CERT-AGID

Scrivere un deoffuscatore

Ripristinare il flusso di esecuzione dei metodi .NET

CERT-AGID

Table of Contents

Introduzione....3
Esempi....5
Attacco a CFF....7
- Guardare l’IL....7
- Riconoscere i pattern....8
  - Codice di smistamento....8
  - Salto incondizionale dipendente....9
  - Salto incondizionale indipendente....10
  - Salto condizionale dipendente....10
  - Salto condizionale indipendente....10

Scrivere il codice....11
- dnlib....11
- Trovare lo switch....12
- Ricostruzione del flusso....14

Risultato....20

CERT-AGID

Introduzione

Chiunque nella sua carriera di analista di sicurezza si sarà trovato di fronte ad un assembly .NET offuscato. Gli offuscatori si distinguono tra di loro per le proprietà dell'assembly che vengono offuscate, ne sono esempi il nome dei metodi o le stringhe, e per la tecnica usata per implementare l'offuscamento.

Un possibile bersaglio dell'offuscatore può persino essere il flusso di esecuzione stesso, o per meglio dire la sua struttura o linearità[1], rendendo molto difficili da leggere e da seguire (persino con un debugger) i metodi dell'assembly. Il continuo saltare da una parte all'altra del metodo non solo è fiosiologicamente snervante ma, quando la destinazione del salto non è di immediata lettura, rende anche molto difficile ragionare *sul* codice del metodo.

Quando l'offuscamento riguarda il flusso di esecuzione, piuttosto che *ciò* che viene eseguito[2], viene detto Control-flow Obfuscation (CFO) ma limiti precisi per questa definizione non sono applicabili. Possibili tecniche di CFO includono l'inserimento di istruzioni non pertinenti, l'inserimento di salti per spezzare la linearità del flusso, l'utilizzo di caratteristiche esotiche del linguaggio o interprete e, oggetto di questo articolo, la riscrittura degli statement di controllo (`if`, `for` e simili).

Ci focalizzeremo su come scrivere un deoffuscatore per un particolare tecnica di CFO detta Control-flow Flattening (CFF) ed ancora più nello specifico ci limiteremo ad analizzare l'implementazione più diffusa di CFF: l'utilizzo di uno `switch` centrale per lo smistamento del flusso di esecuzione.
Ad ogni esecuzione è associata una variabile di stato s, il flusso di esecuzione entra nello `switch` e da qui viene dirottato ad un suo ramo in base ad una funzione $f(s)$. Ogni ramo contiene un frammento del codice originale e, alla sua fine, aggiorna la variabile di stato con una funzione $s = g(s, k)$ dove k è una costante specifica del ramo.

Questa tecnica è usata da ConfuserEx e molto diffusa tra gli offuscatori .NET, chiunque abbia analizzato malware scritti in .NET se l'è trovata di fronte almeno una volta.

La figura a pagina seguente mostra il risultato di un offuscamento, manuale, tramite CFF. A sinistra il codice originale (una versione particolarmente improntata alla leggibilità di FizzBuzz) e a destra il codice offuscato.
In questo semplice esempio f è la funzione identità e $g(s, k) = k$, inoltre le condizioni degli `if` sono state tradotte interamente in un unico ramo ma in realtà, quando fatto da strumenti automatici, gli operatori che possono dare vita dei salti (tipo le congiunzioni, `&&`) finiscono in rami diversi, complicando ulteriormente il codice.

1 O anche, profondità, in quanto alcune di queste tecniche si dicono di *Control-flow flattening*; per via di come appare il flow chart risultante.

2 Un offuscatore che codifica il bytecode non usa intuitivamente un offuscamento del flusso di esecuzione, piuttosto delle istruzioni stesse. Una volte decodificate il flusso è quello originale.

Nell’esempio dato è piuttosto facile seguire il flusso del programma perchè la funzione *g* è così banale da rendere evidente qual è il prossimo ramo di esecuzione.

Nonostante queste semplificazioni il codice è notevolmente più lungo e più difficile da temere a mente poichè manca la “località visiva” (parti di codice connesse che non sono più vicine).
Se nel primo ramo anzichè usare `s = 7` avessimo usato `s = 2901330439 % 68`, e similmente per gli altri, sarebbe già stato più tedioso ricostruire il flusso. Eseguire a mente calcoli complessi solo per ottenere il prossimo ramo di esecuzione è una tecnica efficace per rallentare l’analisi.

```
for (int i = 1; i <= 30; i++)
  if (i % 3 == 0 && i % 5 == 0 )
    printf("FizzBuzz\n");
  else if (i % 3  == 0 && i % 5 != 0 )
    printf("Fizz\n");
  else if (i % 3  != 0 && i % 5 == 0 )
    printf("Buzz\n");
  else if (i % 3  != 0 && i % 5 != 0 )
    printf("%d\n", i);
```

```
int s = 0;
int i;

for (;;)
{
  switch (s)
  {
    case 0:
      i = 1;
      s = 7;
      continue;

    case 1:
      s = i % 3 == 0 && i % 5 != 0 ? 4 : 3;
      continue;

    case 2:
      i++;
      s = 0;
      continue;

    case 3:
      s = i % 3 != 0 && i % 5 == 0 ? 10 :
8;
      continue;

    case 4:
      printf("Fizz\n");
      s = 0;
      continue;

    case 5:
      break;

    case 6:
      printf("FizzBuzz\n");
      s = 0;
      continue;

    case 7:
      s = i <= 30 ? 9 : 5;
      continue;

    case 8:
      s = i % 3 != 0 && i % 5 != 0 ? 12 :
11;
      continue;

    case 9:
      s = i % 3 == 0 && i % 5 == 0 ? 6 : 1;
      continue;

    case 10:
      printf("Buzz\n");
      s = 0;
      continue;
```

CERT-AGID

```
                    case 11:
                      s = 0;
                      continue;

                    case 12:
                      printf("%d\n", i);
                      s = 0;
                      continue;
                  }
                }
```

A sinistra il codice originale, a destra una volta offuscato con una CFF banale.

Lo scopo di questo documento è scrivere un deoffuscatore, relativamente a codice IL di .NET, per tornare dal codice di destra a quello di sinistra.

Va precisato che offuscatori del genere già esistono, questo articolo si differenzia rispetto ai precedenti in quanto ha valore didattico più che operativo.

