Reverzní inženýrství

4. Disassembling a obfuskace

Ing. Tomáš Zahradnický, EUR ING, Ph.D.
Ing. Martin Jirkal

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Verze 18. listopadu 2017

Obsah I

Obsah II

- Odstranění knihovních volání a programovacích idiomů
- Tabulková interpretace
- Přidání redundantních argumentů
- Paralelizace kódu

5 Obfuskační techniky — Transformace agregace

- Inlining
- Outlining
- Interleaving
- Klonování
- Transformace smyček

Disassembling

Disassembling

Disassembling je proces překladu binárního kódu do kódu v člověkem čitelném assembleru cílového CPU.

- Aplikace, která disassembling provádí, se nazývá disassembler.
- Disassembly (výsledek disassemblingu) se používá k provádění statické analýzy kódu a v rámci debuggeru i pro dynamickou (živou) analýzu.
- Disassembly:
  - obsahuje obrovské množství řádek kódu;
  - postrádá množství informace, která existovala v čase kompilace (názvy proměnných, ladící informace, názvy návěští), kvůli jejich odstranění během kompilace;
  - míchá kód a data — jak je můžeme rozlišit?
- K disassemblingu jsou dva základní přístupy — **lineární průchod** (linear sweep) a **rekurzivní průchod** (recursive traversal) [4]. Každý má své klady a zápory.

Lineární průchod

Lineární průchod

Lineární průchod disassembluje lineárně, bajt po bajtu, od začátku sekce `.text` až do jejího konce.

- Rychlý a jednoduchý.
- PowerPC má instrukce s pevnou délkou 4 B.
- x86 používá instrukce s proměnlivou délkou, 1–15 B. Teoreticky je možné vytvořit nekonečně dlouhé instrukce pomocí redundantních prefixů, ale cokoliv delšího než 15 B vyvolá výjimku.
  - `cc` — `int3`.
  - `cd 03` — `int 3`.
  - `f0 65 81 84 d8 78 56 34 12 ef cd ab 90` — `lock add dword ptr gs:[eax+ebx*8+12345678h], 90abcdefh`.
- Jakmile je jedna instrukce disasemblována, pokračuje se další instrukcí. Tento proces je opakován přes celou sekci `.text`.

Nejednoznačný tok kódu

Původní kód

```
      xor eax, eax
+-----jz good // Vždy skočit
|     _emit 0xf0
|     _emit 0x3e
|     _emit 0x81
|     _emit 0x84
v     _emit 0x98
good: mov eax, ebx
      mov eax, ebx
      mov eax, ebx
      mov eax, ebx
```

WinDbg 6.12.0002.633

```
00402cd8 33c0            xor eax,eax
00402cda 7405            je  Tokens!main+0xb1 (00402ce1)
00402cdc f03e8184988bc3  lock add dword ptr ds:[eax+ebx*4-3C743C75h],
           8bc38bc38bc3    0C38BC38Bh
```

IDA Pro 6.7

```
00402cd8 xor eax, eax
00402cda jz short near ptr loc_402CDC+5
00402cdc db 3Eh
00402cdc lock add dword ptr [eax+ebx*4-3C743C75h], 0C38BC38Bh
```

OllyDbg 2.01

```
00402cd8  33c0  xor eax,eax
00402cda  7405  je short 00402ce1
00402cdc  f0    db f0        // Správně!
00402cdd  3e    db 3e        // Správně!
00402cde  81    db 81        // Správně!
00402cdf  84    db 84        // Správně!
00402ce0  98    cwde         // Špatně!
00402ce1  8bc3  mov eax,ebx  // Správně!
00402ce3  8bc3  mov eax,ebx  // Správně!
00402ce5  8bc3  mov eax,ebx  // Správně!
00402ce7  8bc3  mov eax,ebx  // Správně!
```

Promíchaný kód & data

```
0:000> u 011d4780 L200
  011d4780 55              push ebp
  ...
  011d4838 83bdd0feffff03  cmp  dword ptr [ebp-130h],3
  011d483f 7727            ja   011d4868
  011d4841 8b8dd0feffff    mov  ecx,dword ptr [ebp-130h]
  011d4847 ff248d0c491d01  jmp  dword ptr ds:[011d490c+ecx*4]
  011d484e e85bc9ffff      call func0 (011d11ae)
  011d4853 eb13            jmp  011d4868
  011d4855 e82cc9ffff      call func1 (011d1186)
  011d485a eb0c            jmp  011d4868
  011d485c e820c9ffff      call func2 (011d1181)
  011d4861 eb05            jmp  011d4868
  011d4863 e8fbc8ffff      call func3 (011d1163)
  ...
  011d4868 33c0            xor  eax,eax
  ...
  011d4897 5d              pop  ebp
  011d4898 c21000          ret  10h
  011d489b 90              nop
  011d489c 05000000a4      add  eax,0A4000000h
  011d48a1 48              dec  eax
  011d48a2 1d01f4ffff      sbb  eax,0FFFFF401h
  011d490c 4e              dec  esi                          // Data v sekci .text!
  011d490d 48              dec  eax                          // Data v sekci .text!
  011d490e 1d0155481d      sbb  eax,1D485501h                // Data v sekci .text!
  011d4913 015c481d        add  dword ptr [eax+ecx*2+1Dh],ebx // Data v sekci .text!
  011d4917 016348          add  dword ptr [ebx+48h],esp      // Data v sekci .text!
  011d491a 1d01cccccc      sbb  eax,0CCCCCC01h               // Data v sekci .text!
```

Lineární průchod - pokračování

- Jak jsme viděli, lineární průchod slepě disassembloval jednu instrukci po druhé. To bylo funkční, pokud jsme s disassemblováním začali na začátku funkce (`push ebp` na `011d4780`). Výsledky byly zpočátku OK, později se ale disassembler nechal zmást:
  1. nejednoznačným tokem kódu;
  2. daty uvnitř kódu.
- Oba případy vedly na nesmyslnou disassembly!
  - → Disassembler používající poze linární průchod (WinDbg, gdb, objdump [3]) nedokáže odlišit kód od dat.

→ Lineární průchod je sám o sobě nedostatečný → potřebujeme robustnější metodu.

Rekurzivní průchod I

Rekurzivní průchod, na rozdíl od lineárního, disassembluje kód na základě sledování jeho toku. Průchod začíná typicky na vstupním bodu programu, kde se disassembluje jeho první instrukce. Po zpracování každé instrukce je adresa nesoucí tuto instrukci označena jako navštívená. Pokud navíc šlo o instrukci způsobující skok, zapamatuje se návratová adresa (pokud existuje) a analýza rekurzivně začíná na cílové adrese skoku. Poté, co byla analýza této větve dokončena, pokračuje analýza zapamatovanou návratovou adresou.

Tento přístup:

- je pomalý;
- rozlišuje kód a data:
  - všechny nenavštívené adresy jsou považovány za data

Rekurzivní průchod II

Rekurzivní průchod, na rozdíl od lineárního, disassembluje kód na základě sledování jeho toku. Průchod začíná typicky na vstupním bodu programu, kde se disassembluje jeho první instrukce. Po zpracování každé instrukce je adresa nesoucí tuto instrukci označena jako navštívená. Pokud navíc šlo o instrukci způsobující skok, zapamatuje se návratová adresa (pokud existuje) a analýza rekurzivně začíná na cílové adrese skoku. Poté, co byla analýza této větve dokončena, pokračuje analýza zapamatovanou návratovou adresou.

Tento přístup:

- je pomalý;
- částečně rozlišuje kód a data:
  - všechny nenavštívené adresy jsou považovány za data
  - ale — co tabulky skoků, nepřímo volané funkce, atd.

Rekurzivní průchod III

Rekurzivní průchod [4]

```
procedure Disassembluj( adresa, seznamInstr ) {
  if( adresa.navštívena )
    return;

  do {
    instr = DekódujInstr( adresa );
    adresa.navštívena = true;
    přidej instr do seznamInstr;
    if ( instr je větvení nebo volání funkce ) {
      T = množina možných následníků instr v toku kódu;
      for each cíl in T do {
        Disassembluj( cíl, seznamInstr);
      }
    }
    else
      adresa += instr.délka; /* adr. další instrukce */
  } while adresa je platná adresa instrukce;
}
```

Rekurzivní průchod — shrnutí

- **Rekurzivní průchod** je založen na předpokladu, že pro všechny instrukce v toku kódu dokážeme identifikovat všechny možné následníky této instrukce v toku.
- Tok kódu je sledován a všechny adresy navštívené **rekurzivním průchodem** jsou označeny jako kód.
- Jak spočítat možné toky kódu, pokud narazíme na nepřímý skok (skok na adresu danou tabulkou skoků, vypočítané goto nebo volání prostřednictvím VMT)? S CFG by to bylo snadné, ale my ho nemáme!
- Co dělat se zbytkem kódového segmentu, který nebyl navštíven?
  - **Rekurzivní průchod** se ani nepokusí tuto část disassemblovat a nedokáže o ní bez dalších analýz nic říci.
  - **Lineární průchod** by ji disassembloval, některé části dobře, jiné ne, protože představují data.
- **Rekurzivní průchod** má komplementární síly a slabiny ve srovnání s **lineárním průchodem**. Který tedy máme použít? Pokusme se vylepšit oba algoritmy.

