Program Analysis

Lecture 05: *Reconstructing Information*
Winter term 2011/2012

Prof. Thorsten Holz

Announcements

- Second exercise will be published today
  - First exercise will be discussed after the lecture
    - Grading for first exercise soon
- Next Wednesday: talk by @kkotowicz
  - See http://www.nds.rub.de/teaching/lectures/471/

Last Week

- x86 subroutines
- Calling conventions
- Higher-level structures
- Control-flow structures

Top-controlled Loops

High-level language

```
c = 0;
while( c < 0x100 )
{
   Function1();
   c++;
}
```

Assembler

```
   mov ecx, 0
lab_start:
   cmp ecx, 0x100
   jae lab_exit
   call function1
   inc ecx
   jmp lab_start
lab_exit:
   …
```

Bottom-controlled Loops

High-level language

```
c = 0;
do
{
   Function1();
   c++;
}
while( c < 0x100 );
```

Assembler

```
   mov ecx, 0
lab_start:
   call function1
   inc ecx
   cmp ecx, 0x100
   jl lab_start
   …
```

Loop Control

High-level language

```
c = 0;
while( TRUE )
{
   c++;
   if( c == 3 )
      continue;
   if( c == 5 )
      break;
   Function1();
}
```

Assembler

```
   mov ecx, 0
lab_start:
   inc ecx
   cmp ecx, 3
   jnz lab_not3
   jmp lab_start
lab_not3:
   cmp ecx, 5
   je lab_exit
   call function1
   jmp lab_start
lab_exit:
   …
```

For Loops

High-level language

```
c = 3;
for( int i=0; i<c; i++)
{
   Function1();
}
```

Assembler

```
   mov [var_c], 3
   mov [var_i], 0
   jmp after_inc
loop:
   mov eax, [var_i]
   add eax, 1
   mov [var_i], eax
after_inc:
   mov eax, [var_i]
   cmp eax, [var_c]
   jge  exit_loop
   call Function1
   jmp loop
exit_loop:
```

Optimizations

- Modern compilers optimize a lot
  - Eliminate dead code and variables
  - Computer formulars during compile time
  - Avoid instructions and variables if possible
  - Optimize loops, e.g., by unrolling them
  - Inlining, use registers whenever possible
- Code gets harder to understand and analyze

Finding Data Structures

Data Structure Reconstruction

- Different types of data structures
  - Elementary data types
  - Arrays (Strings)
  - Structs and unions
  - Classes and objects
- Memory location for data structures
  - Variable or fixed

Memory Location for Variables

- Global variables / static variables
  - Stored in data section
  - Fixed address, known at compile time
  - For example, `mov eax, [0x00402030]`

Memory Location for Variables

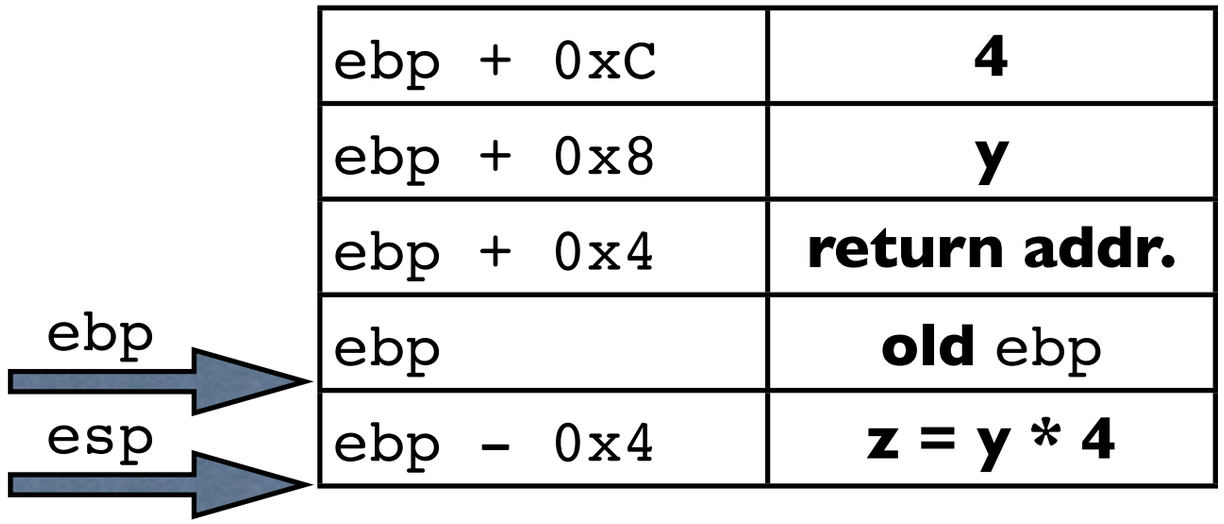
- Local variables
  - Stack variables

```
mov eax, [ebp-0x6] ; negative offset to EBP
mov eax, [esp+0xC] ; positive offset to ESP
```

