Reverse Engineering: Towards Malware Analysis

Lecture – Understanding Code Constructs

Computer Security Practice

Outline

- What are Constructs?
- Why do we care?
- Global vs Local
- IF Statements
- Loops
- etc

Advice

- Analyze instructions as groups to obtain a high-level picture
- “The best reverse engineers do not evaluate each instruction individually”
- This skill takes time to develop and requires practice
- Malware code is most commonly written in C
- Learn to program in C and C++

What are code constructs?

- Code abstraction level defines a functional property
- Programs are broken down into individual constructs
- Types
  - `if` statement
  - `for` loops
  - linked lists
  - `switch` statements
  - etc

Global Vs. Local Variables

Global

- Defined outside of a function
- Can be accessed and used by any function in a program
- Referenced by memory location

```
int x = 9;
main() {
  do_stuff(x);
}
```

```
mov     dword 409ABC, eax
```

Local

- Defined inside of a function
- Accessible only to the function in which it was defined
- Referenced by stack location

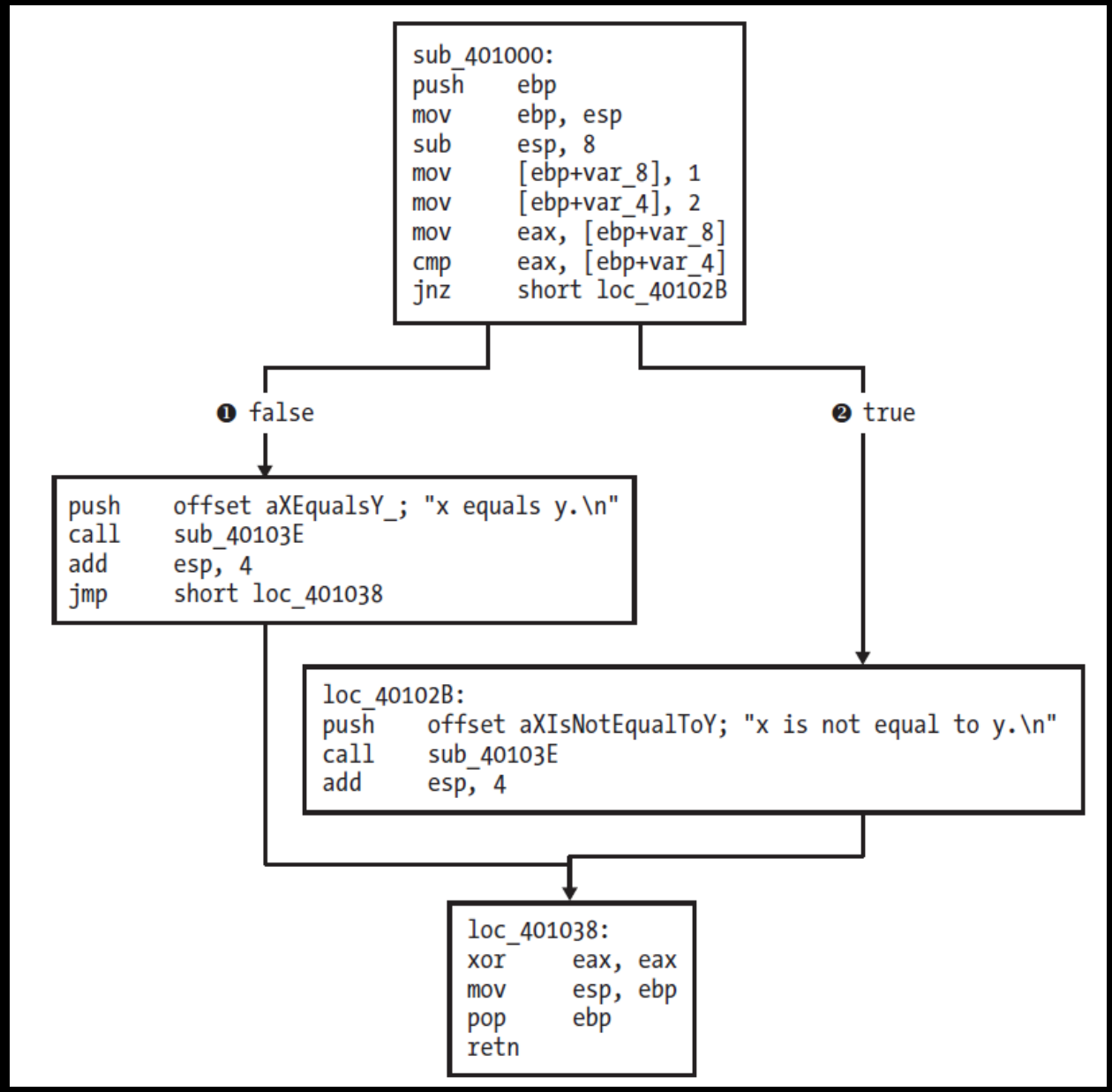
```
main() {
  int x = 9;
  do_stuff(x);
}
```

```
mov     [ebp+var_18], esp
```

