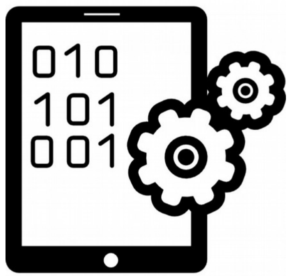
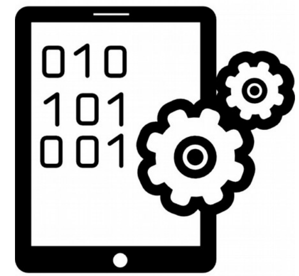
Reverse Engineering
Class 3

Executable Binaries

Binaries Analysis



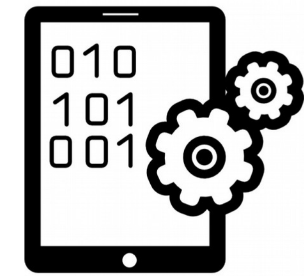
- Static analysis based on the executable format
  - Exported functions and variables
  - Imported functions and variables
  - Symbols and Strings tables
  - Debug information
- But, not everything is exported and has symbols!



Binaries Analysis

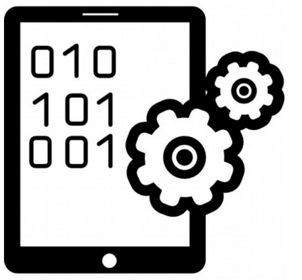
- When compiling and linking information is lost
  - Function and variables names, comments
  - Variables types
  - Non exported functions location (static) and relocation information
  - Functions parameters
  - This lost may be on purpose: strip a binary for release
  - Compiling is a many-to-many operation
    - Same assembly code, different source code (or viceversa)

Binaries Analysis



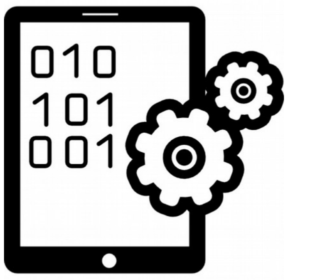
- Static analysis on executable code
  - Disassembly heuristics
  - Functions identification
  - Function parameters identification
  - Local and global variables identification
  - “basic blocks” identification (function flow)
  - Cross-references identification
  - All of this can be automated!

Binaries Analysis



- Disassembly heuristics
  - Linear Sweep
    - From a starting point (I.e. function symbol, .text section start or binary entry point) a linear disassembly is done
      - Instructions and operands of variable but known length (x86) or fixed length (ARM)
    - I.e. mov, add, push, etc.

Binaries Analysis



- Disassembly heuristics
  - Recursive Descent
    - Conditional branching (if, while, for, switch)
      - One branch is disassembled and the other one is marked for future disassemble
    - Unconditional branching (jmp, call)
      - Problem: is the jump target known?

Binaries Analysis

- Disassembly heuristics
  - Recursive Descent
    - Unconditional branching (jmp, call)
      - If we know it, disassemble the target. If not, we have a problem.
      - In a call we assume that a “return” to the next instruction exists. Thus, next address is marked as pending for future disassembly.

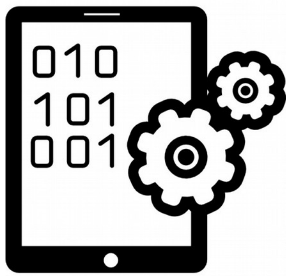
Binaries Analysis

Entry point (known) in binary stream

```
.text:0040101A 68 18 80 41 00
.text:0040101F 8B 45 FC
.text:00401022 50
.text:00401023 FF 15 00 10 41 00
.text:00401029 89 45 F8
.text:0040102C 83 7D F8 00
.text:00401030 74 17
.text:00401032 FF 55 F8
.text:00401035 89 45 F4
.text:00401038 8B 4D F4
.text:0040103B 51
.text:0040103C 68 20 80 41 00
.text:00401041 E8 4A 00 00 00
```

Variable (but known) length for opcodes and operands on this architecture

Binaries Analysis



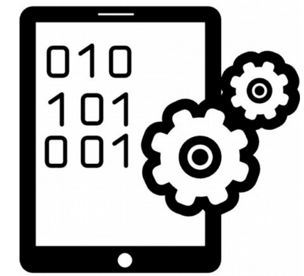

```
.text:0040101A 68 18 80 41 00       push    offset ProcName ;
.text:0040101F 8B 45 FC             mov     eax, [ebp+hModule]
.text:00401022 50                   push    eax             ;
.text:00401023 FF 15 00 10 41 00    call    ds:GetProcAddress
.text:00401029 89 45 F8             mov     [ebp+var_8], eax
.text:0040102C 83 7D F8 00          cmp     [ebp+var_8], 0
.text:00401030 74 17                jz      short loc_401049
.text:00401032 FF 55 F8             call    [ebp+var_8]
.text:00401035 89 45 F4             mov     [ebp+var_C], eax
.text:00401038 8B 4D F4             mov     ecx, [ebp+var_C]
.text:0040103B 51                   push    ecx
.text:0040103C 68 20 80 41 00       push    offset aReturnD ;
.text:00401041 E8 4A 00 00 00       call    sub_401090
```

Binaries Analysis

```
.text:0040101A 68 18 80 41 00       push    offset ProcName ;
.text:0040101F 8B 45 FC             mov     eax, [ebp+hModule]
.text:00401022 50                   push    eax             ;
.text:00401023 FF 15 00 10 41 00    call    ds:GetProcAddress
.text:00401029 89 45 F8             mov     [ebp+var_8], eax
.text:0040102C 83 7D F8 00          cmp     [ebp+var_8], 0
.text:00401030 74 17                jz      short loc_401049
.text:00401032 FF 55 F8             call    [ebp+var_8]
.text:00401035 89 45 F4             mov     [ebp+var_C], eax
.text:00401038 8B 4D F4             mov     ecx, [ebp+var_C]
.text:0040103B 51                   push    ecx
.text:0040103C 68 20 80 41 00       push    offset aReturnD ;
.text:00401041 E8 4A 00 00 00       call    sub_401090
```

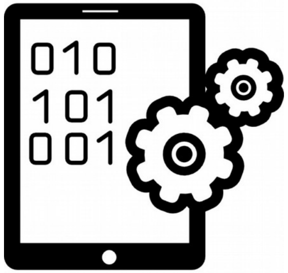
Where to continue disassembling? CALL to an address held in a local variable, only known in run time

Binaries Analysis



- In CISC architectures like x86/x86_64 (with extended instructions sets), many opcodes may be valid.
- However, not every instruction is equally likely or frequent. Executable binary type may provide hints: are we expecting floating point instructions?
- Can we differentiate an executable binary manually written in assembly from one generated by a compiler? Can we identify idioms or patterns?

Binaries Analysis



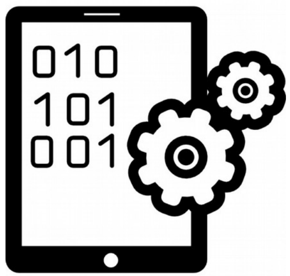
- Compilers tend to use certain instructions more frequently and generate specific patterns following conventions or binary interfaces (ABIs).
- It’s important to be able to make a judgment about the correctness of a disassembly
  - And provide a hint to the disassembler regarding where to start.

Binaries Analysis

- Where should we start disassembling?

```
.text:004012FF                db     0
.text:00401300                db   89h
.text:00401301                db   15h
.text:00401302                db  0D4h
.text:00401303                db   87h
.text:00401304                db   41h
.text:00401305                db     0
.text:00401306                db  0E8h
.text:00401307                db     5
.text:00401308                db  0FFh
.text:00401309                db  0FFh
.text:0040130A                db  0FFh
.text:0040130B                db   83h
.text:0040130C                db  0F8h
.text:0040130D                db  0FFh
.text:0040130E                db   75h
.text:0040130F                db     5
```

Binaries Analysis



- Does it look correct?

```
.text:004012FE                 db  41h ; A
.text:004012FF                 db    0
.text:00401300                 db  89h ; ë
.text:00401301                 db  15h
.text:00401302 ; ---------------------------------------------------------------
.text:00401302                 aam     87h
.text:00401304                 inc     ecx
.text:00401305                 add     al, ch
.text:00401307                 add     eax, 83FFFFFFh
.text:0040130C                 clc
.text:0040130D                 push    dword ptr [ebp+5]
.text:00401310                 or      eax, 0FFFFFFFFh
.text:00401313                 jmp     short loc_401370
.text:00401313 ; ---------------------------------------------------------------
.text:00401315                 db  6Ah ; j
.text:00401316                 db    0
.text:00401317                 db  6Ah ; j
```

Binaries Analysis

- Does it look correct? no

```
.text:004012FE                 db  41h ; A
.text:004012FF                 db    0
.text:00401300                 db  89h ; ë
.text:00401301                 db  15h
.text:00401302 ; ---------------------------------------------------------------------------
.text:00401302                 aam     87h
.text:00401304                 inc     ecx
.text:00401305                 add     al, ch
.text:00401307                 add     eax, 83FFFFFFh
.text:0040130C                 clc
.text:0040130D                 push    dword ptr [ebp+5]
.text:00401310                 or      eax, 0FFFFFFFFh
.text:00401313                 jmp     short loc_401370
.text:00401313 ; ---------------------------------------------------------------------------
.text:00401315                 db  6Ah ; j
.text:00401316                 db    0
.text:00401317                 db  6Ah ; j
```