- Register variables
  - *register* keyword and/or compiler decision

Memory Location for Variables

- Local variables
  - Stack variables




```
mov eax, [ebp-0x6] ; negative offset to EBP
mov eax, [esp+0xC] ; positive offset to ESP
```

- Register variables
  - *register* keyword and/or compiler decision

Memory Location for Variables

- Imported variables
  - Global variables from another module
  - Access via Import Address Table (IAT)

```
mov ecx, [IAT_ENTRY_Variable1]
mov eax, [ecx]
```

Memory Location for Variables

- Thread variables
  - Stored in Thread Local Storage (TLS)
  - Access via TLS API
    - *TlsGetValue(), TlsSetValue()*
  - Or via compiler-specific extensions

```
declspec( thread )
```

Elementary Data Types

- Mapping to generic 32/64 bit data words
  - 8 bit (BYTE), 16 bit (WORD), and 32 bit (DWORD)
  - Boolean
  - Pointer
- Floating points
  - There are specific floating point instructions
  - Not covered in detail, typically not relevant for us

Arrays

- List of uniform data elements
- Sequential storage
- Easy and fast access
  - Element address = Start + index * element size
- Recognizing arrays
  - Easy if index is not constant but computed
  - Else: Array looks like consecutive variables

Local Array

High-level language

```
void main()
{
   WORD LocalArray[10];
   DWORD i = 6;
   LocalArray[0] = 0x10;
   LocalArray[1] = 0x20;
   LocalArray[2] = 0x30;
   LocalArray[i] = 0x60;
}
```

ebp	old ebp
ebp - 0x2	LocalArray[9]
...	...
ebp - 0x14	LocalArray[0]
ebp - 0x18	i

Assembler

```
push   ebp
mov    ebp, esp
sub    esp, 18h
mov    dword ptr [ebp-0x18], 6
mov    word ptr [ebp-0x14], 10h
mov    word ptr [ebp-0x12], 20h
mov    word ptr [ebp-0x10], 30h
mov    eax, [ebp-0x18]
mov    word ptr [ebp-0x14+eax*2], 60h
mov    esp, ebp
pop    ebp
retn
```

Global Array

High-level language

```
WORD GlobalArray[10];

void main()
{
   DWORD i = 6;
   GlobalArray[0] = 0x10;
   GlobalArray[1] = 0x20;
   GlobalArray[2] = 0x30;
   GlobalArray[i] = 0x60;
}
```

Assembler

```
push    ebp
mov     ebp, esp
sub     esp, 4h
mov     [ebp-4], 6
mov     word_406120, 10h
mov     word_406122, 20h
mov     word_406124, 30h
mov     eax, [ebp-4]
mov     word_406120[eax*2], 60h
mov     esp, ebp
pop     ebp
retn
```

Remember: global variables have fixed address, known at compile time

Strings

- There are different ways to store strings
- Typical approach: array of CHAR/WIDECHAR
- But how can we store the length of the string?
  - Zero-terminated C string
  - Pascal strings: first element is length
  - Structs

```
typedef struct{ int Len; char *Buffer } ANSI_STRING;
typedef struct{ int Len; WCHAR *Buffer } UNICODE_STRING;
```