`if` Statements

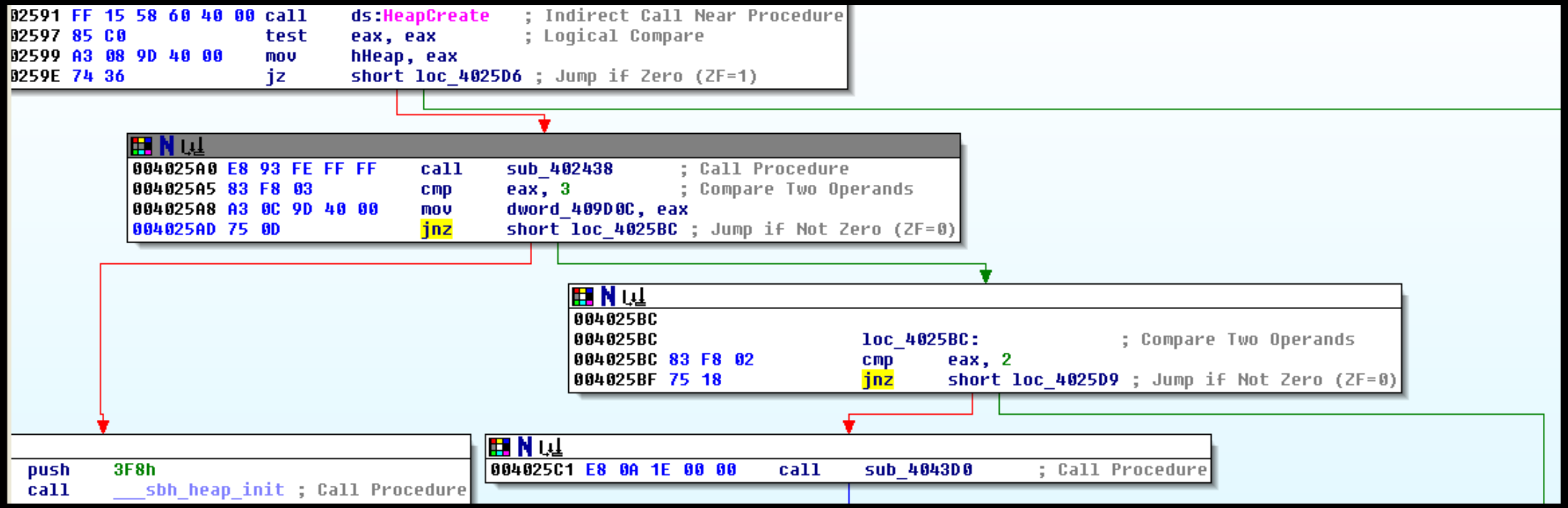
- Alters program flow based on a certain condition(s)
- `if` statements can be nested in `if` statements
- Usually a conditional jump for an `if` statement
  - Sometimes a cmov instruction

