CSCE 790
Introduction to Software Analysis

Code Obfuscation II

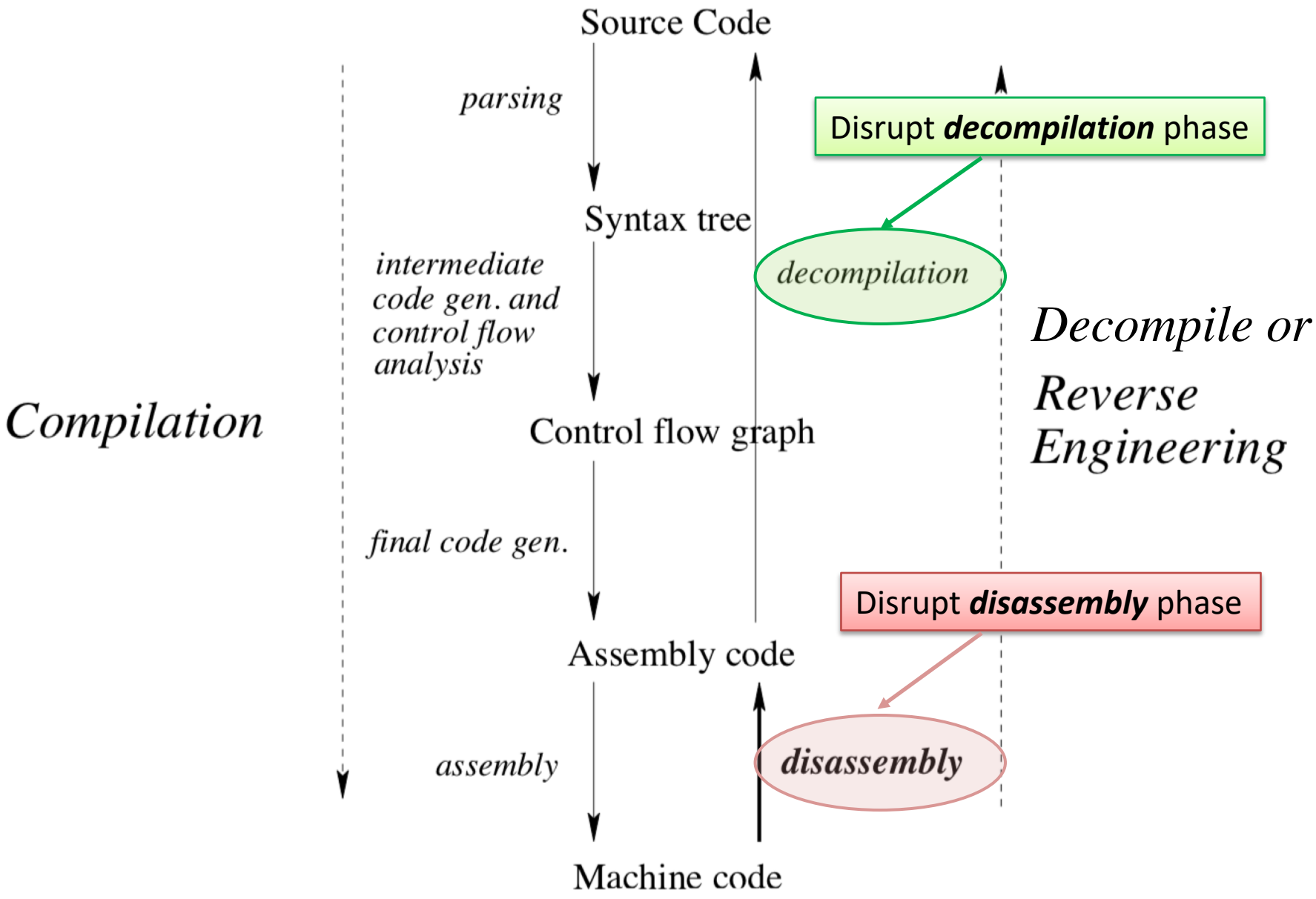
Professor Lisa Luo

Fall 2018

UNIVERSITY OF
SOUTH CAROLINA

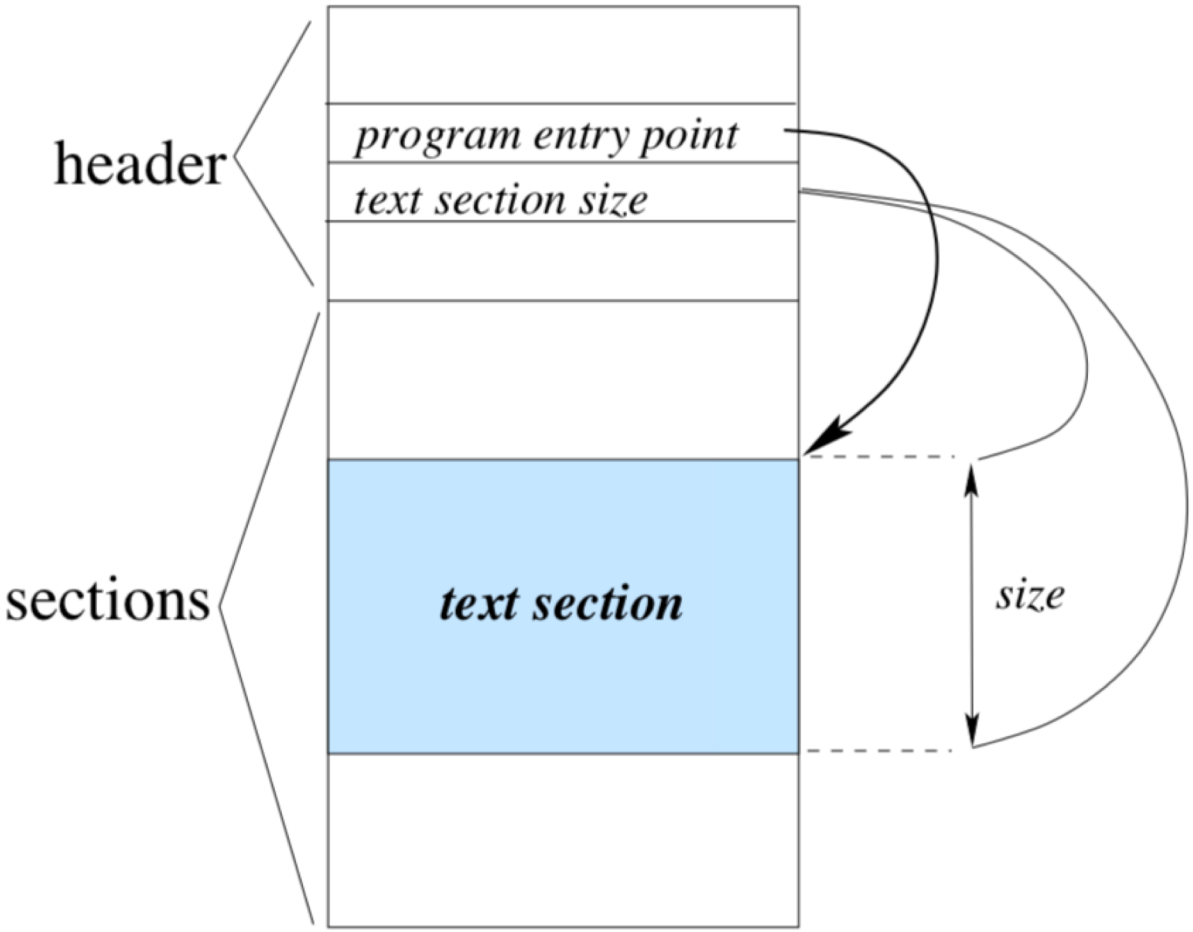
Summary

- What is code obfuscation and its applications?
  - Code obfuscation can be used to protect code, but it is also used by malware to evade detection
- Obfuscation thwarts decompilation
  - Control flow obfuscation
    - Opaque predicates,
    - Control flow flattening
    - Function inline and outline
    - Function clone
  - Data flow obfuscation
    - Converting static data to procedural data
    - Encoding integers
    - Spliting variables
    - Restructuring arrays


Source Code
parsing
Syntax tree
intermediate code gen. and control flow analysis
Control flow graph
final code gen.
Assembly code
assembly
Machine code
Compilation
Disrupt decompilation phase
decompilation
Decompile or Reverse Engineering
Disrupt disassembly phase
disassembly

Background of Disassembly

Structure of an Executable File



Two Approaches of Disassembly

- Static disassembly
  - Executable is not executed
  - Pro: can process the entire file all at once
- Dynamic disassembly
  - Executable is executed on some input and monitored by an external tool (e.g., a debugger)
  - Con: only the instructions that are being executed can be identified.

