CODE OBFUSCATION

ATTACK STRATEGIES

Roberto Giacobazzi

Dipartimento di Informatica
Università degli Studi di Verona
Italy

ASP 2009
Most of the slides are taken from:

*Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*

Christian Collberg  
Jasvir Nagra  
ISBN-10: 0321549252  
Addison-Wesley Professional 2010, 792 pp.
The problem: Protection

An attack typically goes through multiple phases:

- The *black box* phase
- The *dynamic analysis* phase
- The *static analysis* phase
- The *editing* phase
- The *automation* phase
THE BLACK BOX PHASE

We feed the program with inputs and treat the program as an oracle

- Record the input/output behaviour
- Make a very first idea on the code behaviour
- In Architecture independent code (Java or C#) the black box phase can be locally applied to code portions
- Simple and obvious!!
The dynamic analysis phase

Code cracking is *almost* like code debugging

There is indeed a difference:

- Locate the undesired behaviour (protection) then alter the behaviour (make it clean) and finally test it!
THE DYNAMIC ANALYSIS PHASE

Attacking protection mechanism and license check:

- Locate protection: e.g., "Please enter your program activation code"

- In Windows: set a breakpoint in `GetDlgItemInt()` that translates the input code into an integer

- Look up at the call stack to find the location of the user code that called the function: likely this will be close to the validity check

- Deactivate the validity check

- Test the cracked program
THE STATIC ANALYSIS PHASE

Essential tool to understand code structure and internal functionality:

- We need a disassembler
- We can derive the CFG
- We can derive information on live variables and their values
- We can derive the structure of the memory at a given program point
AN EXAMPLE: CRACKING A DRM SYSTEM

Essential tool to understand code structure and internal functionality:

- Decrypt
- Decode
- License check
- Tamper detect
- Violation response
- User key
- Player key
- Activation code
typedef unsigned int uint;
typedef uint* waddr_t;
uint player_key = 0xbabeca75;
uint the_key;
uint* key = &the_key;
FILE* audio;
int activation_code = 666

void FIRST_FUN(){}

uint hash (waddr_t addr, waddr_t last) {
    uint h = *addr;
    for (;addr <= last; addr++) h ^= *addr;
    return h;
}

void die (char* msg) {
    fprintf(stderr, "%s!\n", msg);
    key = NULL;
}
```c
uint play(uint user_key, uint encrypted_media[], int media_len) {
    int code;
    printf("Please enter activation code: ");
    scanf("%i", &code);
    if (code != activation_code) die("Wrong code");
    *key = user_key ^ player_key;
    int i;
    for (i=0; i<=media_len; i++) {
        uint decrypted = *key ^ encrypted_media[i];
        asm volatile ("jmp L1 \n	" "\n\t" ".align 4 \n\t" "\n\t" "long 0xb0b5b0b5\n\t" "L1: \n\t"");
        if (time(0) > 1221011472) die("expired");
        float decoded = (float)decrypted;
        fprintf(audio,"%f\n",decoded); fflush(audio)
    }
}
```
void LAST_FUN(){}  
uint player_main (uint argc, char *argv[]) {  
  uint user_key = ............  
  uint encrypted_media[100] = ............  
  uint media_len = ............  
  uint hashVal = hash((waddr_t)FIRST_FUN,(waddr_t)LAST_FUN);  
  if (hashVal != HASH) die ("tampered");  
  play(user_key, encrypted_media, media_len);  
}
ATTACKING THE DRM PLAYER

Learning about the executable by standard OS primitives:

```bash
> file player
player: ELF 64-bits LSB executable, dynamically linked

> objdump -T player
DYNAMIC SYMBOL TABLE
0xa4  scanf
0x90  fprintf
0x12  time

> objdump -f player | grep start
start address 0x4006a0
```
ATTACKING THE DRM PLAYER: BLACK BOX

Feeding the program with inputs:

```bash
> player 0xca7call5 1 2 3 4
```

Please enter activation code: 42 expired!

segmentation fault

We know that the executable is stripped and dynamically linked!

