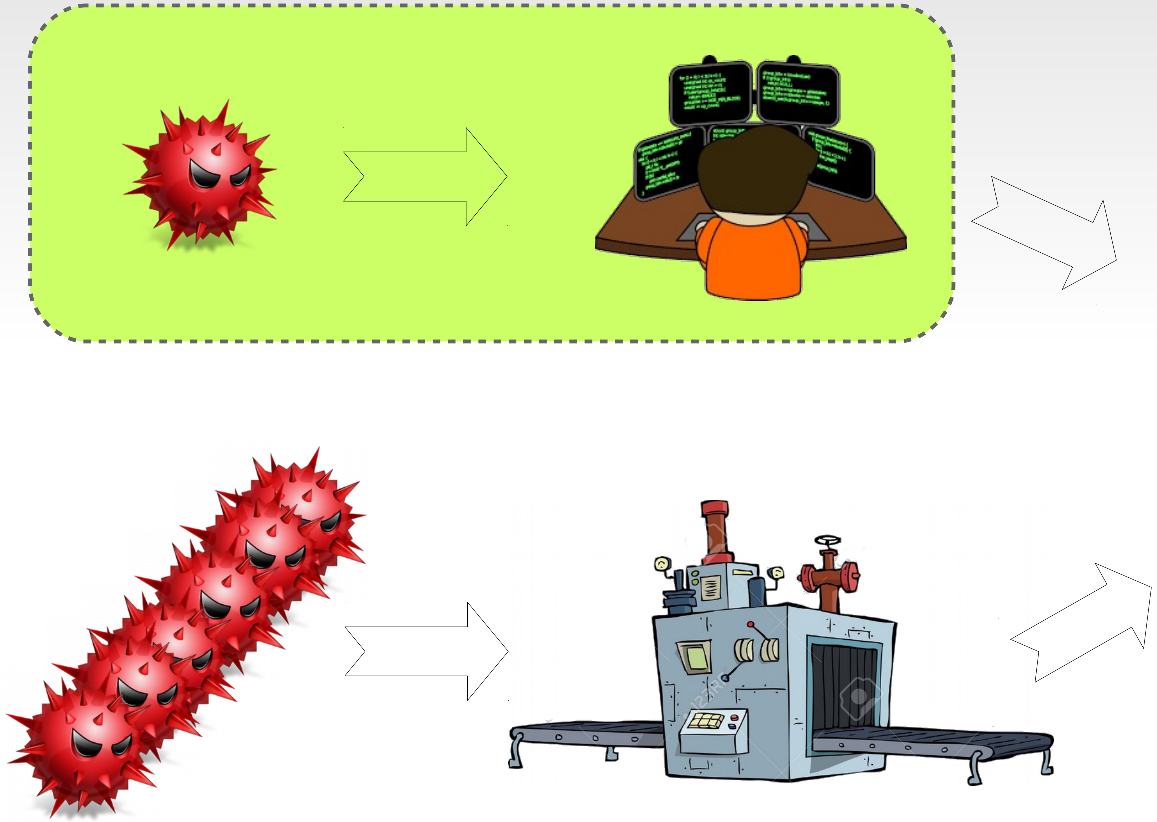
01110111011100101
01101100011101100
01000000001100100
01101111001101110
01100101010100001

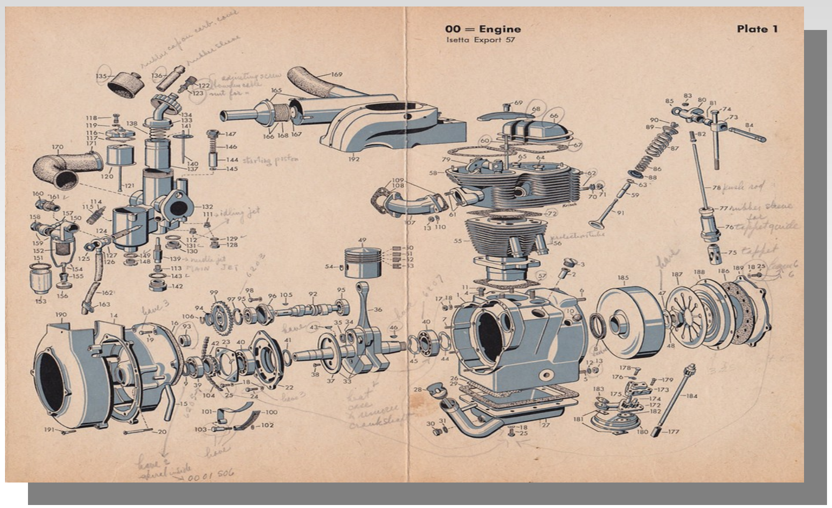
Static Analysis
(part B)

Davide Balzarotti
davide.balzarotti@eurecom.fr

Malware Analysis



Behavior
Triggers
Malware or Goodware?
Type and Family



- Assembly 101
- Disassembly algorithms
  - → Linear sweep and recursive traversal
  - → Detecting function prologues
- Decompilation
- Language Constructs
  - → Assembly and C
  - → Assembly and C++
- Limit of Static Analysis
  - → General limitations
  - → Anti-disassembly
  - → Packing

Assembly 101

Machine Instruction Sets

- RISC (Reduced Instruction Set Computing)
  - Small, highly-optimized set of instructions
  - Fixed length instructions
  - E.g., IBM PowerPC, ARM, Sparc, MIPS

- CISC (Complex Instruction Set Computing)
  - Complex instructions capable of performing multi-step operations
  - Multibyte instructions
  - E.g., x86, x86-64, Motorola 68K

x86 Assembler

- Human-readable form of machine instructions
  - It makes machine instructions readable, it does not make them simple
  - The programmer still has to understand the hardware architecture and the memory model

```
48 83 ec 30  →  sub rsp, 0x30
```

x86 Assembler

- Human-readable form of machine instructions
  - It makes machine instructions readable, it does not make them simple
  - The programmer still has to understand the hardware architecture and the memory model
- Each instruction is in the form:

```
mnemonic arguments
```

  - The mnemonic defines the operation to perform, and the arguments (0 to 2) define its parameters
  - Arguments can be *constants*, *registers*, or *memory addresses* (but they cannot be both memory addresses)
  - The resulting instruction has a variable length (between 1 and 13 bytes in 32bit mode)

Intel vs AT&T Syntax

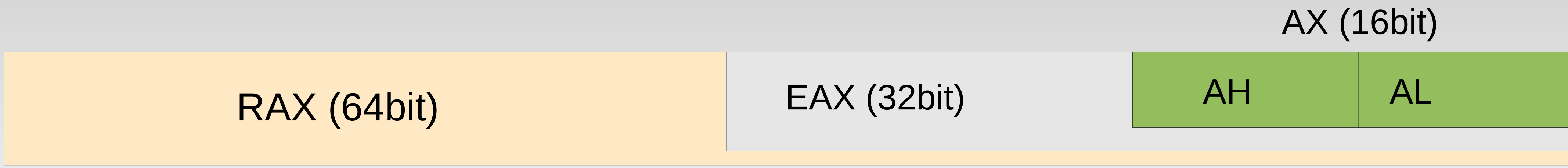
INTEL	AT&T
instr dest,source	*instr source,dest*
`mov eax,1`	`movl $1, %eax`
`int 80h`	`int $0x80`
`mov ax,bx`	`movw %bx, %ax`
`mov eax,[ebx+3]`	`movl 3(%ebx),%eax`
`mov eax, dword ptr [ebx]`	`movl %ebx),%eax`
`mov eax,[ebx+20h]`	`movl 0x20(%ebx),%eax`
`sub eax,[ebx+ecx*4h-20h]`	`subl -0x20(%ebx,%ecx,0x4),%eax`

Registers

- Like variables built-in in the processor
  - Used to store temporary data
  - Can be used explicitly by the user, or implicitly by certain instructions
- The IA32 architecture has 16 basic 32bit registers
  - General purpose registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
  - Segment registers: CS, SS, DS, ES, FS, GS
  - Instruction pointer: EIP
  - Flags register: EFLAGS
- Special purpose registers:
  - CR3, DR0-7, 80bit Floating Point Registers, MMX, test registers, ...

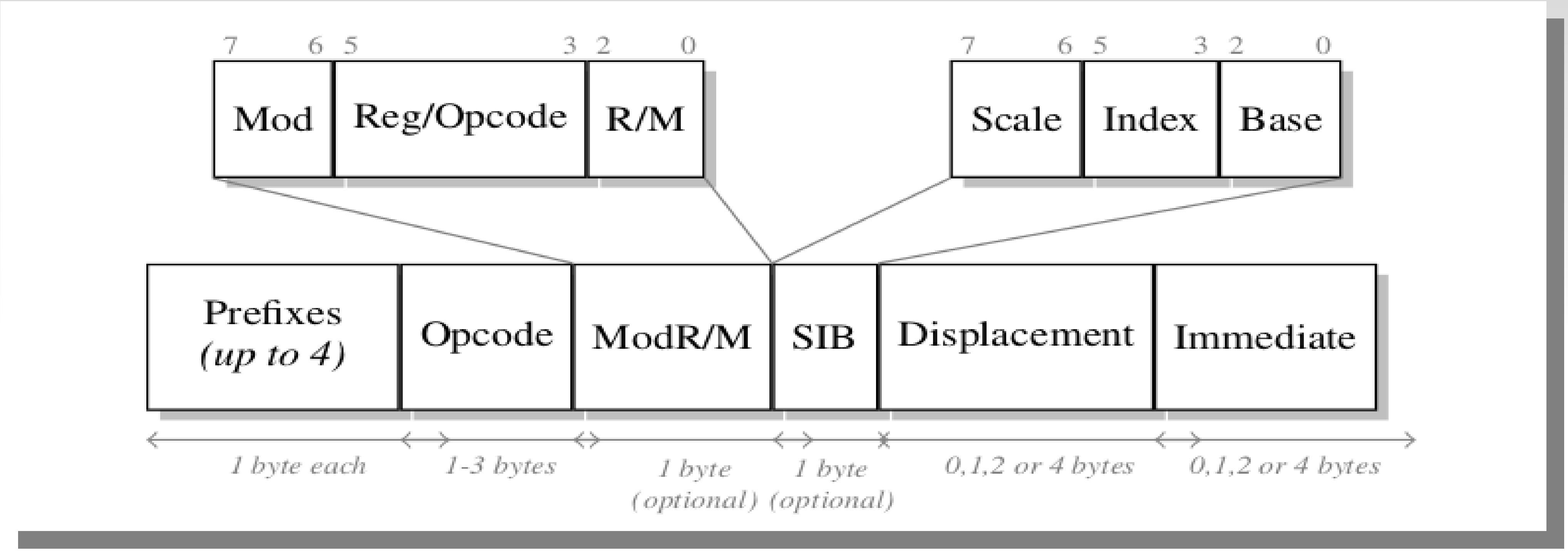
Registers

- Register access



- Main Status Flags
  - Are automatically changed according to the result of certain operations
  - **C** (carry) → unsigned result is too large or below zero
  - **O** (overflow) → signed result is too large or small
  - **S** (sign) → result sign (1=neg, 0=pos)
  - **Z** (zero) → result is zero

Intel x86 instruction format



Instructions

- Data movement

  MOV, XCHG, PUSH, POP, LEA, ...