Structs

- Elements are accessed via name instead of index
  - Compiler transforms name into numerical offset relativ to starting address of struct
- Alignment / packing
  - Fields are typically aligned on pre-determined boundary (e.g., 0x0, 0x4, 0x8, ...)
  - Alignment can be changed to reduce memory requirements (called *packing*)

Local / Globale Structs

High-level language

```
typedef struct {
   WORD w;
   DWORD dw;
   CHAR c; }
MY_STRUCT;

MY_STRUCT gs;

void Struct1()
{
   MY_STRUCT s;
   s.w = 12;
   s.dw = 4;
   s.c = 8;
   gs.w = 3;
   gs.dw = 5;
   gs.c = 6;
}
```

Assembler

```
push    ebp
mov     ebp, esp
sub     esp, Ch
mov     [ebp-Ch], Ch
mov     [ebp-8], 4
mov     [ebp-4], 8
mov     word_406120, 3
mov     dword_406124, 5
mov     byte_406128, 6
mov     esp, ebp
pop     ebp
retn
```

Pointer to Structs

High-level language

```
typedef struct
{
   WORD w;
   DWORD dw;
   CHAR c;
}
MY_STRUCT;

void Struct3( MY_STRUCT *s)
{
   s->w = 12;
   s->dw = 4;
   s->c = 8;
}
```

Assembler

```
push    ebp
mov     ebp, esp
mov     eax, [ebp+4]
mov     word ptr [eax], 0Ch
mov     eax, [ebp+4]
mov     dword ptr [eax+4], 4
mov     eax, [ebp+4]
mov     byte ptr [eax+8], 8
mov     esp, ebp
pop     ebp
retn
```

Optimization

Motivation

- Up to now we covered
  - Machine code
  - x86 basics
  - Assembler programs
  - Finding important info
- Now we focus on *optimization*, i.e., how can a compiler transform code such that it is more efficient? How does this affect the analysis process?

Optimization

- Code Optimizations
  - Inlining
  - Unrolling
  - ...
- Processor Features
  - Pipelined execution
  - Instruction cache
  - Branch prediction

Constant Propagation

- Very simple idea: when a variable has a constant value, the compiler can insert it wherever the variable is used (until it is modified)
- Can be used repeatedly to eliminate more code

Example

```
void main() {
   int size = 256;
   UpdateSize(size * 5);
}
```

Optimized

```
void main() {
   UpdateSize(1280);
}
```

Dead Code

- Delete code that can not be reached

Example I

```
void main() {
   printf("foo\n");
   return;

   printf("bar\n");
}
```

Example II

```
void main() {
   int A = 42;
   int Size = 32 * 5;

   UpdateSize(Size * 5);
}
```

- Compiler can detect that second printf() can not be reached and automatically removes it
- Might be an advantage since we do not need to analyze this code

Inlining

- Compiler can use *inline expansion* for a particular function such that stack overhead is reduced
- Compiler inserts complete body of a function in every place in the code where this function is used
  - Space benefit for small functions
  - Enables other kinds of optimization
- Many advantages compared to macros

Inlining

Example

```
int Increment(int x) {
   return x+1;
}

int main(int argc, char *argv[]) {
   return Increment(argc);
}
```

Without Inlining

```
.text:00401000 sub_401000      proc near               ; CODE XREF: _main+7↓p
.text:00401000
.text:00401000 arg_0           = dword ptr  8
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 mov     eax, [ebp+arg_0]
.text:00401006                 add     eax, 1
.text:00401009                 pop     ebp
.text:0040100A                 retn
.text:0040100A sub_401000      endp
.text:0040100A
.text:0040100A ; ---------------------------------------------------------------------------
.text:0040100B                 align 10h
.text:00401010
.text:00401010 ; =============== S U B R O U T I N E =======================================
.text:00401010
.text:00401010 ; Attributes: bp-based frame
.text:00401010
.text:00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401010 _main           proc near               ; CODE XREF: ___tmainCRTStartup+10A↓p
.text:00401010
.text:00401010 argc            = dword ptr  8
.text:00401010 argv            = dword ptr  0Ch
.text:00401010 envp            = dword ptr  10h
.text:00401010
.text:00401010                 push    ebp
.text:00401011                 mov     ebp, esp
.text:00401013                 mov     eax, [ebp+argc]
.text:00401016                 push    eax
.text:00401017                 call    sub_401000
.text:0040101C                 add     esp, 4
.text:0040101F                 pop     ebp
.text:00401020                 retn
.text:00401020 _main           endp
```