- library functions by name
- check by calling `time()`
- ... there should be some code like: `if (time(0) > value)` ...
- set a breakpoint on `time(0)`, run the code and check who called it!
ATTACKING THE DRM PLAYER: DYNAMIC ANALYSIS

We use the **gdb** debugger:

```
> gdb -write -silent --args player 0xca7cal15 1000 2000 3000 4000
(gdb) break time
Breakpoint 1 at 0x400680
(gdb) run
Please enter activation code: 42
Breakpoint 1, 0x400680 in time()
(gdb) where 2
#0 0x400680 in time
#1 0x4008b6 in ??
(gdb) up
#1 0x4008b6 in ??
(gdb) disassemble $pc -5 $pc+7
0x4008b1 callq 0x400680
0x4008b6 cmp $0x48x72810, %rax
0x4008bc jle 0x4008c8
```
ATTACKING THE DRM PLAYER: DYNAMIC ANALYSIS

We can patch the executable by replacing:

```
0x4008bc jle 0x4008c8
```

with:

```
0x4008bc jg 0x4008c8
```

The player does not say expired! but still says: wrong code

```bash
> player 0xca7call5 1 2 3 4
tampered!
Please enter activation code: 99
wrong code!
Segmentation fault
```

Idea: Let us search for "wrong code" in the binary!!
ATTACKING THE DRM PLAYER

The code should look like:

```assembly
addr1: .ascii "wrong code"
       .....  
cmp read-val,activation-code
je somewhere
addr2: move addr1,reg0
call printf
```
ATTACKING THE DRM PLAYER

Search the data-segment of line addr1 then search for an instruction that contains that address:

(gdb) find 0x400ba8,+0x84,"wrong code"
0x400be2
(gdb) find 0x4006a0,+0x4f8,0x400be2
0x400862
(gdb) disassemble 0x40085d 0x400867
0x40085d  cmp  %eax,%edx
0x40085f   je   0x40086b
0x400861  mov  $0x400be2,%edi
0x400866  callq 0x4007e0

Bingo! we can replace the conditional jump:

0x40085f   je   0x40086b

with an unconditional (always) jump:

0x40085f   jmp   0x40086b
ATTACKING THE DRM PLAYER: watching mem

Still crash due to access to illegal memory location (unix), e.g. dereferencing a NULL pointer:

> player 0xca7call5 1 2 3 4 tampered!
Please enter activation code: 55
Segmentation fault

Look at what address is trying to load or write: Look into memory!

(gdb) run
Program received signal SIGSEGV
0x40087b in ?? ()
(gdb) disassemble 0x40086b 0x40087d
0x40086b mov 0x2009ce(%rip),%eax #0x601240
0x400872 mov 0x2009c0(%rip),%edx #0x601238
0x400878 xor -0x14(%rbp),%edx
0x40087b mov %edx,(%rax)

The code tries to write to an address loaded from 0x601240 with failure!
ATTACKING THE DRM PLAYER: watching mem

Watchpoint on 0x601240:

(gdb) watch *0x601240
(gdb) run
Hardware watchpoint 2: *0x601240
Old value = 6296176
New value = 0

0x400811 in ?? ()

(gdb) disassemble 0x400806 0x400812
0x400806 movq $0x0,0x200a2f(%rip) #0x601240
0x400811 leaveq

Setting key = NULL in die() forces instruction at location 0x400806 to the location at 0x601240 to 0.
ATTACKING THE DRM PLAYER: TAMPERING AGAIN

We bypass the key = NULL by overwriting NOP (x86 opcode 0x90) operations:

(gdb) set {unsigned char} 0x400806 = 0x90
(gdb) set {unsigned char} 0x400810 = 0x90

Test:

(gdb) disassemble 0x400806 0x400812
0x400806 nop
    ......
0x400810 nop
0x400811 leaveq

code cracked!!!
IS REVERSING LEGAL?

see: E. Eilam. Reversing, Secrets of reverse engineering, Wiley 2005

Trial: SEGA vs. ACCOLADE

In 1990 SEGA released the Genesis Console with secret interface.

ACCOLADE reverse engineered the Genesis interface, for producing his own games running on Genesis.

In October 1991 SEGA put the question on court for copyright infringement: The copies made by ACCOLADE during reverse engineering process (intermediate copying) violated copyright laws.

The court ruled in favor of ACCOLADE for:
- The SW developed by ACCOLADE did not contain code from SEGA.
- The opening of Genesis Console to the market opens the market itself.