Esempi

Durante le nostre attività abbiamo incontrato un malware .NET in cui il codice IL di alcuni metodi era stato codificato. L'inizializzatore di modulo si presentava offuscato con CFF ma data la sua brevità non era comunque difficile seguire il flusso di esecuzione, calcolatrice alla mano.

```
static <Module>()
{

<Module>.\u206E\u200C\u202D\u200D\u202B\u200D\u200F\u206C\u202E\u200C\u206B\u200C\u200D\u202D\u202D\u206E\u202B\u202C\u206C\u206E\u2
00F\u206F\u206C\u200F\u200B\u206B\u202E\u202E\u206E\u206D\u206F\u200E\u206C\u202C\u206E\u202A\u202A\u206C\u202D\u202E();
	for (;;)
	{
		IL_05:
		uint num = 2283578620u;
		for (;;)
		{
			uint num2;
			switch ((num2 = (num ^ 4046939290u)) % 5u)
			{
			case 0u:

<Module>.\u206E\u206C\u200D\u206E\u200E\u200F\u206A\u202C\u200D\u200C\u202B\u206D\u200D\u200D\u206C\u200B\u200F\u200C\u200B\u200D\u2
00D\u206E\u202D\u202D\u206E\u206D\u202A\u206F\u200F\u206A\u206D\u200D\u206D\u200F\u202C\u200C\u202D\u202B\u200C\u202E();
				num = (num2 * 2114284431u ^ 2114510973u);
				continue;
			case 1u:

<Module>.\u200D\u200E\u202C\u202C\u206F\u202B\u206C\u202B\u200F\u200D\u206F\u200F\u202A\u206B\u202E\u206F\u200D\u206F\u202E\u206C\u2
02A\u206B\u200B\u200C\u206D\u200C\u200C\u206B\u202B\u202A\u200E\u202A\u206A\u200B\u200F\u206D\u206E\u202B\u200E\u202C\u202E();
				num = (num2 * 2402118695u ^ 1024147650u);
				continue;
			case 3u:

<Module>.\u200D\u200D\u206B\u206F\u202D\u206D\u206F\u200F\u202B\u202E\u202B\u200F\u202C\u200C\u206C\u200D\u202B\u200B\u206F\u200D\u2
00C\u202D\u202B\u206D\u202C\u206C\u206B\u202A\u206D\u200C\u206E\u202A\u202C\u200C\u200C\u200D\u200E\u206B\u206C\u202E();
				num = (num2 * 2249599230u ^ 547203453u);
				continue;
			case 4u:
				goto IL_05;
			}
			goto Block_1;
		}
	}
	Block_1:

<Module>.\u206A\u206B\u206A\u206E\u202D\u202E\u200B\u206A\u200C\u202B\u202D\u206C\u206D\u206D\u206C\u206B\u200B\u206D\u206A\u202A\u2
02E\u206A\u206A\u200F\u200E\u206D\u206E\u206D\u200B\u200E\u202C\u202A\u202E\u200E\u202C\u206B\u206A\u206F\u206F\u206D\u202E();
}
```

L'inizializzatore di modulo offuscato.

Tuttavia la procedura che decodificava il codice IL era già abbastanza lunga da rendere impraticabile la ricostruzione manuale del flusso.

```
internal unsafe static void
\u206E\u200C\u202D\u200D\u202B\u200D\u200F\u206C\u202E\u200C\u206B\u200C\u200D\u202D\u202D\u206E\u202B\u202C\u206C\u206E\u200F\u206F
\u206C\u200F\u200B\u206B\u202E\u202E\u206E\u206D\u206F\u200E\u206C\u202C\u206E\u202A\u202A\u206C\u202D\u202E()
	{
		Module module = typeof(<Module>).Module;
		string fullyQualifiedName = module.FullyQualifiedName;
		bool flag = fullyQualifiedName.Length > 0 && fullyQualifiedName[0] == '<';
		byte* ptr = (byte*)((void*)Marshal.GetHINSTANCE(module));
		for (;;)
		{
			IL_39:
			uint num = 1116222858u;
			for (;;)
			{
				uint num2;
				uint num12;
				uint num13;
				switch ((num2 = (num ^ 250225443u)) % 62u)
				{
				case 0u:
					Environment.FailFast(null);
					num = (num2 * 1316863231u ^ 1721636044u);
					continue;
				case 1u:
				{
					uint[] array;
					uint[] array2;
					array[0] = (array[0] ^ array2[0]);
					array[1] = array[1] * array2[1];
					array[2] = array[2] + array2[2];
					num = (num2 * 4264723871u ^ 110803757u);
					continue;
				}
				case 2u:
				{
					bool flag2;
					<Module>.CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref flag2);
					num = (((!flag2) ? 205472263u : 944656317u) ^ num2 * 2084536022u);
					continue;
				}
				case 3u:
				{
					uint[] array;
					uint[] array2;
					array[10] = array[10] * array2[10];
					num = (num2 * 1560449509u ^ 4060318075u);
					continue;
				}
				case 4u:
				{
					uint[] array;
					uint[] array2;
					array[8] = array[8] + array2[8];
					num = (num2 * 3077807964u ^ 2514662607u);
					continue;
				}
				case 5u:


. . .


				case 60u:
				{
					uint num8;
					uint num5 = num8 >> 11 | num8 << 21;
					num = (num2 * 1106060028u ^ 2995952501u);
					continue;
				}
				case 61u:
					num = (num2 * 759206699u ^ 1523480793u);
					continue;
				}
				goto Block_2;
				IL_729:
				num12 = num13 >> 2;
				num = 1422973378u;
			}
		}
		Block_2:;
	}
```

Altro metodo offuscato con CFF, questa volta, con più di sessanta rami, è impraticabile da deoffuscazione manuale.

Useremo quest’ultimo metodo come esempio per il nostro deoffuscatore giocattolo.

Attacco a CFF

A prima vista il lavoro di ricostruzione del flusso sembra complesso, la manipolazione degli statement C#, con tutti i loro side effect e corner case può, da sola, essere un'impresa degna di nota. Questa impressione si rileva però errata, infatti il codice C# mostrato da strumenti come ILSpy, dnSpy o dotPeek è dovuto al decompilatore, il cui output è, appunto, la vittima dell'offuscatore.

Inquadrare il problema nella giusta prospettiva lo rende attaccabile, per farlo dobbiamo scendere di livello.