Rozšířený lineární průchod

Lineární průchod se nedokázal vypořádat s daty vloženými do sekce `.text`, např. skokovými tabulkami nebo řetězcovými konstantami. Taková data zůsobovala chyby v disassembly v podobě nesprávně identifikovaných instrukcí. **Lineární průchod** ale můžeme vylepšit:

Myšlenka [4]

Pokud máme k dispozici relokační údaje, pak musí mít tabulky skoků v sekci `.text` záznamy v tabulce relokací. Podle těchto záznamů můžeme tabulky skoků rozpoznat a označit je jako data.

Každá záznam a_i v tabulce skoků musí splňovat [4]:

1. paměťové lokace, v nichž a_i leží, jsou označeny jako relokovatelné; a
2. sama adresa a_i musí ležet v sekci `.text`.

Tabulka skoků v programu I

Ověřme si navrženou myšlenku následujícím programem:

Vzorová tabulka skoků

```
switch (intval) {
  case 0:
    func0();
    break;
  case 1:
    func1();
    break;
  case 2:
    func2();
    break;
  case 3:
    func3();
    break;
}
```

Vzorová tabulka skoků (assembler)

```
.text:00414838                cmp  [ebp+intval], 3
.text:0041483F                ja   short loc_414868
.text:00414841                mov  ecx, [ebp+intval]
.text:00414847                jmp  ds:off_41490C[ecx*4]
...
.text:0041484E loc_41484E:
.text:0041484E                call func0
.text:00414853                jmp  short loc_414868
.text:00414855 loc_414855:
.text:00414855                call func1
.text:0041485A                jmp  short loc_414868
...
--------- ZAČÁTEK dat v sekci .text -----------
.text:0041490C off_40C5E0  dd offset loc_41484E // case 0
.text:00414910             dd offset loc_414855 // case 1
.text:00414914             dd offset loc_41485C // case 2
.text:00414918             dd offset loc_414863 // case 3
--------- KONEC dat v sekci .text -------------

...
```

Tabulka skoků v programu II

Když si vypíšeme sekci (`.reloc`) nástrojem `dumpbin`, dostaneme:

```
C:\...\> dumpbin /RELOCATIONS jumptable.exe
...
   14000 RVA,       8C SizeOfBlock
     ...  ...           ...
     90C  HIGHLOW            0041484E
     910  HIGHLOW            00414855
     914  HIGHLOW            0041485C
     918  HIGHLOW            00414863
```

```
--------- ZAČÁTEK dat v sekci .text -----------
.text:0041490C off_40C5E0  dd offset loc_41484E // case 0
.text:00414910             dd offset loc_414855 // case 1
.text:00414914             dd offset loc_41485C // case 2
.text:00414918             dd offset loc_414863 // case 3
--------- KONEC dat v sekci .text -------------
```

V relokační tabulce jsou 4 záznamy relokující 4 po sobě jdoucí ukazatele v sekci `.text`. Protože část x86 instrukce mohou tvořit max. 2 z nich, zbylé dva (adr. 00414914 a 00414918) musí být data. První dvě položky by mohly být kódem.

Pozn.: EXE soubory nepoužívající ASLR normálně sekci `.reloc` vůbec nemají, na rozdíl od DLL. Pokud je ASLR pro program povoleno, musí být binárka relokovatelná a tudíž mít sekci `.reloc`, podobně jako v příkladu výše.

Rozšířený lineární průchod — pokračování

Jako data jsme identifikovali pouze 2 ze 4 položek. Můžeme ale sekci `.text` rozčlenit do segmentů oddělených tabulkami skoků, disassemblovat veškerý kód v těchto segmentech a ověřovat, zda byla každá instrukce úspěšně disassemblována a zda nepřetekla do dat mezi segmenty. V případě přetečení nejde o instrukci, ale o data!

Rozšířený lineární průchod [4]

1. Označ všechny tabulky skoků jako data, kromě prvních dvou záznamů v každé tabulce (ty mohou na x86 být součástí instrukce).
2. Pro každou posloupnost neoznačených adres v sekci `.text`:
   a) Použij **lineární průchod**, zastav při dosažení označené adresy.
   b) Pokud poslední instrukce přetekla do označené adresy, označ ji jako data.
   c) Prozkoumej poslední úspěšně disassemblovanou instrukci a v případě potřeby ji označ jako data.

Hybridní přístup

Rekurzivní průchod má komplementární síly a slabiny vůči **lineárnímu průchodu**. Zkusme zkombinovat to nejlepší z obou přístupů [4] a:

- Použijme **rozšířený lineární průchod** pro prvotní disassemblování;
- Použijme **rekurzivní průchod** pro ověření každé nalezené funkce.

Ověření probíhá následujícím postupem:

1. Na každou instrukci funkce použij **rekurzivní průchod**.
2. Ověř každou instrukci I, získanou na adrese a_I. Pokud nebyla získána také **lineárním průchodem**, vyvolej chybu.
3. Pokud ve funkci nedošlo k chybě, označ funkci za správnou.

Obfuskace — motivace

Motivací pro použití obfuskačních technik je utajení vlastního kódu a ztížení jeho porozumění. Běžný software tyto techniky používá pro zabránění reverzního inženýrství schémat sériových čísel, prolomení ochrany proti kopírování, vynucení použití hardwarových donglů, zajištění DRM, atd. Malware je používá pro zabránění analýze svého kódu.

Cílem těchto technik je:

1. zmást disassembler (statickou analýzu);
2. zmást debugger (dynamickou analýzu);
3. zmást decompiler;
4. zmást člověka, který výše uvedené nástroje používá.

Mezi obfuskační techniky zahrnujeme obfuskaci kódu, transformace toku kódu, šifrování, používání kontrolních součtů, detekci debuggeru, antidebugovací kód, atd. Čím delší čas si analýza vyžádá, tím méně lidí ji bude dělat :-).

Kategorizace ochran [1]

Ochrana duševního vlastnictví může být členěna na:

- právní
- technickou
  - obfuskace
  - šifrování
  - (částečný) běh na serveru
  - důvěryhodný nativní kód

Na transformační cíl můžeme aplikovat:

- obfuskaci rozložení
- obfuskaci řízení
- obfuskaci dat
- preventivní transformaci

Obfuskační metriky I

Obfuskace

Obfuskace činí objekt nesrozumitelným. V počítačovém softwaru jí rozumíme proces změny transformačního cíle tak, aby mu bylo obtížné rozumět, ale zůstala zachována jeho funkcionalita.

Transformační cíl

Transformačním cílem rozumíme objekt, který má být obfuskován. Tímto cílem může být zdrojový kód programu, ale také program v binární podobě.

Příklady transformačních cílů:

- javascriptový kód ve vašich HTML stránkách;
- javové třídy a archívy, zdrojové soubory i spustitelný kód .NET;
- zdrojový kód v PHP, C/C++, ...;
- spustitelné programy.

Obfuskační metriky II

Obfuskovaným produktem rozumíme transformační cíl poté, co na něj byly aplikovány obfuskující transformace. Každá z těchto transformací by měla, vedle zachování původního chování programu, být **potentní**, **odolná**, a její **cena** by měla být co nejnižší.

Potence [1]

Potence $\mathcal{T}_{pot}(P)$ měří, jak moc transformace programu P na obfuskovaný program P' zmate lidského čtenáře:

$$\mathcal{T}_{pot}(P) = E(P')/E(P) - 1,$$

kde $E(P)$ je složitost P podle **nějaké** metriky.

Můžeme použít metriky jako **délka programu**, **cyklomatická složitost**, **složitost vnoření**, **složitost datového toku**, . . . ; obfuskace pak zvyšuje složitost toho, co daná metrika měří.

Obfuskační metriky III

Odolnost (resilience) [1]

Odolnost měří, jak obtížné je odstranit transformaci automatickým deobfuskátorem.

$$\mathcal{T}_{res} = \text{Resilience}(\mathcal{T}_{prace\ deobfuskatoru}, \mathcal{T}_{prace\ programatora}),$$

kde $\mathcal{T}_{prace\ programatora}$ vyjadřuje množství času potřebné pro vytvoření automatického deobfuskátoru, který sníží $\mathcal{T}_{pot}$, a $\mathcal{T}_{prace\ deobfuskatoru}$ vyjadřuje dobu běhu a velikost prostoru, které auto-deobfuskátor použije pro snížení $\mathcal{T}_{pot}$.

Odolnost $\mathcal{T}_{res}$ může být: **triviální**, **slabá**, **silná**, **plná**, nebo **jednosměrná**.

$\mathcal{T}_{pr.\ deob.} \setminus \mathcal{T}_{pr.\ prog.}$	**Lokální**	**Globální**	**Meziprocedurová**	**Meziprocesová**
Polynomiální	**triviální**	**slabá**	**silná**	**plná**
Exponenciální	**slabá**	**silná**	**plná**	**plná**

Tabulka : Odolnost($\mathcal{T}_{prace\ deobfuskatoru}, \mathcal{T}_{prace\ programatora}$) [1].