Reverse Engineering is legal for interoperability!
**Is Reversing Legal?**

---

**see: The Digital Millenium Copyright Act (DMCA) 1988**

- **Interoperability**
- **Encryption research**: allows researchers to circumvent copyright protection if protection interfere with the evaluation of the encryption technology
- **Security Testing**: copyright protection can be reversed and circumvented for the evaluation and improving of the security of a computer system
- **Educational institutions and public libraries**: copyright protection can be reversed and circumvented in order to evaluate if the content can be published
- **Government investigation**: ... of course!!
- **Regulation**: protection technologies may be circumvented for access regulation (minors etc.)
- **Privacy Protection**: protection mechanisms can be violated if the protected code includes (stolen) personal information
An attack typically goes through multiple phases:

- The *black box* phase
- The *dynamic analysis* phase
- The *static analysis* phase
- The *editing* phase
- The *automation* phase
High-level structural information: We know how to do it!

Typical static analysis include:

- CFG generation
- Control Flow analysis
- Data-flow analysis
- Type based analysis
- Invariant generation by Abstract interpretation
- Model checking
**Dynamic analysis**

- **Static analysis**: extract properties that hold for all inputs (or input properties)
- **Dynamic analysis**: extract properties from the run of the program (under some selected input)

Typical dynamic analysis include:

- Debugging
- Profiling
- Tracing
- Emulation
DEBUGGING: SETTING BREAKPOINTS

We consider a *small Linux x86 Debugger* with HW and SW breakpoints, single stepping, print register and continue commands.

**ptrace**: allows a parent process to observe and control another process: examine, modify registers, data, and text segments.
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>

int main()
{
    pid_t child;
    long orig_eax;
    child = fork();

    A process forks a child and the child executes the process we want to trace.
    
    ......
Before running `exec`, the child calls `ptrace` with the first argument, equal to `PTRACE_TRACEME`. This tells the kernel that the process is being traced, and when the child executes the `execve` system call, it hands over control to its parent. The parent waits for notification from the kernel with a `wait()` call. Then the parent can check the arguments of the system call or do other things, such as looking into the registers.

......

```c
if(child == 0) {
    ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    execv("/bin/ls", "ls", NULL);
} else {
    wait(NULL);
    orig_eax = ptrace(PTRACE_PEEKUSER, child, 4 * ORIG_EAX, NULL);
    printf("The child made a system call %ld\n", orig_eax);
    ptrace(PTRACE_CONT, child, NULL, NULL);
}
return 0;
```
Tracing system calls

- Fork a child process
- Signal the kernel that the process will be traced
- Make the system call
- Look into the register (eax)
- Continue

Ingegneria e Scienze Informatiche – Verona – p.29/74
Debug a debugee:

```c
#include <sys/types.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <sys/reg.h>
#include <sys/user.h>

pid_t debugee_pid;
int status;
char *arg[] = {"debugee"};
debugee_pid = fork();
if (debugee_pid <0) error
else if (debugee_pid == 0 ) {
    int err=ptrace(PTRACE_TRACEME,0,NULL,NULL);
    execv("debugee", args);
} else {
    wait(&status);
    db_loop();
}
```

/* debugee_pid = 0
/* Child process:
control and exec debugee
/* debugee_pid > 0
/* Parent process:
wait and debug
void dbg_loop() {
    char op;
    uint32 arg;
    while (1) {
        dbg_parse(&op,&arg);
        switch (op) {
        case 'c': dbg_continue(); break;
        case 'b': dbg_setSWBreakpoint(arg); break;
        case 'B': dbg_setHWBreakpoint(arg); break;
        case 'r': dbg_printRegs(); break;
        case 's': dbg_step(arg); break;
        }
        if (status==1407) {
            enum event e = dbg_getEvent();
            if (e==swBPhit) dbg_handleSWBreakpoint();
            else if (e==hwBPhit) dbg_handleHWBreakpoint();
            dbg_setDbgReg(6,0x0);
        }
    }
}
THE DEBUGGER

void dbg_continue() {
    err = ptrace(PTRACE_CONT, debugee_pid, NULL, NULL);
    wait(&status)
}
/* PTRACE_CONT: Restarts the stopped child process.

void dbg_step(int steps) {
    for (i=1; i<=steps; i++) {
        int err = ptrace(PTRACE_SINGLESTEP, debugee_pid, NULL, NULL);
        wait(&status);
    }
}
/* PTRACE_SINGLESTEP: Restarts the stopped child, but arranges for the
    child to be stopped after execution of a single instruction.