Rare instruction: ASCII Adjust AX After Multiply

Compilers sometimes do “silly” things but not this much

Binaries Analysis

- Does it look correct?

```
.text:00401300 ; ---------------------------------------------------------------
.text:00401300                 mov     dword_4187D4, edx
.text:00401306                 call    sub_401210
.text:0040130B                 cmp     eax, 0FFFFFFFFh
.text:0040130E                 jnz     short loc_401315
.text:00401310                 or      eax, 0FFFFFFFFh
.text:00401313                 jmp     short loc_401370
.text:00401315 ; ---------------------------------------------------------------
```

Binaries Analysis

- Does it look correct?: yes

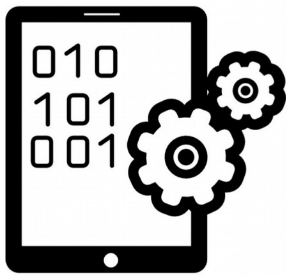
1^{st} parameter for a call (x86_64 ABI)

Call to a verifiable function

Compares function return value against -1

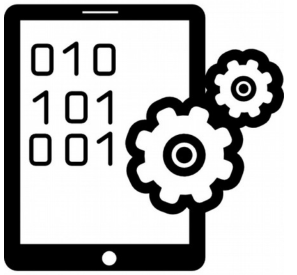
```
.text:00401300 ; ---------------------------------------------------------------
.text:00401300                 mov     dword_4187D4, edx
.text:00401306                 call    sub_401210
.text:0040130B                 cmp     eax, 0FFFFFFFFh
.text:0040130E                 jnz     short loc_401315
.text:00401310                 or      eax, 0FFFFFFFFh
.text:00401313                 jmp     short loc_401370
.text:00401315 ; ---------------------------------------------------------------
```

Binaries Analysis



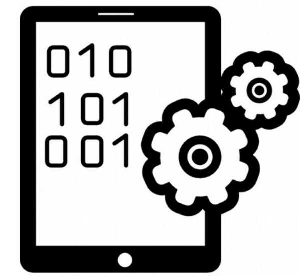
- In previous examples we assume that binary is not obfuscated / packed, and that is genuine compiler assembly
  - Use case example: DLLs or SYS modules diffing from security patches
  - When analyzing malware, these assumptions may not be true
- Part of this is “training”

Binaries Analysis



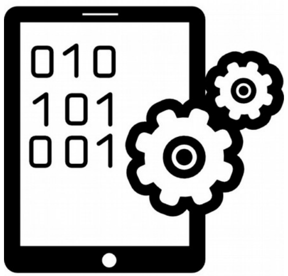
- Functions identification
  - Exported functions
  - CALL instructions targets
  - Epilogues (ABIs)
- Functions parameters identification
  - Calling conventions (I.e. x86 ABI) to determine parameters count
  - “mov” instructions for size

Binaries Analysis



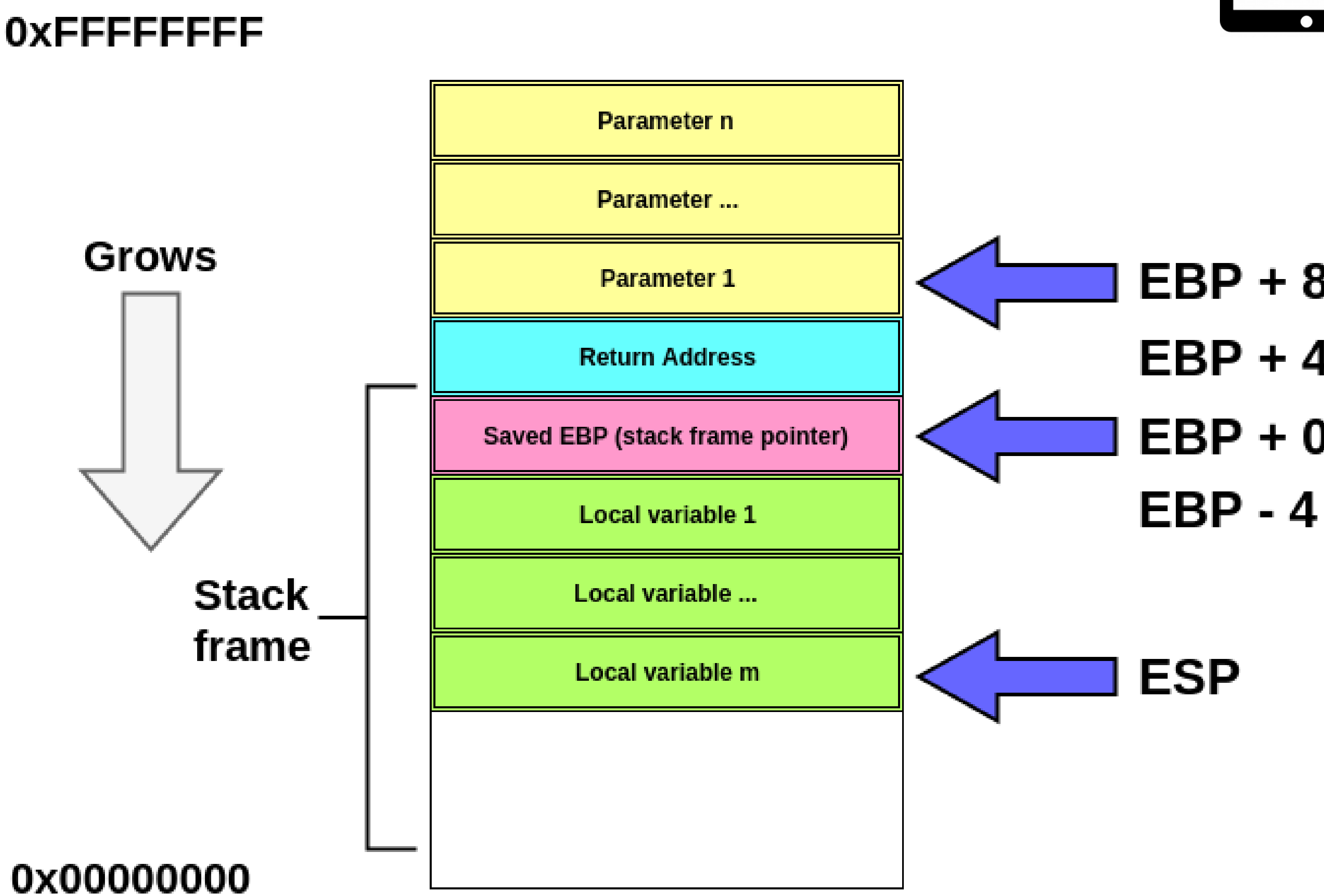
- Functions parameters identification
  - Is up to the reverser to determine:
    - Pointers meaning
    - Structures
      - When are their members written or read? That provides semantic value.
    - Data types
      – I.e. are floating point operations applied on a parameter?

Binaries Analysis



- Calling conventions – Application Binary Interface (ABI)
- How is a function called at the assembly level?
  - Send parameters (values, alignment, structures)
  - Return address
  - Return value
  - Stack balance
  - Which registers are saved? Who is responsible for that?
- A convention is needed: code generated by one compiler may call libraries generated by a different compiler.
- These conventions depend on the CPU architecture and the platform (Windows, Unix, etc.)

Binaries Analysis



x86

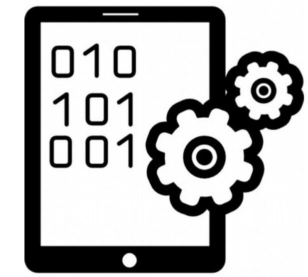
Stack
1 stack in user-space per main thread

Binaries Analysis

```
sub_401060 proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h
arg_C= dword ptr  14h

push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_C]
push    eax
mov     ecx, [ebp+arg_8]
push    ecx
mov     edx, [ebp+arg_4]
push    edx
mov     eax, [ebp+arg_0]
push    eax
```

x86

Binaries Analysis



- Calling conventions x86
  - Cdecl
    - Caller function balances the stack (parameters cleanup)
  - Stdcall
    - Callee function balances the stack (parameters cleanup)
    - Common in Windows API
  - Fastcall
    - Parameters by registers

Binaries Analysis

```c
int __stdcall function_a(int p1) { return ++p1; }

int __cdecl function_b(int p1) { return ++p1; }

int __fastcall function_c(int p1) { return ++p1; }

void main(void) {
    printf("function_a: %d\n", function_a(0));
    printf("function_b: %d\n", function_b(1));
    printf("function_c: %d\n", function_c(2));
}
```

MSVC calling conventions

Binaries Analysis

int __stdcall function_a(int p1) { return ++p1; }

```
push    0
call    sub_401000
push    eax
push    offset aFunction_aD
call    sub_4010F0
```

Parameter 1 pushed to the stack

function_a

Stack is not balanced, callee did it

printf

main function

```
mov     eax, [ebp+arg_0]
pop     ebp
retn    4
```

Callee balances the stack, freeing up space used for the parameter

function_a function

Binaries Analysis

int __cdecl function_b(int p1) { return ++p1; }

```
push    1
call    sub_401020
add     esp, 4
push    eax
push    offset aFunction_bD ;
call    sub_4010F0
```

Parameter 1 pushed to the stack

function_b

Stack is balanced, freeing up space used for the parameter

printf

main function

```
mov     eax, [ebp+arg_0]
pop     ebp
retn
```

Callee does not balance the stack

function_b function

Binaries Analysis

int __fastcall function_c(int p1) { return ++p1; }

```
mov     ecx, 2
call    sub_401040
push    eax
push    offset aFunction_cD
call    sub_4010F0
```

