Reverse Engineering

4. Disassembling and Obfuscation

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

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Czech Technical University in Prague
Faculty of Information Technology
Department of Information Security

Version 2021-09-11

Table of Contents I

1. Disassembling
   - Linear Sweep
   - Recursive Traversal
   - Extended Linear Sweep
   - Hybrid Approach
2. Obfuscation
   - Definition
   - Metrics
3. Layout Obfuscations
4. Control Flow Obfuscations
   - Opaque Predicates and Variables
5. Computation Transformations
   - Insertion of Dead or Irrelevant Code
   - Extension of Loop Conditions
   - Conversion of a Reducible CFG to Non-Reducible

Table of Contents II

- Removal of Library Calls and Programming Idioms
- Table Interpretation
- Addition of Redundant Operands
- Code Parallelization

6. Aggregation Transformations
   - Inlining
   - Outlining
   - Interleaving
   - Cloning
   - Loop Transformations

Disassembling

Disassembling

Disassembling is a process of translation of binary code into a code in a human-readable assembly language of the target CPU.

- An application performing the disassembling is called a disassembler.
- Disassembly (a product of a disassembled executable) is used to perform static code analysis, or for the dynamic (live) code analysis within a debugger.
- Disassembly:
  - contains a huge number of lines of code;
  - lacks a lot of information present at compile time (variable names, debugging information, label names) due to stripping and optimization;
  - mixes code and data — how to distinguish between them?
- There are two approaches to disassembling — a **linear sweep** and a **recursive traversal** [4]. Each of them has its own pros and cons.

Linear Sweep

Linear Sweep

Linear sweep performs disassembly in a linear fashion, byte by byte, from the start of the `.text` section toward its end.

- Fast and simple.
- PowerPC uses fixed length instruction encoding. Each instr. 4 B.
- x86 uses variable length instruction encoding. Its instructions can have 1–15 bytes. Infinitely long instructions can theoretically be created with redundant prefixes, but anything longer than 15 bytes generates an exception.
  - `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`.
- Once an instruction is disassembled, processing of another starts, and so on. This process is repeated for the entire `.text` section.

Ambiguous Control Flow

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
```

Original code

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

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        // Correct!
00402cdd  3e    db 3e        // Correct!
00402cde  81    db 81        // Correct!
00402cdf  84    db 84        // Correct!
00402ce0  98    cwde         // Bogus!
00402ce1  8bc3  mov eax,ebx  // Correct!
00402ce3  8bc3  mov eax,ebx  // Correct!
00402ce5  8bc3  mov eax,ebx  // Correct!
00402ce7  8bc3  mov eax,ebx  // Correct!
```

Mixed Code & 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 in the .text section!
  011d490d 48               dec  eax                         // Data in the .text section!
  011d490e 1d0155481d       sbb  eax,1D485501h               // Data in the .text section!
  011d4913 015c481d         add  dword ptr [eax+ecx*2+1Dh],ebx // Data in the .text section!
  011d4917 016348           add  dword ptr [ebx+48h],esp     // Data in the .text section!
  011d491a 1d01cccccc       sbb  eax,0CCCCCC01h              // Data in the .text section!
```

Linear Sweep Continued

- As we have observed, linear sweep blindly disassembled one instruction after another. This was working fine when we started disassembly at a function's start (`push ebp` at `011d4780`). Initially it was going OK, later the disassembler got confused by the following constructs:
  1. ambiguous control flow;
  2. data in code.
- Both resulted in a meaningless disassembly!
  - → Any disassembler using only linear sweep (WinDbg, gdb, objdump [3]) cannot distinguish code from data.

→ Linear sweep is by itself insufficient → we need a more robust method.

Recursive Traversal I

Recursive traversal, in contrast to the linear sweep, disassembles code by watching its flow rather than linearly parsing all contents of the `.text` section. The traversal starts typically at the executable's entry point and disassembles the first instruction there. The code section with that instruction is marked as visited. If the instruction was a branch instruction, the return address (if any) is remembered and the analysis recursively starts at the branch target location. Once all analyses of the branch target are performed, the analysis resumes after the call/conditional jump instruction if necessary.

This approach:

- is slower;
- distinguishes code from data:
  - all locations not visited are considered to be data

Recursive Traversal II

Recursive traversal, in contrast to the linear sweep, disassembles code by watching its flow rather than linearly parsing all contents of the `.text` section. The traversal starts typically at the executable's entry point and disassembles the first instruction there. The code section with that instruction is marked as visited. If the instruction was a branch instruction, the return address (if any) is remembered and the analysis recursively starts at the branch target location. Once all analyses of the branch target are performed, the analysis resumes after the call/conditional jump instruction if necessary.

This approach:

- is slower;
- partially distinguishes code from data:
  - all locations not visited are considered to be data
  - but — there are jump tables, indirectly called functions, etc.

Recursive Traversal III

Recursive Traversal [4]