Guardare l'IL

Quella che appare una situazione complessa a livello di codice C# si rileva essere una trama fatta di mattoncini semplici, se guardata a livello di IL.

```
/* 0x0008390D 208A358842   */ IL_0039: ldc.i4    1116222858
// loop start (head: IL_003E)
    /* 0x00083912 202323EA0E   */ IL_003E: ldc.i4    250225443
    /* 0x00083917 61           */ IL_0043: xor
    /* 0x00083918 25           */ IL_0044: dup
    /* 0x00083919 1318         */ IL_0045: stloc.s   V_24
    /* 0x0008391B 1F3E         */ IL_0047: ldc.i4.s  62
    /* 0x0008391D 5E           */ IL_0049: rem.un
    /* 0x0008391E
      453E000000DE0200008B020000F7020000A204000023050000CF030000F0050000E203000094060000B5010000050000000D501000[illegible]C5040000[illegible]A0200002707000[illegible]0370200
      F040000AC0300003C01000080050000570200006F0300005F060000CF060000FD0500001E000000A50500005401000071000000C0[illegible]00000C50[illegible]0000F2FEFFFFF[illegible]0100000
      0000D50500004305000026010000EF000000E8060000B906000037000000014070000 */ IL_004A: switch    (IL_0425, IL_0[illegible]D2, IL_0[illegible]3E, IL_05E9, IL_066A,
      IL_086E, IL_037E, IL_056D, IL_0491, IL_04D9, IL_019E, IL_0220, IL_02BC, IL_0368, IL_01D1, IL_070F, IL_077[illegible], IL_05B[illegible], IL_04F3, IL_0283,
      IL_01B8, IL_0207, IL_040C, IL_0039, IL_033C, IL_07C7, IL_06A4, IL_089E, IL_0593, IL_01F1, IL_0650, IL_047[illegible], IL_083[illegible], IL_062D, IL_0580,

    /* 0x00083A1B 3852070000   */ IL_0147: br        IL_089E

    /* 0x00083A20 14           */ IL_014C: ldnull
    /* 0x00083A21 282600000A   */ IL_014D: call      void [mscorlib]System.Environment::FailFast(string)
    /* 0x00083A26 1118         */ IL_0152: ldloc.s   V_24
    /* 0x00083A28 20AD906731   */ IL_0154: ldc.i4    828870829
    /* 0x00083A2D 5A           */ IL_0159: mul
    /* 0x00083A2E 20D17A557C   */ IL_015A: ldc.i4    2085976785
    /* 0x00083A33 61           */ IL_015F: xor
    /* 0x00083A34 38D9FEFFFF   */ IL_0160: br        IL_003E

    /* 0x00083A39 14           */ IL_0165: ldnull
    /* 0x00083A3A 282600000A   */ IL_0166: call      void [mscorlib]System.Environment::FailFast(string)
    /* 0x00083A3F 1118         */ IL_016B: ldloc.s   V_24
    /* 0x00083A41 20022211ED   */ IL_016D: ldc.i4    -317644286
    /* 0x00083A46 5A           */ IL_0172: mul
    /* 0x00083A47 20B99E9414   */ IL_0173: ldc.i4    345284281
    /* 0x00083A4C 61           */ IL_0178: xor
    /* 0x00083A4D 38C0FEFFFF   */ IL_0179: br        IL_003E

    /* 0x00083A52 1109         */ IL_017E: ldloc.s   V_9
    /* 0x00083A54 1F0B         */ IL_0180: ldc.i4.s  11
    /* 0x00083A56 64           */ IL_0182: shr.un
    /* 0x00083A57 1109         */ IL_0183: ldloc.s   V_9
    /* 0x00083A59 1F15         */ IL_0185: ldc.i4.s  21
    /* 0x00083A5B 62           */ IL_0187: shl
    /* 0x00083A5C 60           */ IL_0188: or
    /* 0x00083A5D 130C         */ IL_0189: stloc.s   V_12
    /* 0x00083A5F 1118         */ IL_018B: ldloc.s   V_24
    /* 0x00083A61 20FC22ED41   */ IL_018D: ldc.i4    1106060028
    /* 0x00083A66 5A           */ IL_0192: mul
    /* 0x00083A67 20759B92B2   */ IL_0193: ldc.i4    -1299014795
    /* 0x00083A6C 61           */ IL_0198: xor
    /* 0x00083A6D 38A0FEFFFF   */ IL_0199: br        IL_003E
```

Tre rami dello switch con evidenziato il codice per il ritorno al punto di smistamento (anch'esso evidenziato, di colore diverso).

Evidenziato in figura c'è il codice per il calcolo di *s = g(s, k)* dei primi (in ordine di offset) rami dello `switch` ed il calcolo di *f(s)* per lo smistamento del flusso tramite lo `switch` stesso.

Un modo per deoffuscare il flusso è quello di calcolare il target di ogni ramo e sostituire il codice evidenziato con un salto incondizionale a questo.
Alternativamente (l'approccio preso) è possibile ricordare tutte le istruzioni viste fino al pattern di salto e poi riprendere dal target di questo, incollando le due sequenze di istruzioni per ottenere una sequenza di istruzioni deoffuscate.

Sebbene i tre blocchi evidenziati siano tutti simili, questi non possono essere l'unico tipo di pattern presente; infatti essi rappresetano salti incondizionati, un diverso tipo di codice deve gestire quelli condizionali.

Riconoscere i pattern

Il modo migliore per riconoscere i pattern è seguire il flusso di esecuzione e prendere nota dei vari tipi di "mattoncini" incontrati "per strada".
Se non contiamo il codice per lo smistamento, ovvero per il calcolo di *f(s)*, ci sono quattro pattern: *salto incondizionale dipendente*, *salto incondizionale indipendente*, *salto condizionale dipendente* e *salto condizionale indipendente*.

Codice di smistamento

E' il codice che a partire dallo stato corrente salta al ramo giusto dello switch. Si compone di tre parti: la prima che imposta il valore iniziale di s, la seconda che a partire da s calcola in quale ramo dello `switch` andare (si tratta di *f(s)*, ovvero l'espressione che, in C#, è dentro le parentesi dello `switch`) e la terza è lo switch stesso.

```
IL_0039: ldc.i4 1116222858
// loop start (head: IL_003E)
IL_003E: ldc.i4 250225443
IL_0043: xor
IL_0044: dup
IL_0045: stloc.s V_24
IL_0047: ldc.i4.s 62
IL_0049: rem.un
IL_004A: switch (
        IL_0425, IL_03D2, IL_043E, IL_05E9, IL_066A, IL_0516, IL_0737, IL_0529, IL_07DB, IL_02FC,
        IL_014C, IL_031C, IL_060C, IL_03B1, IL_086E, IL_037E, IL_056D, IL_0491, IL_04D9, IL_019E,
        IL_0220, IL_02BC, IL_0368, IL_01D1, IL_070F, IL_0776, IL_05B6, IL_04F3, IL_0283, IL_06C7,
        IL_039E, IL_04B6, IL_07A6, IL_0816, IL_0744, IL_0165, IL_06EC, IL_029B, IL_01B8, IL_0207,
        IL_040C, IL_0039, IL_033C, IL_07C7, IL_06A4, IL_089E, IL_0593, IL_01F1, IL_0650, IL_0471,
        IL_083C, IL_062D, IL_0580, IL_054C, IL_071C, IL_068A, IL_026D, IL_0236, IL_082F, IL_0800,
        IL_017E, IL_085B)

IL_0147: br IL_089E
```

La prima istruzione (`IL_0039`) imposta il valore inizale della variabile di stato s*, le successive fino allo switch (`IL_004A`) escluso calcolano* f(s) *usata per il test effettuato da questo.*

La prima istruzione (a `IL_0039`) pusha nello stack il valore iniziale di *s* (`1116222858`). A seguire vi sono le istruzioni per il calcolo dell'espressione testata dallo switch.

Questo codice si aspetta un operando, *s*, nello stack, può quindi essere pensato come una funzione. E' immediato vedere che l'operando viene XORato con una costante e salvato in una variabile locale, il risultato è poi passato all'operatore di resto (per 62) ed usato come operando per l'opcode `switch`.
Notare che il resto è effettuato con `rem.un`, che tratta gli operandi come numeri senza segno, per cui è necessario convertirli.

```
public uint f(int s, out int new_s)
{
    new_s = (s ^ 1116222858);
    return ( (uint)new_s ) % 62U;
}
```

La funzione f*, in C#. Notare che non è pura, abbiamo evitato di appesantire la notazione indicando, impropriamente, il suo risultato semplicemente con* f(s)*.*

Tramite questa funzione è possibile, dato un *s*, ottenere l'indice del ramo a cui saltare (altre al nuovo valore di *s*). L'operando dell'opcode `switch`, una lista di offset, è necessario per trasformare il valore ritornato da *f(s)* in un offset nel codice del metodo.