- mov ecx, 2 → Parameter 1 loaded in a register
- call sub_401040 → function_c
- push eax → Stack remains balanced
- call sub_4010F0 → printf

main function

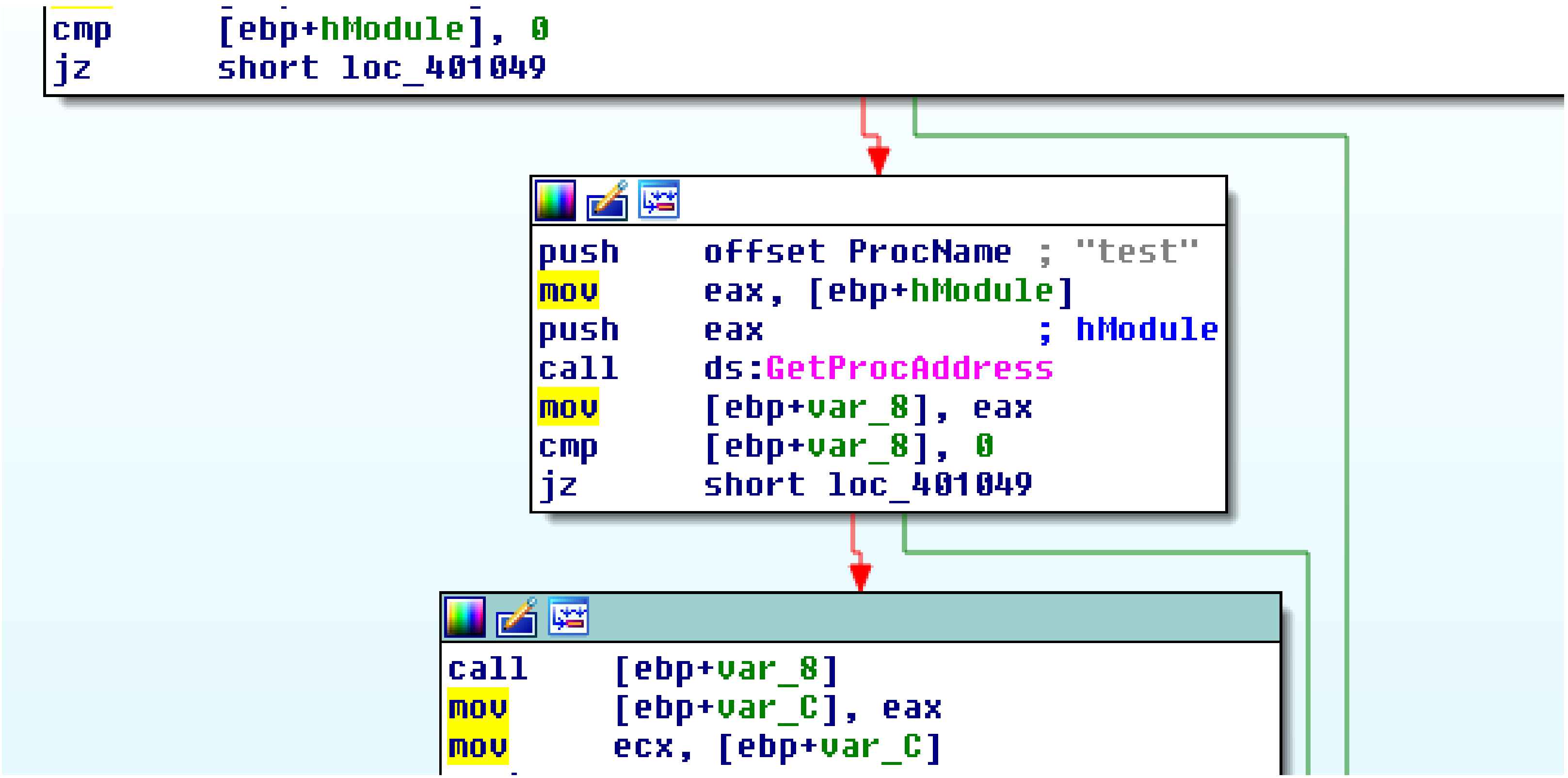
Binaries Analysis

- Variables identification
  - Similar to parameters identification
  - Local variables are referenced (in x86) by EBP - offset
    - Compiler can reference them with ESP
    - Can be held in registers, depending on optimization levels
  - Global variables are references to .data (initialized) and .bss (uninitialized) segments

Binaries Analysis

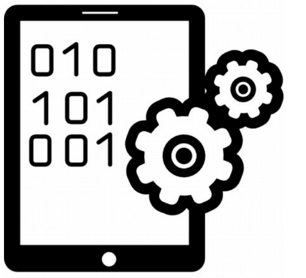
```
sub_4026F4 proc near

var_C= dword ptr -0Ch
var_8= dword ptr -8
var_1= byte ptr -1
arg_0= dword ptr  8
arg_4= dword ptr  0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 0Ch
mov     eax, [ebp+arg_0]
lea     ecx, [ebp+var_1]
mov     [ebp+var_8], eax
mov     [ebp+var_C], eax
```

Binaries Analysis



Basic blocks

Binaries Analysis

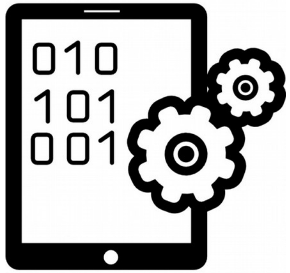


- Cross references identification
  - Based on offsets
    - + symbols information
    - + value (I.e. String)
  - Bidirectional search
  - Good strategy to understand what a function does

Binaries Analysis

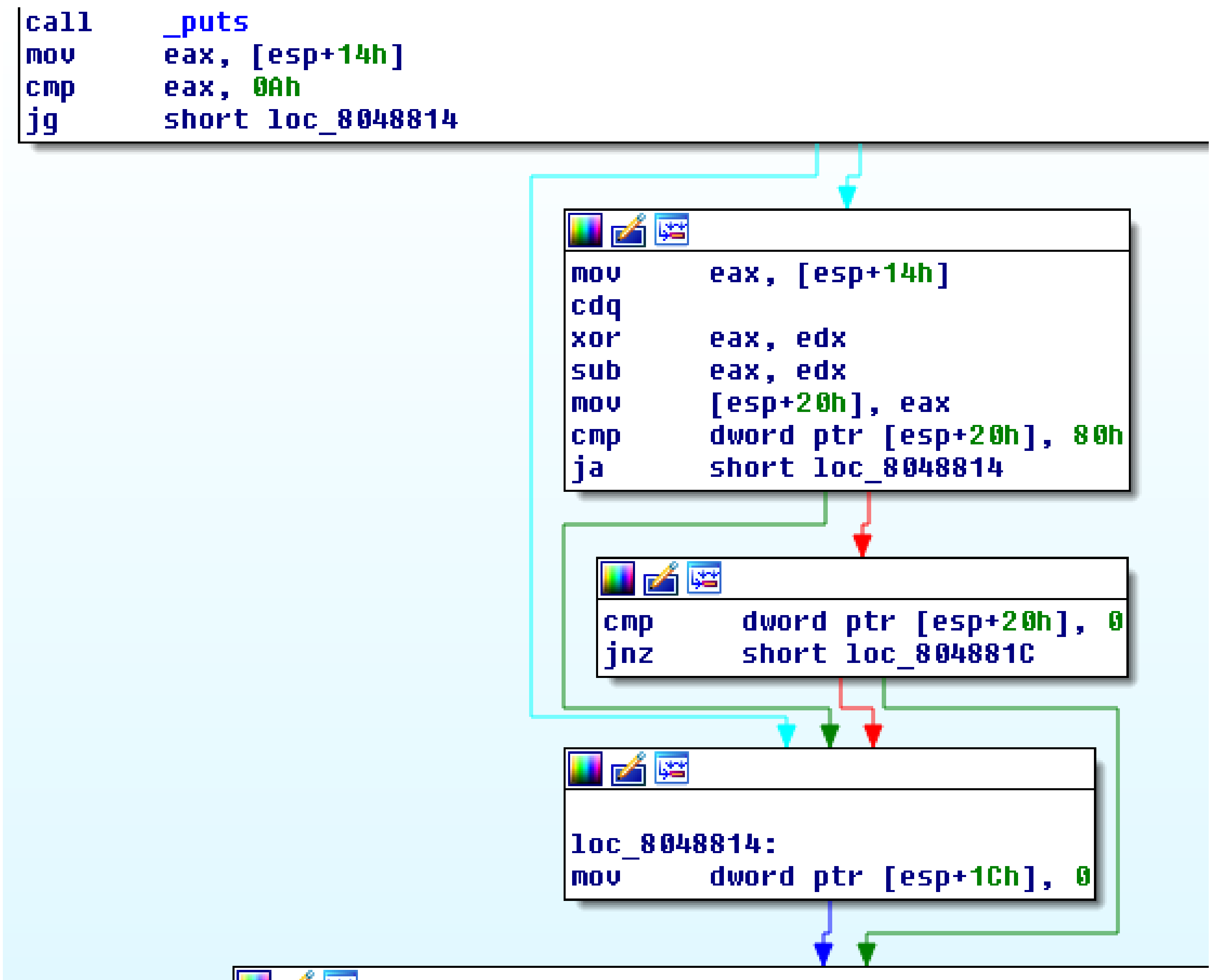
```
push    ebp
mov     ebp, esp
sub     esp, 0Ch
push    offset LibFileName ; "test.dll"
call    ds:LoadLibraryA
```