Obfuskace rozložení (Layout Obfuscations) I

Obfuskace rozložení [1] se zaměřuje na lexikální strukturu aplikace. Ta zahrnuje formátování zdrojového kódu, komentáře, jména proměnných a funkcí, rozložení tříd a datových struktur, atd.

Před obfuskací:

```
static public float PrůměrnáZnámkaStudenta(Student student) {
  float známka = 0.0f; float kredity = 0.0f;
  /* Známka spočítána jako vážený průměr g=sum(známka_i*kredity_i)/sum(kredity_i) */
  if( student.Předměty().Empty() ) return 0.0f;
  for_each(předmět in student.Předměty()) {
    známka += předmět.Kredity() * předmět.Známka();
    kredity += předmět.Kredity();
  }
  return známka/kredity;
}
```

Po obfuskaci (odstraněny komentáře, typy, metody a třídy přejmenovány):

```
static public float a(a b) {
  float c = 0.0f; float d = 0.0f;
  if( b.a().a() ) return 0.0f;
  for_each(e in b.a()) {
    c += e.a() * e.b();
    d += e.a();
  }
  return c/d;
}
```

Obfuskace rozložení (Layout Obfuscations) II

Tranformace	Potence	Odolnost	Cena
Přejmenování identifikátorů	střední	jednosměrná	nulová
Změna formátování	nízká	jednosměrná	nulová
Odstranění komentářů	vysoká	jednosměrná	nulová
Odstranění ladících informací	vysoká	jednosměrná	nulová

Tabulka : Kvalita obfuskací rozložení [1].

Transformace řízení

Transformace řízení obfuskují tok kódu. Můžeme je rozčlenit na tranformace **pořadí**, **agregace**, a **výpočtu** [1].

- **Transformace agregace** rozdělují výpočty, které patří k sobě, a spojují ty, které k sobě nepatří.
- **Transformace pořadí** znáhodňují pořadí výpočtů.
- **Transformace výpočtu** přidávají zbytečný a/nebo mrtvý kód nebo provádějí změny v algoritmech.

Mnohé z obfuskací řízení závisí na použití **neprůhledných proměnných** (opaque variables) a **neprůhledných predikátů** (opaque predicates).

Neprůhledné predikáty a proměnné I

Proměnnou nebo predikát označíme jako neprůhledné, pokud je jejich hodnota nebo vlastnost známa obfuskujícímu v době obfuskování, ale pro deobfuskátora je obtížné ji odhadnout [1].

Neprůhledný konstrukt se skládá z příkazu **if** testujícího hodnotu neprůhledného predikátu nebo proměnné `ovar` na specifickou hodnotu `VAL`, která byla známa v době obfuskování:

```
if( ovar == VAL )
  nikdy nenastane
else
  něco udělej
```

```
if( ovar == VAL )
  něco udělej
else
  nikdy nenastane
```

```
if( ovar == VAL )
  něco udělej
else
  něco udělej
```

Naším hlavním cílem je zmást jak disassembler, tak člověka, který kód studuje. Podívejme se nejprve na disassembler.

Neprůhledné predikáty a proměnné II

Lineární průchod neumí rozlišit kód od dat, a proto je zranitelný daty (smetí, tabulky skoků, . . .) umístěnými uvnitř funkcí nebo mezi nimi. Využijeme této skutečnost k naplnění nikdy neprováděné větve neprůhledného predikátu s nesmyslnými daty. Ta můžeme vytvořit uvnitř bloku `_asm` klíčovým slovem `_emit`:

```
_asm {
    xor     eax, eax // Neprůhledná proměnná v EAX
    cmp     eax, 1   // Zbytečné, mohli jsme rovnou použít ZF nastavený xor-em
    jnz     pokracuj // Vždy skočí

  nikdy_nenastane:
    _emit   0xe9     // E9 je kód instrukce jmp, následují 4 B adresy

  pokracuj:
    imul    edx
}
```

Klíčové slovo `_emit` je použito ke vložení instrukce `jmp` do binárky. Disassemblery používající **lineární průchod** pak typicky pochopí instrukci `imul` jako část operandu instrukce `jmp`.

Neprůhledné predikáty a proměnné III

```
  216 00f0482f 33c0            xor     eax,eax
  217 00f04831 83f801          cmp     eax,1
  218 00f04834 7501            jne     HelloWorld!wWinMain+0xb7 (00f04837)

HelloWorld!wWinMain+0xb6 [c:\users\tomas zahradnicky\documents\visual stud
  219 00f04836 e9f7ea8b45      jmp     467c3332

HelloWorld!wWinMain+0xb7 [c:\users\tomas zahradnicky\documents\visual stud
  220 00f04837 f7ea            imul    edx
```

overlap

Obrázek : WinDbg 6.12.0002.633

```
00F0482F  33C0          XOR EAX,EAX
00F04831  83F8 01       CMP EAX,1
00F04834 v75 01         JNZ SHORT HelloWor.00F04837
00F04836 -E9 F7EA8B45   JMP 467C3332
00F0483B  E4 89         IN AL,89
00F0483D  85D0          TEST EAX,EDX
00F0483F  FE            ???
00F04840  FFFF          ???
```

Obrázek : OllyDbg 1.10

```
00F0482F   33C0          XOR EAX,EAX
00F04831   83F8 01       CMP EAX,1
00F04834   75 01         JNE SHORT 00F04837
00F04836   E9            DB E9
00F04837   F7EA          IMUL EDX
```

Obrázek : OllyDbg 2.01

Skutečnost, že kód ve větvi `nikdy_nenastane` nebude nikdy spuštěn, je snadno detekovatelná **rekurzivním průchodem**.

Neprůhledné predikáty a proměnné IV

```
.text:0041482F                  xor     eax, eax
.text:00414831                  cmp     eax, 1
.text:00414834                  jnz     short near ptr loc_414836+1
.text:00414836
.text:00414836 loc_414836:                              ; CODE XREF:
.text:00414836                  jmp     near ptr 45CD3332h
.text:00414836 ; ---------------------------------------------------
.text:0041483B                  db 0E4h
```

Obrázek : IDA Pro 6.7 původní.

```
.text:0041482F                  xor     eax, eax
.text:00414831                  cmp     eax, 1
.text:00414834                  jnz     short loc_414837
.text:00414834 ; ---------------------------------------------------
.text:00414836                  db 0E9h
.text:00414837 ; ---------------------------------------------------
.text:00414837
.text:00414837 loc_414837:
.text:00414837                  imul    edx
```

Obrázek : IDA Pro 6.7 po opravě.

Pozn.: I když tato transformace může disassembler zmást, je triviálně opravitelná člověkem. Potence je tudíž **nízká**. Použitý konstrukt může být odstraněn automatickým deobfuskátorem v polynomiálním čase v rámci lokální procedury. Z těchto důvodů je odolnost **triviální**. Cena je ovšem **levná**, což činí tuto techniku tak populární.

Příklad I

Použijeme nyní **neprůhledný predikát** ke zmatení jak lineárních, tak rekurzivních disassemblerů. Kód uvedený níže si přečte ze zásobníku security cookie 1 a zxoruje ho s hodnotou ebp 2. Obdobný kód nalezneme na konci funkcí chráněných kanárkem. Náš kód ho používá jako neprůhlednou hodnotu.

```
#define OBFUSKUJ(_random)                          \
  _asm{ mov eax, dword ptr [ebp-4] };     \ // 1
  _asm{ xor eax, ebp };                   \ // 2
  _asm{ jnz _label##_random };            \ // 3
  _asm{ _emit 0x70+ (_random&0xF) };      \ // 4
  _asm{ _emit 0x00 };                     \ // 5
  _asm{ _emit 0xe8+ ((_random>>4)&0x3) }; \ // 6
  _asm{ _label##_random: }                  // 7
```

Výsledek xoru na řádku 2 by teoreticky mohl být nula, ale není to pravděpodobné ($1 : 2^{32}$). Když není nula 3, pokračujeme řádkem 7. To je standardní tok kódu. Disassembler nedokáže určit, zda je podmínka vždy pravdivá nebo nepravdivá, protože je počítána za běhu. `_emit` na řádku 4 vytvoří náhodnou instrukci `jxx` skákající `0x00` bajtů za sebe 5, tj. na začátek dalšího `_emit` na řádku 6. Tento `_emit` vytvoří první bajt náhodné instrukce `jmp` nebo `call`. Zbylé její bajty se vezmou z instrukcí následujících za makrem.

Makro `OBFUSKUJ` přijímá hodnotu `_random` jako argument, aby při každém použití generovalo poněkud odlišný kód. Celkem může vytvořit 64 různých kombinací.

