# Static and Dynamic Code Analysis from Malware Aspects

**Gábor Pék**
**pek@crysys.hu**
Laboratory of Cryptography and System Security (CrySyS)
Budapest University of Technology and Economics
www.crysys.hu

# Outline

- Taxonomy of malware
- Malware analysis
  - Static
    - Linear sweep
    - Recursive traversal
    - Speculative disassembly
  - Dynamic
    - Weaknesses of current approaches
    - Malware analysis platforms
    - Introduction to hardware assisted virtualization
    - Out-of-the-guest malware analyzers

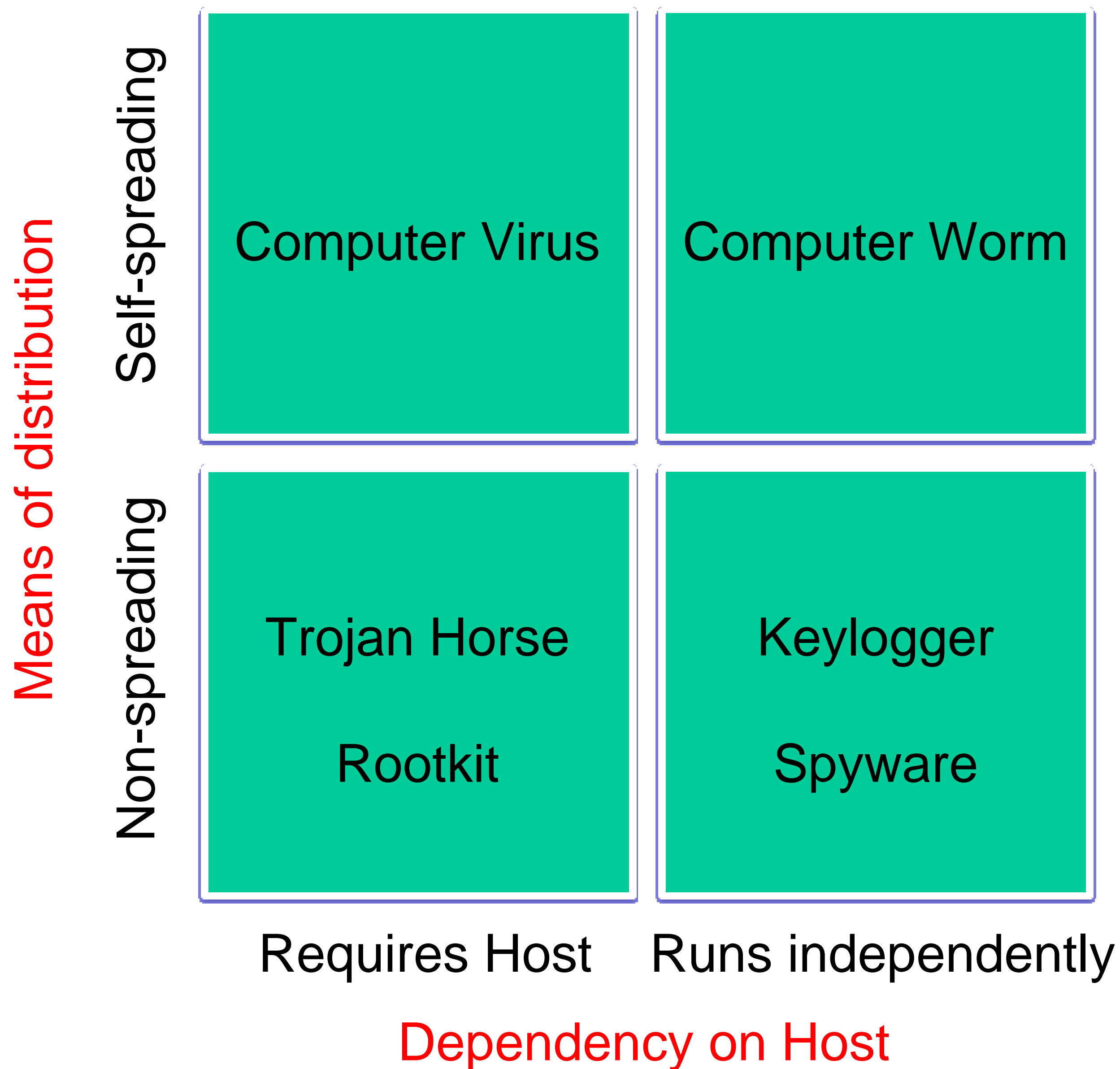
# Malware in General

- Malware is the abbreviation of malicious software
- In short: software to infiltrate or damage a computer system without the informed consent of its owner.
- Comprises many overlapping terms
  - ***Virus**:* Code that replicates itself and uses host file infection
  - ***Worm**:* Network virus, however, a standalone, autoexecutable application
  - ***Germ**:* Virus after first compilation without host file and encryption
  - ***Dropper**:* Installs germs
  - ***Trojan horse**:* Deceives users. Typically backdoors, password stealers
  - ***Exploits**:* Code for special vulnerabilities
  - ***Rootkit**:* Gaining root access over a system and opening a *covert channel*
  - ***Keylogger**:* Sensitive information gathering
  - ***Spammer**:* Money making, advertising
  - ***Flooder**, **Auto-rooter**, **Kits***, etc

# Malware in General

- Other categories (no malicious content):
  - ***Spyware**:* Collecting information of user activity
  - ***Adware**:* Annoying pop-up messages for information gathering
  - ***Phishing**:* Personal information gathering through social engineering
  - ***Tracking cookies**:* Behaviour tracking, information stealing
  - ***LSP** (Layered Service Provider):* Intercepting applications using WinSock the aim of which is information stealing
  - ***Joke programs**:* e.g. random screen locking
  - ***BHO (Browser Helper Objects**):* Takes control over the browser
  - ***Hoax programs**:* Spread untrue stories, e.g., collecting money for sick child.

# Taxonomy



# Static Analysis of Malware

- Malware is <u>not</u> executed
- Analysed code is called *specimen*
- Information (high/low level) is gathered
  - File size
  - Cryptographic hash
  - File format
  - Imported shared libraries, etc…
  - Through disassembling/decompiling
- Advantages
  - Analysis on different platform
  - Whole specimen is included (indpendent from input)
- Disadvantages
  - Time consuming
  - Self-modifying code and packed binaries are not extracted