Static Disassembly

- Two generally used techniques:
  - Linear sweep
  - Recursive traversal

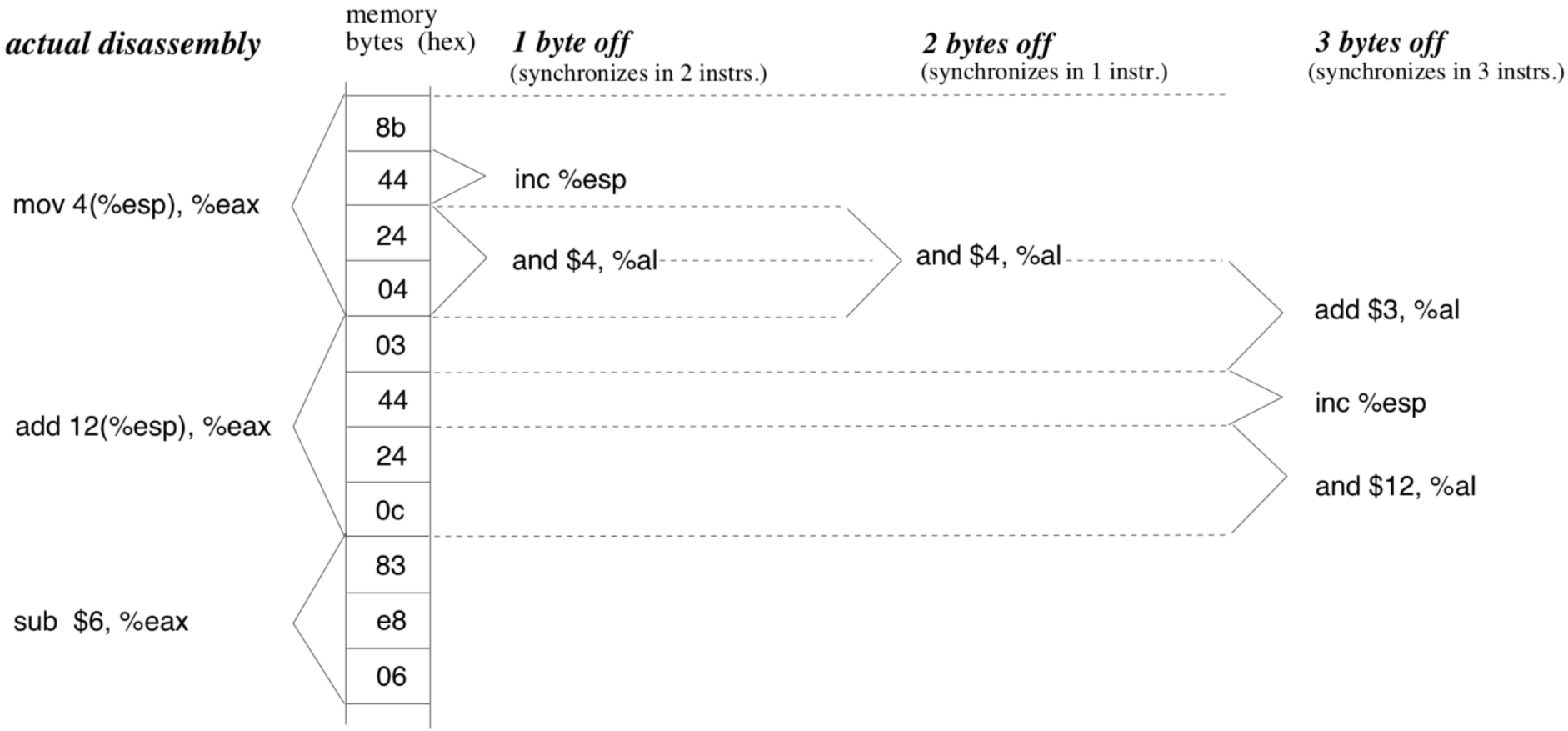
Thwarting Linear Sweep

Linear Sweep

- It begins at the first executable byte, and sweeps through the entire text section disassembling each instruction
- Used by *objdump*

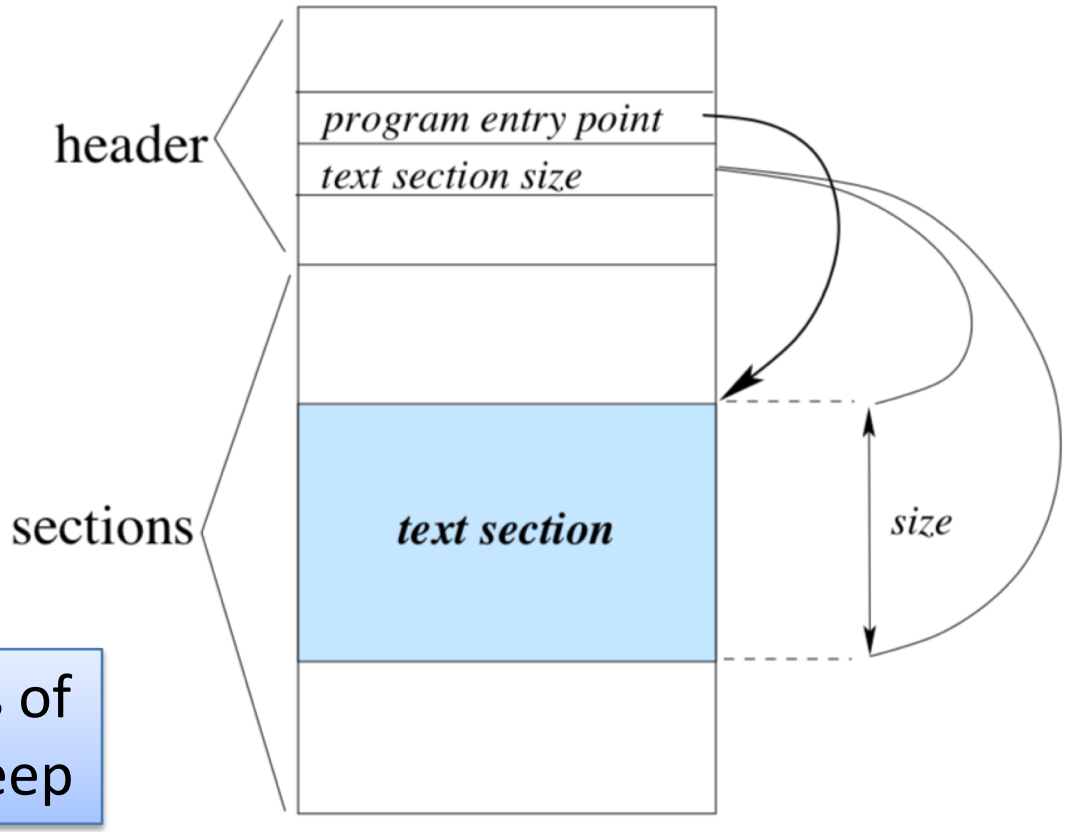
```
global startAddr, endAddr;
proc DisasmLinear(addr)
begin
   while (startAddr ≤ addr < endAddr) do
      I := decode instruction at address addr;
      addr  +=  length(I);
   od
end

proc main()
begin
   startAddr := address of the first executable byte;
   endAddr := startAddr + text section size;
   DisasmLinear(ep);
end
```