Příklad II

Podívejme se, jak tato technika zafunguje na tomto kódu v C:

```
OBFUSKUJ(238);
cbSelf.QuadPart = 0;

OBFUSKUJ(372);
MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);

OBFUSKUJ(118);
CreateConsole();

OBFUSKUJ(235);
```

Příklad II

Podívejme se, jak tato technika zafunguje na tomto kódu v C:

```
OBFUSKUJ(238);
cbSelf.QuadPart = 0;

OBFUSKUJ(372);
MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);

OBFUSKUJ(118);
CreateConsole();

OBFUSKUJ(235);
```

```
010DCC74   . 8B45 FC            MOV EAX,DWORD PTR SS:[EBP-4]
010DCC77   . 33C5               XOR EAX,EBP
010DCC79   . 75 03              JNZ SHORT HelloWor.010DCC7E
010DCC7B   . 7E 00              JLE SHORT HelloWor.010DCC7D
010DCC7D   > EA 0F57C066 0F13   JMP FAR 130F:66C0570F
010DCC84   ? 45                 INC EBP
010DCC85   ? EC                 IN AL,DX
010DCC86   . 8B45 FC            MOV EAX,DWORD PTR SS:[EBP-4]
010DCC89   . 33C5               XOR EAX,EBP
010DCC8B   . 75 03              JNZ SHORT HelloWor.010DCC90
010DCC8D   . 74 00              JE SHORT HelloWor.010DCC8F
010DCC8F     EB                 DB EB
010DCC90   . 8D45 EC            LEA EAX,DWORD PTR SS:[EBP-14]
010DCC93   . 50                 PUSH EAX
010DCC94   . 8D45 10            LEA EAX,DWORD PTR SS:[EBP+10]
010DCC97   . 50                 PUSH EAX
010DCC98   . 8D55 FC            LEA EDX,DWORD PTR SS:[EBP-4]
010DCC9B   . 8D4D F8            LEA ECX,DWORD PTR SS:[EBP-8]
010DCC9E   . E8 1D010000        CALL HelloWor.010DCDC0
010DCCA3   . 83C4 08            ADD ESP,8
010DCCA6   . 8B45 FC            MOV EAX,DWORD PTR SS:[EBP-4]
010DCCA9   . 33C5               XOR EAX,EBP
010DCCAB   . 75 03              JNZ SHORT HelloWor.010DCCB0
010DCCAD   . 76 00              JBE SHORT HelloWor.010DCCAF
010DCCAF     EB                 DB EB
010DCCB0   . E8 EB010000        CALL HelloWor.010DCEA0
```

Příklad II

Podívejme se, jak tato technika zafunguje na tomto kódu v C:

```
OBFUSKUJ(238);
cbSelf.QuadPart = 0;

OBFUSKUJ(372);
MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);
```

```
OBFUSKUJ(118);
CreateConsole();
OBFUSKUJ(235);
```

```
00C1CC74  .  8B45 FC       MOV EAX,DWORD PTR SS:[EBP-4]
00C1CC77  ?  33C5          XOR EAX,EBP
00C1CC79  .  75 03         JNE SHORT 00C1CC7E
00C1CC7B  .- 7E 00         JLE SHORT 00C1CC7D
00C1CC7D  .- EA 8D45EC0F ! JMP FAR C057:0FEC458D
00C1CC84  ?  50            DB 50
00C1CC85     8D            DB 8D
00C1CC86     45            DB 45
00C1CC87     10            DB 10
00C1CC88     66            DB 66
00C1CC89     0F            DB 0F
00C1CC8A     13            DB 13
00C1CC8B     45            DB 45
00C1CC8C     EC            DB EC
00C1CC8D  .  508D55FC      DD FC558D50
00C1CC91     8D            DB 8D
00C1CC92     4D            DB 4D
00C1CC93     F8            DB F8
00C1CC94     E8            DB E8
00C1CC95     F7            DB F7
```

Příklad II

Podívejme se, jak tato technika zafunguje na tomto kódu v C:

```
OBFUSKUJ(238);
cbSelf.QuadPart = 0;

OBFUSKUJ(372);
MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);

OBFUSKUJ(118);
CreateConsole();

OBFUSKUJ(235);
```

```
.text:0040CC74                  mov     eax, [ebp-4]
.text:0040CC77                  xor     eax, ebp
.text:0040CC79                  jnz     short near ptr loc_40CC7D+1
.text:0040CC7B                  jle     short $+2
.text:0040CC7D
.text:0040CC7D loc_40CC7D:                              ; CODE XREF: .text:0040CC7B↑j
.text:0040CC7D                                          ; .text:0040CC79↑j
.text:0040CC7D                  jmp     far ptr 0C057h:0FEC458Dh
.text:0040CC7D ; ---------------------------------------------------------------------------
.text:0040CC84                  dd 10458D50h, 45130F66h, 558D50ECh, 0F84D8DFCh, 0F7E8h
.text:0040CC84                  dd 1D2E800h, 0A6A0000h, 50F4458Dh, 5045E856h, 758BFFFFh
.text:0040CC84                  dd 14C48310h, 5589CE8Bh, 0E8F88BE8h, 224h, 774F685h
.text:0040CCC0                  db 56h, 0FFh, 15h
.text:0040CCC3                  dd offset UnmapViewOfFile
.text:0040CCC7                  db 8Bh
.text:0040CCC8                  dd 358BFC45h
.text:0040CCCC                  dd offset CloseHandle
.text:0040CCD0                  dd 74FFF883h, 0D6FF5003h, 83F8458Bh, 374FFF8h, 0EBD6FF50h
.text:0040CCD0                  dd 0F8C1E901h, 83C03302h, 17501F8h, 83EAF7E9h, 2E7703FFh
.text:0040CCF8 ; ---------------------------------------------------------------------------
```

Příklad II

Podívejme se, jak tato technika zafunguje na tomto kódu v C:

```
OBFUSKUJ(238);
cbSelf.QuadPart = 0;

OBFUSKUJ(372);
MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);

OBFUSKUJ(118);
CreateConsole();

OBFUSKUJ(235);
```

Naše snaha sice zmátla debuggery s **lineárním** i **rekurzivním průchodem** a možná i některé lidi, není ale o mnoho **potentnější** než předchozí verze. Také její **odolnost** je jen mírně lepší; jako neprůhlednou hodnotu používáme kanárka, mohli bychom použít i návratovou adresu, a ani jedno by statické analyzátory neměly dokázat předpovědět, pokud ovšem nepochopí, co skutečně testujeme. **Odolnost** je tedy **slabá**. Jak bychom ji mohli zvýšit?

Transformace výpočtu

Transformace výpočtu patří mezi transformace toku, které **skrývají skutečný tok kódu** za **irelevantními instrukcemi**, přidávají kód, pro který ve vyšším jazyku není vyjádření, **vkládají mrtvý kód**, nebo odstraňují skutečné abstrakce toku či vkládají falešné [1]. To zahrnuje:

- **Vkládání mrtvého nebo nerelevantního kódu**
- **Rozšíření podmínek cyklů**
- **Konverzi reducibilního CFG na nereducibilní**
- **Odstranění knihovních volání a programovacích idiomů**
- **Tabulkovou interpretaci**
- **Přidávání redundantních argumentů**
- **(Nesmyslnou) paralelizaci kódu**

Vkládání mrtvého nebo nerelevantního kódu I

Tyto transformace přidávají kód, který není relevantní pro původní program, aby zmátly čtenáře. Toho dosahují použitím neprůhledných predikátů nebo přidáním kódu, který s původní funkcionalitou vůbec nesouvisí.

Neprůhledné predikáty můžeme vkládat kamkoliv v bloku kódu. Pro další ztížení analýzy můžeme dát do obou větví predikátu stejný kód, ale chráněný odlišnými transformacemi, případně do nesprávné větve dát kód podobný, ale dělající něco (mírně) odlišného.

Nerelevantní kód může například provádět totožný výpočet, ale jeho výsledek ukládat do jiné proměnné.

Vkládání mrtvého nebo nerelevantního kódu II

Příklad použití této techniky už jsme viděli:

```
OBFUSKUJ(238);
cbSelf.QuadPart = 0;

OBFUSKUJ(372);
MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);

OBFUSKUJ(118);
CreateConsole();
```

Kód vložený makrem `OBFUSKUJ` používal neprůhlednou proměnnou, kanárka, k vytvoření mrtvé větve, která nikdy nebude spuštěna.

Další příklad:

```
if( oval == VAL )            /* neprůhledná proměnná */
  cbSelf.QuadPart = 0;
else                         /* mrtvý */
  cbSelf.QuadPart = 1;       /* mrtvý */

if( cbSelf.QuadPart == 0 )   /* vždy pravda */
  MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);

CreateConsole();
```

Rozšíření podmínek cyklů

Rozšíření podmínek cyklů komplikuje rozhodnutí o ukončení smyčky zahrnutím neprůhledného predikátu nebo proměnné do ukončovací podmínky:

```
char array[10];
unsigned int i;

for( i=0; i<10; ++i )
{
  array[i] = i;
}
```

```
.text:0040CDA0 sub_40CDA0     xor  eax, eax
.text:0040CDA2 loc_40CDA2:    mov  [eax+ecx], al
.text:0040CDA5                inc  eax
.text:0040CDA6                cmp  eax, 0Ah
.text:0040CDA9                jb   short loc_40CDA2
.text:0040CDAB                retn
```

Po transformaci:

```
char array[10];
unsigned int i , j, k ;

for( i=0 , j=1, k=0 ;
   (k<170) && ((j%2)==1) ;
   ++i , k+=17  )
{
  array[i] = i;
   j|=k;
}
```

```
.text:0040CD70 sub_40CD70     push esi
.text:0040CD71                mov  esi, ecx
.text:0040CD73                xor  edx, edx       // i
.text:0040CD75                mov  ecx, 1         // j
.text:0040CD7A                xor  eax, eax       // k
.text:0040CD7C                lea  esp, [esp+0]   // align no-op
.text:0040CD80 loc_40CD80:    test cl, 1
.text:0040CD83                jz   short loc_40CD95
.text:0040CD85                or   ecx, eax
.text:0040CD88                mov  [edx+esi], dl
.text:0040CD8B                add  eax, 11h
.text:0040CD8E                inc  edx
.text:0040CD8F                cmp  eax, 0AAh
.text:0040CD94                jb   short loc_40CD80
.text:0040CD96 loc_40CD95:    pop  esi
.text:0040CD97                retn
```

Konverze reducibilního CFG na nereducibilní I

Cílem této transformace je zesložitit graf toku kódu a zabránit dekompilaci.

