Code Obfuscation

L'offuscamento del codice è stato introdotto per la prima volta da Fred Cohen, allo scopo di **diversificare** i programmi per difendersi dagli attacchi automatici (e quindi replicabili) ai sistemi operativi. Senza, però, una protezione fisica è improbabile che si riesca a proteggere completamente un sistema → Fred Cohen sosteneva che: qualsiasi schema di protezione diverso da quello fisico dipende dal funzionamento di una macchina a stati finiti e, in ultima analisi, ogni macchina a stati finiti può essere esaminata e modificata a piacimento,
con tempo e sforzi sufficienti. Il meglio che possiamo fare, quindi, è
ritardare l'attacco aumentando la complessità delle alterazioni desiderate (**sicurezza attraverso l'offuscamento**). Diamo, adesso, una prima definizione di offuscamento:
L'offuscamento è una **trasformazione T di programmi**, che trasforma un programma "p" in un programma "p'", che deve avere due caratteristiche:

1. "p" e "p'" devono avere lo **stesso comportamento osservazionale**, ovverosia l'utente, il quale utilizza sia "p" che "p'", non deve percepire alcuna differenza tra i due programmi → quindi, noi vogliamo programmi estensionalmente equivalenti ed intenzionalmente diversi, in particolare che "p'" sia più complesso di "p";
2. "p" e "p'" devono **terminare producendo gli stessi risultati** → da sottolineare il fatto, che se "p" non termina, allora "p'" può produrre qualsiasi risultato.

L'offuscamento, inoltre, viene valutato secondo tre misure:

1. **Potenza** → misura quanto è più difficile capire il codice offuscato rispetto al codice originale → la prima definizione di Fred Cohen, prova a dare anche una definizione più formale di potenza. In particolar modo, si suppone di avere una misura E(P), che misura la complessità di un programma P e poi facendo il rapporto $Tpot(P) = E(P')/E(P) - 1$ si ottiene la stima della potenza della trasformazione rispetto al programma P. Chiaramente il "punto debole" di questa definizione, è capire come valutare la complessità di un programma. Per colmare questo problema, Cohen ha introdotto alcune misure di Software Complexity, come ad esempio:
    a. la lunghezza del programma → con l'aumentare del numero di operazione e di operandi, la complessità del programma aumenta;
    b. data flow complexity;

c. data structure complexity;

d. nesting complexity → quindi più strutture annidate vi sono all’interno del programma, maggiore sarà la sua complessità.

Queste misure riguardano le **proprietà intenzionali** del programma, ma chiaramente queste misure non catturano la complessità del programma.

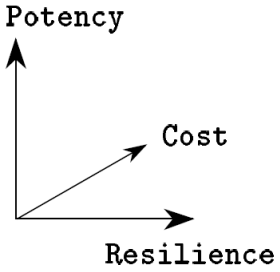
> 💡 Il codice, visto come dato, manifesta due caratteristiche fondamentali: **proprietà intenzionali** ed **estensionali**.
> Le prime riguardano la **sintassi** del codice, quindi com’è scritto il codice e nascondono la sua origine e le trasformazione, come l'offuscamento, che questo ha subito per nascondersi. Le seconde, invece, riguardano come agisce il codice e come si manifesta l'azione dell'attacco.
> Entrambe queste caratteristiche devono essere considerate quando si vuole tracciare la provenienza di un attacco e scoprirne la genesi.

2. **Resistenza** → misura quanto bene una trasformazione regge sotto un attacco automatico di un deobfuscatore. In base alle misure di complessità del software, che abbiamo visto, sembra facile aumentare la potenza, per esempio aumentando il numero di istruzioni if. Queste trasformazioni, però, sono **inutili**, perché possono essere facilmente annullate. La resistenza di una trasformazione T è data dalla combinazione di:

   a. **Sforzo del programmatore (Programmer effort)** → ovverosia la quantità di tempo necessaria per costruire un deobfuscatore automatico (il quale è in grado di eliminare il rumore introdotto nel codice), in grado di ridurre la potenza di T;

   b. **Sforzo del deobfuscatore (Deobfuscator effort)** → ovverosia il tempo di esecuzione e lo spazio richiesto da tale deobfuscatore automatico per ridurre efficacemente la potenza di T.

3. **Costo** → misura il tempo di esecuzione in termini di tempo/spazio, che una trasformazione comporta su un'applicazione applicazione offuscata, ovvero misura il degrado delle performance. Quindi, il costo di una trasformazione offuscante è la penalizzazione in termini di tempo/spazio di esecuzione del programma offuscato rispetto a quello originale.



Oltre a queste tre misure, vi è da introdurre il concetto della **stealthiness**, ovverosia della **furtività/invisibilità** → Sebbene una trasformazione resistente possa non essere soggetta ad attacchi da parte di deobfuscatori automatici, può essere comunque soggetta ad attacchi umani. In particolare, se una trasformazione introduce del nuovo codice, che differisce ampiamente da quello presente nel programma originale, sarà facile da individuare per un attaccante e quindi sarà facile fare reverse engineering. La stealthiness è una nozione sensibile al contesto, in quanto il codice può essere invisibile in un programma, ma estremamente poco invisibile in un altro → la stealthiness, quindi, fa sì che per un attaccante umano non sia evidente localizzare l'offuscamento o capire che effettivamente è avvenuto un offuscamento.

Quando sviluppiamo e valutiamo un offuscamento, dobbiamo prendere in considerazione:

- Qual è l'obiettivo della protezione;
- Chi è l'attaccante e quali azioni può svolgere;
- Definire la tecnica di offuscamento;
- Testare la qualità dell'offuscamento proposto.

In generale, vi sono tre classi di offuscamento, **classificate in base al tipo di informazione che vanno a modificare**:

- **Layout obfuscation** → vanno ad agire sul layout, andando a modificare o rimuovere informazioni utili dal codice, senza però influire sulle istruzioni reali. In questo senso, quindi, si può andare a:
  - eliminare i commenti;
  - rinominare le variabili, andando ad utilizzare nuovi nomi, i quali (ossia i nomi) utilizzano numeri, caratteri non-stampabili oppure caratteri invisibili. I nomi, inoltre, molto spesso vengono sovrascritti, ovverosia vengono utilizzati gli stessi nomi per variabili, che hanno scopi completamenti differenti, in modo

tale da confondere l’attaccante → questo ci fa capire, che il layout obfuscation ha:

- una bassa potenza contro i tool di analisi, poichè c’è poco contenuto semantico nella formattazione;
- è potente contro gli attacchi umani.

Il layout obfuscation è utilizzato in molti offuscatori commerciali e si tratta di una **trasformazione unidirezionale (trasformazione one-way)**, in quanto la formattazione originale non può più essere recuperata e quindi è una tecnica molto resistente e per quanto riguarda il costo è gratuito.