`if` statement example



Nested `if` Example

- `if` statement within an `if` statement



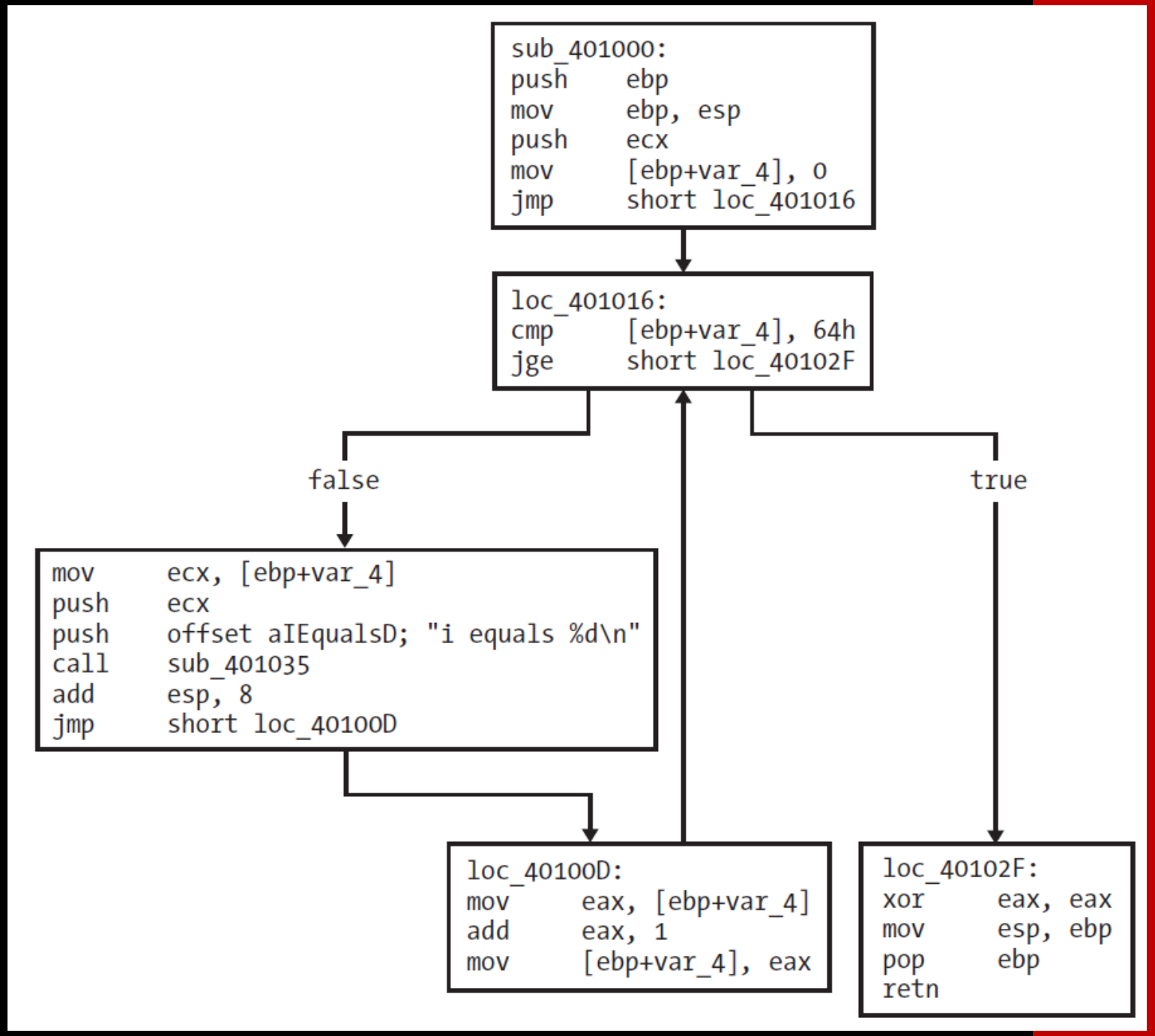
Loops

- Perform repetitive tasks
- Common in most programs
- Two types
  - `for` - initialization, comparison, execution instructions, increment or decrement
  - `while` - initialization, comparison, execution

`for` Loop Example

```
int i;

for(i=0; i<100; i++)
{
   printf("i equals %d\n", i);
}
```



`while` Loop Example

```
int status=0;
int result = 0;

while(status == 0){
     result = performAction();
     status = checkResult(result);
}
```

```
00401036          mov     [ebp+var_4], 0
0040103D          mov     [ebp+var_8], 0
00401044 loc_401044:
00401044          cmp     [ebp+var_4], 0
00401048          jnz     short loc_401063
0040104A          call    performAction
0040104F          mov     [ebp+var_8], eax
00401052          mov     eax, [ebp+var_8]
00401055          push    eax
00401056          call    checkResult
0040105B          add     esp, 4
0040105E          mov     [ebp+var_4], eax
00401061          jmp     short loc_401044
```

Understanding Calling Conventions

- Calling conventions determine the way the function call happens
- Conventions include
  - Order parameters are placed on the stack
  - Order parameters are placed in registers
  - Whether the caller or the function called is responsible for stack cleanup when complete
- 3 most common are
  - `cdecl`
  - `stdcall`
  - `fastcall`

cdecl

- One of the most popular conventions
- Parameters are pushed from right to left
- Caller cleans up the stack when the function completes
- Return value stored in `EAX`

```
push c
push b
push a
call test
add esp, 12
mov ret, eax
```