- Graf toku funkce je reducibilní, pokud používá pouze strukturované příkazy, např. `for`-smyčky, `do-while`-smyčky, `while`-smyčky, `if`-then(-`else`), `break`, a `continue`.
- Jakmile začneme používat `goto` — snadné v C/C++, nemožné v Javě (ale možné v Javovém bytekódu (!)) — stane se graf toku nereducibilním.
- Nereducibilní grafy toku vedou na odpudivý kód vyšší úrovně. To je zvlášť dobře viditelné v Javě, kde klíčové slovo `goto` není k dispozici.
- Do této kategorie spadá také používání konstruktů, pro které jazyk vyšší úrovně nemá vyjádření.

Konverze reducibilního CFG na nereducibilní II

Začneme s jednoduchým programem v Javě a podíváme se na jeho bytekód [5]:

```
public class a
{
  static public void main(String[] args)
  {
    int i;

    for(i=0; i<10; ++i)
    {
      System.console().writer().
        println("Krok: " + i);
    }
  }
}
```

```
.method public static main([Ljava/lang/String;)V
.limit stack 3
.limit locals 2
  iconst_0
  istore_1 ; met002_slot001

met002_2:
  iload_1 ; met002_slot001
  bipush 10
  if_icmpge met002_42

  invokestatic java/lang/System.console()Ljava/io/Console;
  invokevirtual java/io/Console.writer()Ljava/io/PrintWriter;

  new java/lang/StringBuilder
  dup
  invokespecial java/lang/StringBuilder.<init>()V
  ldc "Krok: "
  invokevirtual java/lang/StringBuilder.append(Ljava/lang/Str
    ing;)Ljava/lang/StringBuilder;

  iload_1 ; met002_slot001
  invokevirtual java/lang/StringBuilder.append(I)Ljava/lang/StringBui
  invokevirtual java/lang/StringBuilder.toString()Ljava/lang/String;
  invokevirtual java/io/PrintWriter.println(Ljava/lang/String;)V
  iinc 1 1
  goto met002_2

met002_42:
```

Konverze reducibilního CFG na nereducibilní III

```
.method public static main([Ljava...
.limit stack 3
.limit locals 2
  iconst_0
  istore_1 ; met002_slot001

met002_2:
  iload_1 ; met002_slot001
  bipush 10
  if_icmpge met002_42

  invokestatic java/lang/System.con...
  invokevirtual java/io/Console.wri...
  new java/lang/StringBuilder
  dup
  invokespecial java/lang/StringBui...
  ldc "Krok: "
  invokevirtual java/lang/StringBui...
  iload_1 ; met002_slot001
  invokevirtual java/lang/StringBui...
  invokevirtual java/lang/StringBui...
  invokevirtual java/io/PrintWriter...
  iinc 1 1
  goto met002_2

met002_42:
  return
.end method
```

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
import java.io.Console;
import java.io.PrintWriter;

public class a
{
  public a() { }

  public static void main(String args[])
  {
    for(int i = 0; i < 10; i++)
      System.console().writer().println((new StringBuilder()).
        append("Krok: ").append(i).toString());
  }
}

// Decompiled by JD-GUI 1.0.0, JD-Core 0.7.1
// JD home page: http://jd.benow.ca/
import java.io.Console;
import java.io.PrintWriter;

public class a
{
  public static void main(String[] paramArrayOfString)
  {
    for (int i = 0; i < 10; i++) {
      System.console().writer().println("Krok: " + i);
    }
  }
}
```

Konverze reducibilního CFG na nereducibilní IV

Jak jsme viděli, dekompilovaný výsledek byl velmi přesný. Nyní zkusíme přidat neprůhledný predikát a příkaz `goto`. Predikát zavolá metodu `java.util.Random.nextInt()`, aby načetl náhodné číslo. Výsledek bude vždy v intervalu [0, 9].

```
new java/util/Random                          ; Vytvoř novou instanci třídy
dup                                           ; Zdvoj ji na vrcholu zásobníku
invokespecial java/util/Random.<init>()V      ; Volej ctor, výsledek je void
bipush 10                                     ; Dej bajt na zásobník
invokevirtual java/util/Random.nextInt(I)I    ; Volej nextInt(10), výsledek je int
```

Na vrcholu zásobníku je nyní náhodné číslo. Porovnáme ho s 10.

```
bipush 10                                     ; Dej bajt na zásobník
if_icmpge never                               ; If random >= 10 GOTO never
```

V první iteraci otestujeme jinou horní mez:

```
iload_1
bipush 12
goto test_bounds
```

Konverze reducibilního CFG na nereducibilní V

```
.method public static main([Ljava/lang/String;)V
.limit stack 4
.limit locals 2
  iconst_0
  istore_1 ; met002_slot001

  new java/util/Random
  dup
  invokespecial java/util/Random.<init>()V
  bipush 10
  invokevirtual java/util/Random.nextInt(I)I
  bipush 10
  if_icmpge never
  iload_1
  bipush 12
  goto test_bound

loop_start:
  iload_1
  bipush 10
test_bound:
  if_icmpge met002_42
  invokestatic java/lang/System.console()Ljava/io/Console;
  invokevirtual java/io/Console.writer()Ljava/io/PrintWriter;
  new java/lang/StringBuilder
  dup
  invokespecial java/lang/StringBuilder.<init>()V
  ldc "Krok: "
  invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
  iload_1
```

Konverze reducibilního CFG na nereducibilní VI

```
    invokevirtual java/lang/StringBuilder.append(I)Ljava/lang/StringBuilder;
    invokevirtual java/lang/StringBuilder.toString()Ljava/lang/String;
    invokevirtual java/io/PrintWriter.println(Ljava/lang/String;)V
  never:
    iinc 1 1
    goto loop_start

  met002_42:
    return

  .end method

// Decompiled by JD-GUI 1.0.0, JD-Core 0.7.1
// JD home page: http://jd.benow.ca/
import java.io.Console;
import java.io.PrintWriter;
import java.util.Random;

public class b
{
  public static void main(String[] paramArrayOfString)
  {
    int i = 0;
    if (new Random().nextInt(10) < 10)
    {
      tmpTernaryOp = 12;
      System.console().writer().println("Krok: " + i);
    }
    i++;
  }
}
```

Konverze reducibilního CFG na nereducibilní VII

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
import java.io.Console;
import java.io.PrintWriter;
import java.util.Random;
public class b
{
  public b() { }

  public static void main(String args[])
  {
    int i = 0;
    if((new Random()).nextInt(10) >= 10) goto _L2; else goto _L1
  _L1:
    i; 12; goto _L3
  _L7:
    i; 10;
  _L3:
    JVM INSTR icmpge 65; goto _L4 _L5
  _L4:
    break MISSING_BLOCK_LABEL_31;
  _L5:
    break; /* Loop/switch isn't completed */
    System.console().writer().println((new StringBuilder()).append("Krok: ").append(i).toString());
  _L2:
    i++; if(true) goto _L7; else goto _L6
  _L6:
  }
}
```

Konverze reducibilního CFG na nereducibilní VIII

Dekompilátory se nechaly zmást použitím příkazu goto. Když zkusíme to samé udělat v C, zjistíme, že v něm tato metoda moc účinná není:

```
__declspec(noinline)
void LoopsIred1(volatile char* array) {
  int i;

  _asm{
    mov esi, array
    mov edi, esi
  }

  // Opaque predicate, always false
  if ((__security_cookie
       ^__security_cookie_complement) != 0)
    goto insideloop;

  for (i = 0; i < 10; ++i) {
    array[i] = i;

    insideloop:
    _asm {
      lodsb
      ror al,1
      stosb
    }
  }
}
```

```
.text:0040CDC0  sub  esp, 8
.text:0040CDC3  push esi
.text:0040CDC4  push edi
.text:0040CDC5  mov  [esp+10h+var_8], ecx
.text:0040CDC9  mov  esi, [esp+10h+var_8]
.text:0040CDCD  mov  edi, esi
.text:0040CDCF  mov  eax, ___security_cookie_complement
.text:0040CDD4  xor  eax, ___security_cookie
.text:0040CDDA  jnz  short never_taken
.text:0040CDDC  xor  edx, edx
.text:0040CDDE  jmp  short loc_40CDE9
.text:0040CDE0  mov  edx, [esp+10h+var_4]
.text:0040CDE4  lodsb
.text:0040CDE5  ror  al, 1
.text:0040CDE7  stosb
.text:0040CDE8  inc  edx
.text:0040CDE9  cmp  edx, 0Ah
.text:0040CDEC  jge  short loc_40CDF3
.text:0040CDEE  mov  [edx+ecx], dl
.text:0040CDF1  jmp  short loc_40CDE4
.text:0040CDF3  pop  edi
.text:0040CDF4  pop  esi
.text:0040CDF5  add  esp, 8
.text:0040CDF8  retn
```