- Arithmetic & Logic

  ADD, SUB, MUL, DIV, INC, DEC, NOT, AND, OR, XOR, SHL, SHR, ROL, ROR

- Control Flow
  - JMP, JA, JAE, JB, JBE, ...
  - CALL, RET

- Check the Intel Architecture Software Developer's Manual (vol1-4)
  - Freely available online
  - <u>Everything</u> is in there

Control Flow

- In C
  - `if-then-else`
  - `for` loop
  - `while` and `do-while` loops
  - `switch`
- In Assembly
  - conditional (on one of the flags) and unconditional jumps
  - can be direct (e.g., `jmp 0x45`) or indirect (e.g., `jmp *eax`)
  - Control flow
    1. Test on operands
    2. Jump to a location if true
    3. Continue to the next instruction if not true

Linux Process Memory Layout

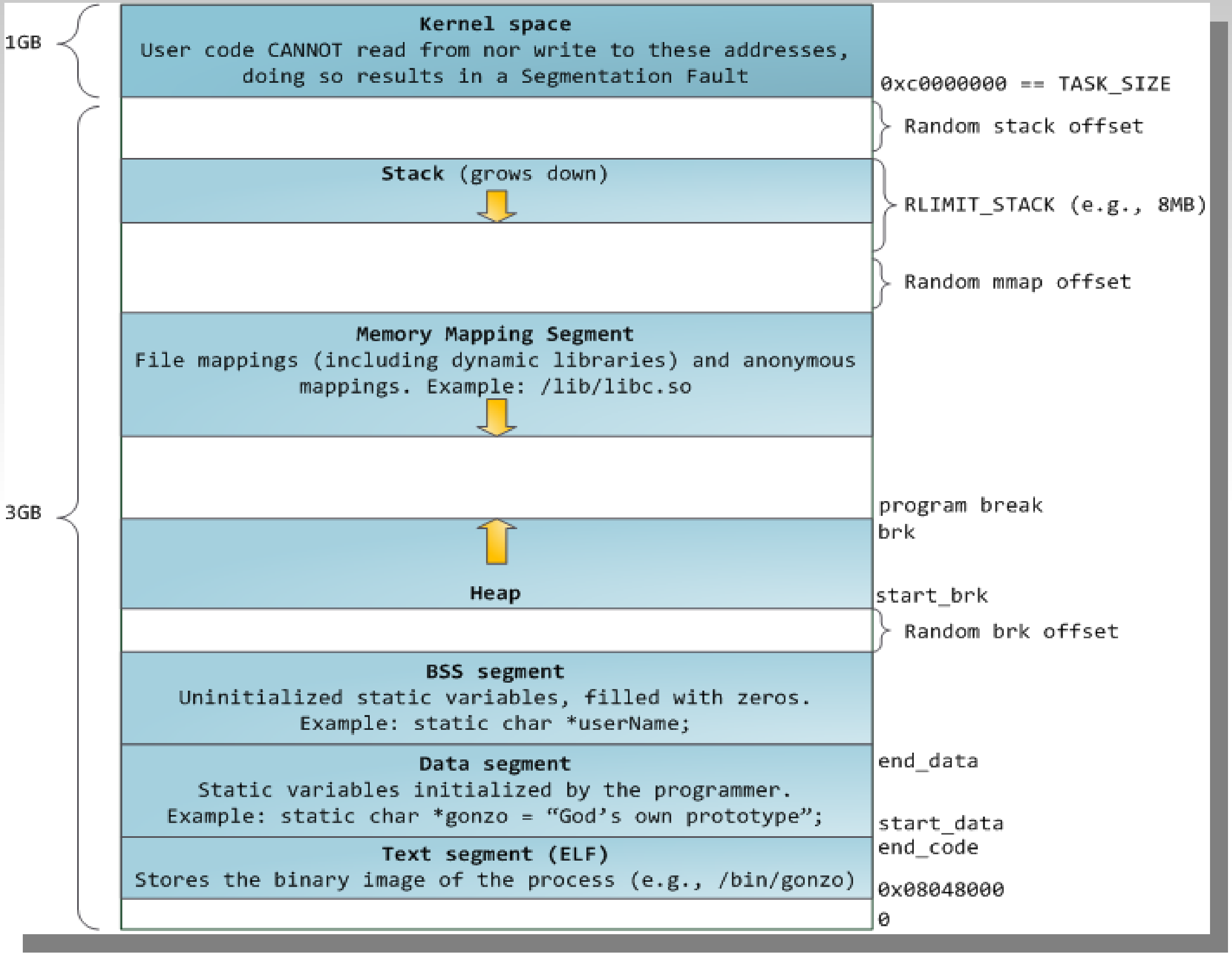
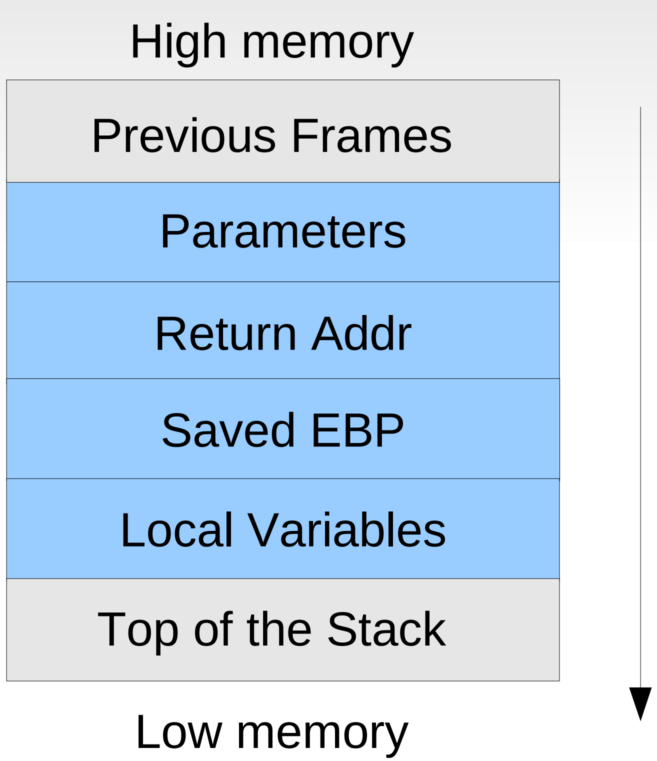


Image from Gustavo Duarte

Stack Frames

- The stack is normally used to store information for the running functions
- All the information of a function are grouped in a stack frame, that contains:
  - The Return Address
  - The Local variables (strictly not part of the frame)
  - The address of the previous frame
  - The Parameters
- ESP points to the top of the stack
- EBP is **normally** used to point to the current frame
  - This allows the code to reference parameters and local vars with fixed offset over EBP



Non-Standard Stack Frames

- Sometimes local variables and parameters are accessed through the ESP register
  - `-fomit-frame-pointer` in GCC
  - Save one register (EBP) for other uses
  - Hard to read the disassembly: since ESP is not constant during the function execution, all the offsets change as well
    - The same variable may be accessed as ESP+0x20 in one instruction and ESP+0x32 in another
- Sometimes the function modifies the saved `ebp` / return address inside his own frame
  - Often used by malware writers
  - … but also the `libc` has a couple of those :(

Calling Conventions

- Conventions (sometime standards) that specify:
  1. The way (where and in which order) the arguments are passed to a function
  2. How the result is passed back to the caller
  3. Who is responsible to set up and remove the stack frame
  4. Which registers can be used by the function without saving their values first on the stack
- There are many, many calling conventions :(

Main Calling Conventions

- CDEL (standard C)
  - Argument passed on the stack right-to-left
  - Return value is placed in the EAX register
  - The calling function clean the stack
  - EAX, ECX, EDX are free to use

Main Calling Conventions

- CDEL (standard C)
  - Argument passed on the stack right-to-left
  - Return value is placed in the EAX register
  - The calling function clean the stack
  - EAX, ECX, EDX are free to use
- STDCALL (defined by Microsoft for Win32 APIs)
  - Same as CDEL, but the called function clean the stack
- FASTCALL (not standard)
  - Same as CDEL, but the first 2 (or 3) parameters are passed in registers

Main Calling Conventions

- CDEL (standard C)
  - Argument passed on the stack right-to-left
  - Return value is placed in the EAX register
  - The calling function clean the stack
  - EAX, ECX, EDX are free to use
- STDCALL (defined by Microsoft for Win32 APIs)
  - Same as CDEL, but the called function clean the stack
- FASTCALL (not standard)
  - Same as CDEL, but the first 2 (or 3) parameters are passed in registers
- THISCALL (used by C++ for method invocation)
  - Same as CDEL, but the pointer to the class object (this) is passed in the ECX register

64bit Calling Convention

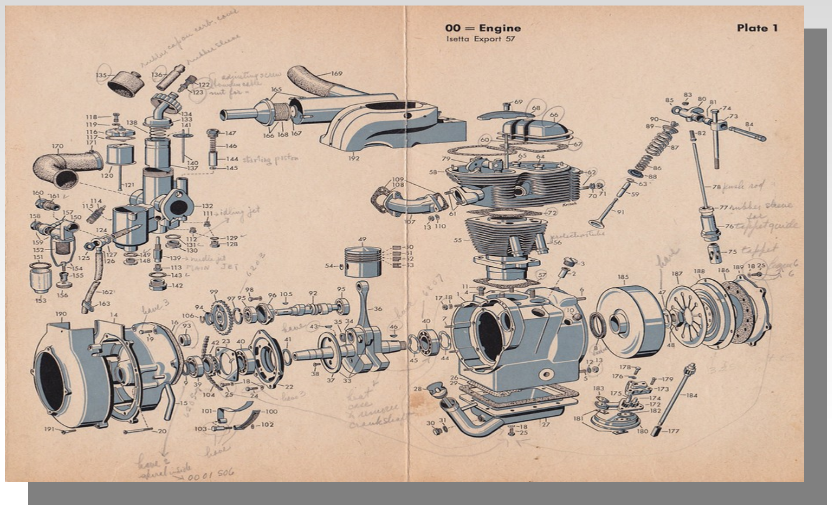
- Microsoft x64 (Windows, UEFI)
  - RCX, RDX, R8, R9 hold the first four parameters
  - The remaining are passed on the stack right to left
  - Return value in RAX
  - Shadow space of 32 bytes set up by the caller right before calling the function (and after pushing the parameters on the stack)

    → used to spill the first four parameters on the stack when debugging

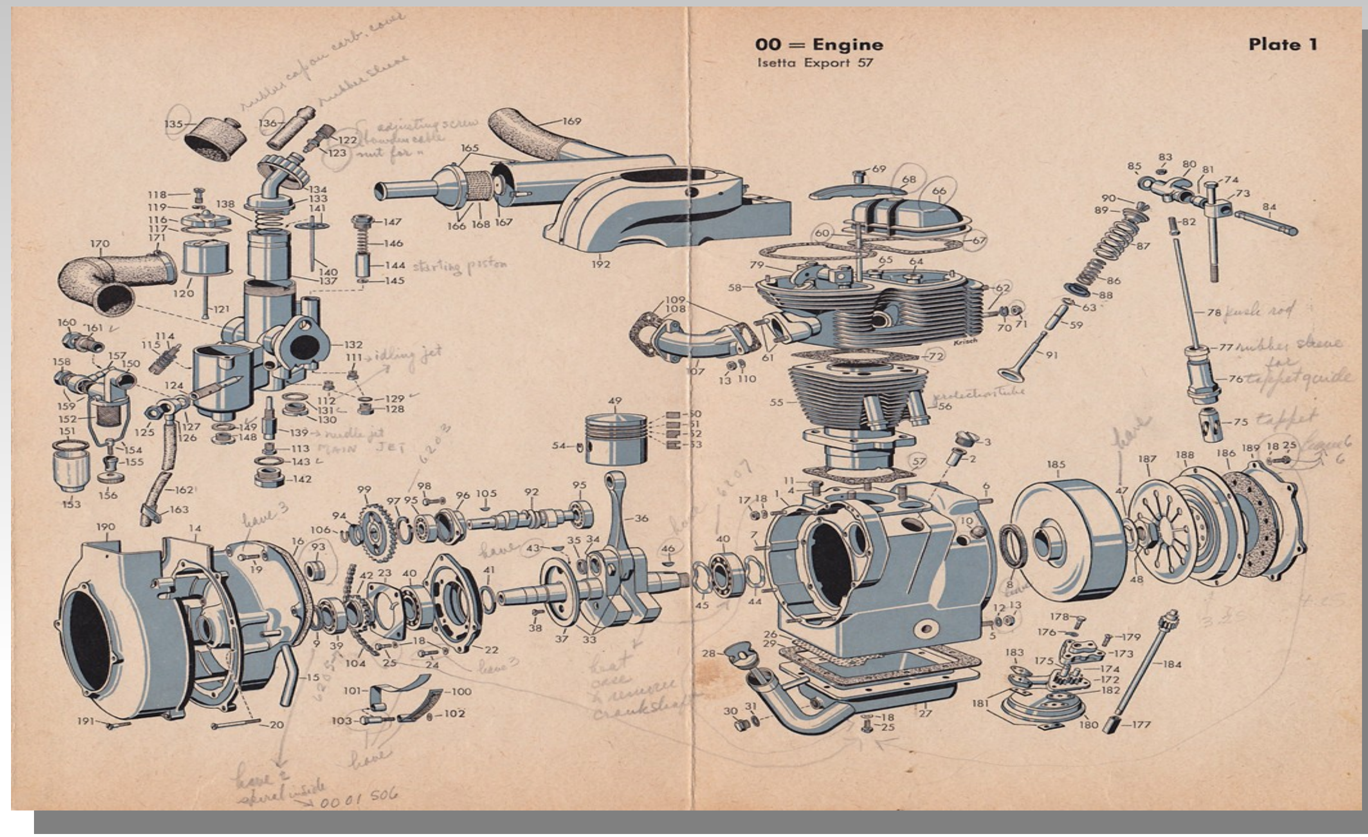
- SystemV AMD64 ABI (linux, Solaris, BSD, MacOS X)
  - RDI, RSI, RDX, RCX, R8, R9 for the first six parameters
  - Remaining parameters on the stack
  - Return value on RAX (or RAX and RDX for 128bits)

ABI

- Application Binary Interface
  - Defines the interface between software modules or userspace to kernel communication
  - Similar to an API, but at the binary level
- It defines
  - The calling convention
  - The layout and alignment of data types
  - The system call invocation and calling convention