The behaviour is largely hidden inside the ptrace system call!!
DEBUGGING: SETTING HW-BREAKPOINTS

The x86 has 4 registers D0, D1, D2, D3 for breakpoint address and D7 for controlling the debugging registers:

D7=1 trigger when execution hits the byte in address D0
D7=0 turn off the breakpoint: back to single step

```c
uint32 dbg_getDbgReg(int reg) {
    return ptrace(PTRACE_PEEKUSER, debugee_pid,
                  offsetof(struct user, u_debugreg[reg]), NULL);
}

/* PTRACE_PEEKUSER: Reads a word at offset addr in the child’s USER area, which holds the registers and other information. */
```
The x86 has 4 registers $D0, D1, D2, D3$ for breakpoint address and $D7$ for controlling the debugging registers:

- $D7=1$ trigger when execution hits the byte in address $D0$
- $D7=0$ turn off the breakpoint: back to single step

```c
void dbg_setDbgReg(int reg, uint32 val) {
    int err=ptrace(PTRACE_POKEUSER, debugee_pid,
                   offsetof(struct user,u_debugreg[reg]), val);
}

/* PTRACE_POKEUSER: Copies the word data to offset addr in the child’s USER area. */
```
The x86 has 4 registers $D_0, D_1, D_2, D_3$ for breakpoint address and $D_7$ for controlling the debugging registers:

- $D_7=1$ trigger when execution hits the byte in address $D_0$
- $D_7=0$ turn off the breakpoint: back to single step

```c
void dbg_setHWBreakpoint(uint32 addr) {
    dbg_setDbReg(0, addr);
    dbg_setDbReg(7, 0x1);
}

void dbg_handleHWBreakpoint(uint32 addr) {
    dbg_setDbReg(7, 0x0);
    dbg_step(1);
    dbg_setDbReg(7, 0x1);
}
```
DEBugging: setting Memory-watchpoints

The x86 has only 4 registers $D_0, D_1, D_2, D_3$ for breakpoint address. So we can only monitor 4 memory words:

```c
void dbg_getRegs(struct user_regs_struct *regs) {
    int err = ptrace(PTRACE_GETREGS, debugee_pid, NULL, regs)
}
/* PTRACE_GETREGS: Copies the child’s general purpose to location data in the parent. */

void dbg_setRegs(struct user_regs_struct *regs) {
    int err = ptrace(PTRACE_SETREGS, debugee_pid, NULL, regs)
}
/* PTRACE_SETREGS: Copies the child’s general purpose registers from location data in the parent. */
```
The x86 has only 4 registers D0, D1, D2, D3 for breakpoint address. so we can only monitor 4 memory words:

```c
void dbg_printRegs() {
    struct user_regs_struct regs;
    dbg_getRegs(&regs);
    printf("edp=0x%x,esp=0x%x,eax=0x%x,ebx=0x%x,ecx=0x%x,edx=0x%x\n",
            regs.edp, regs.esp, regs.eax, regs.ebx, regs.ecx, regs.edx);
}
```
Idea: replace the instructions at the breakpoint address with one that will generate a signal giving control back to the debugger

```c
uint32 dbg_readText(uint32 addr) {
    return ptrace(PTRACE_PEEKTEXT, debugee_pid, addr, NULL);
}

void dbg_writeText(uint32 addr, uint32 instr) {
    return ptrace(PTRACE_POKETEXT, debugee_pid, addr, NULL);
}

uint32 origInstr, trapInstr swBPAddr;
void dbg_setSWBreakpoint(uint32 addr) {
    swBPAddr = addr;
    trapInstr = origInstr = dbg_readText(swBPAddr);
    ((char*) &trapInstr[0] = 0xCC; /* int 3
    dbg_writeText(swBPAddr, trapInstr);
}
```
Debugging: setting SW-breakpoints

Idea: when the breakpoint is hit: restore the original instruction, single step on it and restore the breakpoint for future use.

```c
int dbg_handleSWBreakpoint() {
    struct user_reg_struct regs;
    dbg_getRegs(&regs);
    dbg_writeText(swBPAddr, origInstr);
    regs.eip--;
    dbg_setRegs(&regs);
    dbg_step(1);
    dbg_writeText(swBPAddr, trapInstr);
}
```
**Reverse Debugging**

- Run the program
- Set a breakpoint in the code near the pointer setting
- Look-up the call stack

**Cracked CODE!!**

- The program crash?
  - No
  - Yes

**Removing a license check**

...and tamperproofing

**Time consuming!!**

...we need backward execution!