Binaries Analysis

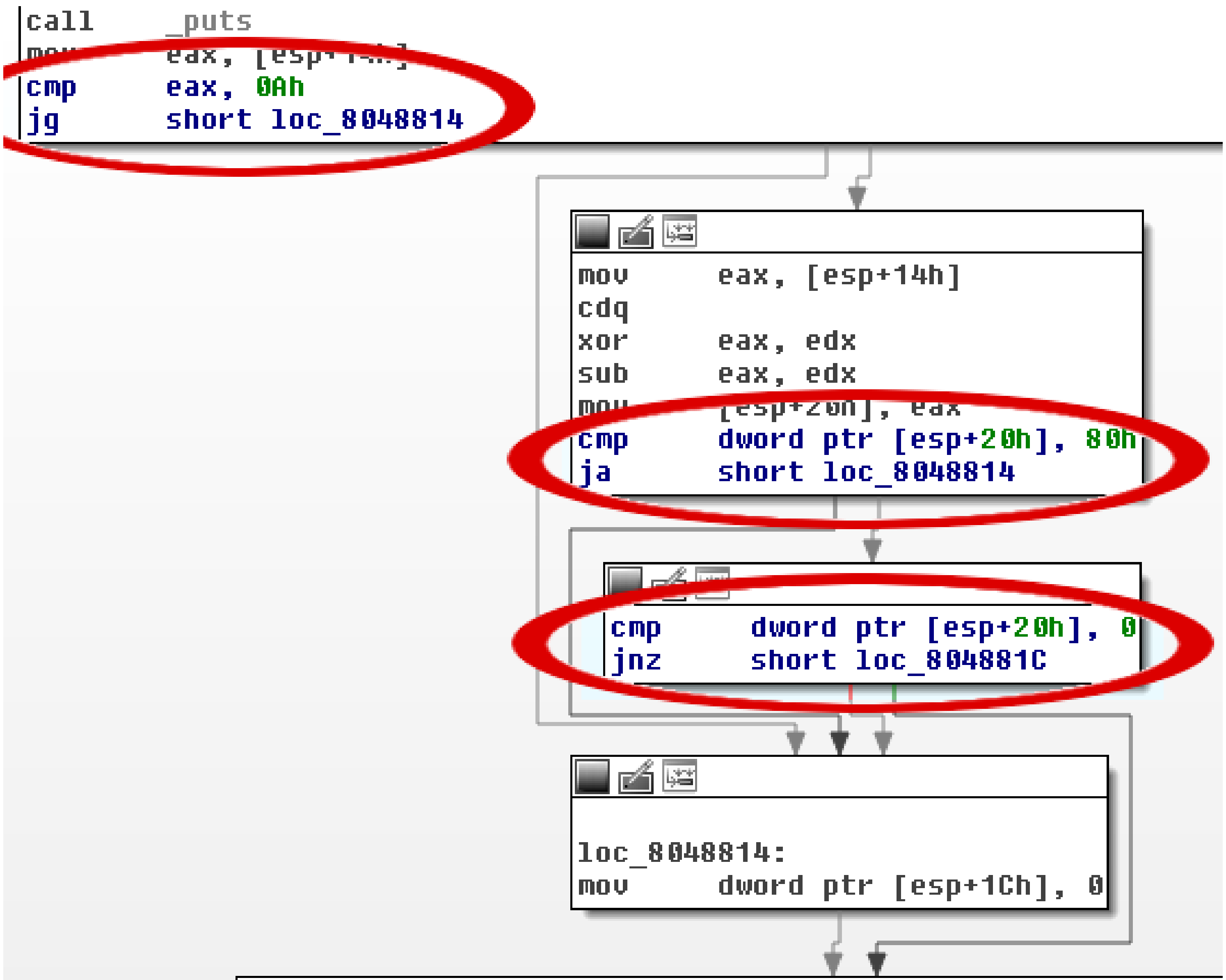


- Patterns identification
  - From assembly to source code
    - A disassembler parses opcodes and shows the instructions mnemonic.
    - A decompiler makes high level abstractions to show C code or pseudo-code.