- **Data obfuscation** → sicuramente un passaggio importante per comprendere il codice e il suo funzionamento, è riuscire a capire come manipolare i dati utilizzati. In questo senso, la Data obfuscation prende la codifica in chiaro del dato (ovvero quella che il programmatore scrive) e la sostituisce con un altro modo di rappresentare qual dato → da notare, che è diverso dalla cifratura, in quanto se il dato viene cifrato, quest’ultimo non può essere utilizzato nel codice per svolgere operazioni, bensì si tratta di una codifica “meno naturale” del dato. Il concetto fondamentale della Data obfuscation è che abbiamo due trasformazioni:
  - encoding;
  - decoding.

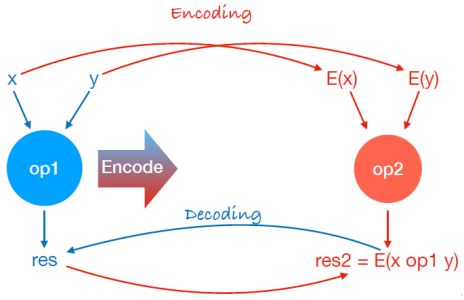
  che trasformano un tipo di dato in un altro tipo di dato e questo comporta, che tutte le operazioni, che avevo sul tipo di dato originale, dovranno essere reinterpretate sul nuovo tipo di dato e questo lo si può fare in due modi:
  - andando a fare la delle operazioni sul tipo di dato originale e poi la codifica sul nuovo tipo di dato;
  - oppure cercando di interpretare direttamente le operazioni sul nuovo tipo di dato.

  Quindi, abbiamo la rappresentazione originale del dato e la vogliamo trasformare in una rappresentazione meno chiara, che in un qualche modo va a rompere le astrazioni logiche oppure va a mascherare dei valori. Un esempio classico di Data obfuscation è l’utilizzo dello **XOR**, in cui abbiamo che l’encoding e il decoding sono la stessa operazioni e possono dipendere da un parametro “p” → questo parametro, quindi, diventa un’informazione sensibile e quindi deve essere nascosto. Per nasconderlo vi sono vari modi, per esempio:

- possiamo fare in modo, che il valore di “p” venga calcolato dinamicamente e che quindi non sia una costante nel codice;
- potrebbe essere una proprietà invariante in un qualche punto del programma, per cui viene svelata solo durante l’esecuzione.

Ci immaginiamo, quindi, di avere i nostri valori “x” e “y”, che vengono combinati attraverso una qualche operazione. Ne facciamo l’encoding e come abbiamo detto prima, possiamo:

- deoffuscare i valori prima dell’operazione e poi ri-offuscarli successivamente → i valori deoffuscati, quindi, sono presenti in memoria fino a che non vengono nuovamente offuscati;
- oppure possiamo andare a reinterpretare l’operazione sui valori codificati → idea simile all’Homomorphic Encryption, la quale permette di svolgere operazioni su dati cifrati. Diciamo che è simile, in quanto nel nostro caso i dati non sono cifrati, bensì sono codificati e quindi posso reinterpretare le operazioni sul nuovo tipo di dato.



Vi sono, però, alcuni problemi:

- la Data Encoding modifica il tange delle variabili e quindi bisogna fare attenzione a non andare in **overflow**;
- deve essere gestita la **codifica dei diversi valori** → quindi se ho valori, che vengono codificati in modi diversi e poi questi vengono combinati, questi devono essere gestiti, in modo tale da preservare la semantica del codice → da notare, che per preservare la semantica del codice, ci deve essere a monte una fase di analisi prima di offuscare il codice, in cui si va a capire:
  - quali sono le variabili che dipendono dalla variabile di cui vogliamo fare l’encoding;
  - come propagare l’encoding anche alle variabili dipendenti.

Per fare la codifica dei **numeri interi**, vi sono diverse possibilità:

- trucchi di teoria dei numeri → ad esempio, un numero intero y può essere rappresentato come (p * N + y) dove N è il prodotto di due numero primi vicini, mentre p è un numero casuale. Il deoffuscamento consiste, quindi, nell'andare ad eliminare (p * N) riducendo il modulo N;

Per fare, invece, la codifica dei valori booleani, essendo solamente due si possono utilizzare più valori per codificare il vero e più valori per codificare il falso. Per esempio:

- True → tutti i numeri divisibili per 2;
- False → tutti i numeri divisibili per 3.

Se quindi andiamo ogni volta ad utilizzare valori diversi, andiamo certamente a confondere l'attaccante.

Un altro modo di codificare i booleani è quello di aggregare più variabili booleane, ovvero: il valore booleano "v" viene codificato da due variabili booleane "p" e "q" e la regola può essere, ad esempio, la seguente:

v	p	q
T	F	F
F	F	T
F	T	F
T	T	T

naturalmente, dovrò aggiornare tutte le operazioni → possiamo, allora, capire che la potenza, la resistenza e il costo della suddivisione delle variabili crescono con il numero di variabili in cui viene suddivisa la variabile originale. La resistenza può essere ulteriormente migliorata selezionando la codifica a tempo di esecuzione e non a tempo di compilazione (cosa che le renderebbe soggette ad analisi statica).

Posso anche decidere di fare l'encoding delle stringhe, che potrebbero attirare l'attenzione dell'attaccante, andando a definire tali stringhe non all'interno del codice, bensì generarle a tempo di compilazione andando, ad esempio, a definire una funzione, che quando viene eseguita mi restituisce le stringhe d'interesse → il concetto è che **un oggetto statico (come appunto la stringa) viene sostituito con un oggetto calcolato a tempo di esecuzione**.

Infine, vi è una famiglia di Data Obfuscation che hanno a che fare con gli array (e le matrici), ovvero l'Array Obfuscation, che vanno a rompere l'astrazione logica della struttura dati. Vi sono due categorie di Array Obfuscation:

1. **Riordinare** gli elementi dell'array → questo rompe l'ordine lineare che un attaccante si aspetterebbe → la potenza viene data dal fatto, che viene utilizzato un ordine innaturale degli elementi;
2. **Ristrutturare** l'array suddividendolo in parti → consiste nell'andare a dividere l'array in diverse parti, oppure unire l'array originale con altri array non correlati oppure modificando la sua dimensionale, quindi trasformandolo in matrice, per nascondere all'attaccante la struttura, che il programmatore intendeva originariamente.

In particolare, quando parliamo di Array Obfuscation, dobbiamo parlare anche:

- **Aggregazione** → essa può essere fatta anche sulle variabili ed in particolar modo, con l'aggregazione andiamo a prendere due variabili e memorizzarle all'interno di un'unica variabile, in cui so che:
  - nei primi tot bit vi è memorizzato il valore della prima variabile;
  - nei secondi tot bit vi è memorizzato il valore della seconda variabile.