*Can we run backward until we find the location that has affected the value of variable X?*

**Idea: Backward watchpoint**
REVERSE DEBUGGING


long step_cnt = 0;
long sc_stop_val = -1;

void step() {
  step_cnt++;
  if (step_cnt == sc_stop_val)
    trap to the debugger
}

int call_depth = 0;

void enter() {call_depth++;}
void leave() {call_depth--;

The algorithm is counter based!

Counter step_cnt for every source line executed

Keep track of the depth of call-stack:
enter and leave inc/dec call_depth
Example: execute `proc(2)`

```
0 int proc(int x) {
1    enter();
2    int R,L;
3    STEP[2]();
4    int k=0;
5    STEP[3]();
6    while (k < x) {
7        STEP[4]();
8            R = foo(k);
9        STEP[5]();
10       L = L+R;
11        k++; }
12    STEP[6]();
13    leave(); return L;
14 }

10 int foo(int w) {
11    enter();
12    int R;
13    STEP[12]();
14    R = 2*W;
15    STEP[13]();
16    leave(); return R
17 }
```
Example: setting a breakpoint at line \( n \)

```c
int brkpt_cnt = 0;
long bc_stop_val = -1;

void brkpt() {
    step_cnt++;
    brkpt_cnt++;
    if (brkpt_cnt == bc_stop_val ||
        step_cnt == sc_stop_val)
        trap to the debugger
}

void dbg_set_breakpoint(int line) {STEP[line] = brkpt;}
void dbg_clear_breakpoint(int line) {STEP[line] = step;}
void dbg_continue(int n) {bc_stop_val = brkpt_cnt + n;}
void dbg_bcontinue(int n) {bc_stop_val = brkpt_cnt - n;}
```

Replace the call to `step()` at line \( n \) with a call to `brkpt()`
Example: `continue bcontinue`

```c
int brkpt_cnt = 0;
long bc_stop_val = -1;
void brkpt() {
    step_cnt++;
    brkpt_cnt++;
    if (brkpt_cnt == bc_stop_val ||
        step_cnt == sc_stop_val)
        trap to the debugger
}
void dbg_set_breakpoint(int line) {STEP[line] = brkpt;}
void dbg_clear_breakpoint(int line) {STEP[line] = step;}
void dbg_continue(int n) {bc_stop_val = brkpt_cnt + n;}
void dbg_bcontinue(int n) {bc_stop_val = brkpt_cnt - n;}
```

In `bcontinue` re-execute the program: it will stop at $n^{th}$ previous breakpoint.
The profine of the execution of a program \( P \) is a record of the number of times its different parts have executed.

**Frequency spectrum analysis:** execution count for program parts (e.g., functions, slices, edges in CFG etc.): function \( \text{foo}() \) is called 2345 times as well as function \( \text{baz}() \)

**Idea:** Instrument the code, i.e., add code that allows us to count the use of relevant structures: \( \mathcal{I} : \text{Program} \rightarrow \text{Program} \)

1. Input \( P \);
2. Choose program input \( x \);
3. Run the instrumented code \( \mathcal{I}(P)(x) \);
4. Collect count-log for \( x \) and \( P \) in \( C_x \);
5. goto 2 until \( \varphi \) is satisfied
6. Make statistics out of \( \{C_{x_1}, \ldots, C_{x_n}\} \)
Tracing a program \( P \) is collecting blocks (and instructions) truly executed.

Problem: Huge size of traces!

Idea: Compress traces of \( P \) into a CFG \( G \): the Directed Acyclic Graph (DAG) of \( G \) is called Whole-Program Path, WPP of \( P \).

Consider the trace: 1424252525252523:

```
CFG:
```

```
{s} → 1 A A C C B 3
A → 42
B → 52
C → BB
```
Tracing a program $P$ is collecting blocks (and instructions) truly executed.

**Problem:** Huge size of traces!

**Idea:** Compress traces of $P$ into a CFG $G$: the Directed Acyclic Graph (DAG) of $G$ is called *Whole-Program Path*, WPP of $P$.