Linear Sweep



Linear Sweep

- *Drawbacks:* If the text session contain data, the data will be treated as code and disassembled as instructions

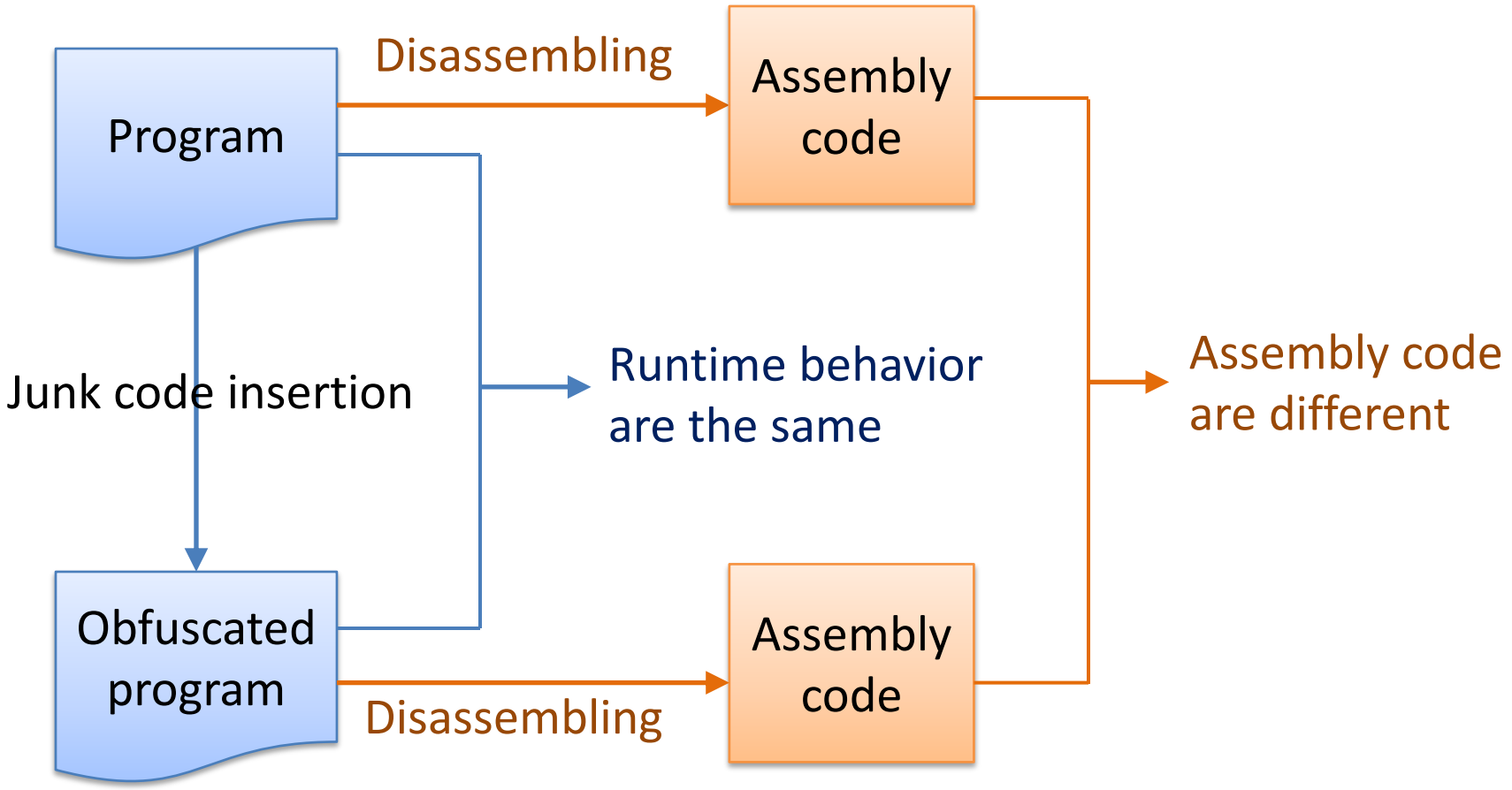


Obfuscation makes uses of this to thwart linear sweep

Thwarting Linear Sweep

- Junk code insertion
- Two properties:
  1. Inserted junk code should NOT effect the program's runtime behaviors
     - *The junk code should unreachable at runtime*
  2. Inserted junk code should confuse the static disassembler, such that the generated disassemble code is not correct