A partire da questo offset è possibile seguire il codice fino a trovare un salto che torna al calcolo di *f(s)*, calcolo che in questo esempio si trova (come evidenziato da dnSpy) all'offset `IL_003E`. Ogni volta un simile salto è incontrato significa che siamo in presenza di un pattern per il calcolo di *s = g(s, k)* e quindi da ricordare (se mai riscontrato prima).
Risparmiamo ai lettori la cronostoria dei vari pattern incontrati e ne riportiamo subito la tipologia.

Notare infine che dopo l'opcode `switch` vi è un opcode di salto incondizionato, questo è di fatto il `default` dello `switch` ma dato che ci sono tanti rami quanti i possibili valori di *s* (62 in questo caso), il default può essere ignorato (che difatti rimanda alla fine della funzione).

Salto incondizionale dipendente

Si tratta di un pattern che rappresenta un salto incondizionale al prossimo (in ordine di programma originale) ramo dello `switch` ma il calcolo di questo dipende dal valore corrente di *s*. E' la tipologia di pattern più comune.

```
IL_0152: ldloc.s V_24
IL_0154: ldc.i4 828870829
IL_0159: mul
IL_015A: ldc.i4 2085976785
IL_015F: xor
IL_0160: br IL_003E
```

```
s = (s * K1) ^ K2;
```

Le istruzioni IL a sinistra ed il codice C# a destra. Le prime saranno necessarie per il riconoscimento del blocco, il secondo per la sua emulazione.

Salto incondizionale indipendente

Analogo al pattern precedente in quanto si tratta di un salto incondizionale ma il suo target non dipende da s.

```
IL_01FD: ldc.i4 951865456                s = K;
IL_0202: br IL_003E
```

Le istruzioni IL a sinistra ed il codice C# a destra. Le prime saranno necessarie per il riconoscimento del blocco, il secondo per la sua emulazione.

Salto condizionale dipendente

Si tratta di un pattern che rappresenta un salto condizionale a due possibili rami successivi (in ordine di programma originale) dello `switch` ma il calcolo di questi dipende dal valore corrente di *s*.

```
IL_0452: brfalse.s IL_045C               s = (s * K1) ^ (cond ? K2 : K3);

IL_0454: ldc.i4 944656317
IL_0459: dup
IL_045A: br.s IL_0462

IL_045C: ldc.i4 205472263
IL_0461: dup

IL_0462: pop
IL_0463: ldloc.s V_24
IL_0465: ldc.i4 2084536022
IL_046A: mul
IL_046B: xor
IL_046C: br IL_003E
```

Le istruzioni IL a sinistra ed il codice C# a destra. Le prime saranno necessarie per il riconoscimento del blocco, il secondo per la sua emulazione.

Salto condizionale indipendente

Analogo al pattern precedente in quanto si tratta di un salto condizionale ma il suo target non dipende da s.

```
IL_01BB: bge.s IL_01C5                   s = cond ? K1 : K2;

IL_01BD: ldc.i4 886947246
IL_01C2: dup
```

```
IL_01C3: br.s IL_01CB

IL_01C5: ldc.i4 2068195830
IL_01CA: dup

IL_01CB: pop
IL_01CC: br IL_003E
```

Le istruzioni IL a sinistra ed il codice C# a destra. Le prime saranno necessarie per il riconoscimento del blocco, il secondo per la sua emulazione.

Scrivere il codice

Per la manipolazione degli assembly .NET, incluso il loro codice IL, abbiamo usato dnlib. Nel deoffuscatore didattico che vogliamo creare, carichiamo un file specifico ed otteniamo un riferimento al metodo che vogliamo deoffuscare tramite il RID. Fatto questo, leggiamo le sue istruzioni alla ricerca del codice di smistamento, salviamo tutte le informazioni utili da questo e poi iniziamo a collezionare le istruzioni dei singoli rami fino ad incontrare un pattern di quelli mostrati precedentemente. Quando questo succede, proseguiamo la scansione (ricorsivamente) dal prossimo ramo, o dai prossimi rami, e poi incolliamo il risultato alle istruzioni già collezionate.

Come di prassi lasciamo al lettore il compito di rendere il codice riusabile.

dnlib

Per usare dnlib è necessario scaricare i sorgenti e compilarla, a tale scopo è possibile usare Visual Studio CE o, se non si vuole occupare troppo spazio, JetBrains Raider.

Una volta compilata, l’assembly va aggiunto tra le referenze.

Le principali classi che useremo saranno:

- `Instruction`. Descrive un’instruzione, ci interessano principalmente le proprietà `OpCode` e `Operand`. Quest’ultimo può essere un’oggetto qualsiasi, nel caso di salti si tratta a sua volta di un’instanza di `Instruction`, che rappresenta l’istruzione target. In questo modo è possibile manipolare il codice senza dover gestire gli offset.
- `OpCodes`. Contiene la lista di tutti gli opcode.
- `MethodDef`. Descrive un metodo, in particolare la sua proprietà `Body` contiene le informazioni sul corpo del metodo, inclusa la lista di istruzioni tramite la proprietà `Instructions` (che ha tipo `IList<Instruction>` e può essere modificata a piacere).

- ModuleDef. Descrive un modulo .NET. Permette di risolvere metodi, tipi, ed altri metadati in base al loro metadata token o RID.

I passi per caricare, trovare un metodo e salvare un assembly .NET con dnlib sono ben documentati online e non presentano particolari difficolta.
A scopo didattico qui l'assembly caricato è fisso, così come il RID del metodo da deoffuscare.

Ogni "componente" di un assembly .NET (metodi, costanti, membri, tipi, moduli e così via) ha un metadata token di 32 bit. Esso è composto da due parti: il MSB indica il tipo di metadato (ad esempio 0x06 per i metodi) e i restanti 3 byte sono identificativo univoco del metadato, detto RID (Relative ID).

```
//Legge il modulo
ModuleContext modCtx = ModuleDef.CreateModuleContext();
ModuleDefMD module = ModuleDefMD.Load(@"C:\users\labbe\desktop\input.exe",
modCtx);

//Ottiene un riferimento al metodo da deoffuscare
MethodDef m = module.ResolveMethod(0x000003A);

//... TODO ...

//Scrive il modulo su un altro file
ModuleWriterOptions o = new ModuleWriterOptions(module);
o.MetadataOptions.Flags |= MetadataFlags.KeepOldMaxStack;
o.Logger = DummyLogger.NoThrowInstance;
module.Write(@"C:\users\labbe\desktop\output.exe", o);
```

Il codice sopra carica un'assembly e ottiene il riferimento al metodo da deoffuscare (tramite il suo RID), da qui è possibile passare al riconoscimento del codice di smistamento.