Consider the trace: 1424252525252523:

![Graph](image)

**WPP:**

**input:** Trace $T$;

1. Create a CFG grammar with $Start := S \rightarrow \varepsilon$ and $G = \{Start\}$;
2. $c := \text{hd}(T)$;
3. $Start := Start :: c$;
4. If $P \in G$ and digram $DD$ is twice in $P$: $G := G[DD/R] \cup \{R \rightarrow DD\}$; ($\pi_1$)
5. If $R \rightarrow \gamma \in G$ and $R$ is only used once in $G$: $G := G[R/\gamma] \setminus \{R \rightarrow \gamma\}$; ($\pi_2$)
6. goto 4 until $\pi_1 \land \pi_2$ hold;
7. goto 2 until $T = \emptyset$

**return:** CFG $G$;

**Invariant:**

$\pi_1$: no digram (pair of adjacent symbols) appears more than once
$\pi_2$: every rule is useful
Compressing trace $T = \text{ababababababa}$:

Result:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow 0 \rightarrow 1 1 1 a \ \backslash \n$</td>
<td>ababab</td>
</tr>
<tr>
<td>$1 \rightarrow 2 2$</td>
<td>abab</td>
</tr>
<tr>
<td>$2 \rightarrow a b$</td>
<td>ab</td>
</tr>
</tbody>
</table>
RECONSTRUCTING SOURCE

The typical development cycle:

and its inverse:
The disassembled is a kind of compiler: Let us see how translation of an IA-32 instruction works!

- **Opcode translation table**
- **Format translation table**
- **Code-symbol converter**
# Disassembly: Opcode Translation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Operands</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>call</td>
<td>addr</td>
<td>function call to <code>addr</code></td>
</tr>
<tr>
<td>1</td>
<td>calli</td>
<td>reg</td>
<td>function call to address in <code>reg</code></td>
</tr>
<tr>
<td>2</td>
<td>brg</td>
<td>offset</td>
<td>branch to <code>pc + offset</code> if flag for &gt; is set</td>
</tr>
<tr>
<td>3</td>
<td>inc</td>
<td>reg</td>
<td>increment <code>reg</code></td>
</tr>
<tr>
<td>4</td>
<td>bra</td>
<td>offset</td>
<td>branch to <code>pc + offset</code></td>
</tr>
<tr>
<td>5</td>
<td>jmpi</td>
<td>reg</td>
<td>jump to address in <code>reg</code></td>
</tr>
<tr>
<td>6</td>
<td>prologue</td>
<td></td>
<td>beginning of a function</td>
</tr>
<tr>
<td>7</td>
<td>ret</td>
<td></td>
<td>return from function</td>
</tr>
<tr>
<td>8</td>
<td>load</td>
<td>reg₁, (reg₂)</td>
<td>the content pointed to <code>reg₂</code> goes into <code>reg₁</code></td>
</tr>
<tr>
<td>9</td>
<td>loadi</td>
<td>reg, imm</td>
<td>the value <code>imm</code> goes into <code>reg</code></td>
</tr>
<tr>
<td>10</td>
<td>cmpi</td>
<td>reg, imm</td>
<td>compare <code>reg</code> and <code>imm</code> and set flag</td>
</tr>
<tr>
<td>11</td>
<td>add</td>
<td>reg₁, reg₂</td>
<td>the sum of <code>reg₁</code> and <code>reg₂</code> goes into <code>reg₁</code></td>
</tr>
<tr>
<td>12</td>
<td>brge</td>
<td>offset</td>
<td>branch to <code>pc + offset</code> if flag ≥</td>
</tr>
<tr>
<td>13</td>
<td>breq</td>
<td>offset</td>
<td>branch to <code>pc + offset</code> if flag =</td>
</tr>
<tr>
<td>14</td>
<td>store</td>
<td>(reg₁), reg₂</td>
<td>the value in <code>reg₂</code> is stored in address in <code>reg₁</code></td>
</tr>
</tbody>
</table>
**DISASSEMBLY: OPCODE TRANSLATION**

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary</th>
<th>Mnemonic</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>[9,0,12]</td>
<td>loadi</td>
<td>r0,12</td>
</tr>
<tr>
<td>3:</td>
<td>[9,1,4]</td>
<td>loadi</td>
<td>r1,4</td>
</tr>
<tr>
<td>6:</td>
<td>[14,0,1]</td>
<td>store</td>
<td>(r0),r1</td>
</tr>
<tr>
<td>9:</td>
<td>[11,1,1]</td>
<td>add</td>
<td>r1,r1</td>
</tr>
<tr>
<td>11:</td>
<td>[3,4]</td>
<td>inc</td>
<td>r4</td>
</tr>
<tr>
<td>14:</td>
<td>[4,-5]</td>
<td>bra</td>
<td>-5</td>
</tr>
<tr>
<td>15:</td>
<td>[7]</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

Code and data are mixed!!!