Junk Insertion

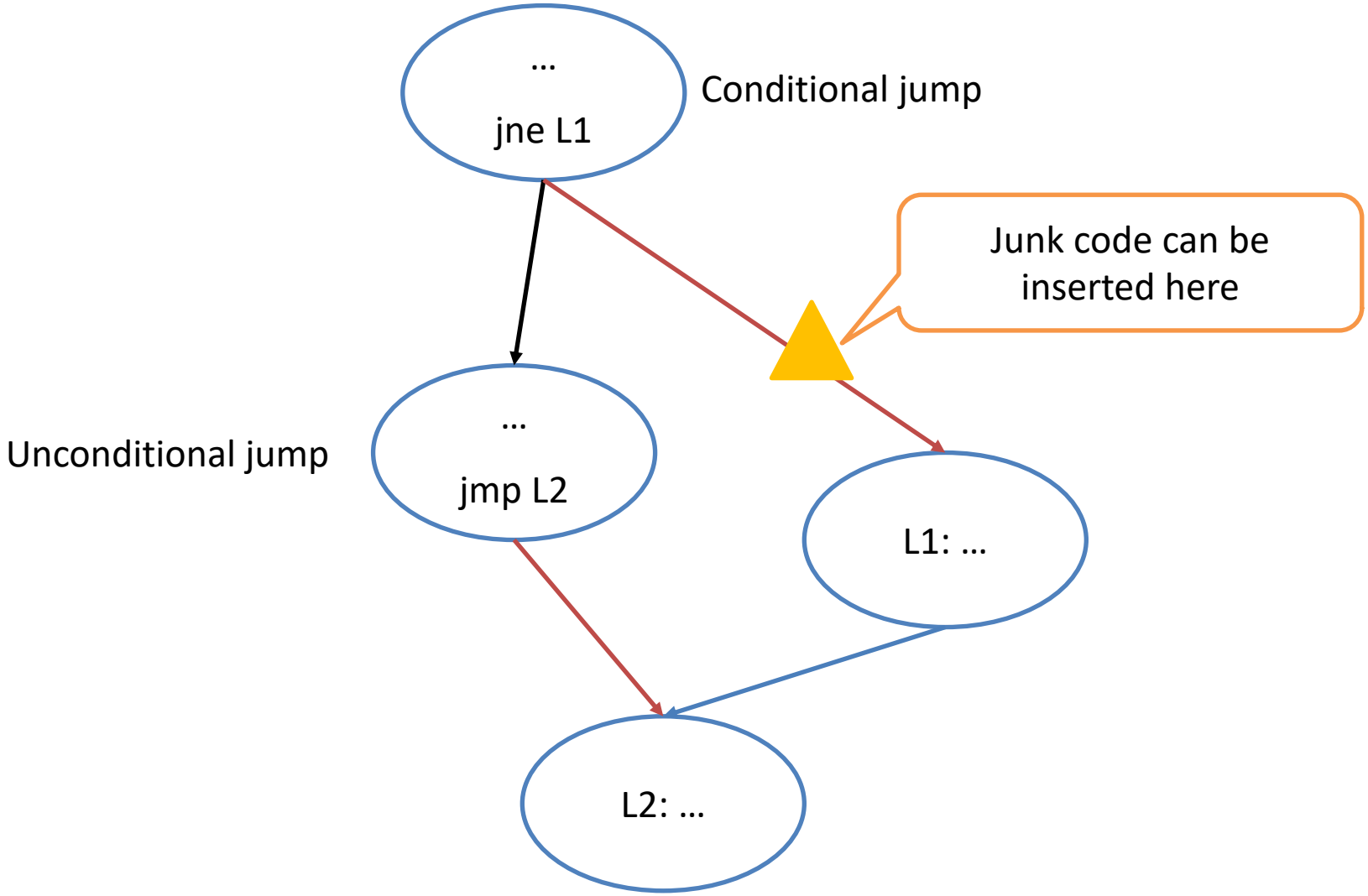


Example

```
    8048000   55              push   %ebp          |  function func(int arg) {
    8048001   89 e5           mov    %esp, %ebp    |      int local_var, ret_val;
                                                   |
    8048003   e8 00 00 74 11  call   19788008 <branch fnct>  |  local = other_func(arg);
                                                   |
                                                   |
    804800a   3c 00           cmp    0, %eax       |      if (local_var == 0)
    804800c   75 06           jne    8048014 <L1>  |
    804800e   b0 00           mov    0, %eax       |          ret_val = 0;
    8048010   eb 07           jmp    8048019 <L2>  |      else
    8048012   0a 05           (junk)               |
L1: 8048014   a1 00 00 74 01  mov    (1740000), %eax  |      ret_val = global_var;
                                                   |
L2: 8048019   89 ec           mov    %ebp, %esp    |      return ret_val;
    804801b   5d              pop    %ebp          |
    804801c   c3              ret                  |
    804801d   90              nop                  |  }
```

```
804800e   b0 00              mov   0, %eax
8048010   eb 07              jmp   8048019
8048012   0a 05 a1 00 00 74  or    740000a1, %al
8048014
8048018   01 89 ec 5d c3 90  adc   %ecx, 90c35dec(%ecx)
8048019
```