`stdcall`

- Similar to `cdecl`
  - Difference is `stdcall` requires the callee to clean up the stack when the function completes
- Functions are compiled differently than `cdecl` when `stdcall` is used
- Standard calling convention for the Windows API
- Code to clean up stack is in the DLL that implement the API function

fastcall

- Varies the most across compilers
- Typically uses two arguments and passes to registers
  - EDX
  - ECX
- Additional arguments loaded from right to left
- Calling function usually responsible for cleaning up the stack

Push vs Move

- Different compilers may choose to use one over the other
- Accomplishes the same thing
- Different settings and options can change the calling conventions of a compiler

Push vs. Move Illustrated

Visual Studio version			GCC version		
00401746	mov	[ebp+var_4], 1	00401085	mov	[ebp+var_4], 1
0040174D	mov	[ebp+var_8], 2	0040108C	mov	[ebp+var_8], 2
00401754	mov	eax, [ebp+var_8]	00401093	mov	eax, [ebp+var_8]
00401757	push	eax	00401096	mov	[esp+4], eax
00401758	mov	ecx, [ebp+var_4]	0040109A	mov	eax, [ebp+var_4]
0040175B	push	ecx	0040109D	mov	[esp], eax
0040175C	call	adder	004010A0	call	adder
00401761	**add**	**esp, 8**			
00401764	push	eax	004010A5	mov	[esp+4], eax
00401765	push	offset TheFunctionRet	004010A9	mov	[esp], offset TheFunctionRet
0040176A	call	ds:printf	004010B0	call	printf

`switch` Statements

- Used to make decisions based on a character or integer
  - Example: Backdoors can select action based on a single byte value
- Commonly compiled in two ways
  - `if` style
  - `jump` tables

```
switch(i)
{
   case 1:
      printf("i = %d", i+1);
      break;
   case 2:
      printf("i = %d", i+2);
      break;
   case 3:
      printf("i = %d", i+3);
      break;
   case 4:
      printf("i = %d", i+3);
      break;
   default:
      break;
}
```