Konverze reducibilního CFG na nereducibilní IX

```
char __usercall sub_40CDC0@<al>(char *pArray@<ecx>) {
  char *v1;         // esi@1
  char *v2;         // edi@1
  char result;      // al@1
  signed int i;     // edx@2
  char v5;          // al@4
  signed int v6;    // [sp+Ch] [bp-4h]@0

  v1 = pArray;
  v2 = pArray;
  result = __security_cookie ^__security_cookie_complement;

  if ( __security_cookie != __security_cookie_complement ) {
    i = v6;
    goto LABEL_4;
  }

  for ( i = 0; i < 10; ++i ) {
    pArray[i] = i;
    LABEL_4:
    v5 = *v1++;
    result = __ROR1__(v5, 1);
    *v2++ = result;
  }

  return result;
}
```

Jak vidíme, naše obfuskace není v C/C++ efektivní. IDA funkci dekompilovala skoro dokonale, včetně použití instrukce `ror`. V Javě byla situace úplně jiná, protože tam nebylo `goto`, které bychom potřebovali použít!

Konverze reducibilního CFG na nereducibilní X

Je zjevné, že potence závisí na jazyku, který obfuskujeme, a na kvalitě neprůhledného predikátu. V Javě je potence **střední** až **vysoká**, protože můžeme používat bytekódové instrukce, které nemají protějšek v původním jazyku. V C je potence **nízká** až **střední**.

Odolnost silně závisí na deobfuskátoru.

Cena této obfuskace je **nízká**.

Odstranění knihovních volání a programovacích idiomů

Volání knihovních funkcí poskytne analytikovi mnoho informací o našem programu. Tato volání navíc nemůžeme přejmenovat. Jejich obfuskace lze dosáhnout naprogramováním vlastních alternativ a použitím těchto alternativ místo původních API. To může zahrnovat funkce pro manipulaci s řetězci, alokaci paměti, třídy STL, atd. Ty všechny můžeme nahradit alternativními implementacemi, které mohou být přejmenované a obfuskované. To navíc zvýší velikost kódu, což činí techniku poměrně potentní (**střední**) a poskytuje odolnost, která je **silná**. Cena závisí na tom, co chceme nahradit.

Do této kategorie spadá také pozměnění běžně používaných programovacích idiomů, jako je procházení spojovým seznamem, polem, atd. Můžeme nahradit spojový seznam něčím jiným?

Tabulková interpretace I

Odolnost této obfuskace je **silná**, ale také je **drahá** cena. Ve své nejjednodušší podobě vyžaduje rozdělení části programového kódu do několika (nebo mnoha) kusů a použití smyčky s mnoha iteracemi, přičemž v každé se spustí jen jeden kus. Vhodné je použít příkaz `switch`, vypočítaná `goto` nebo tabulky skoků. Pokud chceme odolnost ještě navýšit, může mít tabulka dvě nebo více úrovní!

Tento tabulkový přístup si můžeme představit také jako běh nakouskovaného programu v miniaturním virtuálním stroji. Každý kus má své číslo, které reprezentuje operační kód "VM-instrukce". Seznam čísel kusů pak tvoří program VM! Pokud chceme jít ještě dále, můžeme každou instrukci vybavit jejími vlastními operandy, můžeme zavést registry, proměnné, atd.

Tabulková interpretace II

```
int WINAPI _tWinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow
)
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  HANDLE hMapping = INVALID_HANDLE_VALUE;
  LPVOID pSelf = NULL;
  LARGE_INTEGER cbSelf;

  /*1*/ cbSelf.QuadPart = 0;
  /*2*/ MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);
  /*3*/ CreateConsole();
  /*4*/ ExamineRelocations( pSelf, cbSelf );
  /*5*/ UnmapSelfFromMemory(hFile, hMapping, pSelf);

  /*6*/ return 0;
}
```

```
int WINAPI _tWinMainAsTable(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow
)
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  HANDLE hMapping = INVALID_HANDLE_VALUE;
  LPVOID pSelf = NULL;
  LARGE_INTEGER cbSelf;

  // Až do tohoto místa je program beze změny!

  // Nyní nadefinujeme instrukce naší VM.
  // Tzn. pro každý řádek vlevo označený
  // /*#*/ zavedeme samostatnou instrukci.
  // Zavedeme také instrukci pro no-op!
```

Tabulková interpretace III

```
int WINAPI _tWinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow
)
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  HANDLE hMapping = INVALID_HANDLE_VALUE;
  LPVOID pSelf = NULL;
  LARGE_INTEGER cbSelf;

  /*5*/ cbSelf.QuadPart = 0;
  /*1*/ MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);
  /*4*/ CreateConsole();
  /*0*/ ExamineRelocations( pSelf, cbSelf );
  /*3*/ UnmapSelfFromMemory(hFile, hMapping, pSelf);
  /*6*/ {}
  /*2*/ return 0;
}
```

```
int WINAPI _tWinMainAsTable(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow
)
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  HANDLE hMapping = INVALID_HANDLE_VALUE;
  LPVOID pSelf = NULL;
  LARGE_INTEGER cbSelf;

  typedef enum instr {
    instr_init_size = 5, instr_map_file = 1,
    instr_create_console = 4, instr_return = 2
    instr_examine_relocations = 0, instr_nop = 6
    instr_unmap_file = 3
  } instr;


  // Instrukce byly očíslovány náhodně.
  // Rozsah zůstal zachován, ale nejsou v
  // pořadí. To nemá na program vliv.
  // Nyní vytvoříme program pro naši VM.
  // --- posloupnost zavedených instrukcí.
```

Tabulková interpretace IV

```
int WINAPI _tWinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow
)
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  HANDLE hMapping = INVALID_HANDLE_VALUE;
  LPVOID pSelf = NULL;
  LARGE_INTEGER cbSelf;

  /*5*/ cbSelf.QuadPart = 0;
  /*1*/ MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);
  /*4*/ CreateConsole();
  /*0*/ ExamineRelocations( pSelf, cbSelf );
  /*3*/ UnmapSelfFromMemory(hFile, hMapping, pSelf);
  /*6*/ {}
  /*2*/ return 0;
}
```

```
int WINAPI _tWinMainAsTable(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow
)
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  HANDLE hMapping = INVALID_HANDLE_VALUE;
  LPVOID pSelf = NULL;
  LARGE_INTEGER cbSelf;

  typedef enum instr {
    instr_init_size = 5, instr_map_file = 1,
    instr_create_console = 4, instr_return = 2
    instr_examine_relocations = 0, instr_nop = 6
    instr_unmap_file = 3
  } instr;

  static instr g_Instructions[] = {
    instr_nop, instr_init_size, instr_map_file,
    instr_nop, instr_create_console, instr_nop,
    instr_examine_relocations, instr_nop,
    instr_unmap_file, instr_nop, instr_return
  };


  // Nyní přidáme VM jako příkaz switch.
```

Tabulková interpretace V

```
int WINAPI _tWinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow
)
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  HANDLE hMapping = INVALID_HANDLE_VALUE;
  LPVOID pSelf = NULL;
  LARGE_INTEGER cbSelf;

  /*5*/ cbSelf.QuadPart = 0;
  /*1*/ MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);
  /*4*/ CreateConsole();
  /*0*/ ExamineRelocations( pSelf, cbSelf );
  /*3*/ UnmapSelfFromMemory(hFile, hMapping, pSelf);
  /*6*/ {}
  /*2*/ return 0;
}
```

```
typedef enum instr {
  instr_init_size = 5, instr_map_file = 1,
  instr_create_console = 4, instr_return = 2
  instr_examine_relocations = 0, instr_nop = 6
  instr_unmap_file = 3
} instr;

static instr g_Instructions[] = {
  instr_nop, instr_init_size, instr_map_file,
  instr_nop, instr_create_console, instr_nop,
  instr_examine_relocations, instr_nop,
  instr_unmap_file, instr_nop, instr_return
};

for (
  unsigned int i = 0;
  i < sizeof(g_Instructions) / sizeof(instr);
  ++i
)
{
  switch (g_Instructions[i])
  {
    // Větve nejsou prohozeny, pro přehlednost
    case instr_init_size:
      cbSelf.QuadPart = 0;
      break;

    case instr_map_file:
      MapSelfIntoMemory(hFile, hMapping,
        pSelf, cbSelf);
      break;

    case instr_create_console:
      CreateConsole();
      break;
```