Junk code should be inserted before a basic block which is only the target of a *conditional jump*



Thwarting Recursive Traversal

Recursive Traversal

- The problem of linear sweep:
  - It does not take into account the control flow behavior of the program, and thus cannot "go around" data embedded in the text session, and mistakenly interprets them as executable code.

- Recursive traversal fixes this problem, and take into account the control flow behavior of the program

Recursive Traversal

- Whenever a branch is encountered, it determine the possible control flow successors of that instruction, and proceed with disassembly at those addresses

```
global startAddr, endAddr;
proc DisasmRec(addr)
begin
   while (startAddr ≤ addr < endAddr) do
      if (addr has been visited already) return;
      I := decode instruction at address addr;
      mark addr as visited;
      if (I is a branch or function call)
         for each possible target t of I do
            DisasmRec(t);
         od
         return;
      else addr += length(I);
   od
end
```

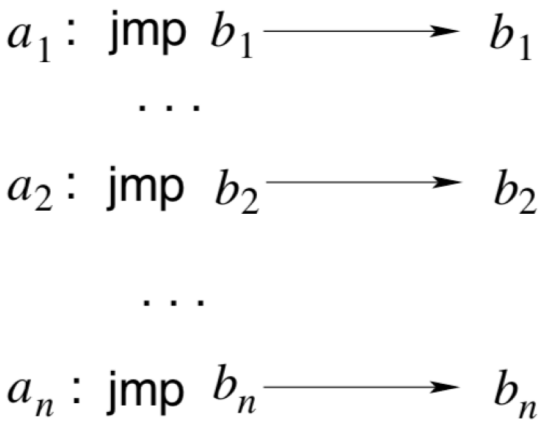
```
proc main()
begin
   startAddr := program entry point;
   endAddr := startAddr + text section size;
   DisasmRec(startAddr);
end
```

Thwarting Recursive Traversal

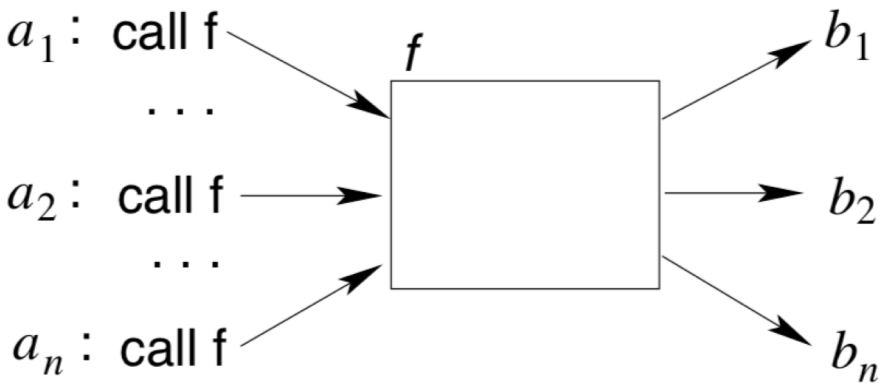
- Branch functions

- When it is called from one of the locations a_i, it transfers the control to the corresponding location b_i
- Then we can insert junk code after the branch functions

Branch Functions



(a) Original code



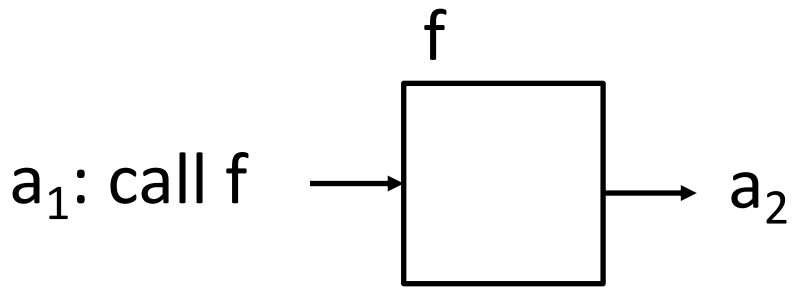
(b) Code using a branch function

At runtime, the branch function *modifies the return address* such that the next instruction is at b_1, b_2, or b_n

Branch Functions

a_1: call function
a_2: ….

(a) Original code



(b) Code using a branch function

At runtime, the branch function *modifies the return address* such that the next instruction is at a_2.