```
procedure Disassemble( addr, instrList ) {
  if( addr.visited )
    return;
  do {
    instr = DecodeInstr( addr );
    addr.visited = true;
    add instr to instrList;
    if ( instr is a branch or function call ) {
      T = set of possible control flow successors of instr;
      for each target in T do {
        Disassemble( target, instrList);
      }
    }
    else
      addr += instr.length; /* addr of next instruction */
  } while addr is a valid instruction address;
}
```

Recursive Traversal Summary

- **Recursive traversal** is based on the assumption that we can identify all possible control flow successors of each control flow instruction.
- Control flow is watched and every address **recursive traversal** visits is marked as code.
- What will be the possible control flow when we encounter an indirect jump (jump to a jump table item, a computed goto, or a call through a VMT)? Easy to tell with a CFG, but we don't have one!
- What to do with the rest of the code segment that was not visited?
  - **recursive traversal** won't even try to disassemble that part and can't tell anything about it without further analyses.
  - **linear sweep** would disassemble it; some portions would disassemble well, others not since they are data.

- **Recursive traversal** has complementary strengths and weaknesses with **linear sweep**. Which one should we use? Let's try to improve both algorithms.

Extended Linear Sweep

Linear sweep could not deal with data embedded in the `.text` section such as jump tables and/or string constants. Such data caused disassembly errors by being improperly identified as instructions. **Linear sweep** can, however, be improved:

The Idea [4]

If we have relocation information available, jump tables in the `.text` section must have their entries in the relocation table. Based on the entries in the relocation table, we can identify possible jump tables within the `.text` section and mark them as data.

Each entry a_i in a jump table must satisfy [4]:

1. the memory locations containing a_i are marked relocatable; and
2. the address a_i itself must be within the `.text` section.

A Jump Table in the Program I

Let us verify the idea with the following program:

A sample jump table

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

A sample a jump table (assembler)

```
.text:00414838              cmp  [ebp+uintval], 3
.text:0041483F              ja   short loc_414868
.text:00414841              mov  ecx, [ebp+uintval]
.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
...
--------- BEGIN of data in the .text section ---------
.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
--------- END of data in the .text section -----------

...
```

A Jump Table in the Program II

If we dump the `.reloc` section with `dumpbin`, we get this output:

```
C:\...\> dumpbin /RELOCATIONS jumptable.exe
...
   14000 RVA,       8C SizeOfBlock
     ...  ...              ...           --------- BEGIN of data in the .text section ---------
     90C  HIGHLOW          0041484E      .text:0041490C off_40C5E0  dd offset loc_41484E // case 0
     910  HIGHLOW          00414855      .text:00414910             dd offset loc_414855 // case 1
     914  HIGHLOW          0041485C      .text:00414914             dd offset loc_41485C // case 2
     918  HIGHLOW          00414863      .text:00414918             dd offset loc_414863 // case 3
                                         --------- END of data in the .text section -----------
```

There are 4 consecutive items in the relocation table relocating 4 consecutive pointers in the `.text` segment. Since no more than 2 of them can form a part of an x86 instruction, the remaining 2 (loc. `00414914` and `00414918`) must be data. The first 2 items could possibly be code.

Note: EXE file images not using ASLR normally do not have the `.reloc` section, unlike DLLs. When ASLR is enabled for an image, it must always be relocatable and thus have the `.reloc` section, as in the example above.

Extended Linear Sweep Continued

We have identified only 2 out of 4 items as data. We can partition the `.text` section into segments delimited by jump tables and then disassemble all code in between the segments, checking whether each disassembled instruction in front of the jump table was successfully disassembled and whether it overflowed into the data in between the segments. In case of an overflow, the instruction isn't instruction but data!

Extended Linear Sweep [4]

1. Mark all jump table items as data, except for the first 2 in each table (on x86, these can form a part of an instruction).
2. For each sequence of unmarked addresses in the `.text` segment:
   a) Use **linear sweep**, stop when a marked location is reached.
   b) If the last instruction overlaps into a marked location, mark it as data.
   c) Examine the last correctly disassembled instruction, mark as data if necessary.

Hybrid Approach

Recursive traversal has complementary strengths and weaknesses with **linear sweep**. Let's combine the best of both approaches [4] and:

- Use **extended linear sweep** for the initial disassembly;
- Use **recursive traversal** to verify the disassembly of each function in the executable.

The verification is performed as follows:

1. **Recursive traversal** is used for each instruction in each function;
2. Each instruction I obtained at address a_I is checked. If it was not obtained also by the **linear sweep**, an error is raised.
3. If no error is produced for the function, report success.

Protection Taxonomy [1]

Intellectual protection can be classified as:

- legal protection
- technical protection
  - obfuscation
  - encryption
  - (partial) server-side execution
  - trusted native code

Definitions

Obfuscation

Obfuscation makes something obscure. Obfuscation in computer software is a process of changing the transformation target into an obfuscated target in a way which is difficult to comprehend while preserving the functionality of the target.

Transformation Target

Transformation target is an object to be obfuscated. The target can be a program's source code as well as its executable in a binary form.

Sample transformation targets:

- javascript code in your HTML pages;
- java classes and archives, .NET source files and executable code;
- PHP source code, C/C++ source code, ...;
- executables.

Motivation for obfuscations

The motivation for using obfuscation techniques is to keep your code secret and to make its comprehension difficult. Regular software uses these techniques to prevent people from reverse engineering serial number schemes, defeating copy protections, disabling hardware dongles, DRMs, etc. Malware uses these techniques frequently to resist analysis.

The purpose of these techniques is to:

1. confuse the disassembler (the static analysis);
2. confuse the debugger (the dynamic analysis);
3. confuse the decompiler;
4. confuse the human reading the disassembly in either tool.

These techniques include code obfuscation, control flow transformations, encryption, checksumming, debugger detection and anti-debugger code, etc. The longer the analysis takes, the fewer people keep doing it :-).

Obfuscation Metrics I

Obfuscated product is the transformation target after the application of a set of obfuscating transformations. Any of these transformations, besides not changing the original program's behavior, should be **potent**, **resilient**, and their **cost** should be minimal.

Potency [1]

The potency $\mathcal{T}_{pot}(P)$ of a transformed program P' measures how much the transformation of a program P confuses a human reader.

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

where $E(P)$ is the complexity of P calculated by **some** metric.

Metrics such as **program length**, **cyclomatic complexity**, **nesting complexity**, **data flow complexity**, ... can be used; obfuscations increase the complexity of whatever a particular metric measures.

Obfuscation Metrics II

Resilience [1]

Resilience measures how difficult it is for an automatic deobfuscator to undo the transformation.

$$\mathcal{T}_{res}(P) := \text{Resilience}(\mathcal{T}_{deobfuscator\ effort}, \mathcal{T}_{programmer\ effort}),$$

where $\mathcal{T}_{programmer\ effort}$ is the amount of time required to create an auto-deobfuscator reducing $\mathcal{T}_{pot}$, while $\mathcal{T}_{deobfuscator\ effort}$ is the execution time and space required by the auto-deobfuscator to reduce $\mathcal{T}_{pot}$.

Resilience $\mathcal{T}_{res}$ can be: **trivial**, **weak**, **strong**, **full**, or **one-way**.

$\mathcal{T}_{deob.\ eff.} \setminus \mathcal{T}_{prog.\ eff.}$	**Local**	**Global**	**Inter-procedural**	**Inter-process**
Polynomial	**trivial**	**weak**	**strong**	**full**
Exponential	**weak**	**strong**	**full**	**full**

Table: Resilience($\mathcal{T}_{deobfuscator\ effort}$, $\mathcal{T}_{programmer\ effort}$) [1].

Obfuscation Metrics III

Cost [1]

Cost expresses how much the execution time and/or required resources increase if an obfuscation transformation is applied.

$$\mathcal{T}_{cost}(P) := \begin{cases} \textbf{dear} & \text{if } P' \text{ requires exponentially more resources than } P \\ \textbf{costly} & \text{if } P' \text{ requires polynomially more resources than } P \\ \textbf{cheap} & \text{if } P' \text{ requires linearly more resources than } P \\ \textbf{free} & \text{if } P' \text{ requires constantly more resources than } P \end{cases}$$

Obfuscation types

Intellectual protection can be classified as:

- legal protection
- technical protection
  - obfuscation
  - encryption
  - (partial) server-side execution
  - trusted native code

A transformation target can be modified using:

- layout obfuscation
- control obfuscation
- data obfuscation
- preventive transformation

Layout Obfuscations I

Layout obfuscations [1] target the lexical structure of the application. This includes the source code formatting, comments, variable and function names, layouts of classes and data structures, etc.

Before obfuscation:

```
static public float AverageStudentGrade(Student student) {
  float grade = 0.0f; float credits = 0.0f;
  /* Grade calculated as a weighted average g=sum(grade_i*credits_i)/sum(credits_i) */
  if( student.Courses().Empty() ) return 0.0f;
  for_each(course in student.Courses()) {
    grade += course.Credits() * course.Grade();
    credits += course.Credits();
  }
  return grade/credits;
}
```

After (comments removed, data types, methods, and classes renamed):

```
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;
}
```

Layout Obfuscations II

Transformation	Potency	Resilience	Cost
Scramble identifiers	medium	one-way	free
Change formatting	low	one-way	free
Remove comments	high	one-way	free
Remove debug info	high	one-way	free

Table: Quality of Layout Obfuscations [1].

Control Flow Obfuscations

Control flow obfuscations obfuscate the control flow. They can be divided into **aggregation**, **ordering**, and **computation** transformations [1].

- **Computation transformations** add redundant and/or dead code or make algorithmic changes.
- **Ordering transformations** randomize the order of computations.
- **Aggregation transformations** break apart computations belonging together and merge computation that do not.

Many control flow obfuscations depend on the use of **opaque variables** and **opaque predicates**.

Opaque Predicates and Variables I

A variable or predicate is opaque if its value or property is known to the obfuscator at the obfuscation time but is difficult for the deobfuscator to deduce [1].

An opaque construct consists of an **if** statement testing the value of an opaque predicate or variable `ovar` for a specific value `VAL` which was known at the obfuscation time:

```
if( ovar == VAL )
  never execute
else
  do something
```

```
if( ovar == VAL )
  do something
else
  never execute
```

```
if( ovar == VAL )
  do something
else
  do something
```

Our primary purpose is to confuse both the disassembler and any human being studying our code. Let's look at the disassemblers first.

Opaque Predicates and Variables II

Linear sweep is poor at distinguishing code from data and thus vulnerable to data (junk, jump tables, ...) inside (or in between) functions. We will use this fact to combine the never-to-be executed branch of an opaque predicate with junk data. We can produce such data within the _asm block with the _emit keyword:

```
_asm {
    xor    eax, eax // Opaque variable in EAX
    cmp    eax, 1   // Redundant, we could use ZF & jz already set by xor
    jnz    continue // Always jumps

  never_execute:
    _emit  0xe9     // E9 is a jmp instruction opcode followed by a 4 B address

  continue:
    imul   edx
}
```

The _emit keyword is used to insert an opcode of a jmp instruction into the binary. **Linear sweep** disassemblers will typically understand the imul instruction as a part of the emitted jmp instruction's address.

Opaque Predicates and Variables 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

Figure: 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          ???
```

Figure: OllyDbg 1.10

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

Figure: OllyDbg 2.01

The fact that the code beyond the `never_execute` label is never executed can be detected with **recursive traversal**.

Opaque Predicates and Variables 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
```

Figure: IDA Pro 6.7 initial.

```
.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
```

Figure: IDA Pro 6.7 corrected.

Note: Although the transformation could fool a disassembler, it is trivially correctable by a human. We find the potency **low**. This construct could be handled by an automatic deobfuscator in a polynomial time within the local procedure scope. For these reasons the resilience is **trivial**. The cost is, however, **cheap** and that's why it is so popular.

An Example I

Let's use an **opaque predicate** to fool both linear and also simple recursive disassemblers. The code below loads security cookie from the stack 1 and exclusively ors it with the ebp value 2. The same code is used at the end of any function using canaries. Our code uses it as an opaque value.