Trovare lo switch

Facendo riferimento alla sezione sui pattern è possibile trovare facilmente il pattern che contiene il codice di smistamento del flusso di esecuzione.

```
public bool findSwitchReflow(IList<Instruction> ins, ref int num, out List<Instruction> reflow)
{
  //Inizializza la lista che conterrà le istruzione deoffuscate
        reflow = new List<Instruction>();

  //Scorri le istruzioni
        for (int i = 0; i < ins.Count; i++)
        {
    //Aggiungile alle istruzioni deoffuscate, finchè non troviamo lo switch, tutte le istruzioni
    //vanno tenute. Poi rimuoviamo quelle del pattern.
                reflow.Add(ins[i]);

    //Segna l'istruzione come già visitata, il parametro è lo stato al momento della visita.
    //Per convenzione usiamo 0 in quanto anche non ha senso parlare di stato s.
                ins[i].setVisited(0);

    //Il pattern di smistamento richiede almeno otto istruzioni
```

```
                if (i < 7)
                        continue;

    //Le otto istruzioni
                var i0 = ins[i - 7];
                var i1 = ins[i - 6];
                var i2 = ins[i - 5];
                var i3 = ins[i - 4];
                var i4 = ins[i - 3];
                var i5 = ins[i - 2];
                var i6 = ins[i - 1];
                var i7 = ins[i - 0];

    //Controlla il tipo di ogni istruzione
                bool i0_ok = i0.IsLdcI4();
                bool i1_ok = i1.IsLdcI4();
                bool i2_ok = i2.OpCode == OpCodes.Xor;
                bool i3_ok = i3.OpCode == OpCodes.Dup;
                bool i4_ok = i4.OpCode == OpCodes.Stloc_S;
                bool i5_ok = i5.OpCode == OpCodes.Ldc_I4_S;
                bool i6_ok = i6.OpCode == OpCodes.Rem_Un;
                bool i7_ok = i7.OpCode == OpCodes.Switch;

    //Se non sono loro, riparti
                if (!i0_ok || !i1_ok || !i2_ok || !i3_ok || !i4_ok || !i5_ok || !i6_ok || !i7_ok)
                {
                        continue;
                }

    //Abbiamo trovato il pattern, rimuoviamo le otto istruzioni del pattern da quelle
    //deoffuscate e recuperiamo le informazioni che ci servono:
    //  - il valore iniziale di s (num)
    //  - il valore con cui è XORato (xor)
    //  - il modulo da effettuare (mod)
    //  - Le istruzioni target dello switch (target_ins)
    //  - L'inizio della funzione f(s) (startOfBlock)

                //Rimuove le otto istruzioni
                for (int j = 0; j < 8; j++)
                        reflow.RemoveAt(reflow.Count - 1);

                startOfBlock = i1;
                num = (int) i0.Operand;
                xor = (int) i1.Operand;
                mod = (sbyte) i5.Operand;
                target_ins = (Instruction[]) switchIns.Operand;

                return true;
        }

  //Non abbiamo trovato lo switch
        return false;

}
```

Il codice non presenta particolari punti di interesse. Le informazioni sono recuperate in variabili di istanza, tranne che per `num` e `reflow` che sono variabili mutevoli.

La variabile `startOfBlock` è utilizzata per riconoscere i salti che ritornano al codice di smistamento, `num`, `xor` e `mod` sono il valore iniziale di *s* e le costanti di *f*.

Inoltre ci servono le istruzioni target dello `switch` in modo da sapere dove trovare i prossimi rami.

La lista `reflow`, creata dalla funzione stessa, contiene le prime istruzioni deoffuscate (che sono tutte quelle prima del pattern di smistamento).

Alla classe `Instruction` sono stati aggiunti dei metodi di estensione, `Visited`, `setVisited` e `Processed`. Questi sono usati per ricordare se un'istruzione è già stata visitata con un

particolare valore di *s* (`Visited` e `setVisited`) e per determinare se è stata visitata con qualsiasi valore di *s* (`Processed`).

Questo è utile per debug ed è necessario per rompere i cicli.

```
public static class Ext
{
        private static HashSet<KeyValuePair<Instruction, int>> visited = new
HashSet<KeyValuePair<Instruction, int>>();
        private static HashSet<Instruction> processed = new
HashSet<Instruction>();

        public static bool Visited(this Instruction ins, int num)
        {

                return visited.Contains(new KeyValuePair<Instruction, int>(ins,
num));
        }

        public static bool Processed(this Instruction ins)
        {

                return processed.Contains(ins);
        }

        public static void setVisited(this Instruction ins, int num)
        {
                processed.Add(ins);
                visited.Add(new KeyValuePair<Instruction, int>(ins, num));
        }
}
```

Ricostruzione del flusso

Per la ricostruzione si effettua seguendo il flusso di istruzioni a partire da un'istruzione di *start* fino al raggiungimento di un pattern di salto. Ci servirà anche il valore corrente di *s*. Le istruzioni sono copiate in una lista locale, tranne quelle del pattern. Raggiunto quest'ultimo, tramite il valore di *s* e le constanti trovate nel codice, possiamo calcolare la prima istruzione del prossimo ramo.

Alcuni accorgimenti riguardano la visita di rami già analizzati: se ci ritroviamo ad analizzare un ramo con gli stessi valori di *start* ed *s*, dobbiamo evitare di rianalizzarlo (che porterebbe ad un ciclo infinito) e **ritornare un salto all'istruzione *start***. Questo funziona perchè ogni ramo contiene almeno un'istruzione che non fa parte di nessun pattern e siccome questi sono sempre alla fine, **la prima istruzione non è mai di nessun pattern**. Inoltre dnlib ci permette facilmente di creare un salto ad un'istruzione anche se non sappiamo l'offset a cui finirà.

L'algoritmo di ricostruzione è ricorsivo, useremo quindi una funzione la cui firmà sarà `List<Instruction> reflow(int s, Instruction start, Ilist<Instruction> ins)`, che prende lo stato al momento dell'arrivo al ramo, la prima istruzione del ramo e le istruzioni del corpo del metodo. Inoltre la funzione ritorna la lista di istruzioni deoffuscate a partire da `start` **fino alla fine della funzione offuscata**.

Il caso base è quando si arriva ad incontrare un’istruzione `ret` o a fine funzione (in questo caso si aggiunge un `ret` artificiale).

Nel caso di salti incondizionati, il risultato è dato dalla concatenazione delle istruzioni del ramo corrente con l’applicazione della funzione `reflow` al ramo successivo.

Nel caso di salti condizionati, il risultato è dato dalla concatenazione tra le istruzioni del ramo corrente, del codice per l’emulazione del salto condizionale e l’applicazione della funzione `reflow` ai due possibili rami successivi.
Il salto condizionale è emulato con un semplice gadget che salta alla prima istruzione di uno dei prossimi rami quando la condizione testata è vera, oppure passa alla prossima istruzione, che è un salto alla prima istruzione dell’altro ramo.

```
IL_0421: brfalse.s IL_042B

IL_0423: ldc.i4 944656317
IL_0428: dup
IL_0429: br.s IL_0431

IL_042B: ldc.i4 205472263
IL_0430: dup

IL_0431: pop
IL_0432: ldloc.s V_24
IL_0434: ldc.i4 2084536022
IL_0439: mul
IL_043A: xor
IL_043B: br IL_003E
```

```
brfalse.s f(g(s, (2084536022, 205472263)))
br f(g(s, (2084536022, 944656317)))
```

Un pattern di salto condizionale ed il relativo gadget (in pseudo-IL) per salto condizionale ai due possibili rami.