Example

At runtime, the branch function *modifies the return address* such that the next instruction is at 804800a

```
    8048000   55              push   %ebp          function func(int arg) {
    8048001   89 e5           mov    %esp, %ebp        int local_var, ret_val;

    8048003   e8 00 00 74 11  call   19788008 <branch fnct>   local = other_func(arg);
    8048008   0a 05           (junk)

    804800a   3c 00           cmp    0, %eax        if (local_var == 0)
    804800c   75 06           jne    8048014 <L1>
    804800e   b0 00           mov    0, %eax            ret_val = 0;
    8048010   eb 07           jmp    8048019 <L2>   else

L1: 8048014   a1 00 00 74 01  mov    (1740000), %eax    ret_val = global_var;

L2: 8048019   89 ec           mov    %ebp, %esp     return ret_val;
    804801b   5d              pop    %ebp
    804801c   c3              ret
    804801d   90              nop                }
```

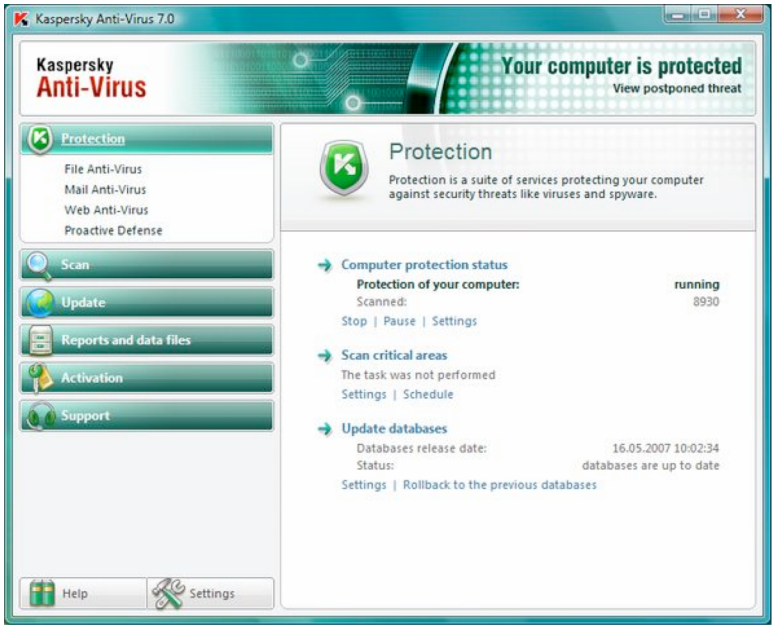
```
e8 00 00 74 11      call   19788008 <branch fnct>
0a 05 3c 00 75 06   or     675003c, %al

b0 00               mov    0, %eax
eb 07               jmp    8048019
```

Code Obfuscation as it Related to Malware

Anti-Virus

- Analyze binary to decide if it is a virus
- Analyze program behavior

- Types:
  - Scanner
  - Real time monitor



1. Scanner : Virus signature

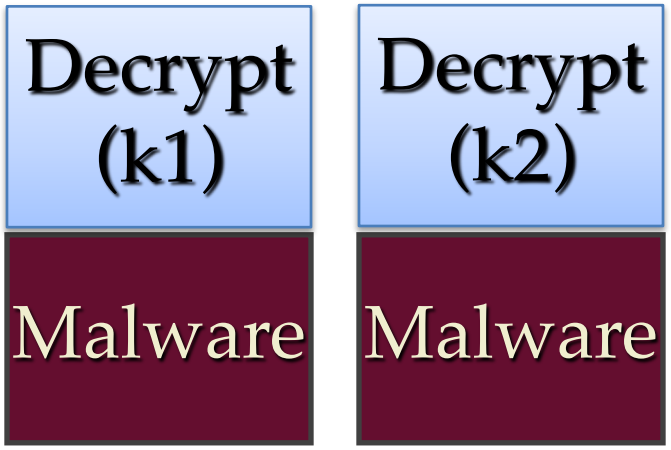
- Find a string that can identify the virus
- Fingerprint like