Conde may change

loadi r0,12
loadi r1,4
store(r0),r1

add r1,r1
inc r4
bra -5
ret

loadi r0,12
loadi r1,4
store(r0),r1

add r1,r1
bra 4
ret
Some good disassemblers (for Microsoft platforms):

- Most debuggers include a disassembler, **OllyDbg**:
  http://www.ollydbg.de/

- **PVDasm**: is a Free, Interactive, Multi-CPU disassembler:
  http://pvdasm.reverse-engineering.net/

- **IDA-Pro**: Interactive disassembler by www.datarescue.com works for:
  IA-32, IA-64 (Itanium), AMD64 etc. and supports: PE (windows), ELF
  (Linux), and XBE (XBOX)

- **ILDasm**: for Microsoft intermediate language (MSL)
The two work together (interactively!): IDA-Pro
DISASSEMBLY AND DEBUGGER

The two work together (interactively!):
We consider disassembly stripped binary (the harder problem):

**see:** DIABLO, http://diablo.elis.ugent.be/

**Static Disassembly:** the file is examined by the disassembler but it is not executed
- Advantage: processes the entire file at once
- Time is proportional to the size of the program

**Dynamic Disassembly:** monitor the execution of the file on some inputs in order to disassemble only the instructions that are executed
- Disassembles a slice of the program
- Time is proportional to the number of executed instructions (much bigger)
We consider disassembly stripped binary (the harder problem):

*see:* DIABLO, [http://diablo.elis.ugent.be/](http://diablo.elis.ugent.be/)

**Static Disassembly: why so hard?**

- In variable length instruction sets, instructions may overlap
- Data is commonly mixed with instructions
- Hard to determine the target of jumps
- Hard to find the beginning of functions with indirect call
- Hard to find the end of a function without `return`
- Hand-written assembly may be different from standard compile-time generated assembly
- Aggressive code compression may destroy code structure (overlapping functions)
- Extremely hard for self-modifying code.
Two main algorithmic approaches

- **Linear Sweep**: begins the disassembly at the input's program entry point and sweeps through the entire text section.
  - Main weakness: miss-interpretation of data embedded in the instruction stream.

- **Recursive Traversal**: takes into account the control flow of the file to disassemble by determining the control flow successors of each branch instruction (not always possible/easy).
The problem is that data in the text segment will be interpreted as code (wrong!!)

```c
uint32 startAddr, endAddr;

void DisLinear(uint32 addr) {
    while (startAddr <= addr <= endAddr) {
        I = decode instruction at addr;
        addr = addr + length(I);
    }
}

void main()
{
    ep = program entry point;
    size = text section size;
    startAddr = ep;
    endAddr = ep + size;
    DisLinear(ep);
}
```
DISASSEMBLY: RECURSIVE TRAVERSAL

recursively follows the branches: you need to correctly catch the targets!!!

```c
uint32 startAddr, endAddr;

void DisRecursive(uint32 addr) {
    while (startAddr <= addr <= endAddr) {
        if (addr has been visited already) return;
        I = decode instruction @ addr;
        mark addr as visited;
        if (I is a branch instruction)
            for each possible target t of I do
                DisRecursive(t);
        return
    }
    else addr = addr + length(I);
}
```
The decompiler translates the CFG into an *Abstract Syntax Tree (AST)* isomorphic to a high-level language structured (*if*, *while*, *for*, *repeat*) of sequences of assignments:

- Basic blocks: sequences of assignments
- CFG structure: basic iteration and conditional structures
- Typically includes its own disassembler (with its own problems!)
DECOMPILATION

What is hard in decompilation?