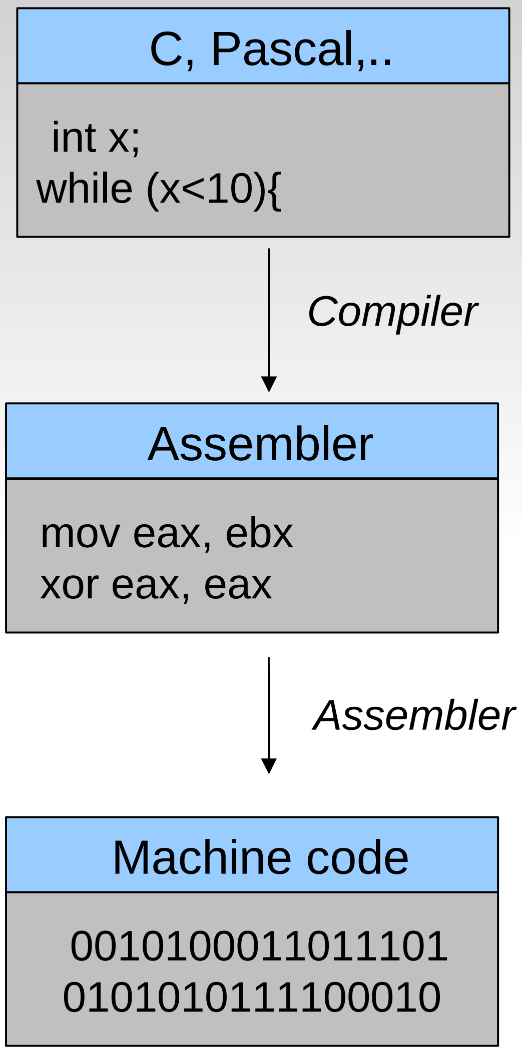


- ✔ Assembly 101
- Disassembly algorithms
  - → Linear sweep and recursive
  - → Detecting function prologues
- Decompilation
- Language Constructs
  - → Assembly and C
  - → Assembly and C++
- Limit of Static Analysis
  - → General limitations
  - → Anti-disassembly
  - → Packing

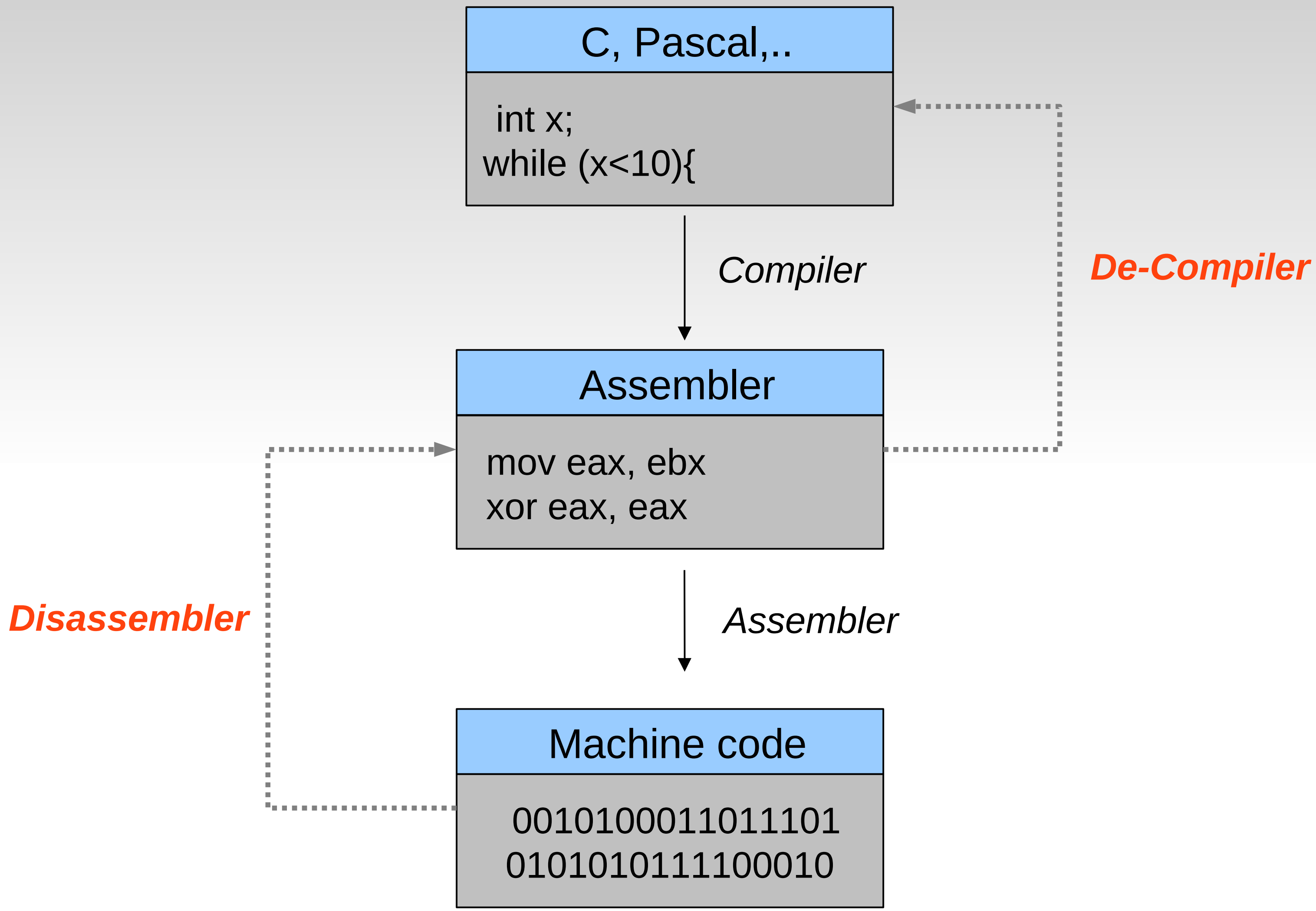


Instruction-Level Static Analysis

Engineering



Engineering



Disassembler

- The core function of a disassembler is to interpret executable files and decode their instructions
  - Every assembly instructions maps univocally to a sequence of bytes.
    The opposite operation should just be as simple but...
- Theoretical limitations
  - Complete and fully automated disassemble/decompilation of arbitrary machine-code is theoretically an undecidable problem
- Practical limitations
  - Binaries (also the good ones) can be very surprising

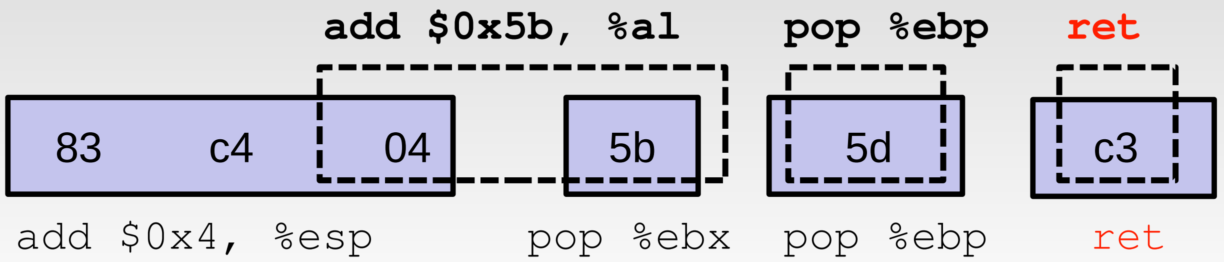
Limitations

- Overlapping instructions – depending from which byte you start disassembling, you can obtain different instructions

83 c4 04	5b	5d	c3
`add $0x4, %esp`	`pop %ebx`	`pop %ebp`	`ret`

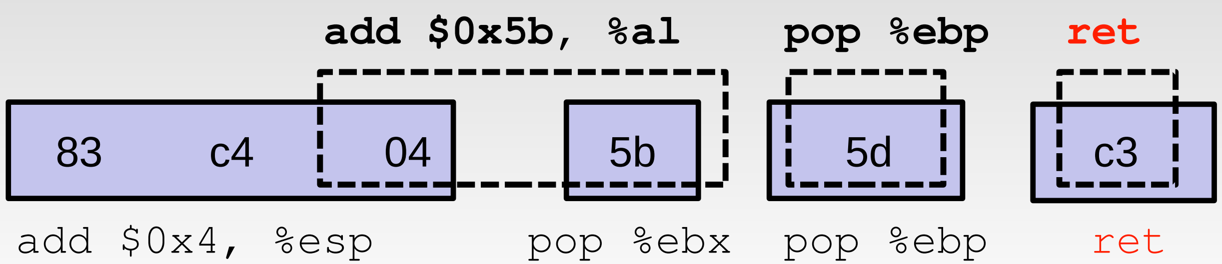
Limitations

- Overlapping instructions – depending from which byte you start disassembling, you can obtain different instructions



Limitations

- Overlapping instructions – depending from which byte you start disassembling, you can obtain different instructions

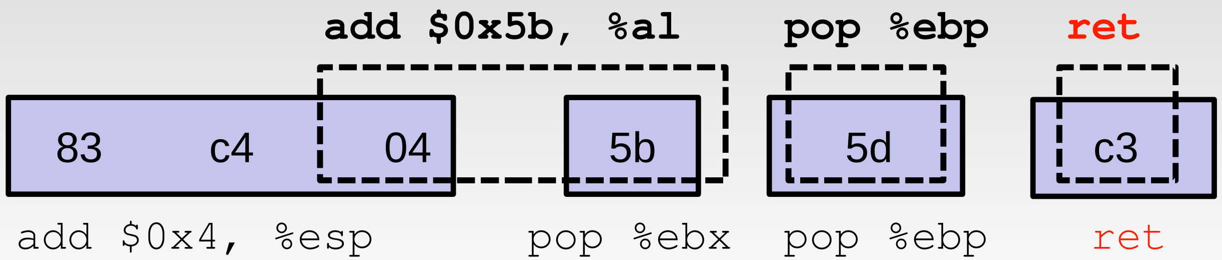


- Recognizing code: data and code are mixed, and it is quite hard to tell what is what

```
0x52 0x40 0x68 0x65
0x61 0x6c 0x74 0x68
0x2e 0x62 0x69 0x7a
```

Limitations

- Overlapping instructions – depending from which byte you start disassembling, you can obtain different instructions



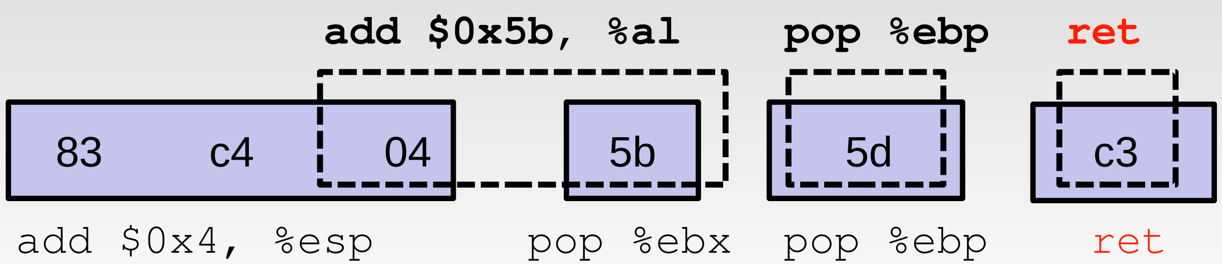
- Recognizing code: data and code are mixed, and it is quite hard to tell what is what

```
0x52 0x40 0x68 0x65
0x61 0x6c 0x74 0x68
0x2e 0x62 0x69 0x7a
```

R@health.biz

Limitations

- Overlapping instructions – depending from which byte you start disassembling, you can obtain different instructions



- Recognizing code: data and code are mixed, and it is quite hard to tell what is what

```
0x52 0x40 0x68 0x65
0x61 0x6c 0x74 0x68
0x2e 0x62 0x69 0x7a
```

R@health.biz

```
push edx
inc eax
push dword 0x746c6165
push dword 0x7a69622e
```

Limitations

- The x86 documentation contains undefined or poorly documented corner cases
  - E.g., the `rep` prefix is undefined when applied to something that is neither a string nor an I/O instruction. `rep dec = ???`

Disass.	Over supported	Not supported *Opc.*	Not supported *Instr.*	Incorrect *Opc.*	Incorrect *Instr.*
diStorm64	10	209	1084	1	1
Ida Pro	461	5	12	49	283
libopcode	331	22	376	105	815
Native Client	479	54	534	133	8232
ndisasm	282	26	388	70	642
OllyDBG	484	136	515	26	176
Udis86	289	4	6	3	4
XED2	44	0	0	12	122

Results from: “*N-version disassembly: differential testing of x86 disassemblers*”, based on 64K byte sequences

Disassembly at Scale

- If properly disassembling a single instruction is hard, disassembling a sequence of instructions (or an entire program) is even harder

- Two main approaches:
  - Linear sweep: disassembles one instruction after the other one, ignoring the control flow
  - Recursive traversal: tries to follow the control flow of the program

Linear Sweep Disassembler

- Disassemble all bytes
  - start at beginning of code (`.text`) section
  - disassemble one instruction after the other
  - assume a well-behaved compiler that tightly packs instructions
  - `objdump, gdb, windbg` use this approach

Linear Sweep Disassembler

- Disassemble all bytes
  - start at beginning of code (`.text`) section
  - disassemble one instruction after the other
  - assume a well-behaved compiler that tightly packs instructions
  - `objdump`, `gdb`, `windbg` use this approach

```
jmp L1
.short 0x4711
L1:
  xor %eax, %eax
```

```
# Correct disassembly
4004cf: eb 02        Jmp 4004d3
4004d1: 11 47        <junk>
4004d3: 31 c0        xor %eax, %eax
```

Linear Sweep Disassembler

- Disassemble all bytes
  - start at beginning of code (`.text`) section
  - disassemble one instruction after the other
  - assume that well-behaved compiler tightly packs instructions
  - `objdump, gdb, windbg` use this approach

```
jmp L1
.short 0x4711
L1:
  xor %eax, %eax
```

```
# Correct disassembly
4004cf: eb 02      Jmp 4004d3
4004d1: 11 47      <junk>
4004d3: 31 c0      xor %eax, %eax
```

```
# As seen by a linear sweep dis.
4004cf: eb 02      jmp 4004d3
4004d1: 11 47 31   adc %eax,0x31(%edi)
4004d3: c0...      sarb ...
```

Recursive Traversal Disassembler

- Disassemble instructions that can be reached from other valid instructions
  - Start at program entry point
  - Disassemble one instruction after the other, until branch or jump is found
  - Recursively follow both (or single) branch (or jump) targets
  - `IDA Pro`, `OllyDbg`, `Radare2` and `HT` use this approach
- Pro:
  - better at dealing with interleaved code and data
- Cons:
  - Does not know what to do with indirect jumps/calls
  - Does not know what to do with unreachable byte sequences (are all data?)