Tabulková interpretace VI

```
int WINAPI _tWinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow
)
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  HANDLE hMapping = INVALID_HANDLE_VALUE;
  LPVOID pSelf = NULL;
  LARGE_INTEGER cbSelf;

  /*5*/ cbSelf.QuadPart = 0;
  /*1*/ MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);
  /*4*/ CreateConsole();
  /*0*/ ExamineRelocations( pSelf, cbSelf );
  /*3*/ UnmapSelfFromMemory(hFile, hMapping, pSelf);
  /*6*/ {}
  /*2*/ return 0;
}
```

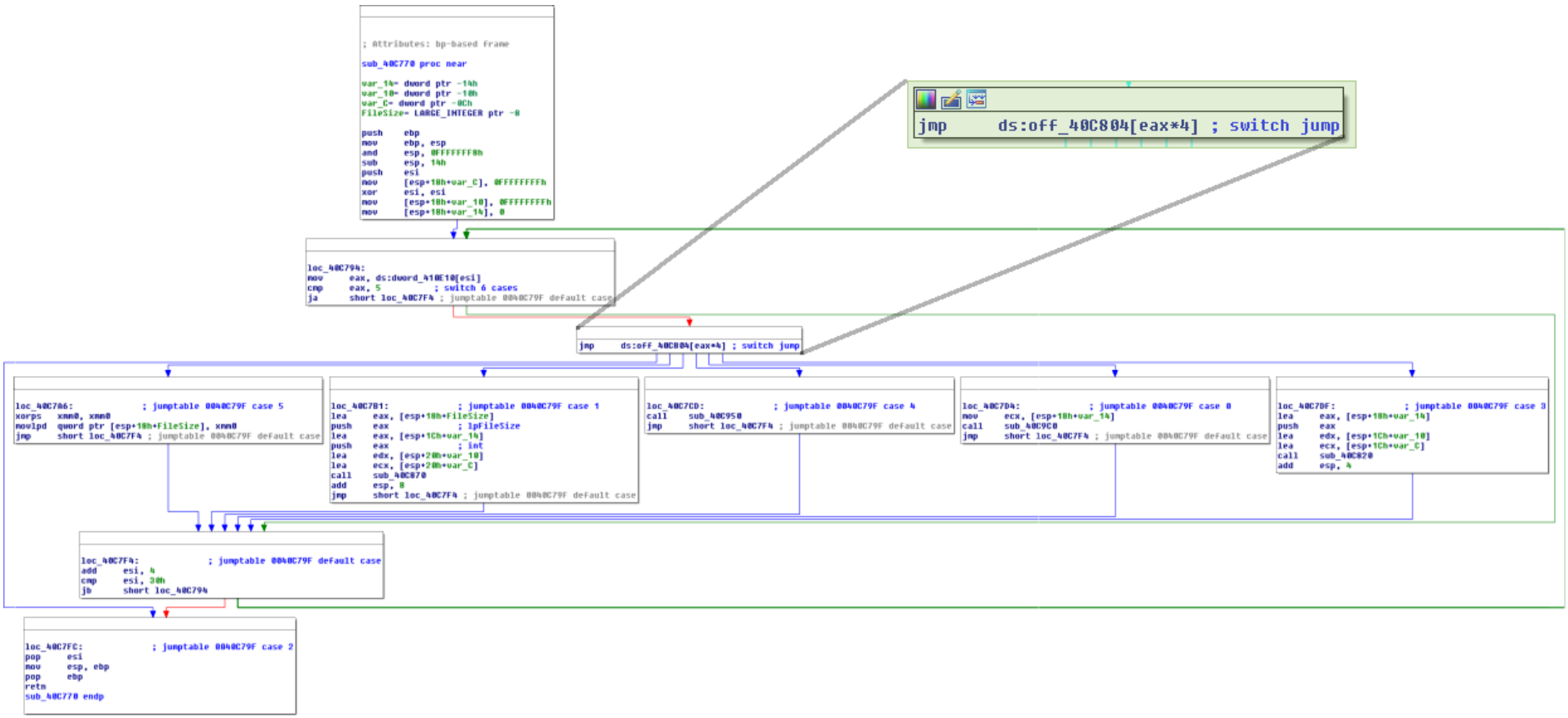
```
static instr g_Instructions[] = {
  instr_nop, instr_init_size, instr_map_file,
  instr_nop, instr_create_console, instr_nop,
  instr_examine_relocations, instr_nop,
  instr_unmap_file, instr_nop, instr_return
};

for (
  unsigned int i = 0;
  i < sizeof(g_Instructions) / sizeof(instr);
  ++i
)
{
  switch (g_Instructions[i])
  {
    // Větve nejsou prohozeny, pro přehlednost
    case instr_init_size:
      cbSelf.QuadPart = 0;
      break;

    case instr_map_file:
      MapSelfIntoMemory(hFile, hMapping,
        pSelf, cbSelf);
      break;

    case instr_create_console:
      CreateConsole();
      break;

    case instr_examine_relocations:
      ExamineRelocations(pSelf, cbSelf);
      break;

    case instr_unmap_file:
      UnmapSelfFromMemory(hFile, hMapping, pSelf);
      break;
```

Tabulková interpretace VII

```
int WINAPI _tWinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow
)
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  HANDLE hMapping = INVALID_HANDLE_VALUE;
  LPVOID pSelf = NULL;
  LARGE_INTEGER cbSelf;

  /*5*/ cbSelf.QuadPart = 0;
  /*1*/ MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);
  /*4*/ CreateConsole();
  /*0*/ ExamineRelocations( pSelf, cbSelf );
  /*3*/ UnmapSelfFromMemory(hFile, hMapping, pSelf);
  /*6*/ {}
  /*2*/ return 0;
}

static instr g_Instructions[] = {
  instr_nop, instr_init_size, instr_map_file,
  instr_nop, instr_create_console, instr_nop,
  instr_examine_relocations, instr_nop,
  instr_unmap_file, instr_nop, instr_return
};
```

```
for (
  unsigned int i = 0;
  i < sizeof(g_Instructions) / sizeof(instr);
  ++i
)
{
  switch (g_Instructions[i])
  {
    // Větve nejsou prohozeny, pro přehlednost
    case instr_init_size:
      cbSelf.QuadPart = 0;
      break;

    case instr_map_file:
      MapSelfIntoMemory(hFile, hMapping,
        pSelf, cbSelf);
      break;

    case instr_create_console:
      CreateConsole();
      break;

    case instr_examine_relocations:
      ExamineRelocations(pSelf, cbSelf);
      break;

    case instr_unmap_file:
      UnmapSelfFromMemory(hFile, hMapping, pSelf);
      break;

    case instr_nop: break;

    case instr_return: return 0;
  }
}
```

Tabulková interpretace VIII



Obrázek : `wWinMain` pomocí tabulkové interpretace.

Tabulková interpretace IX

Jak jsme viděli, náš program se obfuskací kompletně změnil. Srdcem funkce je nyní instrukce:

```
jmp dword ptr ds:[0040c804+4*eax]
```

Tabulka skoků je globální proměnná, stejně jako program pro VM (`00410e10`).

```
.rdata:00410E10 dword_410E10    dd 6, 5, 1, 6
.rdata:00410E20                 dd 4, 6, 6, 0
.rdata:00410E30                 dd 6, 3, 6, 2
```

Množství kódu významně vzrostlo, důsledkem je **vysoká** potence této transformace vzhledem k metrice "velikost kódu". Odolnost transformace je **silná**. Cena za převod programu touto transformací je naneštěstí **drahá**.

Přidání redundantních argumentů I

Redundantní argumenty zvyšují délku programu a jeho složitost. A nemusíme jen přidávat argumenty metodám/funkcním — můžeme rozšiřovat aritmetické výrazy pomocí neprůhledných proměnných.

Pozn.: Při provádění redundantních operací v plovoucí řádové čárce si musíme dávat pozor na dodatečná zaokrouhlení, která ovlivní přesnost spočítané proměnné.

Použití této metody jsme si mohli povšimnout v příkladu na rozšíření podmínek cyklů. Druhý iterátor `j` tam byl redundantní, stejně jako všechny výpočty nad ním.

Přidání redundantních argumentů II

```
int RedundantOps( int iRedundantBound ) {
  int i;
  static unsigned char array[256];
  static unsigned char oarray[256];

  for( i=0; i<iRedundantBound; ++i ) {
    oarray[i]= (i + 1)(i + 1);
    oarray[i]-= (i - 1)(i - 1);
    array[i]=oarray[i]-3*i;
  }

  return 0;
}
```

```
.text:000AC780  xor   ecx, ecx
.text:000AC782  push  ebx
.text:000AC783  lea   edx, [ecx-1]
.text:000AC786  mov   al, dl
.text:000AC788  lea   ebx, [ecx+1]
.text:000AC78B  imul  dl
.text:000AC78D  mov   dl, al
.text:000AC78F  mov   al, bl
.text:000AC791  imul  bl
.text:000AC793  sub   al, dl
.text:000AC795  mov   dl, cl
.text:000AC797  add   dl, dl
.text:000AC799  mov   byte_B3F60[ecx], al
.text:000AC79F  lea   ebx, [edx+ecx]
.text:000AC7A2  sub   al, bl
.text:000AC7A4  mov   byte_B4060[ecx], al
.text:000AC7AA  inc   ecx
.text:000AC7AB  cmp   ecx, 100h
.text:000AC7B1  jl    short loc_AC783
.text:000AC7B3  xor   eax, eax
.text:000AC7B5  pop   ebx
.text:000AC7B6  retn
```

Zde jsme funkci přidali redundantní argument `int iRedundantBound`, a také redundantní neprůhledné pole `static unsigned char oarray`. Argument `iRedundantBound` obsahuje horní mez pole a volající ho vždy nastaví na 256. Neprůhledné pole je použito pouze jako prostor pro výpočet $(i+1)^2 - (i-1)^2 - 3i = i$. Výsledkem pak inicializujeme `array[i]=i`. Množství kódu vzrostlo, takže potence je **střední** až **vysoká**, ale odolnost transformace je **slabá**. Transformation je však **levná**.