- Mismatch between legal assembly code and high-level control structures: (e.g., \texttt{goto} in Java bytecode but not in Java)
- Identify the call to library functions
- Identify idioms of different compilers
- Remove machine dependent artifacts
- Undo compiler optimizations (e.g., loop unrolling)
Example:

```
prolog
loadi r0,100
loadi r1,1
store(r0),r1

loadi r0,100
loadi r1,100
cmpi r1,10
brge B6

loadi r0,100
loadi r1,(r0)
cmpi r1,5
brge B7

loadi r0,100
loadi r1,(r0)
inc r1
store(r0),r1
bra B4

bra B5

call 0x459F8
bra B1
```
Example:

```c
void foo() {
    x=1;
    while (x<=10) {
        if (x<5)
            x ++;
        else
            x+=2;
        printf();
    }
}
```
**Decompile: Recovering Control Flow**

C. Cifuentes and K.J. Gough. *Decompile of binary programs, 1995*:

- **Input**: $X$ stripped executable
  - Parse $X$ distinguishing between *code-text* and *data*
  - Determine the compiler by checking *signatures*
  - Disassemble the *text* and generate a *call graph* $G$
  - Remove from $G$ functions that match the signatures
  - Replace known idioms with high-level constructs
  - Optimize $G$ by removing jump-to-jumps
  - **RECOVERSTATEMENTS($G$)**
  - Classify nodes by **RESTRUCTURELOOPS($G$)** and **RESTRUCTUREIFs($G$)**
  - Traverse $G$ and build the *AST*: for each block which is *head of control structure* traverse his body *depth-first* until the next node
  - Traverse the *AST* and emit the source code
For each Basic Block in $G$:

- Perform **reaching definitions** on condition code and replace machine code tests and branches with *if-goto*:
  
  \[
  i: \text{ cmpi } r, \text{imm} \\
  \downarrow \\
  j: \text{ if } r > \text{imm} \text{ goto lab}
  \]

- Perform **reaching definitions live interprocedural register analysis** and replace temporary registers $tmp$ with its symbolic contents where used:
  
  \[
  i: \text{ add } tmp, v \\
  \downarrow \\
  j: \text{ cmpi } tmp + v, \text{imm} \\
  j: \text{ cmpi } tmp, \text{imm}
  \]

- Replace calls to library functions with corresponding symbolic calls:
  
  \[
  i: \text{ call } \text{addr} \Rightarrow i: \text{ call } \text{name}
  \]
Decomposition: Restructuring Loops (G)


$I(h)$ is an interval graph of $G$ if it is the maximal single-entry subgraph of $G$ having $h$ as the only entry node and where all closed paths include $h$.

Input CFG $G$:
1. Derive the Interval Graphs $\{I_1(h_1), \ldots, I_m(h_m)\}$ of $G$;
2. Determine loop-type: pre-tested, post-tested, endless
   ✓ forall $I(h)$ find the latching node in $I(h)$;
   ✓ match the loop against:
   - Pre-tested: while
   - Post-tested: repeat
   - Endless: loop
3. Repeat from 2 until all intervals in $G$ are collapsed into one (loop) node
DECOMPILATION: DISCOVERING LOOPS
DECOMPILATION: DISCOVERING LOOPS

repeat
....
until ...

Header

Latching

Follow
DECOMPILATION: DISCOVERING LOOPS

while ... {
    repeat
    ....
    until ...
}
DECOMPILATION: RESTRUCTURING $\text{IFS}(G)$

Input CFG $G$:
1. Number the nodes of $G$ in reverse postorder;
2. Determine unresolved nodes:
   \[
   \text{unresolved} := \emptyset
   \]
   forall node $m$ by descending reverse postorder do
   if $m$ is 2-way and not loop header then
   if $\exists n: n = \max \{ i \mid \text{idom}(i) = m \land |\text{inEdge}(i)| \geq 2 \}$
   then follow$(m) := n$
   foreach $x \in \text{unresolved}$ do
   follow$(x) := n$; unresolved := unresolved - \{x\}
   else unresolved := unresolved $\cup \{m\}$
3. Match the if type against:

\[\text{Diagram of CFGs}\]

Header

Follow
DECOMPILATION: DISCOVERING IFS

```
if (...) {  
    ....
} else {
    ....
}
```

```
while ... {
    repeat
    ....
    until ...
}
```
DECOMPILATION: DISCOVERING IFS

```
while {...} {
  repeat
  ....
  until ...
}

if (...) {
  if (...) {
  ....
  }
  else {
  ....
  }
}
```