  Naturalmente dovrò aggiustare tutte le operazioni, in quanto quando faccio, ad esempio, la somma o la moltiplicazione dovrò tenere in considerazione se mi trovo nella parte alta (e quindi dove è stato memorizzato il valore della prima variabile) oppure nella parte bassa (e quindi dove è stato memorizzato il valore della seconda variabile) della variabile → chiaramente questo crea confusione all'attaccante, in quanto il ruolo delle due variabili iniziali non è più ben chiaro, bensì si è unito.

- **Control obfuscation** → vanno a nascondere il flusso, quindi vanno ad agire sul flusso di controllo del codice.

In un paper del 2016 sono state elencate le tecniche di analisi del codice e gli assets interessanti per l'attaccante, andando così a realizzare una tabella per mostrare l'efficacia della tecnica di analisi rispetto all'asset dell'attaccante. In particolar modo, le tecniche di analisi del codice erano le seguenti:

- **Pattern matching** → l'analisi più semplice e veloce. È un'analisi sintattica per l'identificazione di sequenze statiche di istruzioni, espressioni regolari o classificatore basato sull'apprendimento automatico;

- **Analisi statica automatizzata** → ragiona staticamente sulla semantica del programma. Le forme più semplici includono i disassemblatori che interpretano i branch. Spesso viene utilizzata per ricostruire informazioni di alto livello sui programmi;
- **Analisi dinamica automatizzata** → è in grado di ragionare in modo molto preciso sul comportamento del programma lungo le tracce osservate;
- **Analisi assistita dall'uomo** → è l'"analisi" più capace. In questo processo di reverse engineering l'analista si propone di comprendere la struttura e il comportamento del programma con l'ausilio di una serie di strumenti.

Gli assets, invece, erano i seguenti:

- **Individuazione della posizione dei dati** → l'analista desidera recuperare alcuni dati incorporati nel programma (chiave crittografica, chiavi di licenza, certificati, credenziali...);
- **Individuazione della posizione della funzionalità del programma** → l'analista vuole identificare il punto di ingresso di una particolare funzione all'interno del programma offuscato;
- **Estrazione di frammenti di codice** → l'analista vuole estrarre una porzione di codice che includa tutte le possibili dipendenze di codice, comprese tutte le possibili dipendenze, che implementa una particolare funzionalità da un programma offuscato (codice dei concorrenti, routine di decrittazione per rompere il DRM, alterare la funzionalità a runtime);
- **Comprensione del programma** → l'analisi mira a comprendere appieno il programma offuscato (deoffuscamento, trovare vulnerabilità, furto di proprietà intellettuale).

La tabella che è stata costruita è la seguente:

Table II. Analysis of the strength of code obfuscation classes in different analysis scenarios (PM = Pattern Matching LD = Locating Data, LC = Locating Code, EC = Extracting Code, UC = Understanding Code).

	PM		Autom. Static				Autom. Dynamic				Human Assisted			
Name	**LD**	**LC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**
Data obfuscation														
Reordering data			✓				✓							
Changing encodings	✓						✓							
Converting static data to procedures	✓						✓				✓			

Legend		obfuscation breaks analysis fundamentally
		obfuscation is not unbreakable, but makes analysis more expensive
		obfuscation only results in minor increases of costs for analysis
	✓	A checkmark indicates that the rating is supported by results in the literature
		Scenarios without a checkmark were classified based on theoretical evaluation

Nel paper, inoltre, viene fatta una distinzione per quanto riguarda la Code Obfuscation tra:

- **Riscrittura statica del codice (Static code rewriting)** → una riscrittura statica è simile ad un compilatore ottimizzante, in quanto modifica il codice del programma durante l'offuscamento, ma permette al suo output di essere eseguito senza ulteriori modifiche in fase di esecuzione. Tutte le tecniche di offuscamento dei dati descritte in precedenza rientrano nella categoria della riscrittura statica del codice (quindi: sostituzione delle istruzioni, predicati opachi, riordino, scambiare i nomi, **control flow obfuscation**);
- **Riscrittura dinamica del codice (Dynamic code rewriting**) → la caratteristica principale degli schemi di offuscamento del codice di questa categoria è che il codice eseguito si differenzia dal codice staticamente visibile nell'eseguibile. (Quindi comprende: crittografia, modifiche dinamiche del codice, offuscamento del codice assistito dall'HW (binding HW-SW che fa dipendere l'esecuzione del software da un certo tipo di
  l'esecuzione del software dipende da qualche token HW. Senza
  token l'analisi del software fallirà).

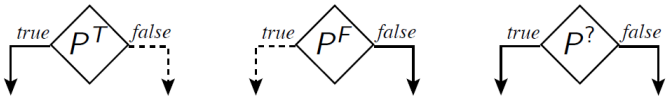
Control Flow Obfuscation

Quando si analizza il codice, moltissime **analisi statiche** funzionano su una rappresentazione a grafo del codice, quindi estraggono una rappresentazione (a grafo) del codice, sulla quale poi si fanno dei ragionamenti. In queste rappresentazioni, quindi, si hanno:

- dei **nodi** → i quali contengono le istruzioni;
- degli **archi** → mi indicano che c'è:

- una **dipendenza di controllo** → ovvero che l'istruzione del nodo sorgente viene eseguita prima dell'istruzione del nodo destinazione;
- oppure una **dipendenza di flusso** → per cui il valore della variabile "x" assunto in un nodo, va riutilizzato in un'altra istruzione.

Le analisi funzionano visitando e seguendo questi archi → confondere il controllo, quindi, significa confondere l'attaccante su quale istruzione verrà eseguita successivamente oppure confonderlo su quali saranno le possibili istruzioni successive ad una certa istruzione (ricordando che stiamo parlando id un'analisi statica). Il mattoncino/blocco base che viene utilizzato moltissimo è il **predicato opaco** → i predicati opachi sono dei predicati, quindi delle espressioni booleane che vengono valutate a vero oppure a falso, che però valutano **sempre** a vero oppure valutano **sempre** a falso (quindi i predicati opachi valutano sempre a vero oppure sempre a falso) → quindi, possiamo dire che i predicati opachi sono dei "finti predicati", in quanto nella realtà sono delle costanti booleane, dato che potrebbero essere sostituiti con vero oppure falso.



L'analizzatore statico, quindi, vede un predicato e mettiamo il caso che tale predicato controlla la guardia di un if. Il successore della guarda dell'if può essere:

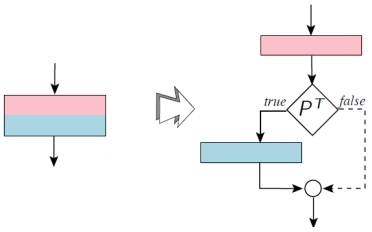
- o la prima istruzione del ramo true;
- oppure la prima istruzione del ramo false.

Con il predicato PT eseguo solo il ramo true, con il predicato PF eseguo solamente il ramo false, mentre per quanto riguarda P?, ovvero i predicati non opachi, vedremo in un secondo momento → i predicati opachi, quindi, sono delle condizioni booleane, che in realtà sono delle costanti → questo però lo sa solamente chi offusca, mentre l'attaccante non lo sa.