Limitations

- The targets of indirect jumps and calls are hard to compute without running the code

```
pop %eax
call *%eax
```

- Overlapping functions – the code of two functions can overlap (resulting in functions with multiple entry points), due to compiler optimization

- Self-modifying code and anti-disassembly tricks are not so rare in the malware world

  → *more on that later...*

Obfuscated Control Flow

```
4004b7: e8 00 00 00 00  call 4004bc
4004bc: 58              pop %eax
4004bd: 83 c0 08        add $0x8,%eax
4004c0: ff e0           jmp *%eax
4004c2: 11 47           <junk>
4004c4: ...             <next instr>
```

(even more) Obfuscated Control Flow

- Using `call` is not the only way to get the program counter
- Another trick is to use the `fnstenv` instruction to inspect the state of the FPU
  - The resulting structure contains (at offset 12) the address of the last floating point instruction that was executed

```
fldz
fnstenv -12(%esp)
popl %ecx
addb 14, %cl
call *%ecx
```

http://www.securityfocus.com/archive/82/344896/2009-02-24/1

Coverage

Recursive traversal approaches are more accurate but often achieve a low coverage

Properly identified Code

Unidentified Area:
- Strings ?
- Jump tables ?
- Missed Functions ?

Speculative Disassembler

- Recursive traversal approaches often implement speculative techniques to increase the coverage
  - Identify gaps and unreachable areas in the `.text` segment
  - Try to disassembly them and see what happen
    - If the result looks (?) reasonable, assume it is code and continue
- Often requires human intervention to fix errors and identify code

Detecting Functions

- Search sequences of bytes associated to known function prologues
  - Generated by the compiler to initialize the stack frame
  - Works well when the prologue is regular (e.g. in GCC) and not so well otherwise (e.g., with Intel cc)

Linux

```
push ebp
mov ebp, esp
sub esp, X     # [Optional]
```

Detecting Functions

- Search sequences of bytes associated to known function prologues
  - Generated by the compiler to initialize the stack frame
  - Works well when the prologue is regular (e.g. in GCC) and not so well otherwise (e.g., with Intel cc)

Linux

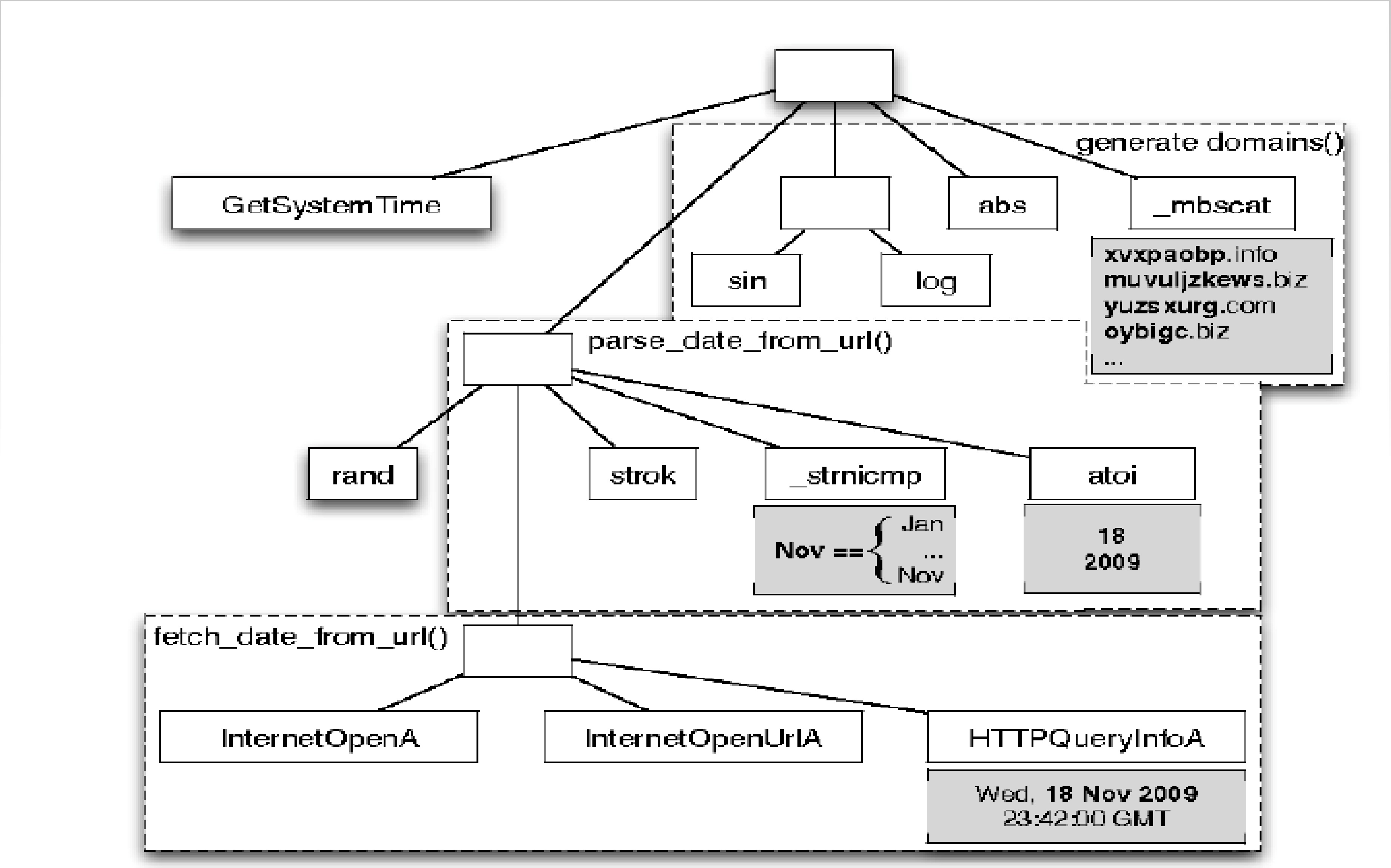
```
push ebp
mov ebp, esp
sub esp, X     # [Optional]
```

Windows

```
nop
nop
nop
nop
nop
mov edi, edi
push ebp
mov ebp, esp
```

`<Function start address>` → `mov edi, edi`

Call Graphs



generate domains()
GetSystemTime
abs
_mbscat
sin
log
xvxpaobp.info
muvuljzkews.biz
yuzsxurg.com
oybigc.biz
...
parse_date_from_url()
rand
strok
_strnicmp
atoi
Nov ==
Jan
...
Nov
18
2009
fetch_date_from_url()
InternetOpenA
InternetOpenUrlA
HTTPQueryInfoA
Wed, 18 Nov 2009
23:42:00 GMT

Decompilation

- Goal: produce high-level code as close as possible to the original source code of the program

- Several limitations
  - Requires a perfect disassembly
  - The compilation process is lossy and many information are lost in the process (variable names, type definitions, …)
  - Complex data structures (whose definition is lost) make the code very hard to read
  - Compilation is a many-to-many operation: there are many ways to translate C to assembly and many to translate assembly back to C
  - Decompilation is often language and compiler specific

- Few options: hexrays (2600 Euros for x86+ARM)

Decompilation

1. Disassembly
2. Lifting and dataflow analysis
   - Translate the assembly to an intermediate language
   - Recognize distinct variables, and detach them from registers or memory addresses
   - Identify function arguments
3. Control flow analysis
   - Recover control flow structure information (e.g., for and while)
4. Type analysis
   - Infer the types for variables and parameters

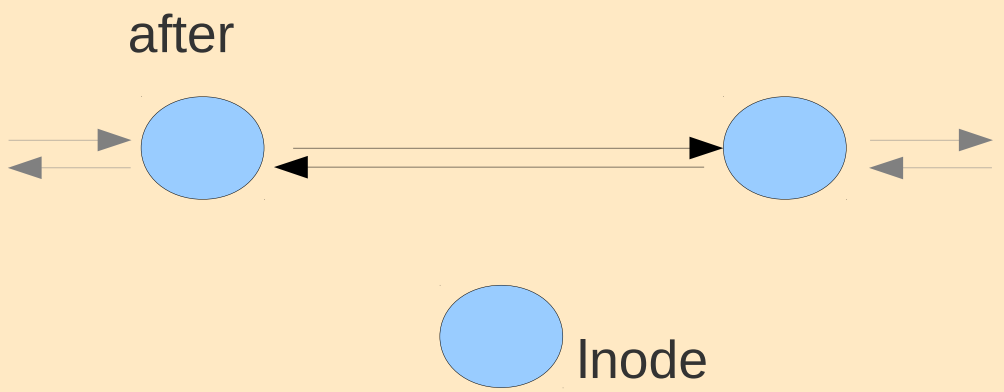
Decompilation

```
struct dllist {
 int number;
 struct dllist *next;
 struct dllist *prev;
};

void insert_node(struct dllist *lnode, struct dllist *after) {
 lnode->next = after->next;
 lnode->prev = after;

 if(after->next != NULL)
  after->next->prev = lnode;
 else
  tail = lnode;

 after->next = lnode;
}
```

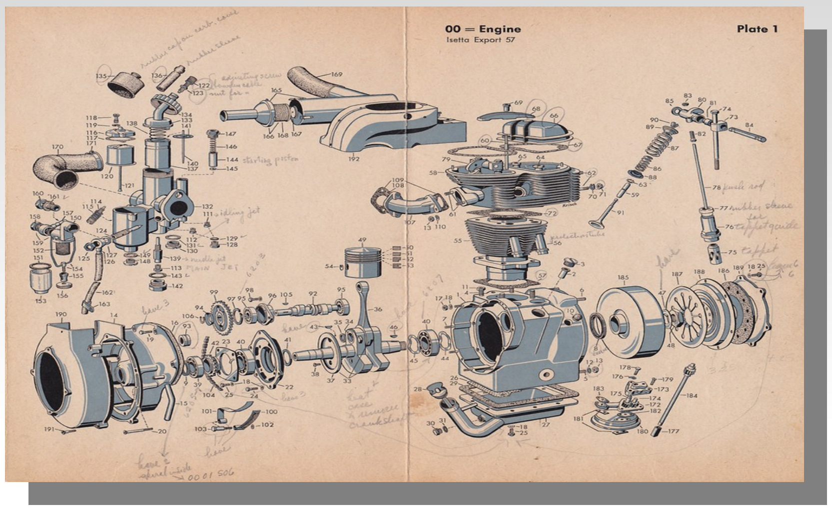


Decompilation

```
int __cdecl insert_node(int a1, int a2)
{
  int result;

  *(_DWORD *)(a1 + 4) = *(_DWORD *)(a2 + 4);
  *(_DWORD *)(a1 + 8) = a2;
  if ( *(_DWORD *)(a2 + 4) )
    *(_DWORD *)(*(_DWORD *)(a2 + 4) + 8) = a1;
  else
    dword_804A028 = a1;

  result = a2;
  *(_DWORD *)(a2 + 4) = a1;
  return result;
}
```



- ✔ Assembly 101
- ✔ Disassembly algorithms
  - → Linear sweep and recursive
  - → Detecting function prologues
- ✔ Decompilation
- Language Constructs
  - → Assembly and C
  - → Assembly and C++
- Limit of Static Analysis
  - → General limitations
  - → Anti-disassembly
  - → Packing