Binaries Analysis



Binaries Analysis

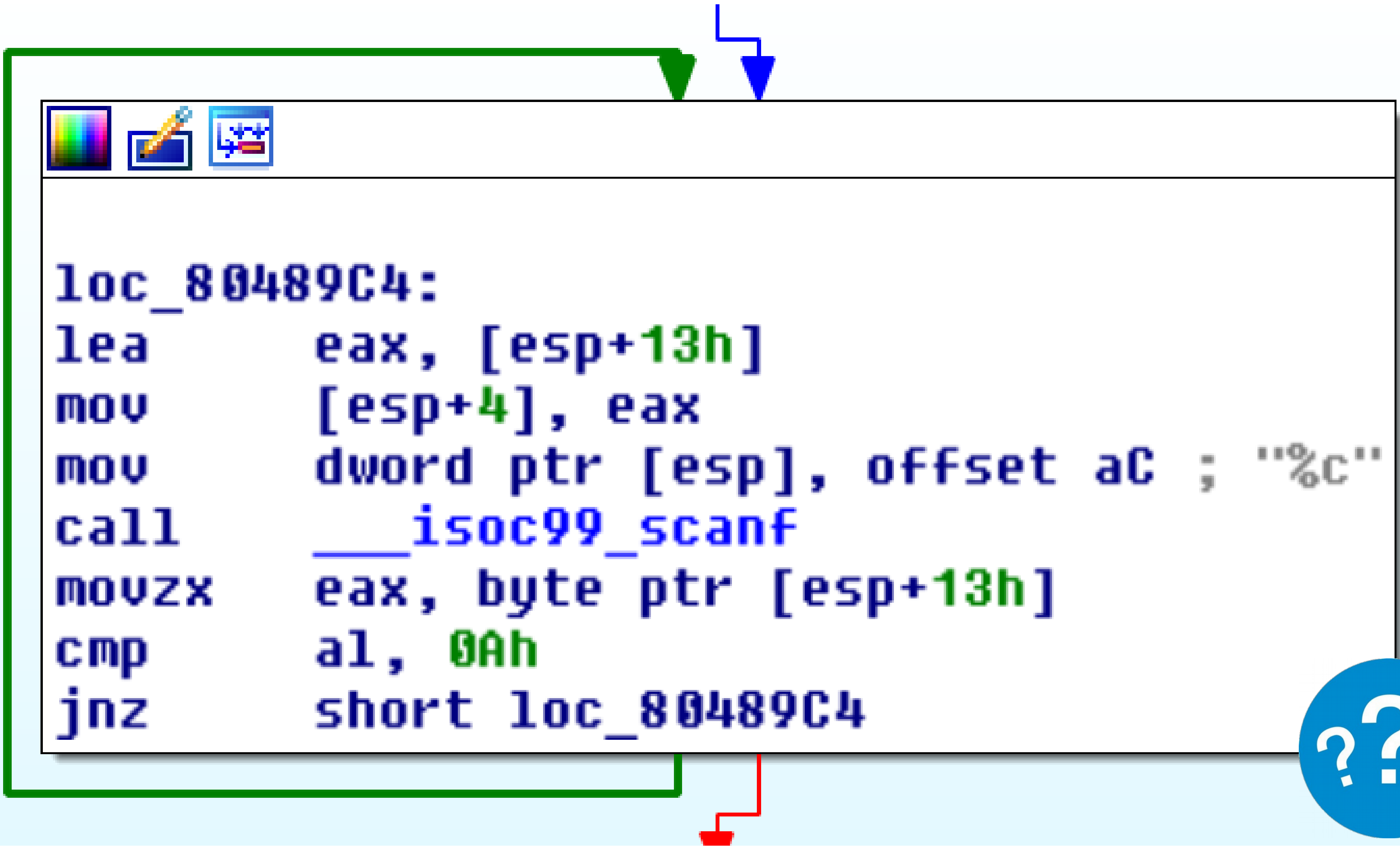


Binaries Analysis

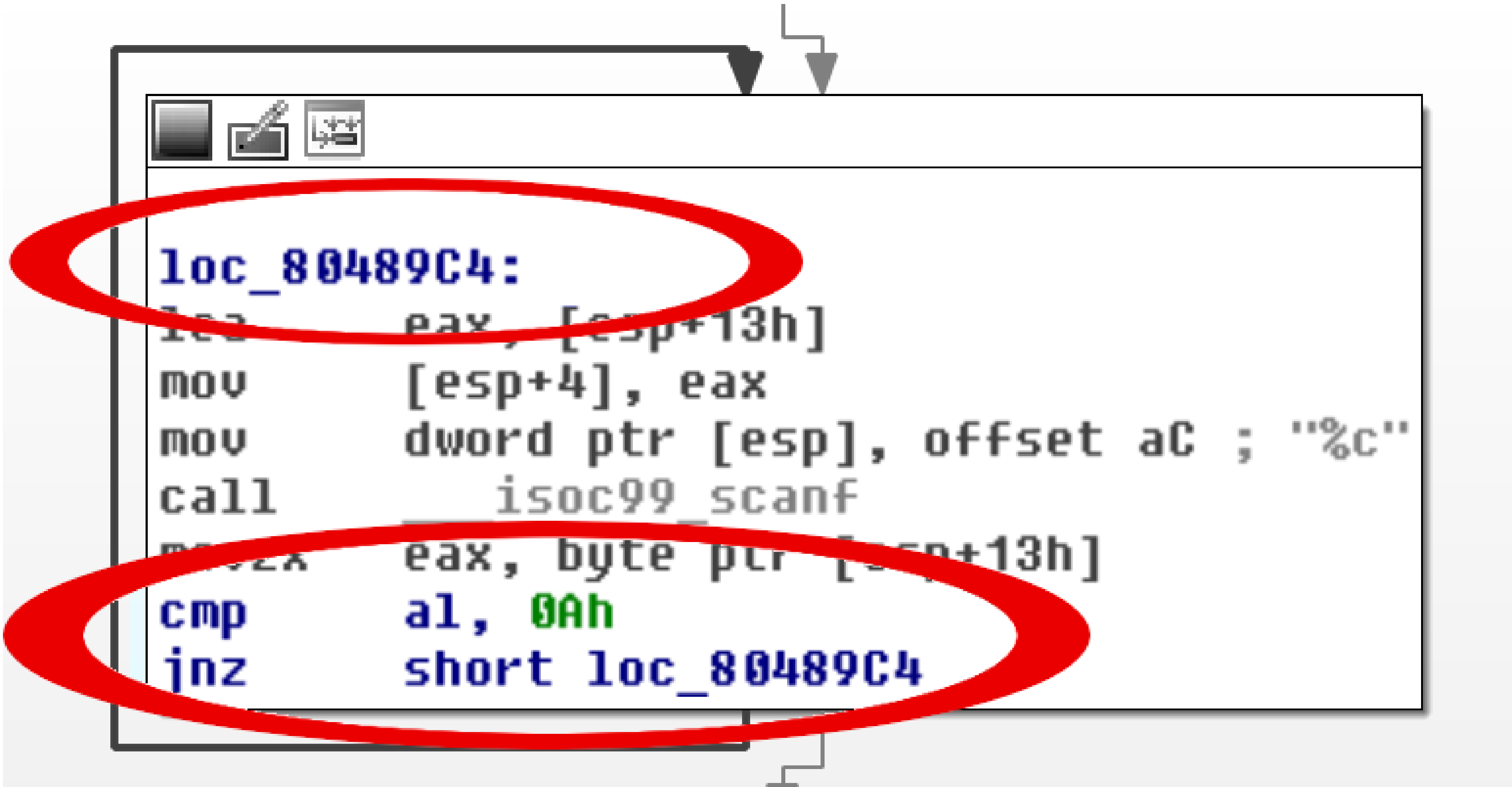
- Patterns identification

```
if ( condition_1 && condition_2 … &&
condition_n) {
    do;
}
```

Binaries Analysis



Binaries Analysis

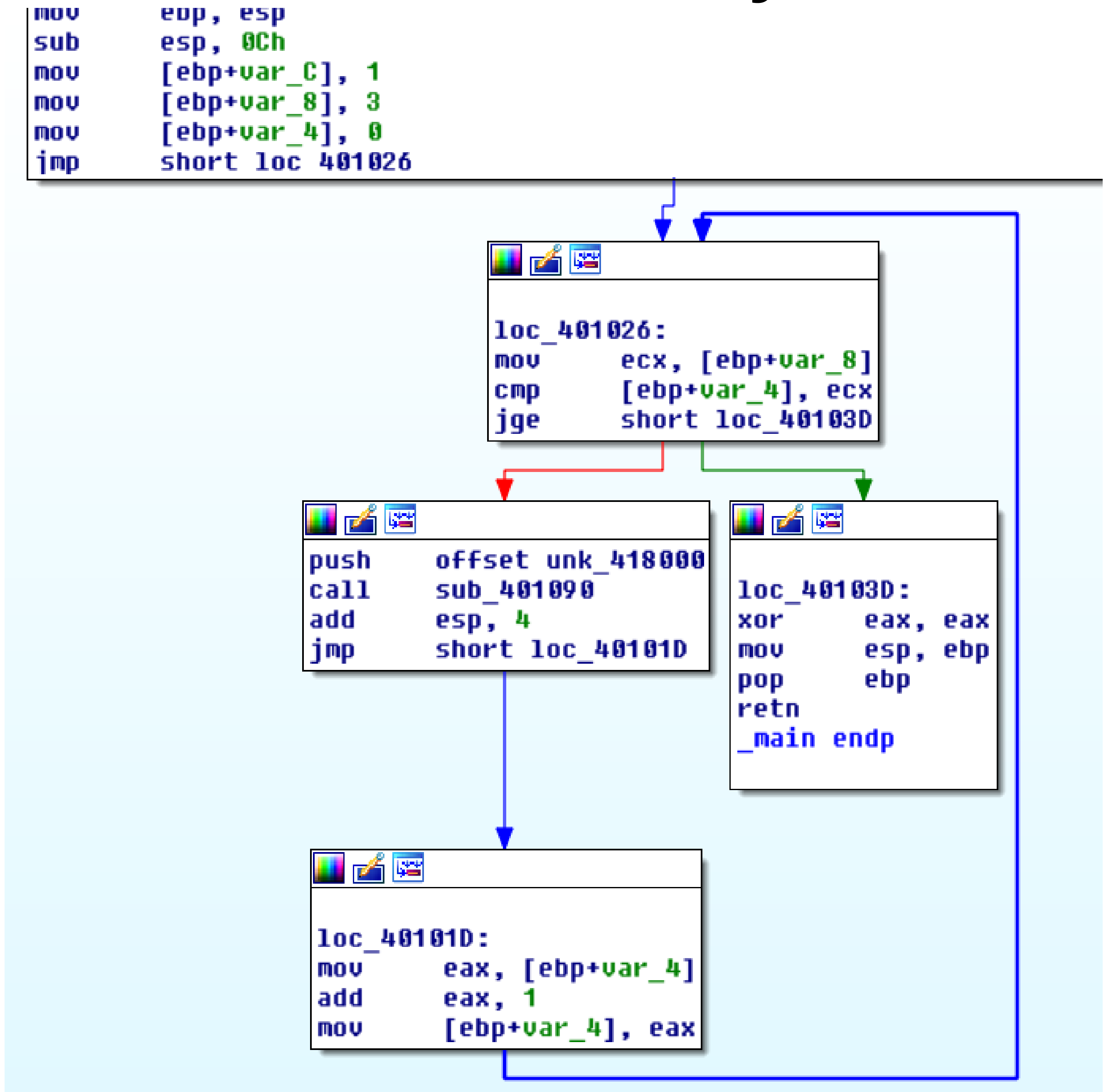


Binaries Analysis

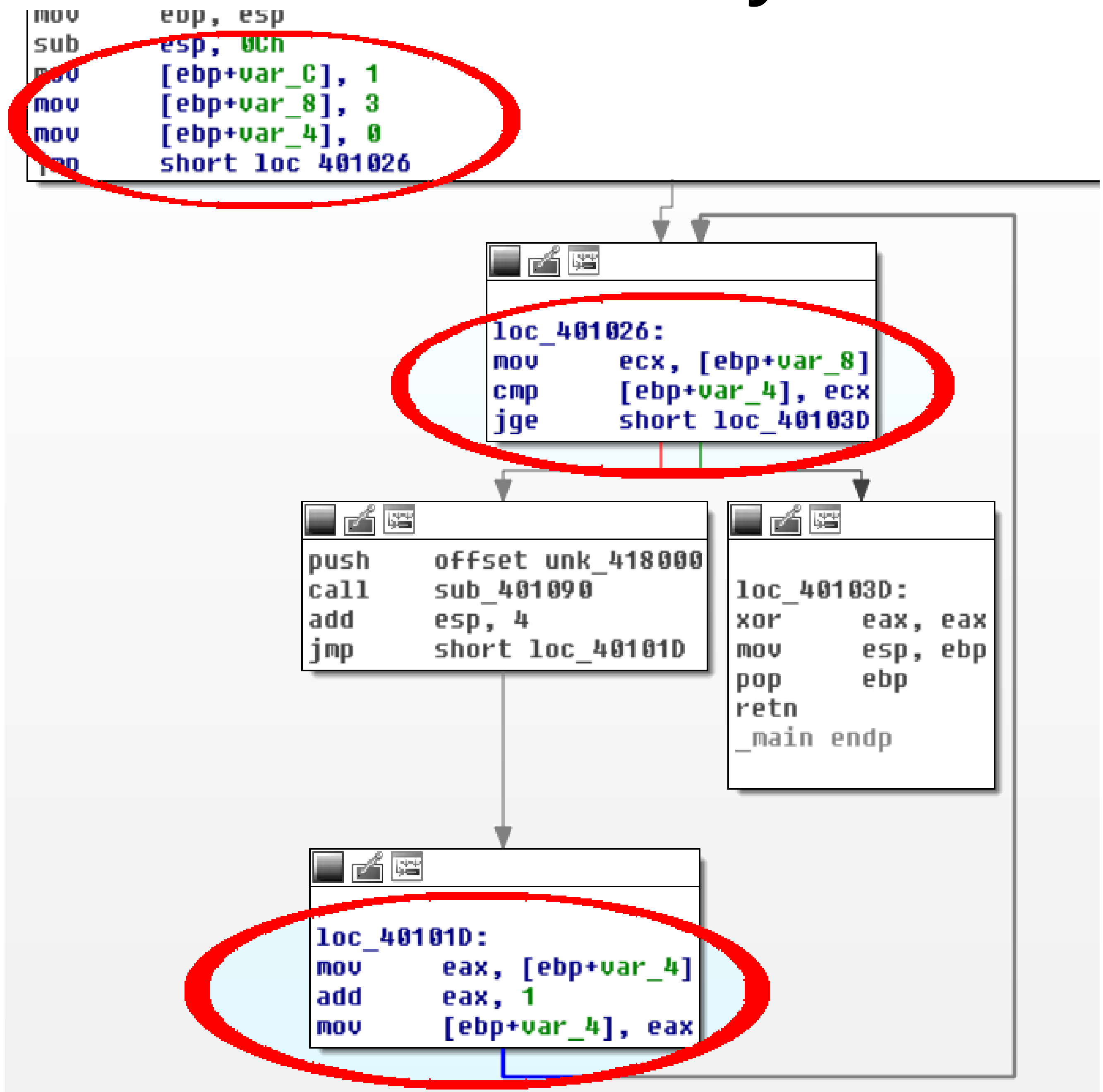
- Patterns identification

```
while ( condition_1 ) {
    do;
}
```