```
#define OBFUSCATE(_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
```

The xor'd value in eax at 2 might theoretically be zero, but it is very unlikely ($1 : 2^{32}$). When nonzero 3, continue to 7. This is the normal control flow. The disassembler is unable to tell whether the condition is always true or not since it is calculated at runtime. The `_emit` at line 4 creates a random `jxx` instruction jumping `0x00` bytes behind it 5, that is at the another `_emit` at line 6. This `_emit` generates the first byte of a random `jmp` or `call` instruction. The remaining bytes are taken from instructions following the macro.

The `OBFUSCATE` macro takes a `_random` value as an argument in order to generate slightly different assembly each time it is used. There are 64 possible combinations it can generate.

An Example II

Let's see how it works on this C code:

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

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

OBFUSCATE(118);
CreateConsole();

OBFUSCATE(235);
```

An Example II

Let's see how it works on this C code:

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

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

OBFUSCATE(118);
CreateConsole();

OBFUSCATE(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
```

An Example II

Let's see how it works on this C code:

```c
OBFUSCATE(238);
cbSelf.QuadPart = 0;

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

OBFUSCATE(118);
CreateConsole();

OBFUSCATE(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
```

An Example II

Let's see how it works on this C code:

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

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

OBFUSCATE(118);
CreateConsole();

OBFUSCATE(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 ; ---------------------------------------------------------------
```

An Example II

Let's see how it works on this C code:

```
OBFUSCATE(238);
cbSelf.QuadPart = 0;
OBFUSCATE(372);
MapSelfIntoMemory(hFile, hMapping, pSelf, cbSelf);
OBFUSCATE(118);
CreateConsole();
OBFUSCATE(235);
```

Though our efforts confused both **linear sweep** and **recursive traversal** disassemblers and perhaps some human being too, our construct is only slightly more **potent** than the previous one. Regarding its **resilience**, it is also slightly better. The opaque value we use is the stack canary. We could use the return address as well. Static analyses should not be able to predict, unless they discover what is really tested, the value of the variable. Anyway the **resilience** is **weak**. How could we improve this?

Computation Transformations

Computation transformations are control flow transformations that **hide the real control-flow** behind **irrelevant statements**, add code for which there are no high level language constructs, **insert dead code**, or remove real control-flow abstractions or introduce spurious ones [1]. This includes:

- **Insertion of dead or irrelevant code**
- **Extension of loop conditions**
- **Conversion of a reducible flow graph to non-reducible**
- **Removal of library calls and programming idioms**
- **Table interpretation**
- **Addition of redundant operands**
- **Code parallelization**

Insertion of Dead or Irrelevant Code I

These transformations insert code which is not relevant to the original code in order to confuse the reader. This can be achieved by insertion of a never-execute branch behind an opaque predicate, or by adding code that is completely irrelevant to the original functionality.

Opaque predicates can be inserted somewhere into a code block. We can go further and place a portion of the original code in the true branch of the predicate and the same thing in the false branch, but with different obfuscation transformations. The result will be the same. Another possibility is to slightly change the false branch to evaluate to something (slightly) different.

Irrelevant code might e.g. perform similar computation but store its result in a different location.

Insertion of Dead or Irrelevant Code II

We have already seen an example of doing this:

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

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

OBFUSCATE(118);
CreateConsole();
```

The code inserted by the `OBFUSCATE` macro used an opaque variable, the stack canary, to create a dead branch that never executed.

Another example:

```
if( oval == VAL )           /* Opaque variable */
  cbSelf.QuadPart = 0;
else                        /* DEAD */
  cbSelf.QuadPart = 1;      /* DEAD */

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

CreateConsole();
```

Extension of Loop Conditions

Extending loop conditions makes loop termination more complex by using an opaque predicate or variable in the bounds checking condition:

```
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
```

May become:

```
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_40CD98
.text:0040CD85                cmp   eax, 0AAh
.text:0040CD9A                jae   short loc_40CD98
.text:0040CD8C                or    ecx, eax
.text:0040CD8F                mov   [edx+esi], dl
.text:0040CD92                add   eax, 11h
.text:0040CD95                inc   edx
.text:0040CD96                jmp   short loc_40CD80
.text:0040CD98 loc_40CD95:    pop   esi
.text:0040CD99                retn
```

Conversion of a Reducible CFG to Non-Reducible I

The purpose of this obfuscation is to make the flow graph of a function complex and prevent its decompilation.

- The flow graph of a function is reducible if it only uses structured statements, e.g. `for`-loops, `do-while`-loops, `while`-loops, `if`-then(-`else`), `break`, and `continue`.
- Once you start using `goto` — easy in C/C++, impossible in Java (but possible in Java bytecode (!)) — the flow graph becomes ireducible.
- Ireducible flow graphs make for a nasty high level code. This is especially evident in Java where `goto` is not available.
- The use of constructs that are not available in the high level language also falls into this category.

Conversion of a Reducible CFG to Non-Reducible II

Let's start with a simple Java program and explore its bytecode [5]:

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

    for(i=0; i<10; ++i)
    {
      System.console().writer().
        println("Step:  " + 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 "Step:  "
  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:
  return
.end method
```

Conversion of a Reducible CFG to Non-Reducible 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 "Step:  "
  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
```

```java
// 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("Step:  ").append(i).toString());
    }
}
```

```java
// 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("Step:  " + i);
    }
  }
}
```

Conversion of a Reducible CFG to Non-Reducible IV

As you have seen, the decompiler was very accurate. Now, let's try to insert an opaque predicate with a goto statement. The predicate will call the `java.util.Random.nextInt()` method to get a random number. The result will always be in the interval [0, 9].

```
new java/util/Random                           ; Create a new instance of the class
dup                                            ; Duplicate the stack top
invokespecial java/util/Random.<init>()V       ; Call the ctor, result is void
bipush 10                                      ; Push byte as int
invokevirtual java/util/Random.nextInt(I)I     ; Call nextInt(10), result is int
```

Now the stack top contains a random number, let's compare it with 10.

```
bipush 10                                      ; Push byte as int
if_icmpge never                                ; If random >= 10 GOTO never
```

Let's test for a different high bound in the first iteration:

```
iload_1
bipush 12
goto test_bounds
```

Conversion of a Reducible CFG to Non-Reducible 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 "Step:  "
  invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

Conversion of a Reducible CFG to Non-Reducible VI

```
    iload_1
    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("Step:  " + i);
    }
    i++;
  }
}
```

Conversion of a Reducible CFG to Non-Reducible 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("Step:  ").append(i).toString());
  _L2:
    i++; if(true) goto _L7; else goto _L6
  _L6:
  }
}
```

Conversion of a Reducible CFG to Non-Reducible VIII

Decompilers got confused by using the goto statement. If we do the same thing in C, we find that this method is not very efficient here:

```c
__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  jz   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
```

Conversion of a Reducible CFG to Non-Reducible 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;
}
```

As you can see, our obfuscation is not effective against C/C++. IDA was able to decompile it almost perfectly, including the `ror` instruction. If we were in Java, the situation would be very different because there wouldn't be a `goto` to use!

Conversion of a Reducible CFG to Non-Reducible X

Apparently potency depends on the language being obfuscated and the quality of the opaque predicate. If we are obfuscating Java, the potency is **medium** to **high**, as we are using bytecode instructions that have no corresponding high-level counterparts. In C, the potency is **low** to **medium**.

Resilience strongly depends on the deobfuscator.

The cost of this obfuscation is **low**.

Removal of Library Calls and Programming Idioms

Calls to library functions can tell reverse engineers a lot about our program. These calls cannot be renamed by the obfuscator. The way to obfuscate them is to provide an obfuscator-developed alternatives and use these alternatives instead of the original APIs. Such APIs include string manipulation functions, memory allocation, STL classes, etc. All of these can be replaced by alternative implementations which can be renamed and obfuscated. This increases program size and is quite potent (**medium**) and provides a **strong** resilience. The cost depends on what we want to replace.

Changing commonly used programming idioms such as traversing a linked list, arrays, etc., also falls under this category. Can we replace a linked list with something else?

Table Interpretation I

This obfuscation has a **strong** resilience, but it is also **costly**. Its simplest form requires splitting a portion of a program's code into several (or many) chunks and using a loop to iterate through many iterations, each of them running only one of the chunks. A `switch` statement, computed `goto`s or jump tables can be used. If we want to add extra resilience, the table might even have 2 or more levels!

This table approach can be interpreted as running the chunked program under a mini virtual machine. Each chunk has its number and represents an opcode of a "VM-instruction". A list of chunk numbers is then the VM's program! If we want to go even further, each of these instructions could have its own operands, there can be registers, variables, etc.

Table Interpretation 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;

  // The program is the same up to this point!

  // Now, let us define instructions of our VM.
  // That is, one instruction for each line
  // on the left marked with an a /*#*/.
  // We will add a no-op instruction too!
```