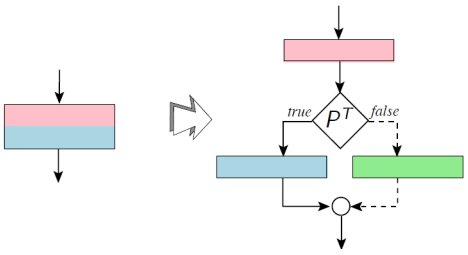
Essendo un mattoncino, i predicati opachi possono essere utilizzati in diversi modi per confondere l'analizzatore statico ed in questo senso possiamo:

- Prendere un blocco del Control Flow Graph, dove il Control Flow Graph consiste nell'andare a suddividere il programma in blocchi e ogni blocco è una lista di operazioni sequenziali (quindi che vengono eseguite tutte) e poi ho una diramazione ogni volta che ho una condizione, quindi ogni volta che ho un ciclo o un if → ogni volta che entro in un blocco, quindi eseguo tutte le istruzioni, per

poi uscire dal blocco e saltare in un altro punto del codice. I predicati opachi nei Control Flow Graph, vengono utilizzati per fare **block splitting**, ovvero:



prendo un blocco, il quale contiene molteplici istruzioni, lo divido in due sotto-blocchi e in mezzo vi metto un predicato opaco → in questo modo, l'attaccante vede che si se verifica una certa condizione, allora viene eseguito il sotto-blocco, altrimenti non viene fatto nulla → chiaramente nel ramo false posso anche inserire delle istruzioni, per esempio delle istruzioni che sono completamente le opposte di quelle nel ramo vero, oppure molte dipendenze con il codice sopra, che vanno ulteriormente a confondere l'attaccante.

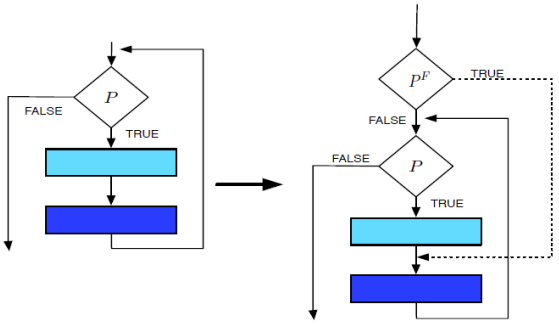


Chiaramente la semantica del codice rimane invariata, in quanto il ramo falso non viene mai eseguito e non si va nemmeno ad appesantire il codice, dato che il ramo falso non viene mai eseguito.

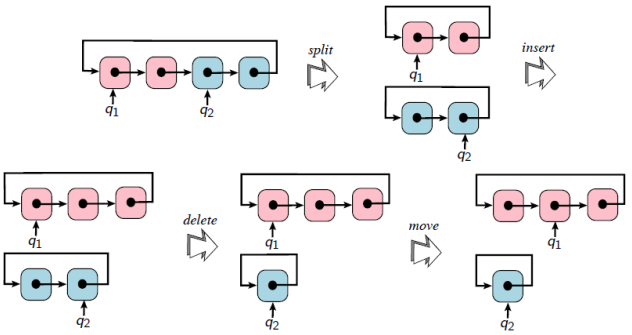
→ i **predicati non opachi** consistono nell'andare a prendere un predicato che valuta sia vero sia falso ed andare ad inserire nei due rami due codici equivalenti, che fanno la stessa cosa in modo diverso (producono cioè lo stesso output, ma in modo diverso) → faccio sembrare, quindi, all'attaccante che vi sia un apparente scelta.

I predicati opachi vengono utilizzati anche nel seguente contesto: Quando facciamo il Control-flow graph di un programma, quello che tipicamente non può mai accadere, è che vi sia un'istruzione di jump nel mezzo del corpo di un loop, perchè poi non sono in grado di andare a scrivere il programma corrispondente di tale Control-flow graph → questi Control-flow graph prendono il nome di **Control-flow graph riducibili** e la cosa che a noi interessa, è che tutti i tool di analisi statica assumo che i Control-flow graph siano di questo tipo (ovvero riducibili), dato che assumono che esso derivi da un programma. Se però, per un qualche motivo, i tool

di analisi trovano un jump nel mezzo del corpo di un loop, vanno a **duplicare il codice**. Se nell'offuscamento, quindi, andiamo ad introdurre dei predicati opachi che vanno a rendere **irriducibile** il Control-flow graph, andiamo ad aumentare (ad esplodere) in maniera esponenziale la dimensione del Control-flow graph per i tool di analisi statica → i **predicati opachi, quindi, vanno a trasformare un Control-flow graph riducibile in un Control-flow graph irriducibile, andando ad inserire un jump nel mezzo del corpo del loop**.



In generale, un attaccante per deoffuscare un programma in cui vi sono dei predicati opachi, dovrà innanzitutto accorgersi della presenza dell'opacità, ovvero dovrà rendersi conto, che dei predicati in realtà sono delle costanti e quindi dei falsi predicati. Chiaramente, il nostro obiettivo è fare in modo, che per gli attaccanti capire che i predicati siano di tipo opaco, sia possibile solamente risolvendo un problema della classe NP, ovvero un problema molto complicato → quindi, se io che offusco ho un problema di analisi complicato e di cui ovviamente conosco la soluzione (dato che sono io quello che offusco), mentre l'attaccante per scoprire la soluzione deve fare un'analisi molto complessa, comporta che quest'ultimo (ovvero l'attaccante) per scoprire che i predicati sono di tipo opaco, deve risolvere un problema complesso → un'analisi molto complicata è la **Pointer Analysis**, ovvero è il processo per determinare se, in un determinato punto del programma, due variabili possono riferirsi alla stessa locazione di memoria. L'idea alla base è, che io offuscatore, vado ad inserire due puntatori e a questi gli cambio valore e/o li sposto, ovvero vado a compiere diverse azioni su questi puntatori, mantenendo però delle invarianti che io offuscatore conoscono, ma che l'attaccante non conosce. Queste invarianti, allora, diventano dei predicati opachi che posso utilizzare e che l'attaccante, per scogliere, deve risolvere un problema di analisi molto complicato. Vediamo un esempio per capire meglio:



Abbiamo le seguenti invarianti:

- G1 (rosa) e G2 (azzurra) sono due linked list circolari;
- q1 punta sempre ad un elemento della lista Q1 e q2 punta sempre ad un elemento della lista Q2;
- q1 sia sempre diverso da q2.