Virus's Defense

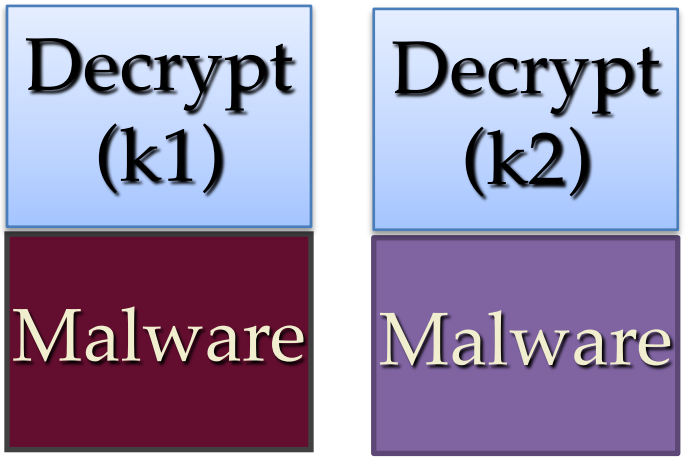
- Change their code as they propagate

- Some Virus types:
  - Polymorphic virus
  - Metamorphic virus

Polymorphic Virus



Metamorphic Virus



Code Obfuscation

Code Obfuscation

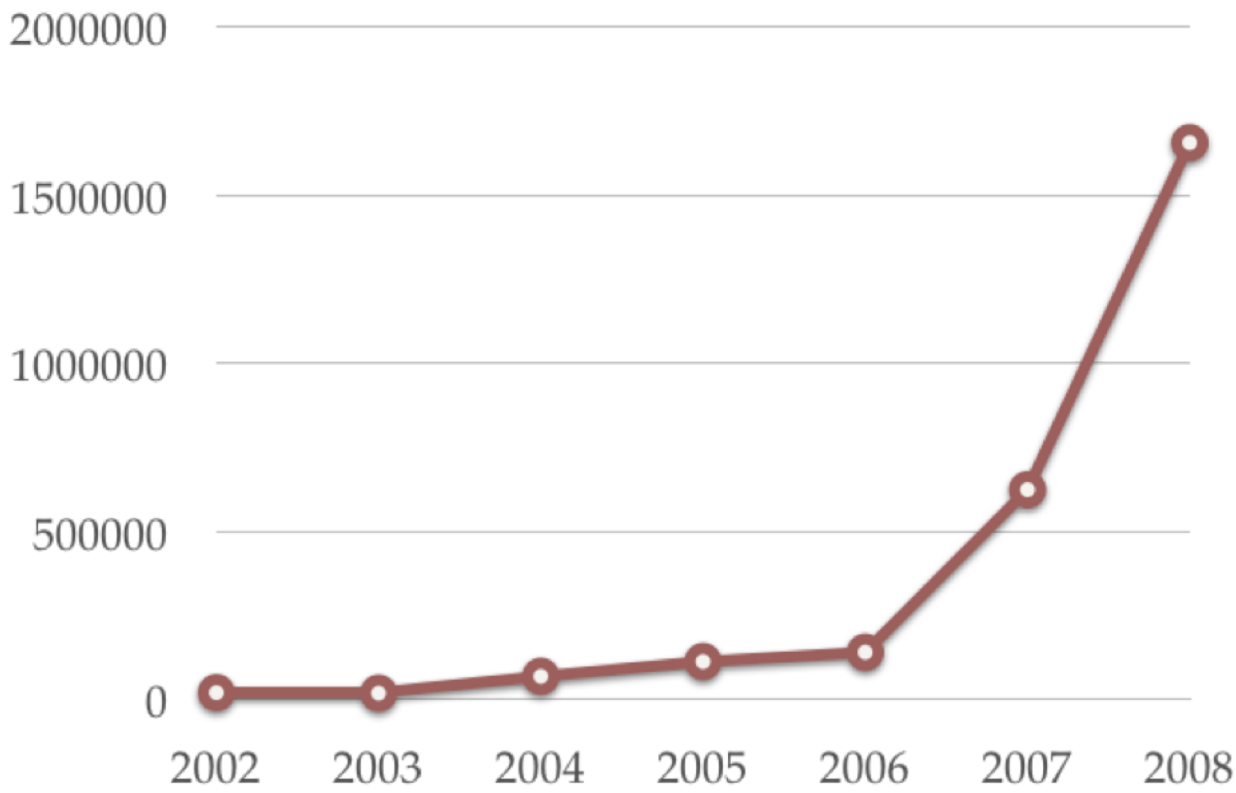
- ***Goal:*** prevent signature-based detection, and reverse-engineering

- Code obfuscation
  - Hard-to-analyze code structures
  - Different code in each copy of the virus
    - Effect of code execution is the same, but this is difficult to detect by static analysis

Code Obfuscation Techniques

- Some examples:
  - Control flow obfuscation
    - Opaque predicates,
    - Control flow flattening
    - Function inline and outline
    - Function clone
  - Data flow obfuscation
    - Converting static data to procedural data
    - Encoding integers
    - Spliting variables
    - Restructuring arrays
  - Junk insertion & branch functions …
- There is no constant, recognizable virus body

Number of malware signatures



Symantec report 2009

2. Real Time Monitor : Heuristics

- Analyze program behavior
  - Network access
  - File open
  - Attempt to delete file
  - Attempt to modify the boot sector

Sandbox analysis

- Running the executable in a VM
- Observe it
  - File activity
  - Network
  - Memory

Virus's Defense

- Anti-debugger detection and VM detection
  - Detect debuggers and virtual machines, and terminate execution

Drawbacks

- Limited code coverage
- Impossible to analyze/execute all the code

Summary

- Two generally used techniques in static disassembly:
  - Linear sweep
  - Recursive traversal
- Obfuscation thwarts disassembly
  - Thwarting linear sweep: junk insertion
  - Thwarting recursive traversal: branch functions
- Obfuscation as it related to viruses
  - Detect malware
    - Static & dynamic
  - Virus Types
    - Polymorphic
    - Metamorphic
  - Obfuscation techniques