verranno ora mostrate delle tecniche che influenzano poco le performance per realizzare l'offuscamento di variabili temporanee.

Offuscamento di variabili temporanee(3)

1. **XOR con costante**
2. **XOR con variabile**
3. **Bit più significativo**
4. **Utilizzo di offsets**
5. **Rappresentazione con modulo**
6. **Rotazione**
7. **XN mod 1**
8. **Divisione delle locazioni di memoria**
10. **Array chasing**
11. **Utilizzo di tabelle per sostituire gli interi**

1) XOR con costante

È lo XOR tra le variabili locali e una costante nota.

- Questo schema garantisce il rispetto dell'obiettivo di performance, dato che la trasformazione è piccola e veloce.
- Fallisce invece per quel che riguarda l'obiettivo di offuscamento se la trasformazione è limitata ad un singolo blocco.

1) XOR con costante(2)

Esempio:

dato il seguente bytecode:

```
ldc 12345678
istore_1
goto loop
start:    iload_1
ldc 12345678
ixor
invokestatic foo
iload_1
ldc 12345678
ixor
invokestatic bar
iload_1
ldc 12345678
ixor
iconst_1
iadd
ldc 12345678
ixor
istore_1
loop:     iload_1
ldc 12345678
ixor
iconst_5
if_icmplt start
```

- Si noti che "ldc 12345678" appare spesso durante il ciclo.
- Purtroppo, mostrare continuamente le costanti chiave nel codice, fa in modo che il segreto possa diventare più facile da identificare.

1) XOR con costante(3)

- Questa trasformazione è efficace quando sono richieste solo operazioni di confronto e di assegnamento.
- Lo XOR con costante ha quindi il vantaggio di permettere rari de-offuscamenti del valore originale, cosa che non è permessa da schemi basati su permutazioni arbitrarie di valori.

2) XOR con variabile

Se la variabile è la costante relativa al numero di esecuzioni del ciclo (non modificata durante il ciclo), è possibile utilizzarla come costante dello schema.

- L'utilizzo di questo schema rende difficile l'utilizzo dell' offuscamento attraverso i basic block, e la probabilità che una variabile resti non modificata diminuisce.

3) Bit più significativo

Trasformiamo un piccolo intero in un vettore di bit, dove l'assegnamento del bit più significativo rappresenta il valore dell'intero.

- Il vantaggio di questo metodo è che le operazioni più comuni possono essere effettuate mentre il valore è ancora nella forma trasformata.

4) Utilizzo di offsets

Gli interi possono essere rappresentati con un offset costante.

- Per esempio l'intero 5 può essere rappresentato come 11 con un offset di 6.

5) Rappresentazione con modulo

Questa tecnica offusca un variabile intera ***x***, trasformandola nella coppia (X, k) così rappresentata:

$(a^k * x \bmod p, k)$

- Dove *a* è il parametro di offuscamento, *x* è l'intero di partenza, *p* è un numero primo e *k* è l'esponente a cui elevare *a*.

5) Rappresentazione con modulo(2)

Presi due interi offuscati *X* e *Y* , (X,k) e (Y,l) dove *k* ed *l* sono i rispettivi esponenti a cui elevare il parametro *a.*

- La **moltiplicazione** è eseguita nel modo seguente:

  $(X, k) * (Y, l) = (X*Y, k+l)$

  - Quindi per effettuare la moltiplicazione bisogna moltiplicare i due interi offuscati e sommare i due esponenti.

5) Rappresentazione con modulo(3)

- La **somma** di due variabili offuscate *W* e *Y* bisogna procedere come segue:

  $(W, k) + (Y, l) = (W + Y *(k-l), k)$

  - Il parametro a deve essere lo stesso per entrambi gli operandi.

6) Rotazione

Questa tecnica si basa sull'idea di invertire le posizioni di un certo numero di bit all'interno del registro che contiene l'intero.

Esempio:
- con un registro di 16 bit, la rappresentazione viene modificata da big-endian a little-endian.

- un vantaggio di questo metodo è che la maggior parte delle operazioni possono essere eseguite mentre il valore è offuscato.

6) Rotazione(2)

- L'**addizione di una variabile offuscata e una costante** richiede la rotazione dei corrispondenti bit della costante.

- Nell'**addizione di due variabili offuscate** l'unica cosa da manipolare è il bit di riporto.

- La **moltiplicazione** deve essere trasformata in sequenza di shift/aggiungi (in modo analogo la **divisione**).

7) XN mod 1

Un'interessante trasformazione dovuta a ***L. Washington*** è rappresentare un intero *x* nel seguente modo:

$X = a * x \bmod 1$

- dove '*a*' è un *numero irrazionale* fissato e rappresenta il parametro di offuscamento e *X* è un *numero reale* compreso nell'intervallo **[0 , 1)**,

- (*a* è scelto in modo da evitare che corrisponda ad un multiplo di X).

7) XN mod 1(2)

Esempio:

Dato *x* = 13.5, allora 12 e 14 corrispondono entrambi a 0 nell'intervallo [0, 1).

Per evitarlo si sceglie un valore per *a* che include una piccola funzione, es. *a = 13.5 + ½^30*.

7) XN mod 1(3)

- L'**addizione** offuscata è semplice ed offre un buon livello di offuscamento se per entrambi gli operandi utilizzando lo stesso parametro di offuscamento.
- Per la **moltiplicazione** offuscata invece esistono due metodi, i quali hanno entrambi dei limiti.

7) XN mod 1(4)

Per esempio dati come operandi:

a*x mod 1* e *a*y mod 1

- Il **primo metodo:**
  - consiste nel moltiplicare entrambi gli operandi per a^{-1} *mod 1* per ottenere *x* e *y*, successivamente moltiplicando *x* e *y* si ottiene *x * y mod 1* e come ultimo passo si converte il risultato nella forma *a*x*y mod 1*.