Binaries Analysis



Binaries Analysis

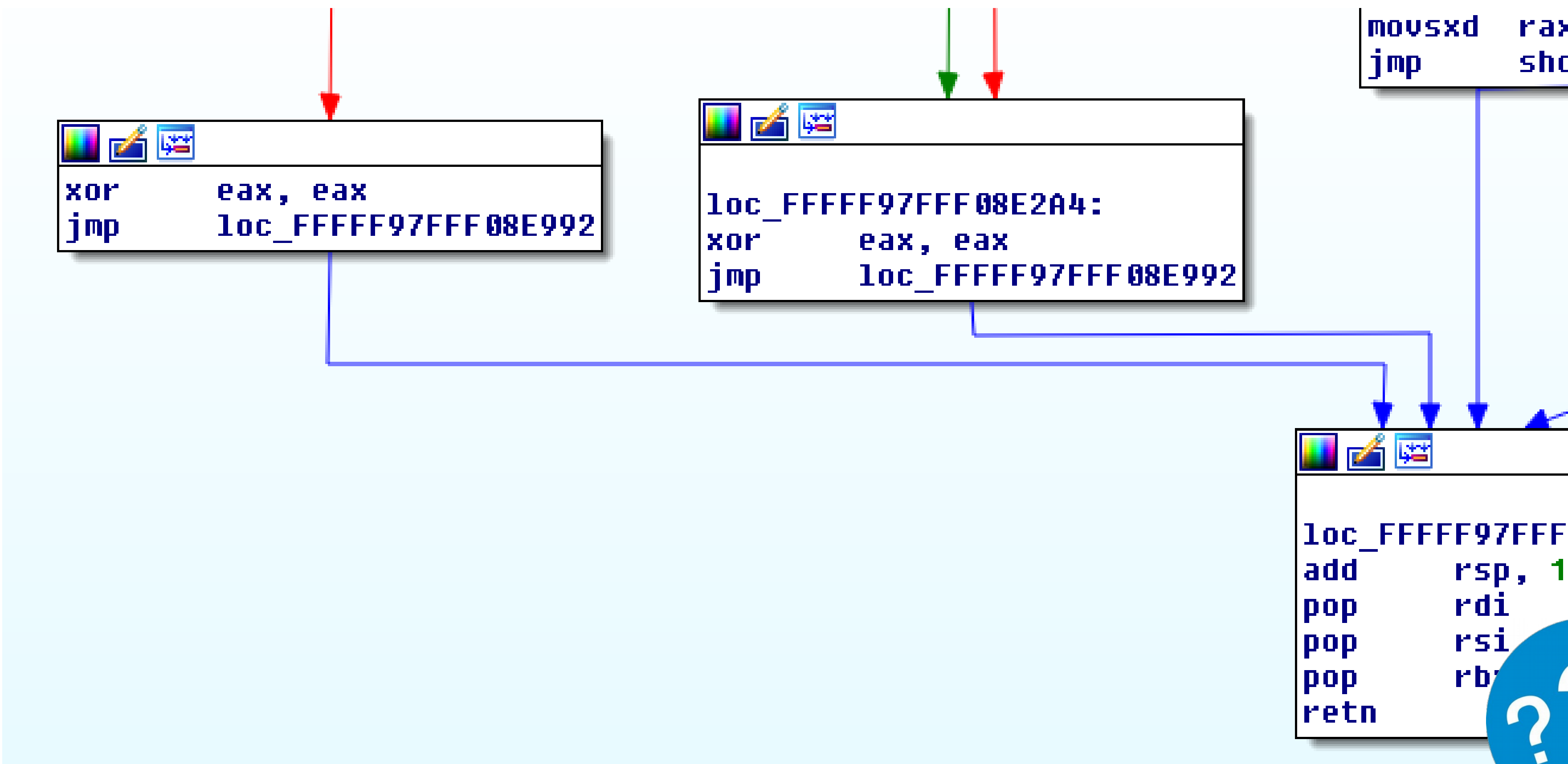


Binaries Analysis

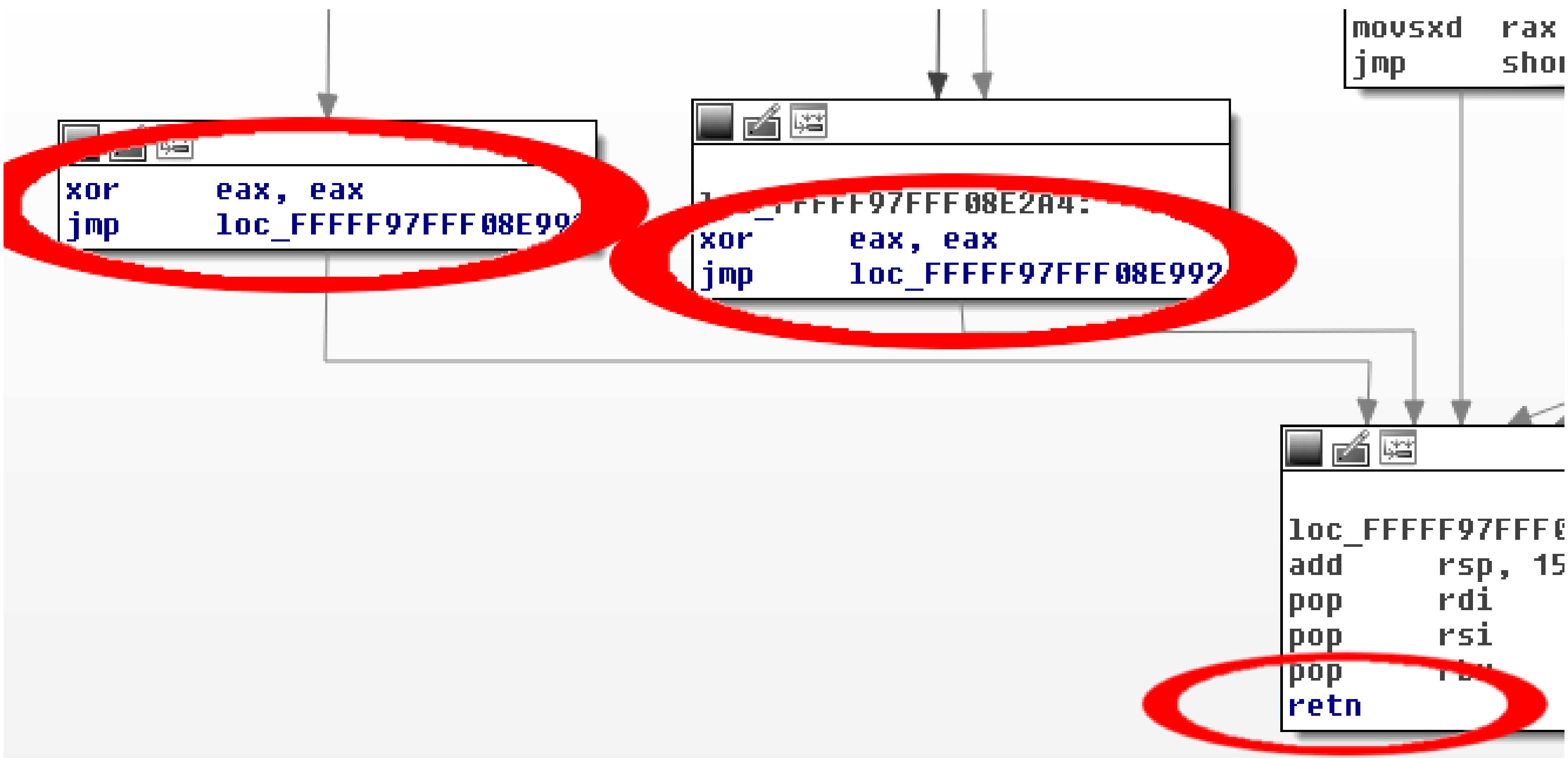
- Patterns identification

```
int max = 3;
for ( int i = 0; i < max; i++ ) {
    ...
}
```