C Disassembly

Locating Main *(the Linux/gcc way)*

- Isn't `main` the application entry point?
  - Not really, a number of setup and initialization tasks must be done before transferring the control to the main function
  - The function located at the entry point is typically called `_start` (and it is the only one you can locate in a stripped binary)
  - The name "*main*" is just a compiler convention

- In a dynamiccally linked binary, `_start` is a `glibc` function that is statically linked to every executable and placed at the beginning of the `.text` segment
  - The only purpose of `_start` is to setup the arguments and call `__libc_start_main`
  - It is possible to build an executable that does not use the `_start` function

_start

```
public _start
xor     ebp, ebp
pop     esi
mov     ecx, esp
and     esp, 0FFFFFFF0h
push    eax
push    esp
push    edx
push    offset __libc_csu_fini
push    offset __libc_csu_init
push    ecx
push    esi
push    offset main
call    ___libc_start_main
hlt
_start endp
```

_start

```
__libc_start_main(int (*main) (int, char**, char**),
   int argc,
   char *__unbounded *__unbounded ubp_av,
   void (*init) (void),
   void (*fini) (void),
   void (*rtld_fini) (void),
   void (*__unbounded stack_end));
```

```
push    offset __libc_csu_fini
push    offset __libc_csu_init
push    ecx
push    esi
push    offset main
call    ___libc_start_main
hlt
_start endp
```

__libc_start_main

- Initialize the thread local storage
- Set up the thread stack guard
- Initialize the `glibc` inself by calling `__libc_init_first`
- Register `__libc_csu_fini`
- Call `__libc_csu_init`
  - Call function pointers in `.preinit_array` section
  - Execute the code in `.init` section, which is usually the `_init` function
    - In GCC, `_init` executes user functions marked as `__attribute__ ((constructor))`
  - Call function pointers in `.init_array` section
  - Call a function responsible to call the constructors for global objects in C++
- Call `main`
- Call `exit`

Initialization Functions

- The linker creates an array of pointers to initialization functions called `__CTOR_LIST__`
  - The first entry is ignored
  - The list is terminated by a null pointer (four `\x00`)
- The same structure applies for termination functions (`__DTOR_LIST__`)

```
.ctors:08049EE0 __CTOR_LIST__
.ctors:08049EE0 dd 0FFFFFFFFh
.ctors:08049EE4 dd 08048484
.ctors:08049EE8 dd 0
```

Locating Main *(the Windows way)*

- The entry point of the PE file normally points to the beginning of the code section
- The first code to be executed is typically a function from the C runtime library (CRT)
  - `mainCRTStartup` – default for console applications
    - Normally calls `GetVersion` and `GetCommandLineA`
    - Calls main
  - `WinMainCRTStartup` – default for GUI applications
    - Normally starts by calling `GetVersion` and `GetModuleHandleA`
    - Calls WinMain

Variables

- Global variables
  - Stored in the `.bss` section (if uninitialized) or in the `.data` section (if initialized)
  - Accessed with their absolute memory address
- Local variables
  - Stored on the stack
  - Accessed using a relative offset from the base pointer (EBP) or sometimes from the current stack pointer (ESP)

```
mov     ds:dword_804A024, 3   ; global variable
mov     [ebp-4], 5            ; local variable
```

IF Statements

- `test` or cmp instruction followed by a conditional jump
  - `test`

    An `and` operation that does not store the result (but update the flags)

```
test    eax, eax           ; set ZF if the result is zero
jz      short loc_80484F2 ; jump if ZF
```

  - `cmp`

    Equivalent to a `sub` that does not store the result

```
cmp   ecx, edx           ; set ZF if the result is zero
je    short loc_80484F2 ; jump if ZF
```

  - There are over 30 conditional jump instructions (`jl, jne, jg, ...`)

FOR Loops

```
# for(i=10; i>0; i--){ … }

080484EB        jmp     short loc_8048505    ; start the loop
080484ED        . . .                        ; loop body
08048501        sub     [ebp+var_C], 1       ; increment
08048505        cmp     [ebp+var_C], 0       ; test
08048509        jg      short loc_80484ED
0804850B        . . .                        ; after the loop
```

Arrays

- Often accessed by using an offset from the base address
  - Very common inside loops

```
# a[i] = i
mov     [ebp+eax*4-64], eax    ; ebp-64 is the base of the array
                               ; eax is the index
                               ; 4 is the size of the elements
```

- Direct access to a particular element use the offset of the element itself
  - Very hard to distinguish from a local variable :(

```
# a[7] = 0
mov     [ebp-36], 0    ; ebp-36 is the address of a[7]
```

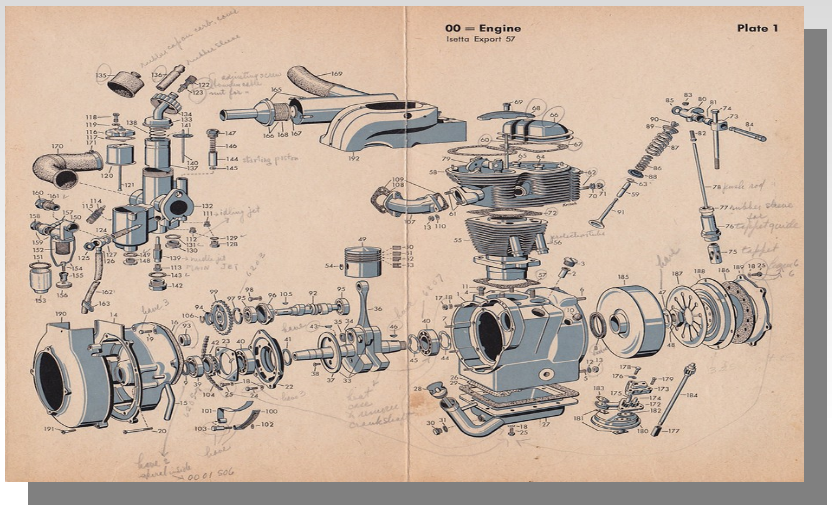
Structures

- Dynamically allocated structures are usually accessed using an offset from the base of the structure
    - The size of the structure can be extracted by monitoring `malloc`

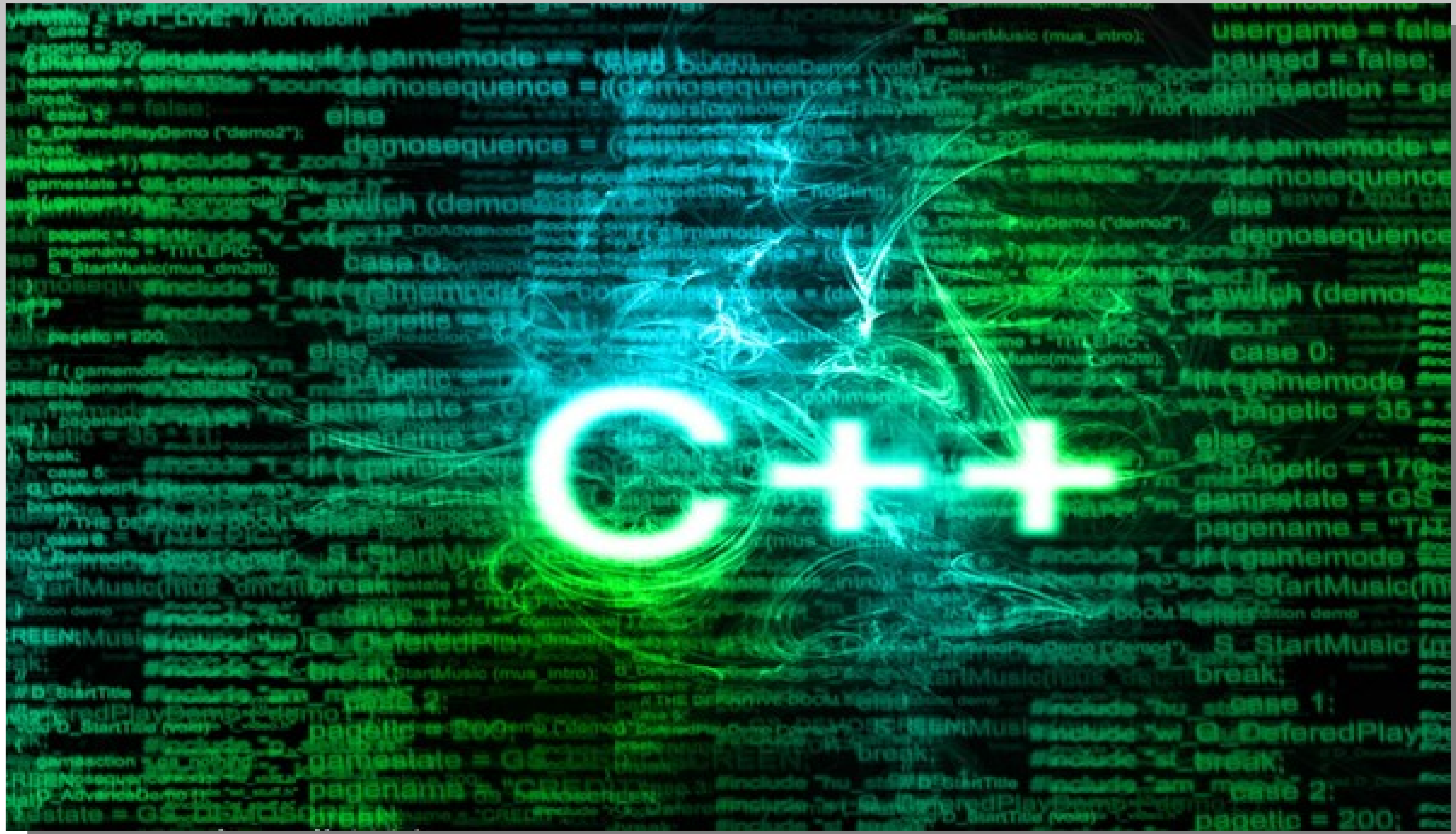
```
# struct {
#  int val1;
#  int val2;
#  float f; }

# s->val2 = 22
mov     dword ptr [eax+4], 16h  ; eax is the pointer to the struct
# s->f = 0.3
fld     ds:dbl_8048760
fstp    qword ptr [eax+8]
```

- Static structures on the stack are basically equivalent to a list of local variables :(



- ✔ Assembly 101
- ✔ Disassembly algorithms
  - → Linear sweep and recursive
  - → Detecting function prologues
- ✔ Decompilation
- ✔ Language Constructs
  - ✔ Assembly and C
  - → Assembly and C++
- ▪ Limit of Static Analysis
  - → General limitations
  - → Anti-disassembly
  - → Packing



C++ Disassembly

- Object creation
- Name mangling
- Methods
- Dynamic dispatching
- VTables
- Exception handling

C++

- Largely used to develop programs (including malware)
- The language definition does not specify how the language functionalities should be implemented in assembly
  - Classes, class relationships, methods, exceptions, …
- Custom compiler-specific implementations makes more difficult to reverse engineering C++ code
  - Name Mangling
  - Dynamic Dispatching
  - Exception Handling

Object Creation

- Statically allocated
  - Placed on the stack
  - The class constructor is called at declaration
  - The destructor is called when exiting the scope (e.g., end of function)

Object Creation

- Statically allocated
  - Placed on the stack
  - The class constructor is called at declaration
  - The destructor is called when exiting the scope (e.g., end of function)
- Dynamically allocated
  - Allocated on the heap
  - Created by invoking the new operator
    - Allocate the memory and call the constructor
  - Destroyed by calling the delete operator
    - Call the destructor and free the memory

Name Mangling

- Encoding and decoration of symbols names
- Used to
  - pass more information from the compiler to the linker
  - change names that could not be understood by the linker otherwise (e.g., in case of function overloading)
- Each compiler uses its own mangling scheme
  - E.g., `void h(int, char)` becomes:

    `_Z1hic` in GCC 4.0

    `?h@@YAXHD@Z` in Microsoft Visual C++

    `@h$qizc` in Borland C++
- To demangle names you can use the `c++filt` command line tool

- ✔ Object creation
- ✔ Name mangling
- Methods
- Dynamic dispatching
- VTables
- Exception handling

Class Methods

- Methods works on a class instance (i.e., an object)
  - A pointer to *this* is passed as a hidden first parameter
  - MSVC by default use the `thiscal` calling convention (passing the *this* pointer in the `ecx` register)
  - GCC (g++) pushes the *this* pointer to the stack as a first argument
- Constructors / Destructors work like normal methods