Questo gadget risulterà piuttosto comune a chi ha programmato in assembly o a basso livello in generale. E’ possibile rimuovere il salto incondizionale concatenando prima le istruzioni derivanti dal `reflow` della destinazione di questo e poi quelle dell’altro ramo.

```
reflow(s, start, ins) -> list:
        pattern, costanti, deoffuscate = leggi_le_istruzioni_fino_a_pattern(start, ins)

        case pattern:
                ret => return deoffuscate

                incondizionale => return deoffuscate +
                        reflow(prossimo_stato(s, costanti), prossima_istruzione(s, costanti), ins)

                condizionale => return deoffuscate + gadget +
                        reflow(prossimo_stato(s, costanti.cond1),
                                prossima_istruzione(s, costanti.cond1), ins) +
                        reflow(prossimo_stato(s, costanti.cond2),
                                prossima_istruzione(s, costanti.cond2), ins)
```

Pseudo codice della funzione di `reflow`.

CERT-AGID

L’implementazione della funzione `reflow` segue.

```
public List<Instruction> reflow(int num, Instruction start, IList<Instruction> ins)
{
        List<Instruction> result = new List<Instruction>();

        //Se l’istruzione è già stata vista, ritorna un salto ad essa
        if (start.Visited(num))
        {
                result.Add(new Instruction(OpCodes.Br, start));
                return result;
        }

        //Dove si trova l’istruzione nella lista?
        int index = ins.IndexOf(start);

        //Scorriamo in avanti fino alla fine, al massimo
        while (index < ins.Count())
        {
                var cur = ins[index];


                cur.setVisited(num);

                //Aggiungi l’istruzione
                result.Add(cur);

                //Abbiamo trovato un salto al codice di smistamento?
                if (cur.IsBr() && cur.Operand == startOfBlock)
                {

                        /*
                                IL_04E8: ldloc.s   V_24
                                IL_04EA: ldc.i4    -51339179
                                IL_04EF: mul
                                IL_04F0: ldc.i4    1662030126
                                IL_04F5: xor
                                IL_04F6: br        IL_003E
                        */
                        if (
                                index >= 5 &&
                                ins[index - 5].OpCode == OpCodes.Ldloc_S &&
                                ins[index - 4].OpCode == OpCodes.Ldc_I4 &&
                                ins[index - 3].OpCode == OpCodes.Mul &&
                                ins[index - 2].OpCode == OpCodes.Ldc_I4 &&
                                ins[index - 1].OpCode == OpCodes.Xor

                        )
                        {
                                //Prendi le constanti e calcola il prossimo s
                                int mul_op = (int) ins[index - 4].Operand;
                                int xor_op = (int) ins[index - 2].Operand;

                                int new_num = (num * mul_op) ^ xor_op;

                                //Rimuovi le istruzioni del pattern
                                for (int j = 0; j < 6; j++)
                                        result.RemoveAt(result.Count - 1);

                                //Calcola le istruzioni successive
                                var next = target(new_num, out new_num);


                                result.AddRange(reflow(new_num, next, ins));

                                return result;
                        }


                        /*
                                IL_0421: brfalse.s IL_042B

                                IL_0423: ldc.i4    944656317
                                IL_0428: dup
                                IL_0429: br.s      IL_0431

                                IL_042B: ldc.i4    205472263
                                IL_0430: dup

                                IL_0431: pop
                                IL_0432: ldloc.s   V_24
                                IL_0434: ldc.i4    2084536022
                                IL_0439: mul
                                IL_043A: xor
                                IL_043B: br        IL_003E
                        */
                        if (
                                index >= 11 &&
                                isCond(ins[index - 11]) && (ins[index - 11].Operand as Instruction).Offset ==
ins[index - 11].Offset + 10 &&
                                ins[index - 10].OpCode == OpCodes.Ldc_I4 &&
                                ins[index - 9].OpCode == OpCodes.Dup &&
                                ins[index - 8].OpCode == OpCodes.Br_S && (ins[index - 8].Operand as
Instruction).Offset == ins[index - 8].Offset + 8 &&
```

```
                                ins[index - 7].OpCode == OpCodes.Ldc_I4 &&
                                ins[index - 6].OpCode == OpCodes.Dup &&
                                ins[index - 5].OpCode == OpCodes.Pop &&
                                ins[index - 4].OpCode == OpCodes.Ldloc_S &&
                                ins[index - 3].OpCode == OpCodes.Ldc_I4 &&
                                ins[index - 2].OpCode == OpCodes.Mul &&
                                ins[index - 1].OpCode == OpCodes.Xor
                        )
                        {
                                int mul_op = (int) ins[index - 3].Operand;
                                int xor_op2 = (int) ins[index - 7].Operand;
                                int xor_op1 = (int) ins[index - 10].Operand;

                                //Prendi l'istruzione di condizionale
                                var conditional = ins[index - 11];

                                 //Prendi le costanti e calcola i nuovi target
                                int new_num1 = (num * mul_op) ^ xor_op1;
                                var next1 = target(new_num1, out new_num1);

                                int new_num2 = (num * mul_op) ^ xor_op2;
                                var next2 = target(new_num2, out new_num2);

                                //Rimuovi le istruzioni del pattern
                                for (int j = 0; j < 12; j++)
                                        result.RemoveAt(result.Count - 1);

                                if (conditional.OpCode == OpCodes.Bge_S)
                                {
                                        xor_op1 = xor_op1;
                                }

                                //Calcola le istruzioni successive
                                var res2 = reflow(new_num2, next2, ins);
                                var res1 = reflow(new_num1, next1, ins);

                                var icon = new Instruction(longCond(conditional), res2[0]);
                                var ijmp = new Instruction(OpCodes.Br, res1[0]);
                                //Riscrivi come: cond T2 / jmp T1
                                result.Add(icon);
                                result.Add(ijmp);

                                result.AddRange(res2);
                                result.AddRange(res1);


                                return result;
                        }


                        /*
                                ldc.i4 <num>
                                br <switch>

                         */
                        if (index >= 1 && ins[index - 1].IsLdcI4())
                        {
                                int lnum = (int) ins[index - 1].Operand;

                                //Remove the last 2 insts
                                for (int j = 0; j < 2; j++)
                                        result.RemoveAt(result.Count - 1);

                                //Add the next instructions
                                var next = target(lnum, out int new_num);
                                result.AddRange(reflow(new_num, next, ins));
                                return result;
                        }


                        /*
                                IL_01BB: bge.s      IL_01C5

                                IL_01BD: ldc.i4     886947246
                                IL_01C2: dup
                                IL_01C3: br.s       IL_01CB

                                IL_01C5: ldc.i4     2068195830
                                IL_01CA: dup

                                IL_01CB: pop
                                IL_01CC: br         IL_003E

                        */
                        if (
                                index >= 7 &&
                                isCond(ins[index - 7]) && (ins[index - 7].Operand as Instruction).Offset ==
ins[index - 7].Offset + 10 &&
                                ins[index - 6].OpCode == OpCodes.Ldc_I4 &&
                                ins[index - 5].OpCode == OpCodes.Dup &&
                                ins[index - 4].OpCode == OpCodes.Br_S && (ins[index - 4].Operand as
Instruction).Offset == ins[index - 4].Offset + 8 &&
                                ins[index - 3].OpCode == OpCodes.Ldc_I4 &&
                                ins[index - 2].OpCode == OpCodes.Dup &&
```

```
                                        ins[index - 1].OpCode == OpCodes.Pop
                                )
                                {
                                        int new_num2 = (int) ins[index - 3].Operand;
                                        int new_num1 = (int) ins[index - 6].Operand;

                                        //Get the conditional jmp
                                        var conditional = ins[index - 7];

                                        var next1 = target(new_num1, out new_num1);
                                        var next2 = target(new_num2, out new_num2);

                                        //Remove the last 8 insts
                                        for (int j = 0; j < 8; j++)
                                                result.RemoveAt(result.Count - 1);

                                        if (conditional.OpCode == OpCodes.Bge_S)
                                        {
                                                new_num2 = new_num2;
                                        }

                                        //Next results
                                        var res2 = reflow(new_num2, next2, ins);
                                        var res1 = reflow(new_num1, next1, ins);

                                        //Rewrite as: cond T2 / jmp T1
                                        var icon = new Instruction(longCond(conditional), res2[0]);
                                        var ijmp = new Instruction(OpCodes.Br, res1[0]);
                                        result.Add(icon);
                                        result.Add(ijmp);
                                        result.AddRange(res2);
                                        result.AddRange(res1);

                                        return result;
                                }


                        }

                        //Next instruction
                        if (cur.OpCode == OpCodes.Ret)
                                return result;

                        if (cur.OpCode == OpCodes.Br)
                        {
                                index = ins.IndexOf(cur.Operand as Instruction);
                        }
                        else
                                index++; // = (cur.OpCode == OpCodes.Br || cur.OpCode == OpCodes.Br_S ) ?
ins.IndexOf(cur.Operand as Instruction) : (index + 1);
        }

        result.Add(new Instruction(OpCodes.Ret));
        return result;
}


public Instruction target(int num, out int new_num)


{
        new_num = (num ^ xor);
        var x = ((uint)new_num) % ((uint)mod);
        return target_ins[x];
}

public bool isCond(Instruction ii5)
{
        return ii5.IsBrfalse() || ii5.IsBrtrue() ||
                        ii5.OpCode == OpCodes.Bge || ii5.OpCode == OpCodes.Beq || ii5.OpCode == OpCodes.Bgt ||
                        ii5.OpCode == OpCodes.Ble || ii5.OpCode == OpCodes.Blt ||
                        ii5.OpCode == OpCodes.Bge_S || ii5.OpCode == OpCodes.Beq_S || ii5.OpCode == OpCodes.Bgt_S ||
                        ii5.OpCode == OpCodes.Ble_S || ii5.OpCode == OpCodes.Blt_S ||
                        ii5.OpCode == OpCodes.Bge_Un || ii5.OpCode == OpCodes.Bgt_Un || ii5.OpCode == OpCodes.Ble_Un
||
                        ii5.OpCode == OpCodes.Blt_Un || ii5.OpCode == OpCodes.Bne_Un ||
                        ii5.OpCode == OpCodes.Bge_Un_S || ii5.OpCode == OpCodes.Bgt_Un_S || ii5.OpCode ==
OpCodes.Ble_Un_S ||
                        ii5.OpCode == OpCodes.Blt_Un_S || ii5.OpCode == OpCodes.Bne_Un_S;
}


public dnlib.DotNet.Emit.OpCode longCond(Instruction ii5)
{
        OpCode op = ii5.OpCode;
        if (op == OpCodes.Bge_S)
                return OpCodes.Bge;

        if (op == OpCodes.Beq_S)
                return OpCodes.Beq;

        if (op == OpCodes.Bgt_S)
                return OpCodes.Bgt;
```