`if` style vs. jump table

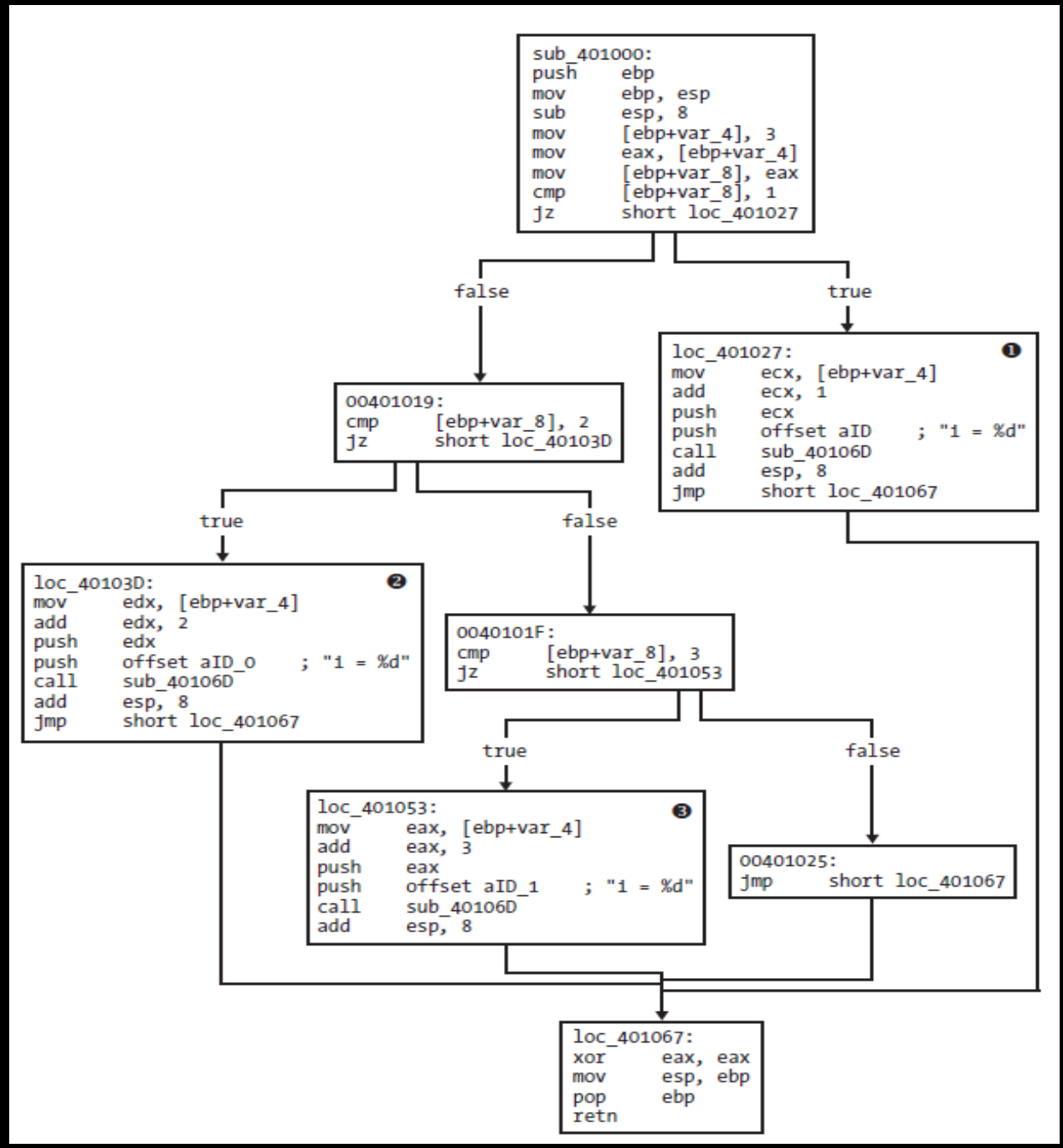
`if` style

- Series of `if` statements
- Comparisons and conditional jumps are shown in assembly
- Each conditional jump leads to instructions followed by an unconditional jump