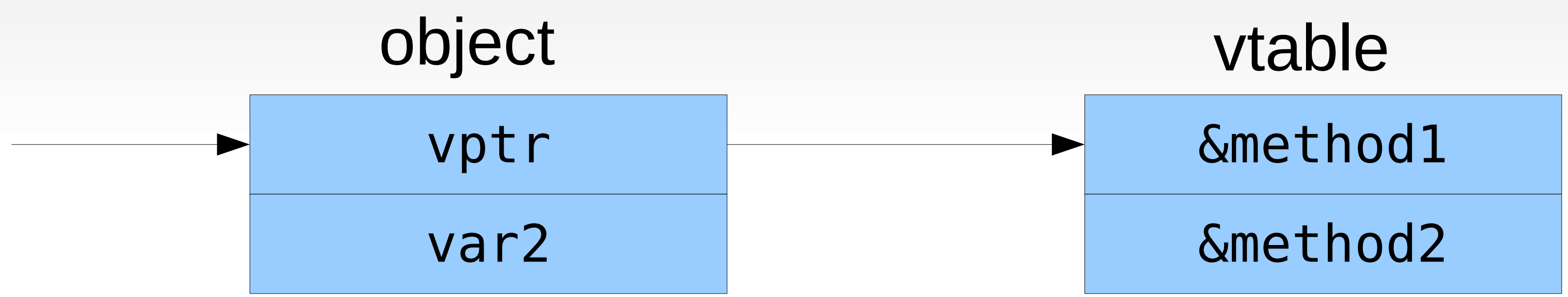
Class Methods

- Virtual and non-Virtual Methods
  - For non-virtual methods, the function to execute is determined at compile time
    - In assembly you see a normal `call` instruction to the function
  - For virtual methods, the function can only be determined at runtime
    - In assembly it needs to be implemented with an indirect call (dynamic dispatch)

Dynamic dispatch

- Normally implemented by using a Virtual Table (`vtable`)
  - The compiler create a separate `vtable` for each class that uses virtual functions (or that is derived from one that does)
  - The `vtable` contains an array of pointers to the virtual methods
  - A pointer to the `vtable` (`vptr`) is added as a first member of the class object
- The class constructor initialize the `vptr` of its objects to the address of the corresponding `vtable`

```
class MyClass3: public MyClass2{
protected:
 int var2;
public:
  MyClass3(int x){ var2 = x; }
  virtual int method1(int x) {return x;}
  virtual int method2(int x) {return x+1;}
}
```



```cpp
class MyClass3: public MyClass2{
protected:
 int var2;
public:
  MyClass3(int x){ var2 = x; }
  virtual int method1(int x) {return x;}
  virtual int method2(int x) {return x+1;}
}
```

```asm
# Constructor MyClass3(int x);
push    ebp
mov     ebp, esp
sub     esp, 18h
...
```

```cpp
class MyClass3: public MyClass2{
protected:
 int var2;
public:
  MyClass3(int x){ var2 = x; }
  virtual int method1(int x) {return x;}
  virtual int method2(int x) {return x+1;}
}
```

```asm
# Constructor MyClass3(int x);
push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, [ebp+arg_0]   ; first arg is this
mov     [esp], eax         ; put this on the stack
call    _ZN8MyClass2C2Ev   ; call constructor of
                             the base class
...
```

```cpp
class MyClass3: public MyClass2{
protected:
 int var2;
public:
  MyClass3(int x){ var2 = x; }
  virtual int method1(int x) {return x;}
  virtual int method2(int x) {return x+1;}
}
```

```asm
# Constructor MyClass3(int x);
push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, [ebp+arg_0]   ; first arg is this
mov     [esp], eax         ; put this on the stack
call    _ZN8MyClass2C2Ev   ; call constructor of
                             the base class
mov     eax, [ebp+arg_0]
mov     dword ptr [eax], 80488D8h ; put vptr as
                                    first field
...
```

```cpp
class MyClass3: public MyClass2{
protected:
 int var2;
public:
  MyClass3(int x){ var2 = x; }
  virtual int method1(int x) {return x;}
  virtual int method2(int x) {return x+1;}
}
```

```
# Constructor MyClass3(int x);
push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, [ebp+arg_0]   ; first arg is this
mov     [esp], eax         ; put this on the stack
call    _ZN8MyClass2C2Ev   ; call constructor of
                             the base class
mov     eax, [ebp+arg_0]
mov     dword ptr [eax], 80488D8h ; put vptr as
                                    first field
mov     eax, [ebp+arg_0]
mov     edx, [ebp+arg_4]   ; second arg is x
mov     [eax+4], edx       ; copy x in var2
leave
retn
```

```cpp
class MyClass3: public MyClass2{
protected:
 int var2;
public:
  MyClass3(int x){ var2 = x; }
  virtual int method1(int x) {return x;}
  virtual int method2(int x) {return x+1;}
}
```

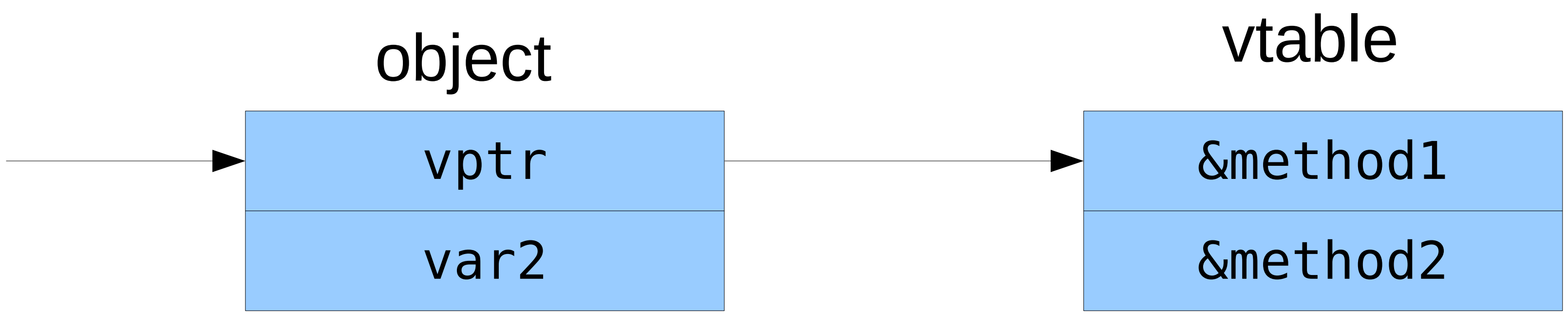
```asm
# eax = Object
# object->method2(8)

mov     edx, [eax]                ; vptr in edx
mov     dword ptr [esp+4], 8  ; push 8 on the stack
mov     [esp], eax                ; push this on the stack
call    [edx+4]                   ; call 2nd entry in
                                    the vtable
```

- ✔ Object creation
- ✔ Name mangling
- ✔ Methods
- ✔ Dynamic dispatching
- VTables
- Exception handling

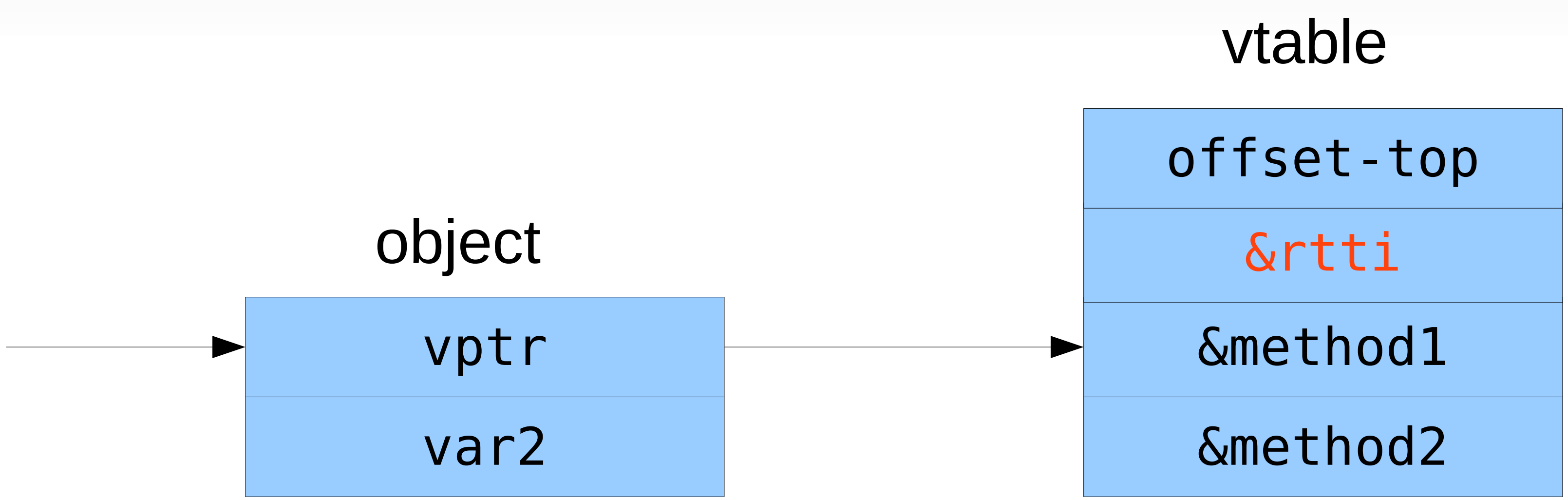
vtable internals

- The `vtable` contains more than just the pointers to the virtual methods



vtable internals

- The `vtable` contains more than just the pointers to the virtual methods
  - offset-to-top (used in multiple inheritance for nested `vtables`)
  - pointer to the `typeinfo struct`
  - pointer to virtual method 1 (`vptr` points here)
  - ...

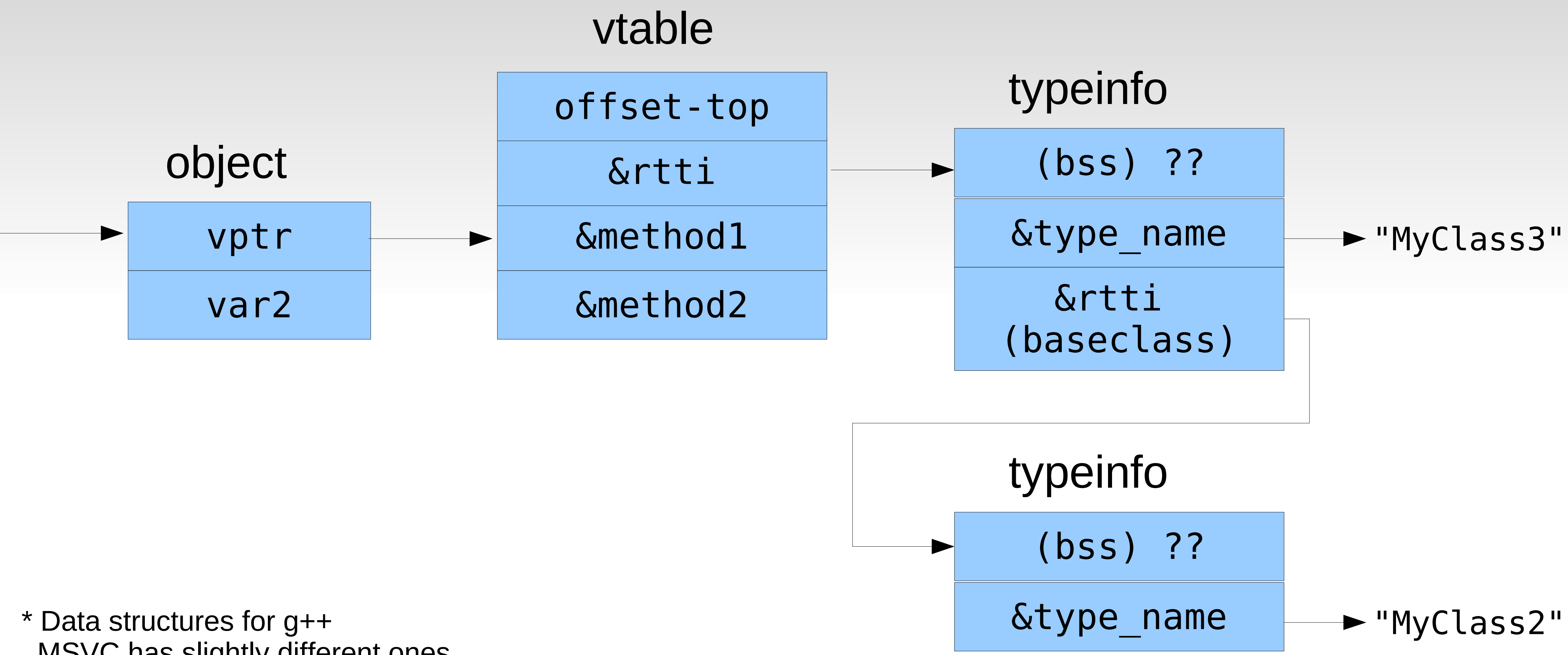


Run-Time Type Identification (RTTI)

- Used to identify objects at runtime
- Special information used to support `dynamic_cast` or operators like `typeid()`
  - Generated by the compiler for each class with virtual functions
  - Can be explicitly suppressed at compile-time if not needed
- Stored in a couple of data structures, reachable from the `vtable`
- Contains extremely useful information for reverse engineering
  - Original name of the class
  - Class hierarchy

    → This information is available also in stripped binaries !!