Binaries Analysis



Binaries Analysis

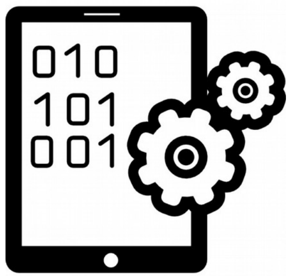


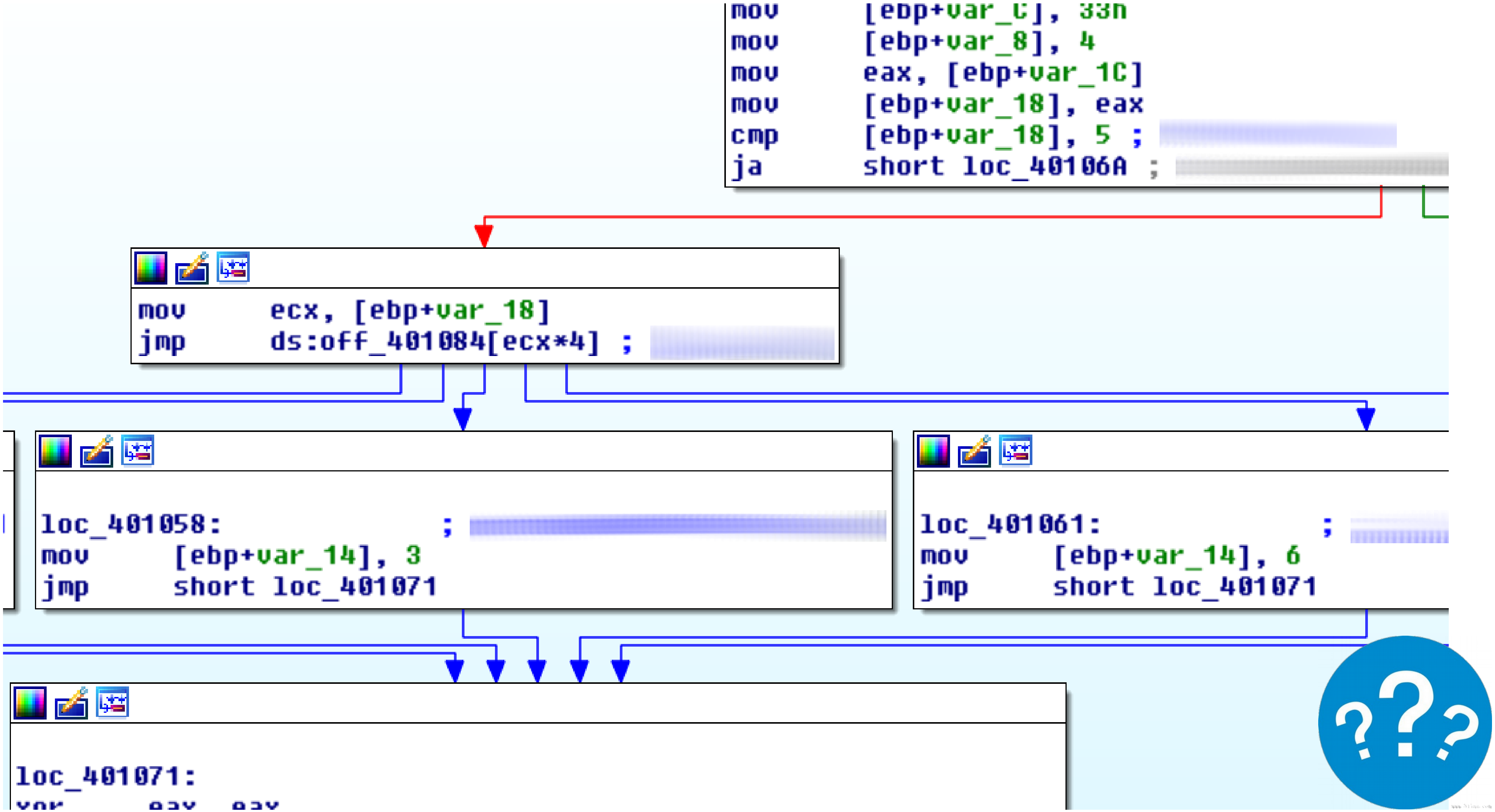
Binaries Analysis

- Patterns identification

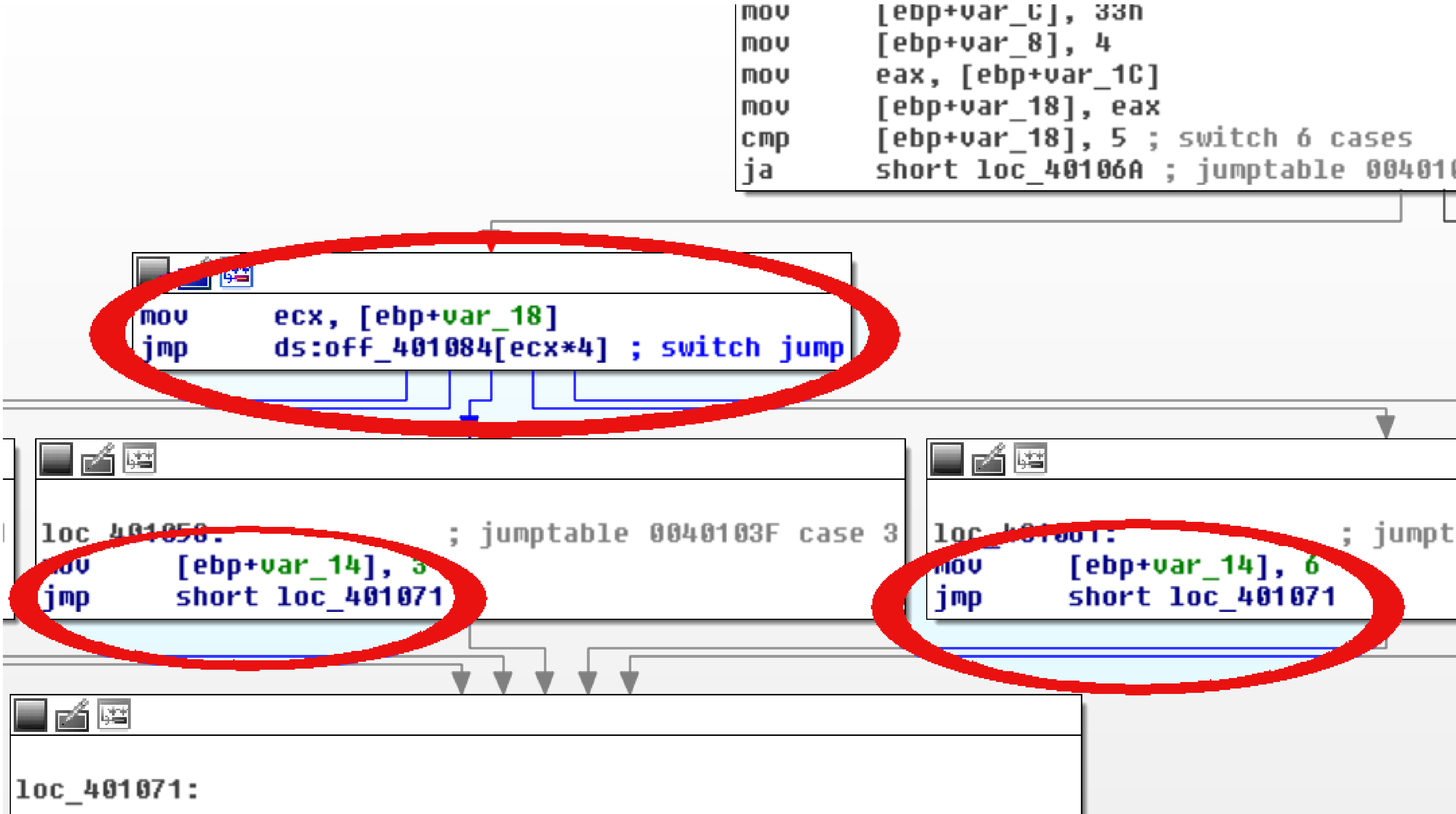
```
if ( condition_1 ) {
    goto error;
}
if ( condition_2 ) {
    goto error;
}
error:
    return 0;
```

Binaries Analysis





Binaries Analysis



Binaries Analysis

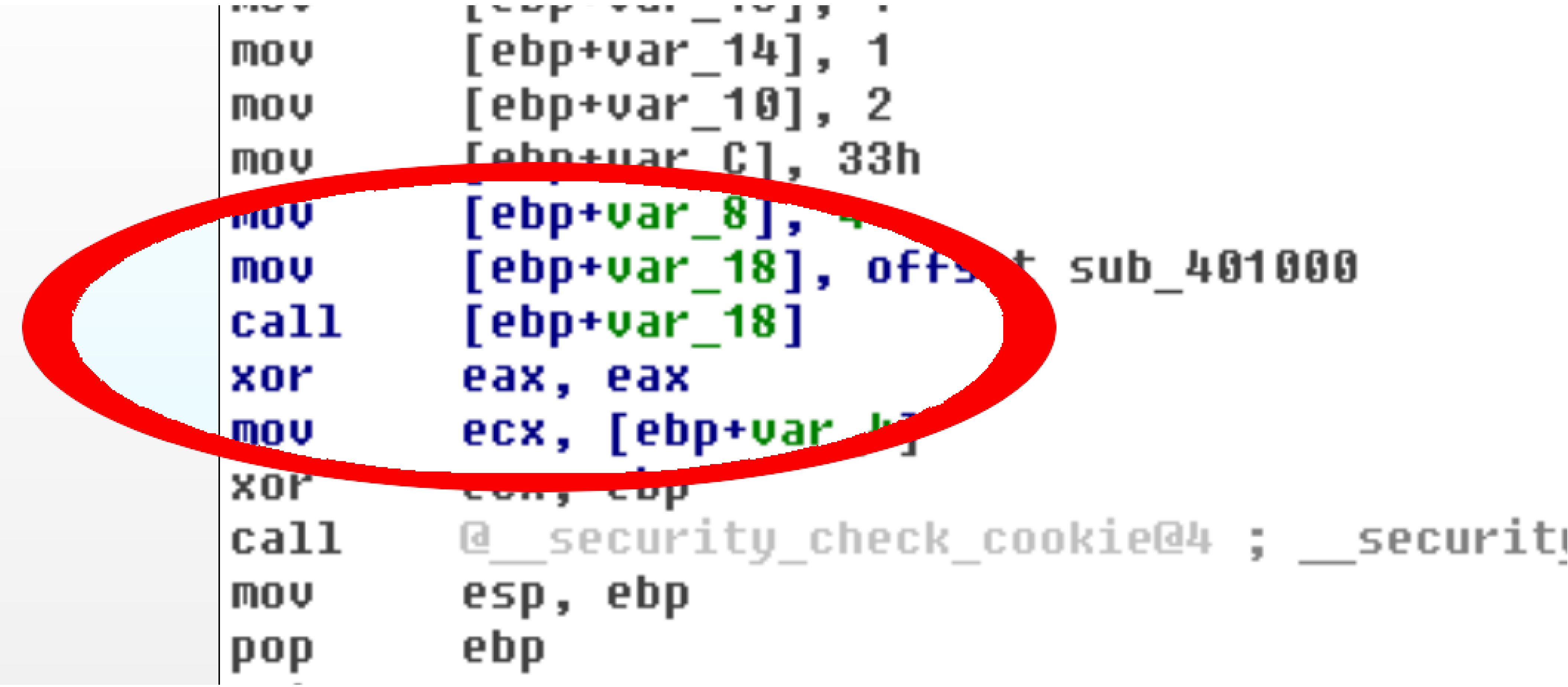
- Patterns identification

```
switch ( variable ) {
    case 0:

        ...

        break;

    case 1:

        ...

        break;

}
```