# Static Analysis/Disassembly

- Process of analysing and represent machine code as text
- Disassemblers face challenges
  - x86 is a von Neumann architecture → code and data are not distinguished → possibilty of decoding data as code
  - x86 architecture is CISC → instructions of varying length (from 1 up to more than 10 bytes) → disassembly errors are difficult to be detected
- Two main types
  - Linear sweep
  - Recursive traversal
- Other approaches
  - Speculative disassembly

# Disassembly/Linear Sweep

- Starts at the begining of program's code section
- Taking one instruction after the other until the end of code section → very simple
- However, does not take program's control flow into account
  - data in instruction stream might be decoded as code
  - so insert junk bytes, but be unreachable at runtime
  - program's execution is the same, but disassembled incorrectly

```
1.      xor eax, eax            ; clear register, sets the zero flag
2.      jz dest                 ; jump to dest if zero flag
3.      db 0xe9                 ; junk byte
4.   dest:
5.      mov eax, 0x7c39542c     ; load 0x7c39542c into eax
6.      jmp eax                 ; jump to the address contained in eax

7.      nop                     ; no operation
8.      nop                     ; no operation
```

# Disassembly/Linear Sweep

- Decoded as:

```
00000000 31 c0               xor eax , eax
00000002 74 01               jz 0x5
00000004 e9 b8 2 c 54 39     jmp 39542cc1
00000009 7 c f f             jl 0xa
0000000b e0 90               loopnz ffffff9d
0000000d 90 nop
```

- Not the same instruction stream!
- Examples for linear sweep disassemblers: WinDbg, SoftICE

# Disassembly/Recursive Traversal

- Takes program’s control flow into account
- Creating CFG (code flow graph) build up from basic blocks
- Basic blocks
  - contains no branch instruction
  - possibly have one or more successor, predecessor blocks
  - if the first instruction is executed in it, so does the last one
- Disassembly is started at the entry point → goes till the first control transfer instruction → determining possible successor blocks → rescursively starts disassembling them
- Seems to solve the problem of interpreting data as code, but
  - Successor blocks are not trivial to determine → indirect branches (target address is stored in registers or in memory)
  - Conditional and unconditional jumps can lead to conflicting instruction sequences (example above: 2nd line, 6th line)
- Examples for recursive traversal disassemblers: OllyDbg, IDA