Nelle due liste inserisco due puntatori q1 e q2 e su queste liste vado a compiere diverse azioni, come per esempio:

- elimino elementi;
- aggiungo elementi;
- divido le due liste;
- sposto degli elementi dalle liste.

ed i due puntatori li faccio muovere all'interno delle liste, mantenendo però sempre il vincolo, che q1 punti sempre ad un elemento della lista Q1 e q2 punti sempre ad un elemento della lista Q2 e che q1 sia sempre diverso da q2 → tali invarianti, allora, le utilizzo come segreti e per l'attaccante statico è difficile scoprirli e comprenderli.

Riassumendo, quindi, possiamo dire che i predicati opachi sono di fatto delle invarianti, che valgono in un certo punto del programma, e il fatto di essere invarianti è qualcosa di conosciuto all'offuscatore e non conosciuto e di difficile comprensione per l'attaccante, il quale quindi non si accorge (o almeno questo è il nostro obiettivo) che si tratta di invarianti. Come invarianti si possono utilizzare:
- proprietà dei numeri → poco resistenti, in quanto sono note;
- quello che, invece, viene fatto è di utilizzare un'invariante conosciuta all'offuscatore e devo fare in modo, che per l'attaccante conoscere l'invariante implichi risolvere un problema NP.

Un altro esempio di utilizzo di predicati opachi è il seguente, in cui viene definito un array di numeri, in cui i numeri memorizzati nelle diverse celle soddisfano determinate proprietà:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
36	58	1	46	23	5	16	65	2	41	2	7	1	37	0	11	16	2

Invariants:

1. every third cell (in pink), starting will cell 0, is $\equiv 1 \bmod 5$;
2. cells 2 and 5 (green) hold the values 1 and 5, respectively;
3. every third cell (in blue), starting will cell 1, is $\equiv 2 \bmod 7$;
4. cells 8 and 11 (yellow) hold the values 2 and 7, respectively.

You can update a pink element as often as you want, with any value you want, as long as you ensure that the value is always $\equiv 1 \bmod 5$!

ovvero che facendo valore della cella modulo 5, il risultato è sempre 1.

L'idea alla base di questo esempio, è che per l'attacante sembra che i valori all'interno delle celle cambino continuamente, ma in realtà il valore cambia in base a delle invariante conosciute solamente dall'offuscatore.

Sorge a questo punto spontanea la domanda: "**Come si fa a riconoscere un predicato opaco?**" Per riconoscere un predicato opaco si dovrebbe innanzitutto prendere un predicato e:

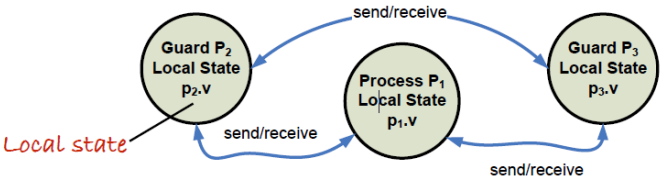
- calcolarsi il backward slice rispetto a dove viene utilizzato il predicato nel branch;
- andare a guardare tutti i possibili valori delle variabili che il predicato può prendere in input;

- con un approccio Brute force andiamo a controllare se effettivamente il risultato è sempre vero oppure sempre falso.

Per completezza, diciamo anche che sono state proposte delle variante ai predicati opachi, tra cui i **predicati opachi dinamici** → i predicati opachi sono vulnerabili all'analisi dinamica, in quanto se un attaccante è in grado di monitorare l'Heap e i registri durante l'esecuzione, allora potrebbe capire rapidamente che un predicato è sempre valutato come vero. I predicati opachi dinamici, sono una famiglia di predicati opachi, che valutano tutti lo stesso risultato in una determinata esecuzione, ma in esecuzioni diverse possono
valutare risultati diversi → quindi l'opacità, ovvero la proprietà invariante, si riferisce ad un insieme di predicati e non più solamente al singolo predicato. Questo comporta che è difficile determinare quali predicati sono correlati tra di loro. Vediamo la seguente immagine per capire meglio:

Predicate	Run 1	Run 2	Run 3	Run 4	Run 5
Pred 1	T	T	F	T	F
Pred 2	T	T	F	T	F
Pred 3	T	T	F	T	F
Pred 4	T	T	F	T	F
Pred 5	T	T	F	T	F

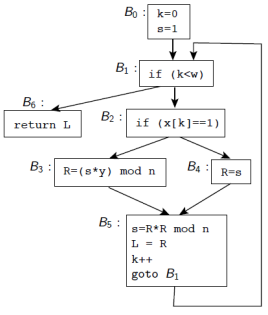
Un'altra variante dei predicati opachi è quella di definire i predicati nei **sistemi concorrenti** e quindi ci immaginiamo non un singolo programma, bensì ci immaginiamo più programmi eseguiti insieme e ci si chiede, in questo caso, come si potrebbe distribuire l'opacità. Nei sistemi concorrenti è importante capire quale processo è in esecuzione e di conseguenza in che ordine vengono eseguite le istruzioni. Programmi concorrenti mal progettati possono incorrere in problemi di race condition, ovvero più processi hanno accesso ad una stessa risorsa/file/oggetto senza prendere il lock. Il predicato opaco distribuito è un predicato opaco, il cui valore dipende dagli stati locali di più processi distribuiti nel sistema distribuito. Consideriamo, allora, un sistema distribuito composto da un insieme di processi intercomunicanti P1,P2,P3, ecc. Per offuscare il flusso di controllo di P1 selezioniamo un certo numero di **processi di guardia** e l'idea alla base di questa variante è di distribuire gli stati locali che si formano nella costruzione del predicato opaco distribuito in P1 tra le guardie, e incorporare un modello di comunicazione sotto forma di chiamate di invio/ricezione, che aggiornano i rispettivi processi di stati locali a valori precedentemente noti.