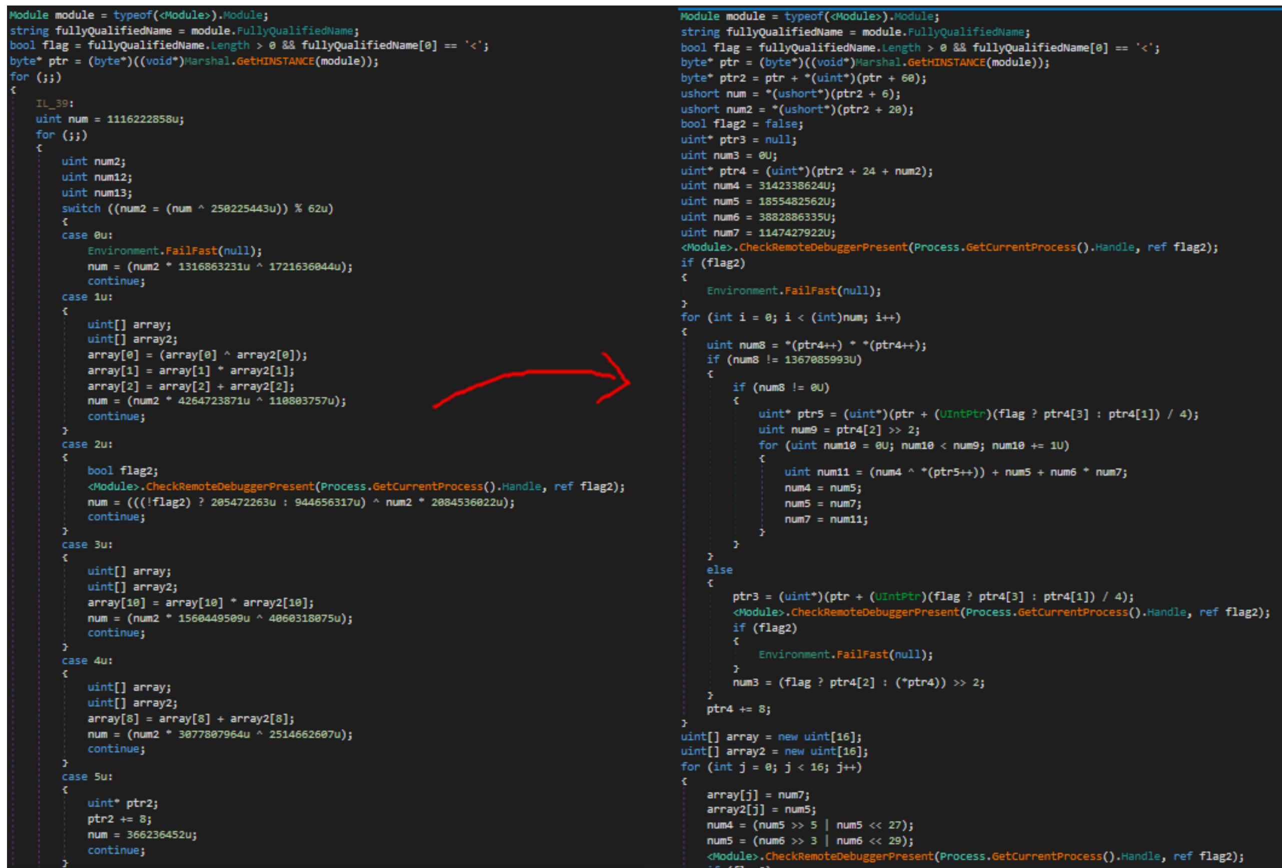
```
        if (op == OpCodes.Ble_S)
                return OpCodes.Ble;

        if (op == OpCodes.Blt_S)
                return OpCodes.Blt;

        if (op == OpCodes.Bge_Un_S)
                return OpCodes.Bge_Un;

        if (op == OpCodes.Bgt_Un_S)
                return OpCodes.Bgt_Un;

        if (op == OpCodes.Ble_Un_S)
                return OpCodes.Ble_Un;

        if (op == OpCodes.Blt_Un_S)
                return OpCodes.Blt_Un;

        if (op == OpCodes.Bne_Un_S)
                return OpCodes.Bne_Un;

        return op;

}
```