Table Interpretation 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;


  // Instructions were numbered randomly.
  // Space conserved, they are not in order.
  // This has no impact on the program.
  // Now let's create a program for our VM.
  // --- a sequence of the above instructions.
```

Table Interpretation 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
  };


  // Now add the VM as a switch statement.
```

Table Interpretation 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])
  {
    // Case branches not shuffled for clarity
    case instr_init_size:
      cbSelf.QuadPart = 0;
      break;

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

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

Table Interpretation 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])
  {
    // Case branches not shuffled for clarity
    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;
```

Table Interpretation 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])
  {
    // Case branches not shuffled for clarity
    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;
  }
}
}
```

Table Interpretation VIII

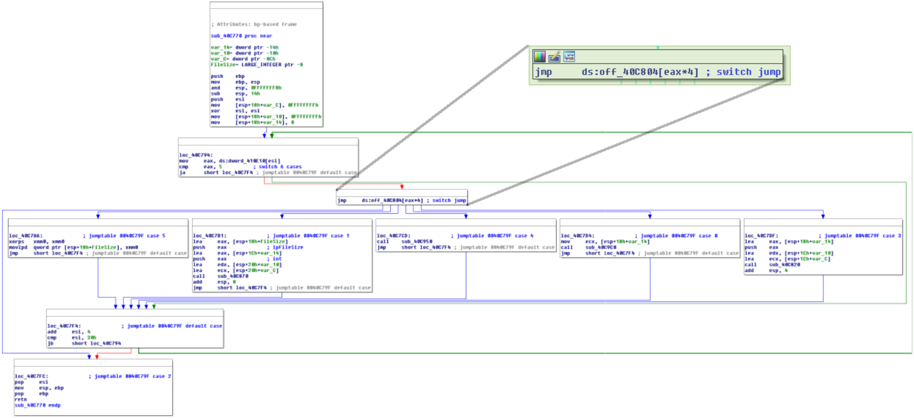

Figure: `wWinMain` as table interpretation.

Table Interpretation IX

As we have seen, our simple program got completely changed by the obfuscation. The heart of the function now lies in the

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

instruction. The jump table is a global variable, as is the VM-program at 00410e10.

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

The amount of code has grown significantly so this transformation's potency is **high** with respect to the code size metric. The transformation has a **strong** resilience. Transforming a program into the table form is, unfortunately, **costly**.

Addition of Redundant Operands I

Redundant operands increase the program's length and complexity. Not only can we add redundant operands to function/method calls, but we can also extend arithmetic expressions by using opaque variables.

Note: We have to be careful as performing redundant operations on floating point numbers will result in additional rounding and will have an impact on the accuracy of the computed variable.

We might have already noticed the use of this method in the example for extending loop conditions. The second loop iterator `j` was redundant, as were all the calculations with it.

Addition of Redundant Operands II

```c
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
```

Here we have added a redundant argument `int iRedundantBound` to the function and also a redundant opaque array `static unsigned char oarray`. The `iRedundantBound` argument contains the upper array bound and will always be set to 256 by the obfuscator. The opaque array is used only as a scratch space to calculate $(i+1)^2 - (i-1)^2 - 3i = i$. The result then initializes `array[i]=i`. The amount of code has grown so the potency of this transformation is **medium** to **high**, but the resilience of the transformation is **weak**. The transformation is **cheap**.

Code Parallelization I

Threads are used to make multiple operations run in parallel by partitioning a problem into smaller chunks, solving them in parallel and the assembling the computed result together. This principle can also be used to obfuscate code:

- If the blocks are data-independent: let each thread evaluate one block from the sequence.
- If the blocks are data-dependent: make a sequence of code blocks and let the first thread execute the first block, then unblock the second thread to execute the second code block, etc.
- Create one or more threads doing nothing useful (updating opaque predicates, throwing exceptions, . . .).