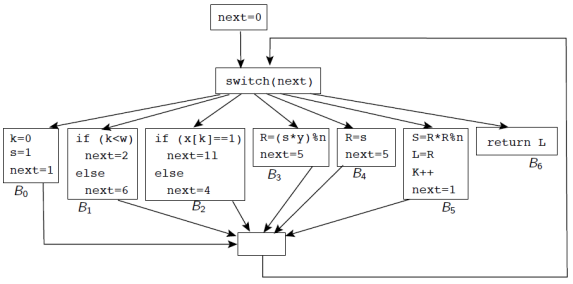
Control Flow Flattening

Dato il Control-flow graph di un programma, il quale ricordiamo è una rappresentazione, utilizzando la notazione grafica, di tutti i percorsi che potrebbero essere attraversati attraverso un programma durante la sua esecuzione e di conseguenza, in un certo modo, esprime l'ordine in cui devono essere eseguite le istruzioni di un programma. A questo punto ci chiediamo: "**Qual è la massima confusione, che posso aggiungere in un Control-flow graph?**" La massima confusione si raggiunge, quando un qualsiasi blocco del Control-flow graph può essere eseguito prima o dopo un altro qualsiasi blocco, ovvero quando non si ha un'idea dell'ordine in cui devono essere eseguite le istruzioni → questo è quello che fa **il Control Flow Flattening, che è la trasformazione rispetto al controllo più potente che abbiamo**. In un qualche modo, il Control Flow Flattening sposta il controllo sui dati, ovvero l'idea è di prendere un programma e far si che ogni blocco possa essere il successore di ogni altro blocco. Il problema è di mantenere la semantica del programma, nonostante l'aggiunta di archi e per risolvere questo problema, viene aggiunta una variabile **Next**, la quale viene aggiornata alla fine di ogni blocco ed è il valore assunto da tale variabile alla fine del blocco, ad indicare quale sarà il blocco successivo che dovrà essere eseguito. Di fatto, quindi, attraverso il Control Flow Flattening ogni programma viene trasformato in uno Switch sul valore della variabile Next, che è quello che mi indica il prossimo blocco da eseguire. Vediamo un esempio di ciò:

```
int modexp(int y,int x[],
           int w,int n) {
   int R, L;
   int k = 0;
   int s = 1;
   while (k < w) {
      if (x[k] == 1)
         R = (s*y) % n;
      else
         R = s;
      s = R*R % n;
      L = R;
      k++;
   }
   return L;
}
```



```
int modexp(int y, int x[], int w, int n) {
   int R, L, k, s;
   int next=0;
   for(;;)
      switch(next) {
         case 0 : k=0; s=1; next=1; break;
         case 1 : if (k<w) next=2; else next=6; break;
         case 2 : if (x[k]==1) next=3; else next=4; break;
         case 3 : R=(s*y)%n; next=5; break;
         case 4 : R=s; next=5; break;
         case 5 : s=R*R%n; L=R; k++; next=1; break;
         case 6 : return L;
      }
}
```

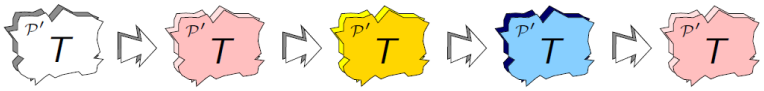


Il Control Flow Flattening, quindi, **è molto efficace come tecnica di offuscamento, ma ha come limitazione il fatto che sia estremamente costoso e di conseguenza deve essere applicato solamente a certe porzioni di codice.**

il Control Flow Flattening, quindi, insieme ai predicati opachi, rappresentano i due principali metodi di offuscamento del controllo. Naturalmente, questi due metodi hanno maggiore efficacia contro un'analisi statica rispetto che ad un'analisi dinamica, dato che:
- l'analisi dinamica, ad esempio, non esegue il ramo falso se il predicato è sempre vero;
- l'offuscamento deve mantenere la semantica del programma.

Dynamic Obfuscation

Mentre l'**offuscamento statico** prende un programma e lo trasforma, attraverso ad esempio il Flattening, in un altro programma più difficile da analizzare per i tool di analisi, i quali quindi restituiscono dei risultati peggiori (dove per risultati peggiori, intendiamo che i risultati contengono il risultato corretto con del rumore). Gli **offuscamenti dinamici**, invece, trasformano un programma in un programma che si auto-modifica, ovvero **trasformano un programma in un altro programma, che quando viene eseguito (quindi a run-time) si aggiorna e quindi si modifica** e questo ha come conseguenza, il fatto che non si ha un oggetto statico che si può analizzare staticamente, perchè l'immagine statica del programma non corrisponde a tutto quello che viene eseguito durante l'esecuzione del programma, dato che alcuni pezzi possono essere aggiunti ed altri modificati durante l'esecuzione (come possiamo vedere dall'immagine sotto, il programma in esecuzione continua a cambiare e da notare anche il fatto, che le varianti al programma sono in un numero limitato).



L’offuscamento dinamico, quindi, confonde l’analisi statica, in quanto solitamente a quest’ultima viene dato in ingresso il programma che poi verrà mandato in esecuzione.

Vediamo alcuni esempi/tecniche di offuscamento dinamico:

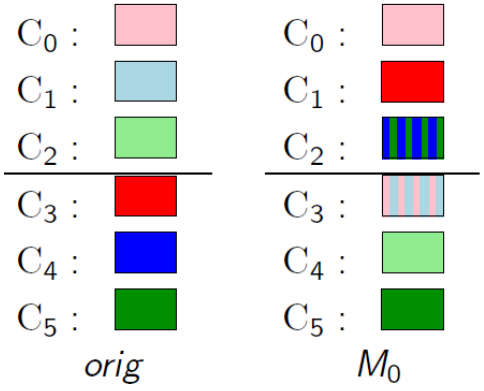
- **Self-modification mechanism** → l’idea alla base è di camuffare molte delle istruzioni originali con delle istruzioni fittizie, ovvero delle istruzioni bacate, attraverso una qualche routine di hiding. Prima però che queste istruzioni bacate vengano eseguite, esse vengono aggiustate attraverso una routine di restore, la quale quindi si occupa di ripristinare le istruzioni bacate al loro valore corretto. L’idea, quindi, consiste in:
  - individuare nel nostro codice delle istruzioni su cui andare ad operare;
  - una volta individuate, andiamo a riscrivere queste istruzioni in maniera sbagliata, ovvero le vado a sostituire con istruzioni non corrette rispetto alla semantica del programma;
  - prima che quest’ultime vengono effettivamente eseguite, avrò delle routine che le andranno a sistemare, ovvero andranno a ripristinare le istruzioni corrette all’interno delle target instructions.

  Per mantenere la semantica, però, bisogna rispettare tre proprietà:

  1. **Ogni cammino che porta ad un’istruzione camuffata deve passare per una routine di restore. Quindi, non può esistere un cammino, che arriva ad un’istruzione camuffata, che non prevede di passare per una routine di restore**;
  2. **Nel cammino tra una routine di restore e la target instruction, non ci deve essere in mezzo una routine di hiding,** perchè altrimenti quest’utlima va nuovamente ad inserire nella target instruction un valore scorretto;
  3. **Ci deve sempre essere una routine di restore in tutti i possibili cammini, che vanno dalla routine di hiding alla target instruction**.

  Il Self-modification mechanism impone un significativo overhead sulle performance e di conseguenza esso deve essere utilizzato secondo un’attenta considerazione del programma e dell’obiettivo della protezione.