7) XN mod 1(5)

- Il **secondo metodo:**
  - consiste nel moltiplicare *ax mod 1* e *ay mod 1* per ottenere (a^2) *x*y mod 1* e successivamente moltiplicare (a^2) *x*y per* a^{-1} *mod 1*per ottenere *a * x * y mod 1*.
- Tuttavia in entrambi i metodi il parametro di offuscamento '*a*' è esposto durante ogni moltiplicazione.

7) XN mod 1(6)

- Per evitare di esporre relazioni tra copie dello stesso valore offuscato con questa tecnica, sarebbero necessari più parametri di offuscamento.
  - Tuttavia usando differenti valori per il parametro di offuscamento diminuisce il livello di offuscamento.

8) Divisione delle locazioni di memoria

l metodo consiste nel dividere la locazione di memoria che contiene il valore della variabile, in più locazioni.

Per deoffuscare la variabile, bisogna conoscere quali sono i bit che in ciascuna locazione la determinano:

- tali bit sono definiti *significativi*.

8) Divisione delle locazioni di memoria(2)

- Il punto focale di questo metodo è: conoscere quali sono i bit significativi.
- Sulle variabili memorizzate in locazioni di memoria divise si possono eseguire molte operazioni.
- Tuttavia introduce una quantità di codice sostanziale e le performance ne risentono.

9) Array chasing

Permette di offuscare i dati di tipo intero.

- Per realizzare il metodo, si utilizzano un array ed un ciclo *for()* supplementari.

9) Array chasing(2)

Esempio:
offuscare i parametri passati ad un funzione bar(foo(7), 2)

- Dato il seguente array:

```
int baz[10] = { 4, 1, 7, 9, 2, 3, 5, 2, 5, 0 };
```

- Invece di usare direttamente *foo(7),* viene utilizzato il seguente ciclo:

```
for (baz[baz[4]] = 0; baz[baz[4]] < 5; ++baz[baz[4]])
        bar(foo(baz[baz[4]]),baz[baz[4]]);
```


9) Array chasing(3)

- Dove anzichè utilizzare la variabile locale i, viene utilizzato *foo[7].*
- inoltre *baz[4]* vale 2 e *baz[2]* è la variabile effettiva, cioè il valore 7 che verrà usato come parametro per la chiamata della funzione foo.

10) Utilizzo di tabelle per sostituire gli interi

- Gli interi con valori particolarmente piccoli, possono essere sostituiti con tabelle.
- Gli operatori unari sono rimpiazzati con array monodimensionali.
- Le operazioni binarie sono rappresentate con tabelle bidimensionali.

10) Utilizzo di tabelle per sostituire gli interi(2)

- Per esempio, nel ciclo

```
for (int i = 0; i < 5;++i)
{
  bar(foo(i),i);
}
```

l'intero ha valori tra 0 e 5. Le operazioni implementate sono: incremento di 1 e <.

10) Utilizzo di tabelle per sostituire gli interi(3)

- La seguente tabella può essere generata a compile time:

Value	Index
0	3
1	4
2	2
3	0
4	1
5	5

10) Utilizzo di tabelle per sostituire gli interi(4)

- Prendendo questi valori iniziali la tabella ***incremento*** avrà i seguenti valori:

1	5	0	4	2	NA

cioè, 0 (indice = 3) incrementato vale 1 (indice = 4). Non è rilevante quale valore andrà nell’indice 5, perché questo valore è al di fuori dell’intervallo usato dal programma.

10) Utilizzo di tabelle per sostituire gli interi(5)

- La tabella dell’**operazione di <** sarà:

0	0	1	1	1	0
1	0	1	1	1	0
0	0	0	1	1	0
0	0	0	0	0	0
0	0	0	1	0	0
1	1	1	1	1	0

10) Utilizzo di tabelle per sostituire gli interi(6)

Il ciclo diventerà qualcosa di simile a quanto segue:

```
iconst_3           # push della costante 0 (index 3)
istore_1           # pop in var 1 ("i")
goto loop

start: iload_1     # push "i"
invokestatic foo   # chiamata a "foo", lasciando retval nello stack
iload_1            # push "i"
invokestatic bar   # chiamata a "bar", lasciando retval nello stack
aload_2            # caricare un riferimento alla tabella inc
iload_1            # push “i”
iaload             # accesso all’ array
istore_1           # pop “i”

loop:
aload_3            # caricare un riferimento alla tabella minore-uguale
iload_1            # push “i”
aaload             # prendere l’i_ma colonna della tabella
iconst_5           # push della constante 5 (index 5)
iaload             # prendere la quinta entry della colonna
ifeq start         # salto a start unless LessThanTable[i,5]
```


10) Utilizzo di tabelle per sostituire gli interi(7)

- Senza conoscere i valori dell’array è impossibile determinare la taglia del ciclo.
- Se l’array è inizializzato dinamicamente dal programma (solo i valori 0 e 5 devono essere costanti).
- Allora il malintenzionato, per effettuare una buona analisi permetterà al programma di funzionare finchè non si sarà inizializzato l’array e solo dopo effettuerà l’analisi della porzione di codice.

10) Utilizzo di tabelle per sostituire gli interi(8)

- Sfortunatamente per questo schema, gli array inizializzati sono relativamente semplici da analizzare.
  - In questo semplice esempio la tabella di “<” è una tavola booleana (valori 0 o 1).
- Se un campo ha tutti 0 e nessun campo ha tutti 1 si tratta di una tabella “<”.
  - Il campo con tutti 0 sarà il valore massimo.