Code Parallelization II

Code parallelization's potency is **high** because parallel programs are difficult to understand. The resilience is **strong** because any automatic deobfuscator must consider all threads and their possible interactions within the program. This can lead to up to $O(n!)$ complexity! The transformation of a program into parallel is **costly**.

Parallelization is usually done by means of threads. For extra resilience (**full**) we can move portions of the code into one or more separate processes and use some form of interprocess communication mechanisms (shared memory, named and anonymous pipes, shared files, local and remote network sockets, RPC endpoints, . . .) to run target's code in parallel.

Code Parallelization 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;
}
```

Here each line of the program is transformed into a separate thread. Thread1 and Thread2 can run in parallel. Thread3 must wait until both finish their work. Thread4 must wait for Thread3 and uses Windows events to do that. Thread5 must again wait for 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(...); /* unblock thread 4 */
  return 0;
}

LPVOID WINAPI Thread4(LPVOID pUserData) {
  WaitForSingleObject(...); /* wait for thread 3 */
  ExamineRelocations( pSelf, cbSelf );
  SetEvent(...); /* unblock thread 5 */
  return 0;
}

...
```

Inlining I

Inlining is a process of copying a function's body to all locations the function is called from.

Compilers use inlining during the optimization phase for short functions, constructors, etc., to save on the prologue and epilogue code, passing arguments and issuing a `call` instruction.

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

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

int G1()
{
  g1;
}
```

→ **inlining** →

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

Functions `F1` and `G1` no longer exist in the inlined product, this information is irreversibly lost and that is why inlining is a **one-way** transformation.

Inlining II

Obfuscators use inlining to make functions more complex. The more local variables and the more lines of code a function has, the more difficult it is to analyze since it removes abstractions the programmer used. The reverse engineer will have no idea which function was inlined and where, especially if all copies of the function got inlined and the original function is no longer present in a non-inlined form in the program.

Inlining is a **cheap** transformation with **medium** potency and **one-way** resilience, particularly when combined with the next technique — outlining.

Outlining

Outlining is a process inverse to inlining. A portion of a function code is removed from the body, put aside, converted into a standalone (outlined) function, and replaced by a `call` instruction. Obfuscators use outlining, particularly in conjunction with inlining, to create potent and resilient transformations:

```
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 is an optimization technique exploiting the physical properties of the given media to increase the access or transfer speeds. In obfuscation, interleaving refers to the practice of combining the code and arguments of two or more functions into a single function, along with a new argument (or a global variable) which allows the caller to select one of the individual branches.

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

int G1()
{
  g1;
}
```

→ **interleaving** →

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

Interleaving II

Interleaving can also be used to hide variables and make it difficult to understand their purpose by combining them with other variables, as well as opaque predicates, into a single blob. Getters and setters can then be used to access the component parts of the blob, incidentally increasing code size and complexity.

Note that this technique is extremely **costly**. Its potency is heavily dependent on the actual interleaving mechanism and resilience tends to be rather **weak** as the removing code needs to be already present in the program.

Cloning

The Method Cloning obfuscation in somewhat similar to opaque predicates in that it inserts unnecessary branching for the reverse engineer to analyze. However, instead of using opaque predicates, we use the method dispatch mechanism of the programming language. This is achieved by transforming a single parent class `C` with a virtual method into a hierarchy (or set) of classes `C1`, `C2`, etc., each created using a different obfuscation transformation. Depending on the actual class instance created, it will appear to the reverse engineer that different methods are being called, while in fact all of them perform exactly the same function.

Loop Transformations

Various loop transformations can be used for the purpose of obfuscation:

- **Loop Blocking** is used in optimization to split the body of a loop into sections which will fit into a cache, improving the cache behavior of the target architecture.
- **Loop Unrolling** replicates the body of the loop multiple times, saving on the code needed for jumps to the next iteration and allowing for better optimization of the use of registers. It is particularly effective in cases where the number of iterations is known in advance.
- **Loop Fission** splits the loop into several loops with the same iteration mechanism, but different bodies. These loops can then e.g. run in parallel.

All of these techniques increase code metrics such as code size or the cyclomatic complexity, which provides their potency. Their resilience is quite **weak** if used on their own, but increases significantly when combined with other obfuscation techniques. The price is **cheap** and sometimes **free**.

Bibliography 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`.

Bibliography II

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