- **Virtualizzazione** → l'offuscamento tramite virtualizzazione è il più potente e di fatto protegge un programma dall'analisi manuale o automatizzata, compilando il programma in un bytecode per un'architettura virtuale randomizzata e di conseguenza il programma diventa l'interprete per quel bytecode. Di fatto, il codice diventa il dato che viene dato in ingresso all'interprete;
- **Aucsmith's Algorithm** → primo vero tentativo di protezione del software nel 1996 e viene classificato come un offuscamento dinamico, in quanto durante l'esecuzione il codice si modifica. **L'idea alla base di questo algoritmo è quella di continuare a fare lo XOR tra pezzi di codice, in modo tale che in un certo instante di tempo solamente alcuni pezzi di codice sono in chiaro ed in particolar modo, siano in chiaro i pezzi di codice che devo eseguire in quel preciso momento**. Vediamo meglio graficamente questo concetto:



Immaginiamoci, quindi, di avere una funzione a cui vogliamo applicare questo algoritmo di protezione e supponiamo di suddividere la funzione in blocchetti di codice (nel nostro esempio 6 blocchetti) e raggruppiamo i blocchetti in:

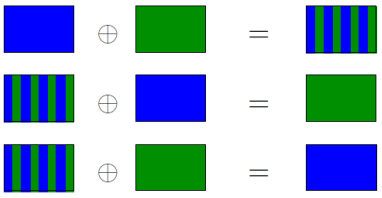
- blocchetti della zona alta (ovverosia quelli al di sopra della barra) e nel nostro esempio sono: C0, C1, C2;
- blocchetti della zona bassa (ovverosia quelli al di sotto della barra) e nel nostro esempio sono: C3, C4, C5.

L'algoritmo, ad ogni **round**, calcola lo XOR della zona alta con la zona bassa e lo memorizza nella zona bassa. Al round successivo, calcola lo XOR della zona bassa con la zona alta e lo memorizza nella zona alta e così via ed in particolare fa lo XOR tra:

- <C0, C3>;
- <C1, C4>;

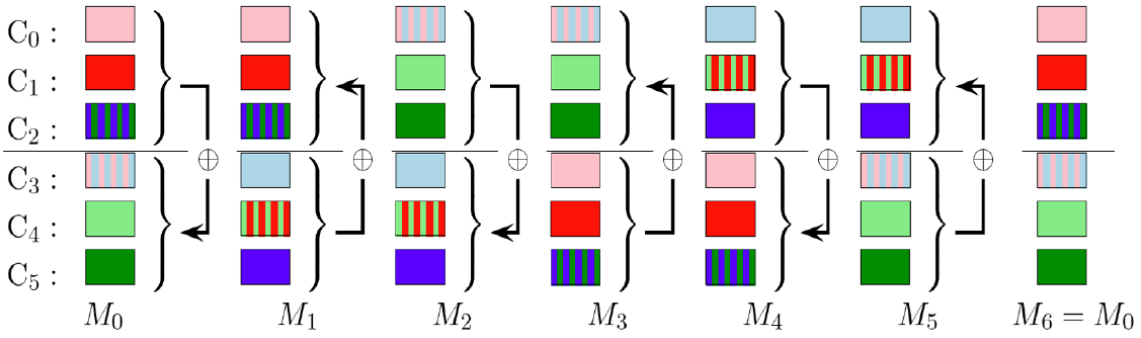
- <C2, C5>.

Il punto fondamentale di questo algoritmo è di **stabilire quali valori devono esserci inizialmente all'interno di C0, C1, C2, C3, C4 e C5 perchè le cose funzionino, ovvero perchè nel momento in cui andiamo ad eseguire tali blocchetti, ci deve essere al loro interno il codice in chiaro da eseguire**. Ricordiamoci, inoltre, che lo XOR ha la proprietà che se viene rifatto due volte sullo stesso oggetto, quest'ultimo ritornerà nella sua forma iniziale. Questa proprietà la si può osservare nella seguente figura:



A questo punto vediamo il funzionamento dell'algoritmo (sul nostro esempio) nelle varie configurazioni, ricordando che:

- **M0 è la configurazione offuscata iniziale;**
- durante i round pari, ogni cella "i" della parte superiore viene xored con la cella "i+3" della parte inferiore. Durante i round dispari, la cella "i" della parte inferiore viene xored con la cella "i-3" della parte superiore.



Notiamo che ad un certo punto M6=M0, ovverosia ad un certo punto si ritorna alla configurazione iniziale.

Come abbiamo detto precedentemente, quindi, l'obiettivo principale è capire cosa deve esserci inizialmente all'interno dei vari blocchetti, in modo tale da avere sempre in chiaro il pezzettino di codice successivo che devo andare ad eseguire.

Thwarting Disassembly

Il Thwarting Disassembly è stato il primo lavoro, che ha posto l'attenzione sul fatto di complicare la fase di disassembly e non la fase di decompilazione. Quando l'offuscamento è stato introdotto, fondamentalmente tutte le tecniche si concentravano nella ricostruzione del codice ad alto livello, ovvero si concentravano nel rendere più complicata la fase di decompilazione del codice → in realtà, l'offuscamento posso applicarlo a qualsiasi livello e il fatto che tante tecniche complicassero la decompilazione, vuol dire che erano tecniche che si immaginavano come attaccante un decompilatore e di conseguenza tali tecniche volevano impoverire il risultato dell'analisi. La stessa cosa può essere fatta a livello del disassemblatore, tanto è che vi sono diversi tool che impediscono oppure offuscano il disassemblaggio.

Il Thwarting Disassembly, quindi, è una tecnica che va a confondere il disassembly ed in particolare facendo credere all'algoritmo di disassemblaggio che in alcuni punti vi sia del codice, anche se in realtà vi sono dei dati, ovvero andando a fargli disassemblare dei valori che in realtà non sono istruzioni. Quindi, tale tecnica va a confondere le instruction boundaries, dato che quello che fa il disassembly è: vede che in un'area di memoria vi sono delle istruzioni e cerca di capire a quali istruzioni assembly corrispondono gli esadecimali. Se però noi andiamo a dire all'algoritmo di disassembly di adoperare una zona di memoria in cui non vi sono istruzioni o di adoperare una zona in cui non ci dovrebbero essere istruzioni, ma noi ne mettiamo di fasulle, andiamo chiaramente a compromettere il risultato del disassemblaggio → si possono, quindi, introdurre errori di disassemblaggio andando ad inserire dei **junk bytes**, ovvero dei byte che possono contenere:

- delle istruzioni parziali;
- oppure delle istruzioni che non verranno mai eseguite.

Questi junk bytes, naturalmente devono preservare la semantica e di conseguenza vengono inseriti in dei blocchi, in cui l'esecuzione non arriverà mai → tali blocchi prendono il nome di **blocchi candidati**, i quali quindi non possono avere un'esecuzione che cade al loro interno e di conseguenza si tratta di blocchi, che vengono raggiunti con un salto diretto, ovverosia sono il risultato di una jump. **Una volta individuati i blocchi candidati, dobbiamo determinare i junk bytes da inserire prima di essi**.

Gli algoritmi di disassembly che dobbiamo riuscire ad ingannare sono due:

1. algoritmo di **Linear Sweep** → essi dicono che le istruzioni si trovano dall'indirizzo A all'indirizzo B, quindi prendono tutti gli indirizzi che stanno tra lo Start Address e l'End Address e quello che fanno è:
    a. decodificano tutti i dati che trovano nel mezzo di tali indirizzi;
    b. una volta decodificata un'istruzione guardano quanto è effettivamente grande l'istruzione ed aggiornano l'indirizzo per andare a decodificare l'istruzione successiva;
    c. se l'indirizzo si trova ancora tra lo Start Address e l'End Address, allora decodificano l'istruzione e così via.