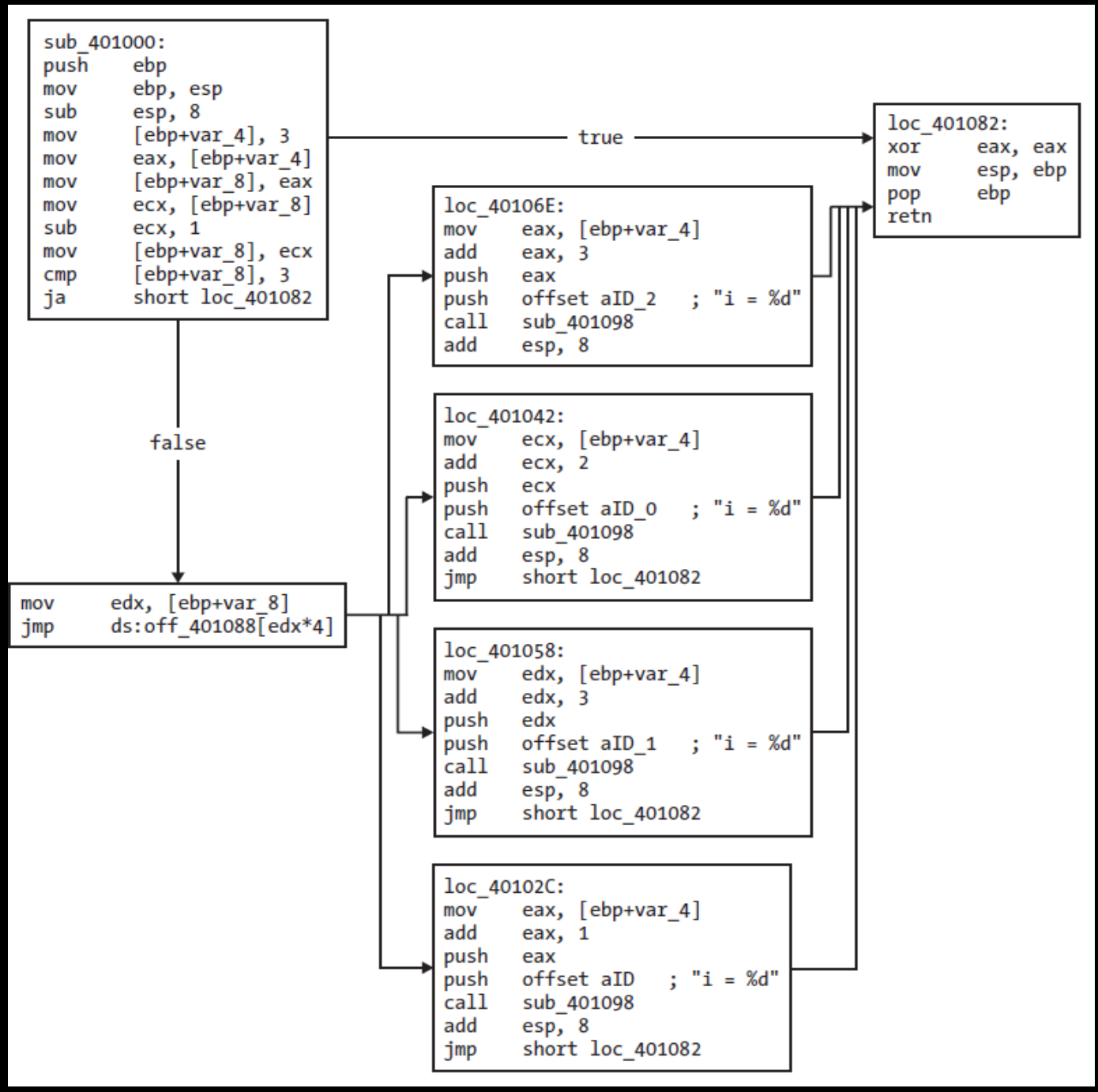
jump table

- Common with large contiguous `switch` statements
- Compiler optimizes code to avoid many comparisons, and is more efficient
- Code is broken down into separate chunks, and the jump table determines which one to use

`switch` statement with `if` style



`switch` statement with jump table



Arrays

- Define an ordered set of similar data
- Sometimes used by malware to point to different host names, which give it options
- Like variables, can be global or local
- Arrays are accessed in assembly using a base address as starting point

Array Example

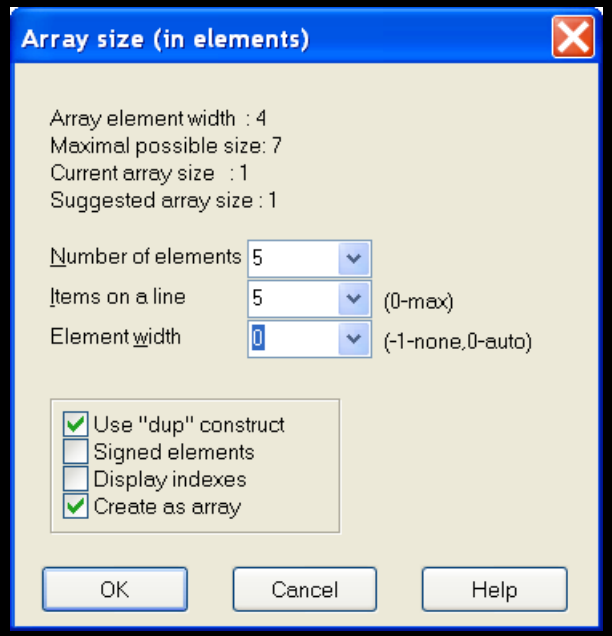
```
int b[5] = {123,87,487,7,978};
void main()
{
   int i;
   int a[5];

   for(i = 0; i<5; i++)
   {
      a[i] = i;
      b[i] = i;
   }
}
```

```
00401006          mov     [ebp+var_18], 0
0040100D          jmp     short loc_401018
0040100F loc_40100F:
0040100F          mov     eax, [ebp+var_18]
00401012          add     eax, 1
00401015          mov     [ebp+var_18], eax
00401018 loc_401018:
00401018          cmp     [ebp+var_18], 5
0040101C          jge     short loc_401037
0040101E          mov     ecx, [ebp+var_18]
00401021          mov     edx, [ebp+var_18]
00401024          mov     [ebp+ecx*4+var_14], edx
00401028          mov     eax, [ebp+var_18]
0040102B          mov     ecx, [ebp+var_18]
0040102E          mov     dword_40A000[ecx*4], eax
00401035          jmp     short loc_40100F
```

Defining Arrays

- Create the first element of array using the data definition commands
- Apply the array command to the item
- Define
  - size
  - display options
  - width



Structures (`struct`)

- Similar to arrays, but are made up of elements of different types
- Commonly used in malware to group information
- It's easier to use a `struct` to maintain many variables, especially if many functions need access to the same group of variables
- Windows API often uses structures
- Accessed by a base address in assembly, same as arrays