The Global Picture*



* Data structures for g++
MSVC has slightly different ones
See http://www.openrce.org/articles/full_view/23

Exception Handling

- The C++ language exception semantics are implemented by calling a set of standard library routines

`___cxa_allocate_exception` : allocates the exception object

`___cxa_throw` : throws the exception (this function never returns). Implement the logic to walk back the stack, looking for a suitable exception handler

`___cxa_begin_catch` : marks the beginning of an catch block

`___cxa_end_catch` : marks the end of the catch block. If the program is not terminated, this call is followed by a jump to the instruction following the catch block

`_unwind_resume` : called to resume the lookup process for an exception handler

`___gxx_personality_v0` : language-specific personality function call by the unwinder to find an appropriate exception handler

```
int test(int a){
    if (a > 2)
       throw a;
    return a-2;}

int main () {
  printf("Starting\n");
  try{
    test(3);
  }
  catch (int e){
    printf("An integer exception occurred: %d\n",e);
  }
  catch (float x){
    printf("A float exception occurred:. %f\n",x);
  }
  printf("I'm done\n");
  return 0;
}
```

```
int test(int a){
    if (a > 2)
       throw a;
    return a-2;}

int main () {
  printf("Starting\n");
  try{
    test(3)
  }
  catch (in
    printf(
  }
  catch (fl
    printf("A float exception occurred!: %f\n",x);
  }
  printf("I'm done\n");
  return 0;
}
```

```
call    ___cxa_allocate_exception
...
mov     dword ptr [esp+8], 0
mov     dword ptr [esp+4], offset _ZTIi@@CXXABI_1_3
mov     [esp], eax
call    ___cxa_throw
```

```
cmov   dword ptr [esp], offset s  ; "Starting"
call   _puts
mov    dword ptr [esp], 3
call   _Z4testi                     ; test(int)
mov    dword ptr [esp], offset aIMDone ; "I'm done"
call   _puts
```

?

```
int test(i
    if (a
       thr
    return

int main () {
  printf("Starting\n");
  try{
    test(3);
  }
  catch (int e){
    printf("An integer exception occurred: %d\n",e);
  }
  catch (float x){
    printf("A float exception occurred:. %f\n",x);
  }
  printf("I'm done\n");
  return 0;
}
```

```
call    _Z4testi                            ; test(int)
mov     dword ptr [esp], offset aIMDone ; "I'm done"
call    _puts
mov     eax, 0
mov     ebx, [ebp+var_4]
leave
retn                                        ; end of main
...
```

```
call    _Z4testi                          ; test(int)
mov     dword ptr [esp], offset aIMDone ; "I'm done"
call    _puts
mov     eax, 0
mov     ebx, [ebp+var_4]
leave
retn                                      ; end of main
...
cmp     edx, 1                            ; exception landing pad
jz      short loc_80486FD
cmp     edx, 2
jz      short loc_8048726
mov     [esp], eax
call    __Unwind_Resume
...
mov     [esp], eax
call    ___cxa_begin_catch
mov     eax, [eax]
mov     [esp+18h], eax
mov     eax, [esp+18h]
mov     [esp+4], eax
mov     dword ptr [esp], offset aAnIntegerExcep ;
call    _printf
call    ___cxa_end_catch
jmp     short loc_80486B3
```

Finding the Landing Pad

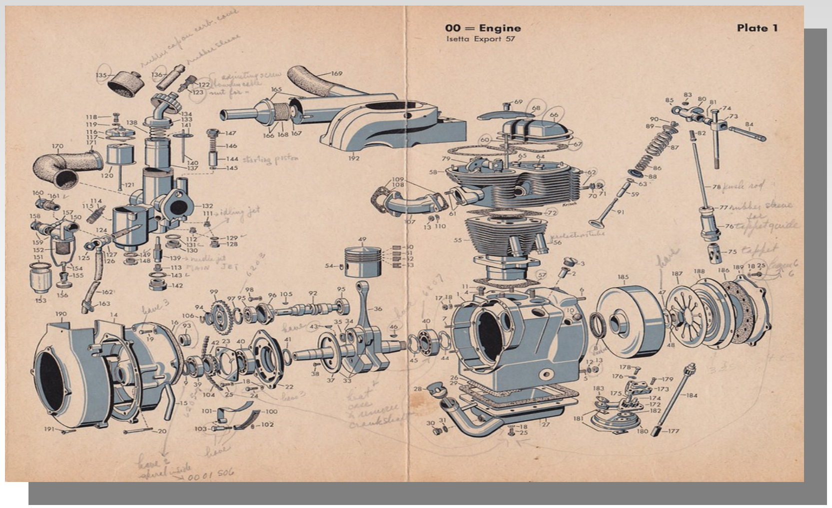
- Poorly documented until the first tool was presented at RECON 2012 (http://recon.cx/2012/schedule/events/247.en.html)
- When `gcc` generates code to handle exceptions, it produces three tables that describe how to unwind the stack and find the landing pads
- These tables are stored in three separate read-only sections:

  `.eh_frame` – main structure that describes how to unwind the stack. It contains a pointer to the personality function and one to a Language Specific Data Area (LSDA)

  `.eh_frame_hdr` – header for the frame section, used to locate the right entry in the table for each given EIP value

  `.gcc_except_table` – the LSDA

More info: http://www.airs.com/blog/archives/460
http://www.airs.com/blog/archives/462
http://www.airs.com/blog/archives/464

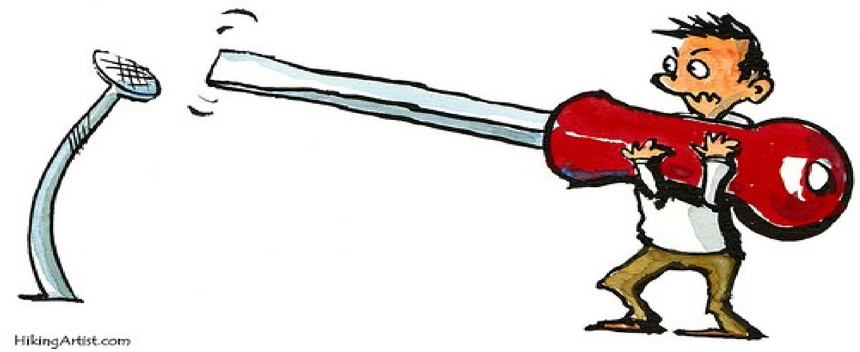


- ✔ Assembly 101
- ✔ Disassembly algorithms
  - → Linear sweep and recursive
  - → Detecting function prologues
- ✔ Decompilation
- ✔ Language Constructs
  - ✔ Assembly and C
  - ✔ Assembly and C++
- ■ Limits of Static Analysis
  - → General limitations
  - → Anti-disassembly
  - → Packing

Limits of Static Analysis

Limitations

- Limits of static analysis
  - Packing
  - Indirect jump prediction and obfuscated addresses
- Limits of the disassembly algorithm
  - Disassemblers rely on a number of assumptions and heuristics to separate code and data
  - For each decision the disassembler has to make, it is possible to write a counter-techniques that undermine the process



Overlapping Instructions

- Certain sequences of instructions do not have a unique interpretation

```
             EB
jmp   -1     FF
                   inc   eax
             C0
             48    dec   eax
```

- The `0xFF` byte is part of two consecutive instructions
  - There is no way to disassemble the bytes so that all the executed instructions are represented
  - The only way out is to replace some of the instructions with NOPs

Overlapping Instructions

```
66 B8 EB 05     mov ax, 0x05eb
31 C0           xor eax, eax
EB FA           jump -6
FF
xx xx
```

Overlapping Instructions

```
66 B8 EB 05     mov ax, 0x05eb
31 C0           xor eax, eax
EB FA           jump -6
FF
xx xx
```

Overlapping Instructions

```
              JMP +5
66 B8 EB 05         mov ax, 0x05eb
31 C0               xor eax, eax
EB FA               jump -6
FF                  <Other code>
xx xx               <Real code>
```

Other Anti-Disassembly Tricks

- Fake conditional jumps
  - Using a conditional jump with a condition that is always true
  - Using a set of conditional jumps to represent an unconditional jump

```
74 0F   JZ +15
75 0D   JNZ +13
...
<JUNK>
...
```

- Playing with the stack pointer
  - Conditionally modifying the stack pointer (ESP) in a function can mess up with many automatic analysis

Other Anti-Disassembly Tricks

- Playing with the `ret` instruction
  - `ret` pops a value from the stack and jump to it, so it could be used instead of a `jmp`
  - Functions can modify their own return address before returning

```
  CALL xxxx

loc_xxxx:
  pop  EBP
  inc  EBP
  push EBP
  retn
```

Solutions

- The binary works just fine, so normally there is no need to touch it
  - Unless you want to help the debugger's disassembler as well

- The disassembler is confused, and it requires some external help to better understand the code
  1. NOP out junk data
  2. Remove some instructions
  3. Connect pieces together with unconditional jumps
  4. Help IDA by adding missing references

Solutions

- The binary works just fine, so normally there is no need to touch it
  - Unless you want to help the debugger's disassembler as well

- The disassembler is confused, and it requires some external help to better understand the code
  1. NOP out junk data `0x90 NOP`
  2. Remove some instructions `0x90 NOP`
  3. Connect pieces together with unconditional jumps `0xEB?? JMP +??`
  4. Help IDA by adding missing references `AddCodeXref()`

Patching

- The easy way
  - IDA6.2 → edit → patch program → apply
  - radare → w

Patching

- The easy way
  - IDA6.2 → edit → patch program → apply
  - radare → w
- The old school way
  - hexedit

Patching

- The easy way
  - IDA6.2 → edit → patch program → apply
  - radare → w
- The old school way
  - hexedit
- The leet way

```
> echo -n -e "\x90\x90\x90" |
  dd seek=100 bs=1 count=3  conv=notrunc of=file
```

Packers

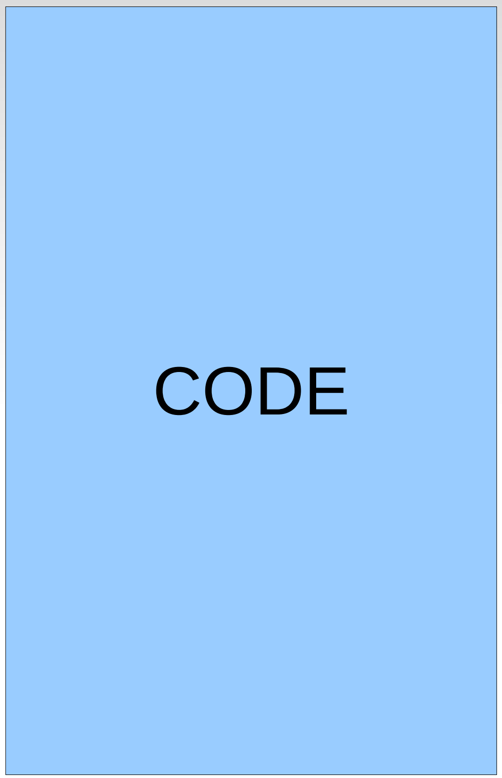
- Goal: compress or encrypt the instructions and data of a program
  - Originally designed to save disk space
  - The new executable decompress in memory the original program and then it jumps into it