Binaries Analysis

```
mov     [ebp+var_14], 1
mov     [ebp+var_10], 2
mov     [ebp+var_C], 33h
mov     [ebp+var_8], 4
mov     [ebp+var_18], offset sub_401000
call    [ebp+var_18]
xor     eax, eax
mov     ecx, [ebp+var_4]
xor     ecx, ebp
call    @__security_check_cookie@4 ; __security
mov     esp, ebp
pop     ebp
```

Binaries Analysis

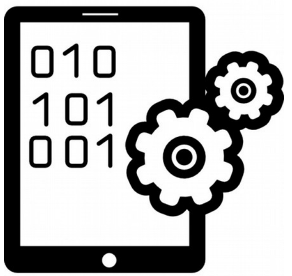


Binaries Analysis

- Patterns identification

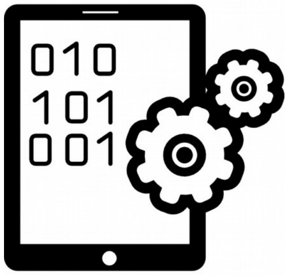
```
int ( * f_ptr ) ( ) = f;

( * f_ptr ) ( );
```

Binaries Analysis



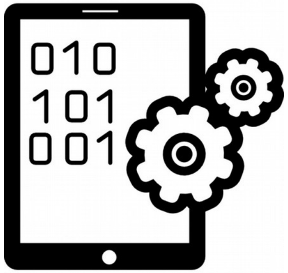
- Dynamic analysis on executable code
  - IDA Pro (debugger)
  - Other debuggers
    - Windbg, gdb, Ollydbg, etc.
  - strace (Linux)
  - API monitor (Windows)
  - Wireshark

Binaries Analysis



- Dynamic analysis on executable code
  - Tools to monitor registry changes (Windows)
  - Tools to monitor filesystem changes
  - Integrated suite: Cuckoo

Binaries Analysis



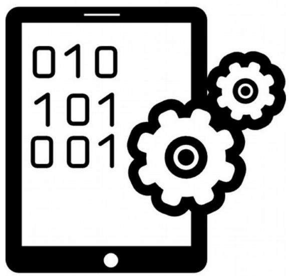
- Execution traces
  - Do not stop execution (in opposition to braekpoints) and record:
    - Instructions execution
    - Memory reads or writes
      - From which instruction was memory accessed
    - Other state changes (i.e. registers)
    - Thread that executed
    - Other information (I.e. call-graph)
  - May generate too much information. Filtering is required.

Binaries Analysis

- Trace example

00000F20				ST0=FFFFFFFFFFFFFFFF ST1=FFFFFFFFFFFFFFFF ST2=FFFFFFF
00000F20	.text:sub_2F13C0+3	sub	esp, 14h	ESP=0042FA34 PF=0
00000F20	.text:sub_2F13C0+6	push	ebx	ESP=0042FA30
00000F20	.text:sub_2F13C0+7	cpuid		EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000
00000F20	.text:sub_2F13C0+9	rdtsc		EAX=DDA53517 EDX=000002FE
00000F20	.text:sub_2F13C0+B	mov	[ebp+var_C], eax	
00000F20	.text:sub_2F13C0+E	mov	[ebp+var_8], edx	
00000F20	.text:sub_2F13C0+11	mov	[ebp+var_4], 0	
00000F20	.text:sub_2F13C0+18	jmp	short loc_2F13E3	
00000F20	.text:sub_2F13C0:loc_2F13E3	cmp	[ebp+var_4], 8	CF=1 AF=1 SF=1
00000F20	.text:sub_2F13C0+27	jnb	short loc_2F13FE	
00000F20	.text:sub_2F13C0+29	mov	ecx, 8	ECX=00000008
00000F20	.text:sub_2F13C0+2E	sub	ecx, [ebp+var_4]	CF=0 AF=0 SF=0
00000F20	.text:sub_2F13C0+31	mov	edx, [ebp+var_4]	EDX=00000000
00000F20	.text:sub_2F13C0+34	mov	al, [ebp+ecx+var_D]	EAX=DDA53500
00000F20	.text:sub_2F13C0+38	mov	byte ptr [ebp+edx+var_14], al	
00000F20	.text:sub_2F13C0+3C	jmp	short loc_2F13DA	
00000F20	.text:sub_2F13C0:loc_2F13DA	mov	eax, [ebp+var_4]	EAX=00000000
00000F20	.text:sub_2F13C0+1D	add	eax, 1	EAX=00000001
00000F20	.text:sub_2F13C0+20	mov	[ebp+var_4], eax	
00000F20	.text:sub_2F13C0:loc_2F13E3	cmp	[ebp+var_4], 8	CF=1 PF=1 AF=1 SF=1
00000F20	.text:sub_2F13C0+27	jnb	short loc_2F13FE	

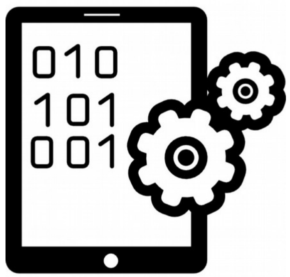
Binaries Analysis



- Trace example (filtering by 0x42FA48)

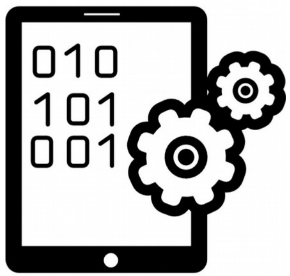
00000F20	.text:sub_2F13C0+45	mov esp, ebp	ESP=0042FA48
00000F20	.text:_main+16	call sub_2F1210	ESP=0042FA48
00000F20	.text:sub_2F1210+1	mov ebp, esp	debug021:0042FA48: 58
00000F20	.text:sub_2F1210+1	mov ebp, esp	EBP=0042FA48
00000F20	.text:sub_2F1000+5E	pop ebp	EBP=0042FA48 ESP=0042FA24
00000F20	.text:sub_2F1000+5E	pop ebp	EBP=0042FA48 ESP=0042FA24
00000F20	.text:sub_2F1000+5E	pop ebp	EBP=0042FA48 ESP=0042FA24
00000F20	.text:sub_2F1210:loc_2F12E2	mov esp, ebp	ESP=0042FA48
00000F20	.text:_main+27	push 0 ; bInitialOwner	ESP=0042FA48
00000F20	KERNELBASE:kernelbase_CreateMutexA+A	jz short near ptr unk_768717C8	debug021:0042FA48: 00
00000F20	.text:_main+38	push 0 ; dwCreationFlags	ESP=0042FA48
00000F20	kernel32:kernel32_CreateThread+D	push dword ptr [ebp+14h]	debug021:0042FA48: 00
00000F20	.text:_main+50	push 0 ; dwCreationFlags	ESP=0042FA48
00000F20	kernel32:kernel32_CreateThread+D	push dword ptr [ebp+14h]	debug021:0042FA48: 00
00000F20	.text:_main+6B	push eax ; hHandle	ESP=0042FA48
00000F20	kernel32:kernel32_WaitForSingleObject+D	call near ptr kernel32_WaitForSingleObjectEx	debug021:0042FA48: 34

Binaries Analysis



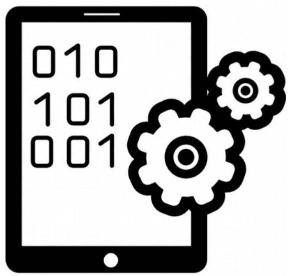
- Which is the proper strategy to analyze an...
  - “stripped” binary? (no symbols)
  - obfuscated or packed binary?
- Code-coverage in dynamic analysis:
  - how can we trigger every possible execution flow?

Binaries Analysis



- Answer is on case-by-case basis and will probably involve a combination of different techniques
  - Static analysis may require a high effort: too much information to analyze!
  - Dynamic analysis based on debugging may require a high effort too
  - Dynamic analysis based on monitoring tools may not be enough

Question



Which approach would you use to analyze a binary that encrypts communications with a custom cryptographic algorithm?

Lab 3.1

Analyze the binary, describe the logic and extract communicated data

References

- https://github.com/cuckoosandbox/cuckoo
- The IDA Pro Book