    Essendo che questo algoritmo decodifica tutti i dati presenti tra i due indirizzi, possiamo andare ad inserire dei dati e/o delle istruzioni fasulle tra lo Start Address e l'End Address ed esse verrebbero decodificate.
2. algoritmo di **Recursive Traversal** → utilizza sempre uno Start Address e un End Address, ma una volta che ha decodificato la prima istruzione, l'indirizzo successivo a cui decodificare l'istruzione viene calcolato in base all'istruzione che è stata appena decodificata → ovvero l'algoritmo vede qual è il successore dell'istruzione appena decodificata (quindi se era un fall true va a decodificare l'istruzione successiva, mentre se era una jump va a decodificare l'istruzione all'indirizzo destinatario della jump), ovvero viene seguito il flusso delle istruzioni. Il punto debole di questo algoritmo è il fatto, che molto spesso non è semplice determinare qual è l'indirizzo successivo a cui decodificare l'istruzione e quindi verranno fatte delle assunzioni ed ovviamente l'offuscamento ha come obiettivo quello di sfruttare tali assunzioni → un modo per far creare queste assunzioni è di implementare i jump come delle chiamate a funzione, le quali naturalmente non hanno una return e al posto della return posso andare ad inserire del rumore.

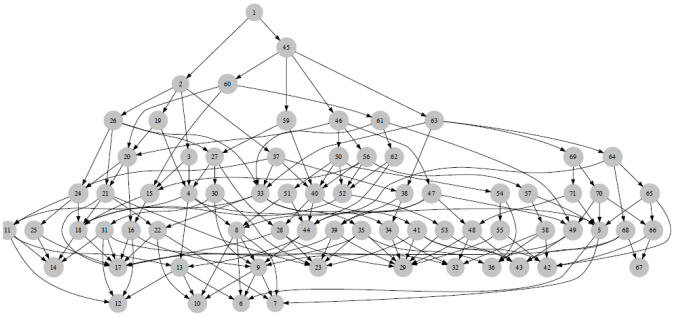
Obstruct Dynamic Analysis

Come abbiamo detto in precedenza, capire cosa significhi offuscare in base all'analisi dinamica è un concetto molto complesso da comprendere e per questo motivo, vi sono dei lavori che danno una determinata accezione (un determinato significato) a cosa significhi offuscare in base all'analisi dinamica. Uno di questi lavori prende il nome di "**Code Obfuscation against Static and Dynamic Reverse Engineering**" e consiste:

- nell’inserire dei **jump indiretti**, ovverosia dei jump risolti dinamicamente. Chiaramente, l’analisi statica non è in grado di capire come proseguirà l’esecuzione del codice e di conseguenza tale analisi viene molto confusa;
- nell’utilizzare le **branch function**, in modo tale da rendere più difficile la costruzione del Control-flow graph;
- per confondere l’analisi dinamica, tale lavoro introduce il concetto di **diversificazione rispetto all’input**, ovvero: l’idea alla base di questo lavoro è che l’analisi dinamica guarda un numero finito di tracce e osservando alcune esecuzioni delle tracce, l’analisi dinamica vuole comprendere la proprietà/la regola generale (quindi vuole comprendere qualcosa di sempre vero) che lega le variabili del programma. **Per ostacolare questo processo, potremmo specializzare il programma per ogni input, in modo tale da avere un programma diverso per ogni input** e di conseguenza la proprietà che l’analisi dinamica impara dall’esecuzione di una traccia, non da alcuna informazione sull’esecuzione delle altre tracce.

Per riuscire, quindi, ad ostacolare l’analisi statica e dinamica, nel lavoro vengono introdotti dei **gadget**, ovverosia dei pezzi di programma (ovvero delle sequenze di istruzioni) e di conseguenza il programma viene suddiviso il gadget. Una volta che il programma è suddiviso in gadget, per confondere l’analisi statica viene fatta una sorta di Control-flow flattening, ovverosia la relazione tra i vari gadget e il loro ordine di esecuzione viene nascosto attraverso l’introduzione di variabili, chiamate **signature**, il cui valore finale, che dipende dal valore dei signature precedenti, determina quale sarà il gadget successivo che verrà eseguito.

L’analizzatore dinamica, però, esegue le varie signature e quindi capisce l’ordine corretto di esecuzione dei gadget e quindi non può essere ingannato in questo modo. Per ingannare l’analizzatore dinamico, allora, viene introdotto il concetto di **gadget diversification**, ovverosia vanno a diversificare i gadget (ovverosia i pezzi di codice, quindi delle sequenze di istruzioni) rispetto all’input da cui viene eseguito → quindi input diversi eseguono gadget diversi, in modo tale da **prevenire che un cammino che sia valido per un input, non sia valido anche per un altro input**. Chiaramente, se il programma viene suddiviso in piccoli gadget (quindi pezzi di codice con un basso numero di istruzioni), naturalmente ce ne saranno di più e conseguentemente ci saranno più interconnessioni tra di loro e quindi viene introdotta maggiore confusione per l’analizzatore dinamico (maggiore è la dimensione dei gadget, minori saranno nel programma e di conseguenza vi sono meno interconnessioni e quindi viene introdotta meno confusione).



Vediamo l’efficienza delle tecniche di offuscamento dette fino ad ora, per quanto riguarda sia l’analisi statica sia per quanto riguarda l’analisi dinamica:

Table II. Analysis of the strength of code obfuscation classes in different analysis scenarios (PM = Pattern Matching LD = Locating Data, LC = Locating Code, EC = Extracting Code, UC = Understanding Code).

	PM		Autom. Static				Autom. Dynamic				Human Assisted			
Name	**LD**	**LC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**
Dynamic code rewriting														
Packing/Encryption	■	✓	■	✓	■	■		✓	✓			✓		✓
Dynamic code modifications	■	■	■	■	■	■		▒	▒	▒				
Environmental requirements	■	■	■	■	■	■	▒	▒	▒	▒	▒	▒	▒	▒
Hardware-assisted code obfuscation	■	■	■	■	■	■	▒	▒	▒	▒	✓	▒	▒	▒
Virtualization	■	■	✓	✓	■	■		✓	▒	▒		✓	▒	✓
Anti-debugging techniques			▒	▒	▒	✓	■	✓	✓	■	▒	▒	▒	✓

Legend	■	obfuscation breaks analysis fundamentally
	▒	obfuscation is not unbreakable, but makes analysis more expensive
		obfuscation only results in minor increases of costs for analysis
	✓	A checkmark indicates that the rating is supported by results in the literature
		Scenarios without a checkmark were classified based on theoretical evaluation