Packers

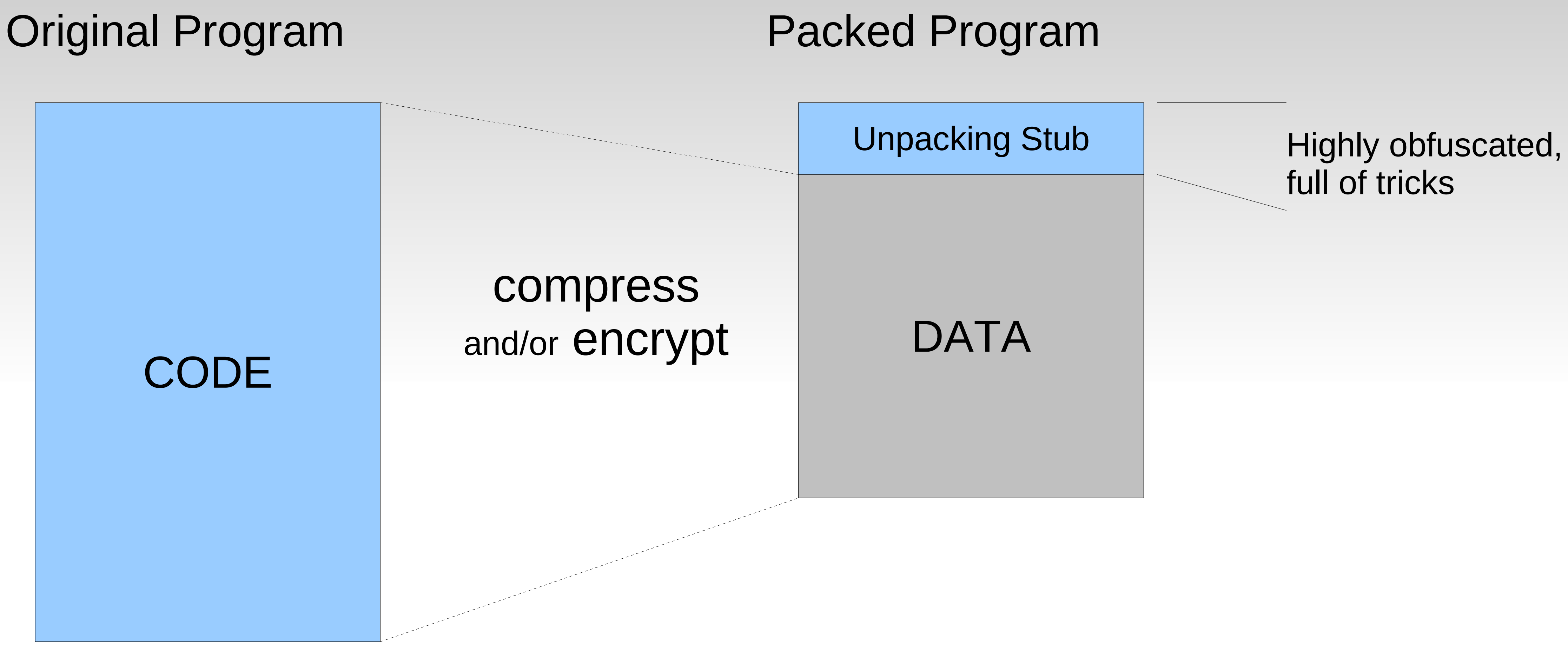
- Goal: compress or encrypt the instructions and data of a program
  - Originally designed to save disk space
  - The new executable decompress in memory the original program and then it jumps into it
- Very useful for malware writers
  - The code must be decrypted before static analysis can be applied
  - Changing the encryption key produces a completely different executable (polymorphism)
  - Many packers automatically include anti- (disassebly, debugging, VM) techniques to further complicate the analysis

Packers Overview

Original Program



Packers Overview



The Unpacking Stub

1. Retrieves, decompresses, and/or decrypts the original code
   - The packed code is often stored in weird PE sections

The Unpacking Stub

1. Retrieves, decompresses, and/or decrypts the original code
   - The packed code is often stored in weird PE sections
2. Resolves the imports of the executable
   - If the import table is packed, the loader cannot resolve the imports and load the corresponding DLLs
   - Option1: the stub only imports `LoadLibrary` and `GetProcAddress` and uses them to manually perform the linking after unpacking the IAT
   - Option2: it does not import anything and uses shellcode-like techniques to located the basic functions to use to resolve all the other imports

The Unpacking Stub

1. Retrieves, decompresses, and/or decrypts the original code
   - The packed code is often stored in weird PE sections
2. Resolves the imports of the executable
   - If the import table is packed, the loader cannot resolve the imports and load the corresponding DLLs
   - Option1: the stub only imports `LoadLibrary` and `GetProcAddress` and uses them to manually perform the linking after unpacking the IAT
   - Option2: it does not import anything and uses shellcode-like techniques to located the basic functions to use to resolve all the other imports
3. Transfers back the control to the Original Entry Point (OEP)
   - The stub usually ends with a tail jump: a control flow instruction (`jump, call, ret`, ..) that transfer the execution to the unpacked code

Other Packing Techniques

- Recursive layers
  - Multiple stages of packing applied on top of each other
  - Force the analyst to locate multiple tail jumps
- Interleaved packing
  - There is not a clear tail jump anymore, but the code of the packer is interleaved with the code of the original application

Other Packing Techniques

- Partial unpacking
  - Unpack only one function/page/block/... at the time
  - Sometimes it re-pack each piece of code after it is executed
  - The entire executable is never in memory
- Emulation
  - The packer translates the original instructions into a bytecode that uses a randomly-generated instruction set
  - The stub includes an interpreter for the bytecode

Packer Identification

- Signatures
  - There are more than 100 families of packers, with many existing variations for each family

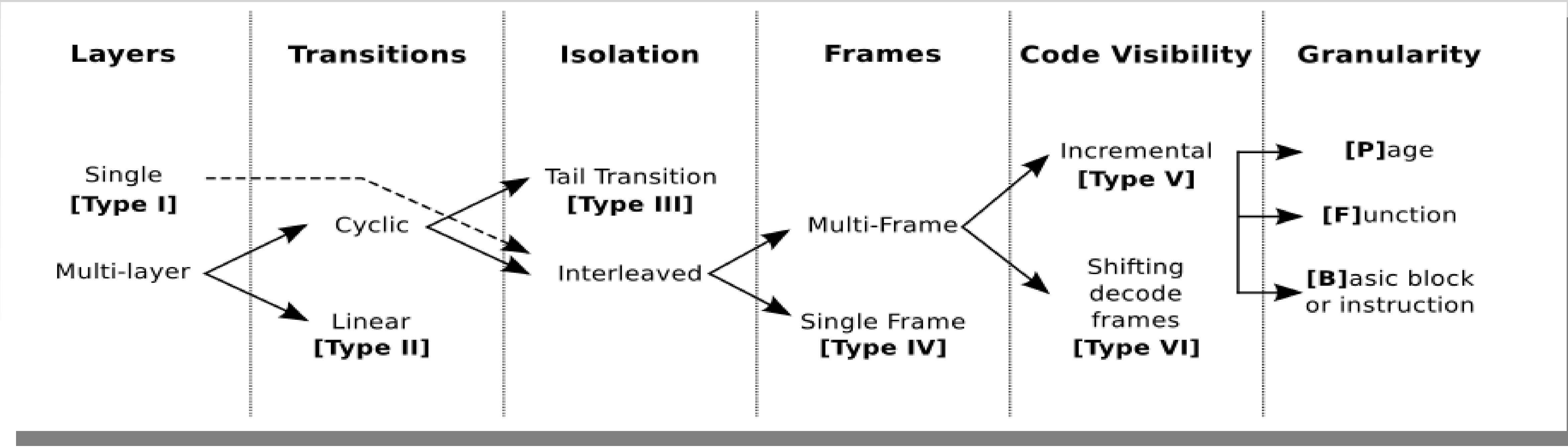
- Heuristics
  - Very few functions
  - Few entries in the import table
  - Missing or compressed string table
  - Very small code
  - Code section that requires more space in memory than on disk
  - Weird sections names
  - Sections with very high entropy

Packer Identification

- Sigbuster
  - Used in the Anubis system
  - Available only to malware researchers and law enforcement
- PeiD
  - Support and development stopped 3 year ago :(
  - The signatures file (`UserDB.txt`) can still be found on other websites
  - The PeID signatures are supported by `pefile`

```
import peutils
sig = peutils.SignatureDatabase('UserDB.TXT')
matches = sig.match_all(pe, ep_only = True)
```

A Packer Taxonomy



Distribution on the Wild

Type	Off-the-shelf	Custom
Type I	25%	7%
Type II	8%	12%
Type III	51%	66%
Type IV	13%	14%
Type V	1%	1%
Type VI	2%	0.2%

Unpacking

- Automatic, Static
  - Possible in very few cases (e.g., UPX) that are not security-oriented

Unpacking

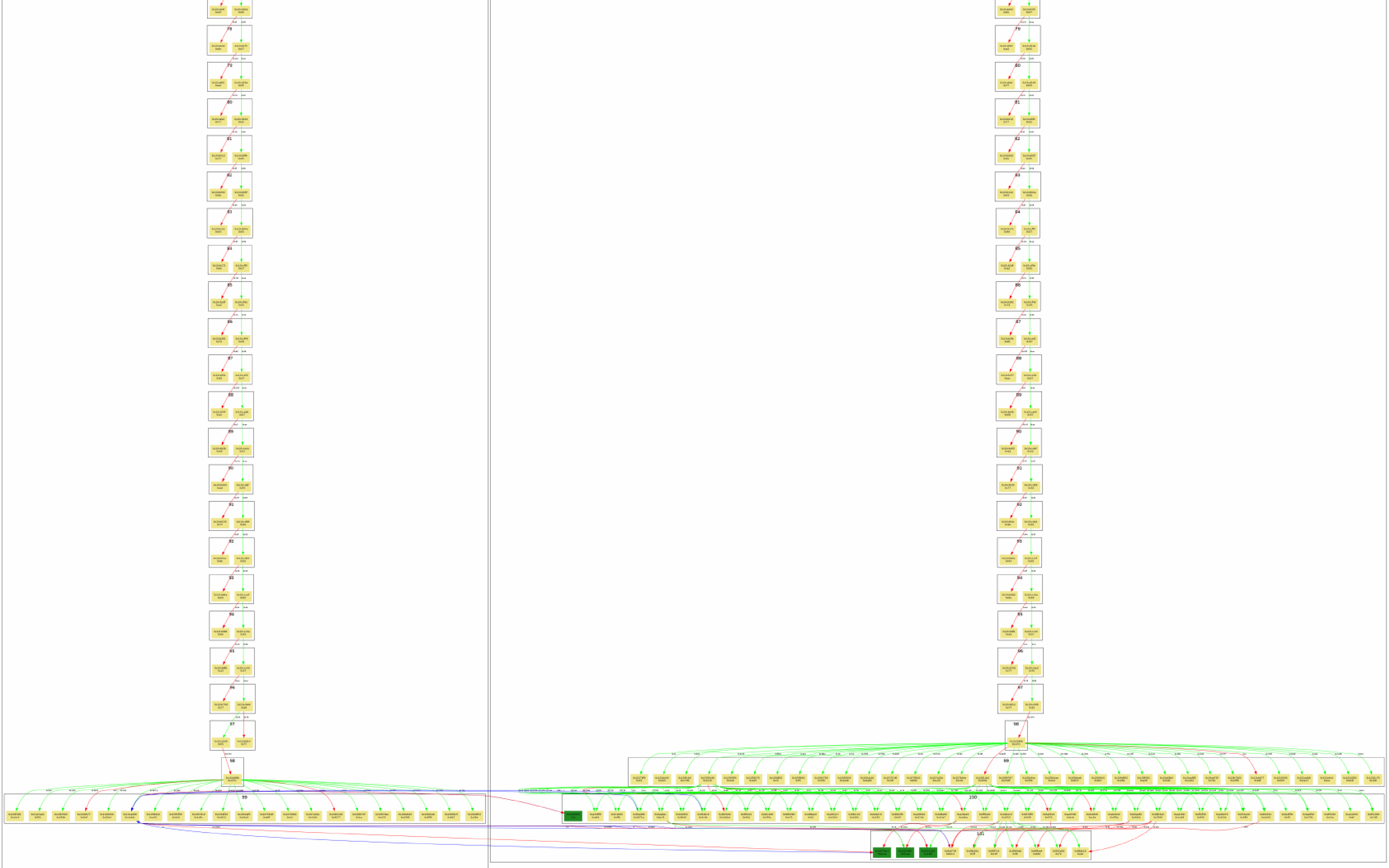
- Automatic, Static
  - Possible in very few cases (e.g., UPX) that are not security-oriented
- Automatic, Dynamic
  - Heuristics to detect the OEP
  - Dump the memory containing the unpacked binary
  - Works well for simple/medium packers

Unpacking

- Automatic, Static
  - Possible in very few cases (e.g., UPX) that are not security-oriented
- Automatic, Dynamic
  - Heuristics to detect the OEP
  - Dump the memory containing the unpacked binary
  - Works well for simple/medium packers
- Manual, Static
  - Requires the analyst to reverse the stub and write a tool that applies the opposite technique
  - Hard and time consuming

Unpacking

- Automatic, Static
  - Possible in very few cases (e.g., UPX) that are not security-oriented
- Automatic, Dynamic
  - Heuristics to detect the OEP
  - Dump the memory containing the unpacked binary
  - Works well for simple/medium packers
- Manual, Static
  - Requires the analyst to reverse the stub and write a tool that applies the opposite technique
  - Hard and time consuming
- Manual, Dynamic
  - Use a debugger to manually identify the OEP and dump the memory



The Research Corner

- [SOK] Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers
  X. Ugarte-Pedrero, Davide Balzarotti, Igor Santos, Pablo G. Bringas
- Binary-Code Obfuscations in Prevalent Packer Tools
  Kevin A. Roundy and Barton P. Miller - ACM Computing Surveys 2012
- Static Disassembly of Obfuscated Binaries
  C Kruegel, W Robertson, F Valeur and G Vigna - 2004 USENIX Security Symposium
- Disassembly of executable code revisited
  B Schwarz, S Debray, G Andrews - Conference on Reverse Engineering 2002
- A taxonomy of self-modifying code for obfuscation
  N Mavrogiannopoulos, N Kisserli, B Preneel - Elsevier Computers & Security 2011
- Labeling library functions in stripped binaries
  ER Jacobson, N Rosenblum, BP Miller - Program analysis for software tools 2011
- The provenance hierarchy of computer programs
  N Rosenblum – Ph.D. Dissertation - 2011