Adesso possiamo completare il codice principale del deoffuscatore.

```
//Legge il modulo
ModuleContext modCtx = ModuleDef.CreateModuleContext();
ModuleDefMD module = ModuleDefMD.Load(@"C:\users\labbe\desktop\input.exe",
modCtx);

//Ottiene un riferimento al metodo da deoffuscare
MethodDef m = module.ResolveMethod(0x000003A);

//Ottiene le informazioni necessarie allo smistamento
int num = 0;
int to_patch = 0;
var ins = m.Body.Instructions;

if (!findSwitchReflow(ins, ref to_patch, ref num, out var result))
{
        Console.WriteLine("Not found");
        return;
}


//Segue il flusso a partire dal primo ramo
var firstBranch = target(num, out int new_num);
result.AddRange(reflow(new_num, firstBranch, ins));

//Cambia tutte le istruzioni del metodo, usando quelle collezionate
m.Body.ExceptionHandlers.Clear();
m.Body.Instructions.Clear();
foreach (var i in result)
        m.Body.Instructions.Add(i);

//Scrive il modulo su un altro file
ModuleWriterOptions o = new ModuleWriterOptions(module);
o.MetadataOptions.Flags |= MetadataFlags.KeepOldMaxStack;
o.Logger = DummyLogger.NoThrowInstance;
module.Write(@"C:\users\labbe\desktop\output.exe", o);
```

Risultato

La funzione vista nella sezioni degli esempi adesso è completamente deoffuscata.



Il metodo deoffuscato, nella sua interezza.

```
internal unsafe static void
\u206E\u200C\u202D\u200D\u202B\u200D\u200F\u206C\u202E\u200C\u206B\u200C\u200D\u202D\u202D\u206E\u202B\u202C\u206C\u206E\u200F\u206F\u206C\u200F\u200B\u206B\u202E\u202E\u206E\u206D\u206F\u200E\u206C\u202C\u206E\u202A\u202A\u206C\u202D\u202E()
{
        Module module = typeof(<Module>).Module;
        string fullyQualifiedName = module.FullyQualifiedName;
        bool flag = fullyQualifiedName.Length > 0 && fullyQualifiedName[0] == '<';
        byte* ptr = (byte*)((void*)Marshal.GetHINSTANCE(module));
        byte* ptr2 = ptr + *(uint*)(ptr + 60);
        ushort num = *(ushort*)(ptr2 + 6);
        ushort num2 = *(ushort*)(ptr2 + 20);
        bool flag2 = false;
        uint* ptr3 = null;
        uint num3 = 0U;
        uint* ptr4 = (uint*)(ptr2 + 24 + num2);
        uint num4 = 3142338624U;
        uint num5 = 1855482562U;
        uint num6 = 3882886335U;
        uint num7 = 1147427922U;
        <Module>.CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref flag2);
        if (flag2)
        {
                Environment.FailFast(null);
        }
        for (int i = 0; i < (int)num; i++)
```

```
        {
                uint num8 = *(ptr4++) * *(ptr4++);
                if (num8 != 1367085993U)
                {
                        if (num8 != 0U)
                        {
                                uint* ptr5 = (uint*)(ptr + (UIntPtr)(flag ? ptr4[3] : ptr4[1]) / 4);
                                uint num9 = ptr4[2] >> 2;
                                for (uint num10 = 0U; num10 < num9; num10 += 1U)
                                {
                                        uint num11 = (num4 ^ *(ptr5++)) + num5 + num6 * num7;
                                        num4 = num5;
                                        num5 = num7;
                                        num7 = num11;
                                }
                        }
                }
                else
                {
                        ptr3 = (uint*)(ptr + (UIntPtr)(flag ? ptr4[3] : ptr4[1]) / 4);
                        <Module>.CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref flag2);
                        if (flag2)
                        {
                                Environment.FailFast(null);
                        }
                        num3 = (flag ? ptr4[2] : (*ptr4)) >> 2;
                }
                ptr4 += 8;
        }
        uint[] array = new uint[16];
        uint[] array2 = new uint[16];
        for (int j = 0; j < 16; j++)
        {
                array[j] = num7;
                array2[j] = num5;
                num4 = (num5 >> 5 | num5 << 27);
                num5 = (num6 >> 3 | num6 << 29);
                <Module>.CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref flag2);
                if (flag2)
                {
                        Environment.FailFast(null);
                }
                num6 = (num7 >> 7 | num7 << 25);
                num7 = (num4 >> 11 | num4 << 21);
        }
        array[0] = (array[0] ^ array2[0]);
        array[1] = array[1] * array2[1];
        array[2] = array[2] + array2[2];
        array[3] = (array[3] ^ array2[3]);
        array[4] = array[4] * array2[4];
        array[5] = array[5] + array2[5];
        array[6] = (array[6] ^ array2[6]);
        array[7] = array[7] * array2[7];
        array[8] = array[8] + array2[8];
        array[9] = (array[9] ^ array2[9]);
        array[10] = array[10] * array2[10];
        array[11] = array[11] + array2[11];
        array[12] = (array[12] ^ array2[12]);
        array[13] = array[13] * array2[13];
        array[14] = array[14] + array2[14];
        array[15] = (array[15] ^ array2[15]);
        uint num12 = 64U;

<Module>.\u202C\u200E\u200D\u200C\u202D\u200E\u202D\u206A\u202E\u206C\u200F\u202B\u202E\u206C\u206E\u202B\u206E\u
206B\u206C\u202B\u202A\u202A\u200C\u200C\u202E\u200D\u202B\u200B\u206B\u200B\u202E\u206A\u200C\u200D\u206C\u200B\
u206E\u202B\u200C\u202B\u202E((IntPtr)((void*)ptr3), num3 << 2, num12, ref num12);
        if (num12 == 64U)
        {
                return;
        }
        uint num13 = 0U;
        for (uint num14 = 0U; num14 < num3; num14 += 1U)
        {
                *ptr3 ^= array[(int)(num13 & 15U)];
                array[(int)(num13 & 15U)] = (array[(int)(num13 & 15U)] ^ *(ptr3++)) + 1035675673U;
                <Module>.CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref flag2);
                if (flag2)
                {
                        Environment.FailFast(null);
                }
                num13 += 1U;
        }
}
```