Paralelizace kódu I

Vlákna se používají pro provádění více operací souběžně tak, že rozčleníme problém na menší kusy, vyřešíme je samostatně a pak sloučíme výsledky. Tentýž princip můžeme použít pro obfuskaci:

- Pokud jsou bloky datově nezávislé: necháme každé vlákno vyhodnocovat jeden blok posloupnosti.
- Pokud jsou bloky datově závislé: vytvoříme posloupnost bloků kódu, necháme ve vlákně spustit první blok, pak odblokujeme druhé vlákno a provedeme druhý blok, atd.
- Můžeme vytvořit několik vláken, které nedělají nic užitečného, jen používají neprůhledné predikáty, vyhazují výjimky, . . .)

Paralelizace kódu II

Potence paralelizace kódu je **vysoká**, protože paralelním programům je těžké pozorumět. Odolnost je **silná**, protože automatický deobfuskátor musí zvažovat všechna vlákna a jejich možné interakce v programu, což může vést až ke složitosti $O(n!)$! Cena transformace programu na paralelní je **drahá**.

Paralelizaci obvykle provádíme prostřednictvím vláken. Pro zvýšení odolnosti (**plná**) můžeme části kódu přesunout do samostatných procesů a využít nějakou formu meziprocesové komunikace (sdílená paměť, pojmenované či anonymní roury, sdílené soubory, lokální či vzdálené sockety, koncové body RPC, . . .) pro synchronizaci.

Paralelizace kódu III

```
int WINAPI _tWinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow
)
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  HANDLE hMapping = INVALID_HANDLE_VALUE;
  LPVOID pSelf = NULL;
  LARGE_INTEGER cbSelf;

  cbSelf.QuadPart = 0;
  MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);
  CreateConsole();
  ExamineRelocations( pSelf, cbSelf );
  UnmapSelfFromMemory(hFile, hMapping, pSelf);
  return 0;
}
```

Každý řádek programu transformujeme na samostatné vlákno. Thread1 a Thread2 mohou běžet souběžně. Thread3 musí počkat, než obě skončí. Thread4 musí počkat na Thread3 a používá pro to události Windows. Thread5 musí počkat na Thread4.

```
volatile LONG __declspec(align(8))
  g_ThreadOneTwoDone = 0;

LPVOID WINAPI Thread1(LPVOID pUserData) {
  LARGE_INTEGER* pcbSelf = (LARGE_INTEGER*)pUserData;
  pcbSelf->QuadPart = 0;
  InterlockedIncrement( &g_ThreadOneTwoDone );
  return 0;
}

LPVOID WINAPI Thread2(LPVOID pUserData) {
  CreateConsole();
  InterlockedIncrement( &g_ThreadOneTwoDone );
  return 0;
}

LPVOID WINAPI Thread3(LPVOID pUserData) {
  while( g_ThreadOneTwoDone != 2 )
    ;

  MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);
  SetEvent(...); /* odblokuj vlákno 4 */
  return 0;
}

LPVOID WINAPI Thread4(LPVOID pUserData) {
  WaitForSingleObject(...); /* počkej na vlákno 3 */
  ExamineRelocations( pSelf, cbSelf );
  SetEvent(...); /* odblokuj vlákno 5 */
  return 0;
}

...
```

Inlining I

Inliningem rozumíme proces kopírování těla funkce do všech míst, odkud je funkce volána.

Kompilátory používají inlining v rámci optimalizace pro krátké funkce, konstruktory, atd., aby ušetřily na režii prologu a epilogu, předávání argumentů a provádění instrukce `call`.

```
int main()
{
  m1;
  F1();
  m2;
  G1();
  m3;
}
```

```
int F1()
{
  f1;
  f2;
}

int G1()
{
  g1;
}
```

→ **inlining** →

```
int main()
{
  m1;
  f1;
  f2;
  m2;
  g1;
  m3;
}
```

Funkce `F1` a `G1` v transformovaném produktu vůbec neexistují a informace o nich je nenávratně ztracena — inlining je **jednosměrná** transformace.

Inlining II

Obfuskátory používají inlining pro zvýšení složitosti funkcí. Čím víc lokálních proměnných a řádků kódu funkce má, tím obtížnější je ji analyzovat, protože obfuskace odstranila abstrakce použité programátorem. Reverzní inženýr nebude vědět, jaká funkce byla inlinovaná a kam, obzvlášť pokud byly inlinovány všechny kopie funkce a originál už se v programu vůbec nevyskytuje.

Inlining je **levná** transformace, její potence je **střední** a odolnost **jednosměrná**, obzvlášť pokud ji zkombinujeme s následující technikou — outliningem.

Outlining

Proces outliningu je inverzní k inliningu. Část kódu funkce je vytržena z těla funkce, postavena stranou, zkonvertována na samostatnou funkci a na původním místě nahrazena instrukcí `call`. Použití outliningu vede, zejména při souběžném použití s inliningem, k potentním a odolným transformacím:

```
int main()
{
  m1;
  f1;
  f2;
  m2;
  g1;
  m3;
}
```

→ **outlining** →

```
int main()
{
  F1();
  G1();
  H1();
}
```

```
int F1()
{
  m1;
  f1;
}
int G1()
{
  f2;
  m2;
  g1;
}
int H1()
{
  m3;
}
```

Interleaving I

Interleaving je optimalizační technika využívající fyzickýxh vlastností média k zvýšení přístupových nebo přenosových rychlostí. U obfuskaci rozumíme interleavingem techniku, kdy se kód a argumenty dvou nebo více funkcí spojují do funkce jediné, společně s novým argumentem (nebo globální proměnnou), který dovolí volajícímu vybrat jednu z větví.

```
int F1()
{
  f1;
  f2;
}

int G1()
{
  g1;
}
```

→ **interleaving** →

```
int H1(int which)
{
  switch (which) {
    case 1:
      f1;
      f2;
      break;
    case 2:
      g1;
      break;
    };
}
```

Klonování

Obfuskace pomocí klonování metod se poněkud podobá použití neprůhledných predikátů v tom, že vytváří nadbytečná větvení, která musí reverzní inženýr analyzovat. Nepoužívá však neprůhledné predikáty, ale mechanismy programovacího jazyka pro výběr metod. Toho je dosaženo tranformací jediné rodičovské třídy `C` s virtuální metodou na hierarchii (nebo sadu) tříd `C1`, `C2`, atd., přičemž každá vznikla z `C` pomocí jiné obfuskační transformace. Reverznímu inžěnýrovi se bude zdát, že volání závisí na konkrétní instanci třídy, ve skutečnosti ovšem provádějí všechny instance přesně totéž.

Transformace smyček

Pro účely obfuskace lze dále využít různé transformace smyček:

- **Loop Blocking** označuje techniku rozdělení těla smyčky do několika částí tak, aby se vešly do cache, a tím lépe využily vlastností cacher dané architektury.
- **Loop Unrolling** replikuje tělo smyčky několikrát po sobě, čímž eliminuje potřebu skoků na další iteraci a umožňuje lepší optimalizaci využití registrů. To je obzvlášť efektivní v případech, kdy je počet iterací předem znám.
- **Loop Fission** dělí smyčku na několik samostatných smyček se stejným iteračním mechanismem, ale různými těly. Tyto smyčky pak mohlou například běžet paralelně.

Všechny tyto techniky zvyšují metriky kódu jako je velikost nebo cyklomatická složitost, což zajišťuje jejich potenci. Jejich odolnost je **slabá**, pokud jsou použity izolovaně, ale výrazně roste v kombinaci s dalšími obfuskačními technikami. Cena je **levná**, někdy **nulová**.

Literatura I


Collberg C., Thomborson C., Low, D.: *A Taxonomy of Obfuscating Transformations*, Technical Report #148, University of Auckland, Auckland, New Zealand, 1997/2009. Available online at `https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf`.


Eilam, E.: *Reversing — Secrets of Reverse Engineering*, Wiley Publishing, Inc., 2005.


Eagle, C.: *The IDA Pro Book — The unofficial guide to the world's mode popular disassembler*, 2nd ed., No Starch Press, 2011.


Schwarz, B., Debray, S., Andrews, G.: *Disassembly of Executable Code Revisited*, Proceedings of the IEEE Working Conference on Reverse Engineering, Oct. 2002, pp. 45-54. Available online at `http://ftp.cs.arizona.edu/~debray/Publications/disasm.pdf`.

Literatura II


Wikipedia Foundation, Inc.: *Java bytecode instruction listings*, Available online at `https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings`, 2015.