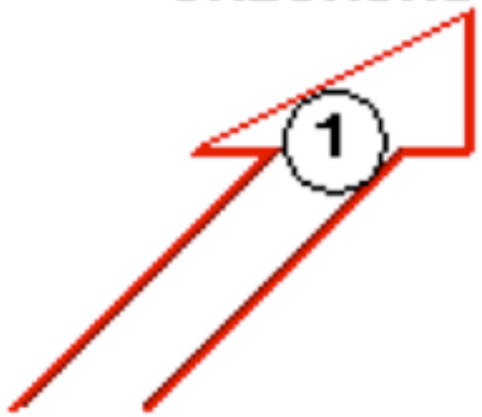
With Inlining


```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main           proc near               ; CODE XREF: ___tmainCRTStartup+10A↓p
.text:00401000
.text:00401000 arg_0           = dword ptr  4
.text:00401000
.text:00401000                 mov     eax, [esp+arg_0]
.text:00401004                 inc     eax
.text:00401005                 retn
.text:00401005 _main           endp
```

Iterative Optimization I


```
void caller()
{
    /* ... */
    strmute(p, 1);
    /* ... */
}

inline void
strmute(char *str, bool bCase)
{
    if(bCase == 1)
    {
        /* make uppercase */
    }
    else
    {
        /* make lowercase */
    }
}
```




```
void caller()
{
    /* ... */
    char *str = p;
    bool bCase = 1;
    if(bCase == 1)
    {
        /* make uppercase */
    }
    else
    {
        /* make lowercase */
    }
    /* ... */
}
```

Source: Rolf Rolles - Binary Literacy

Iterative Optimization II


```
void caller()
{
    /* ... */
    char *str = p;
    bool bCase = 1;
    if(bCase == 1)
    {
        /* make uppercase */
    }
    else
    {
        /* make lowercase */
    }
    /* ... */
}
```

CONSTANT PROPAGATION

(2)

```
void caller()
{
    /* ... */
    char *str = p;
    if(1 == 1)
    {
        /* make uppercase */
    }
    else
    {
        /* make lowercase */
    }
    /* ... */
}
```

Source: Rolf Rolles - Binary Literacy

Iterative Optimization III


```
void caller()
{
    /* ... */
    char *str = p;
    if(1 == 1)
    {
        /* make uppercase */
    }
    else
    {
        /* make lowercase */
    }
    /* ... */
}
```

UNREACHABLE CODE ELIMINATION

3

(IF SIMPLIFICATION)

```
void caller()
{
    /* ... */
    char *str = p;
    if(1 == 1)
    {
        /* make uppercase */
    }
    else
    {
        /* make lowercase */
    }
    /* ... */
}
```

Source: Rolf Rolles - Binary Literacy

Iterative Optimization IV


```
void caller()
{
    /* ... */
    strmute(p, 1);
    /* ... */
}
```

```
void caller()
{
    /* ... */
    char *str = p;
    /* make uppercase */
    /*    ...    */
}
```

We have inlined half of a function

Source: Rolf Rolles - Binary Literacy

Questions?

Contact:
Prof. Thorsten Holz
thorsten.holz@rub.de
@thorstenholz on Twitter

More information:
http://syssec.rub.de
http://moodle.rub.de

Sources

- Lecture *Software Reverse Engineering* at University of Mannheim, spring term 2010 (Ralf Hund, Carsten Willems and Felix Freiling)
- Rolf Rolles: "Binary Literacy", 2007
  - Highly recommended reading!
  - See link in Moodle