# More thwarts on recursive traversal

- Conditional & unconditional jumps (back to example)

```
1. xor eax, eax  ; eax is set to zero, but disassembler extracts both
2. jz dest       ; branches → there can be a direct jump to the 2nd
3. db 0xe9       ; line → eax might not be 0 → Two basic blocks are
4. dest:         ; created: the first starts from line 3 & the second starts from
                 ; line 4 → blocks are contradicting due to the junk byte
                 ; and meet at the second nop instruction
5. mov eax, 0x7c39542c ; Loading address to eax, but disassembler
6. jmp eax       ; can believe that there is a direct jump to line 6
7. nop           ; so cannot make sure that eax is 0x7c39542c.
8. nop           ; However, heuristics can be used to determine eax
```

# More thwarts on recursive traversal

- Obfuscating subroutine calls by modifying return address on the stack
  - CALL instruction pushes the address of the following instruction to the stack before branching
  - RET pops up this address from the stack and using it for return.
  - Miscreants can modify this value on the stack to an arbitrary address
  - Disassemblers do return to that address so being confused
- Example:

```
int main(void){
    int a, b, c;
    a=7;
    b=3;
    c = addnum(a,b);
    printf("Result is: %d", a+b);
}
```

```
int addnum(int a, int b){
    int c = 4;
    c = a + b;
    return c;
}
```

# Disassembly/Other Approaches

- Speculative disassembly
  - Based on recursive traversal, however, if there is no more branch target then it restarts disassembly in every gap in the code section
  - Invalid instructions are neglected if being encountered
- Another solution (Suggested by Kruegel et al.)
  - Tries to find functions in the code stream to reduce complexity
  - Decodes every byte within these functions and identifies intra-procedural branches (having at least one successor basic block inside the function)
  - These basic blocks build up the skeleton of the intra-procedural code flow graph which is the superset of the real CFG.
  - The two graphs can have conflicting basic blocks. E.g., Basic blocks with overlapping address ranges, but their instruction stream cannot be merged
  - An algorithm is created to remove nodes that are not in the real CFG

# Dynamic Analysis

- Executes an unknown program and examines its behaviour.
- Applies *sacrificial lab* (real machine with no or separated, limited network connection) to run malicious code → easy to be cleaned up and reused
- Another solution: Virtual machines
  - More flexible, easy to be handled
  - However, slower and if hostile code can detect the environment, then it modifies its control flow
- The simplest methodology:
  - Taking a snapshot of the pure system and every changes made by a malware are recorded
- A more fine grained way is *tracing*

# Dynamic Analysis/Tracing

- Tracing system calls
  - Works fine on *NIX systems where being documented well
  - Windows documentation does not contain the library of *ntdll.dll* which wraps the system calls. Only Windows API uses it, which is well documented, to reach the kernel of the OS.
- Tracing library calls
  - For programs implemented in high-level programming languages
  - These applications use standard library functions rather than system calls, however, the latter can be inspected as well.
- Instruction-level tracing (internal call tracing)
  - Typically the use of a debugger
  - Taking the control flow of a program into account
  - Time-consuming, but can be an aid for static analysis.

# Dynamic Analysis/Anti-debugging

- Anti-debugging techniques aims at detecting the presence of debugger and if so they change their behaviour. E.g.,
  - Terminating with/without error message
  - Taking another control flow path and so on.
- Detecting debugger through Win API calls
  - Windwos API *isDebuggerPresent()* function → returns the *BeingDebugged* flag from the *PEB (Process Environment Block)* of the current process.
  - Accessing the TRAP (EFLAGS) flag directly, without API call. However, easy to circumvent (resetting the flag)
  - *NtGlobalFlag* in PEB can also be used for the purpose

```
typedef struct _PEB {
    BOOLEAN InheritedAddressSpace;
    BOOLEAN BeingDebugged;
    ULONG NtGlobalFlag;
    PVOID ImageBaseAddress; …
}
```

# Dynamic Analysis/Anti-debugging

- Other methods use less wide-spread API calls, such as
  - *CheckRemoteDebuggerPresent()*
  - *NtQueryInformationProcess()*
- Detecting debuggers with exception handlers
  - Debuggers handle exception that don't pass to the debugged program.
  - These include debug (INT 01) or breakpoint (INT 03) exceptions
  - An exception handler should be implemented in the application
  - Set the value of a flag in program from that handler
  - Generate INT 01 or INT 03 interrupts and check the value of the flag.
  - If it is unchanged there might be a debugger presents
- Detection with timing
  - If debugger is present → ”slower” response

# Dynamic Analysis/Anti-debugging

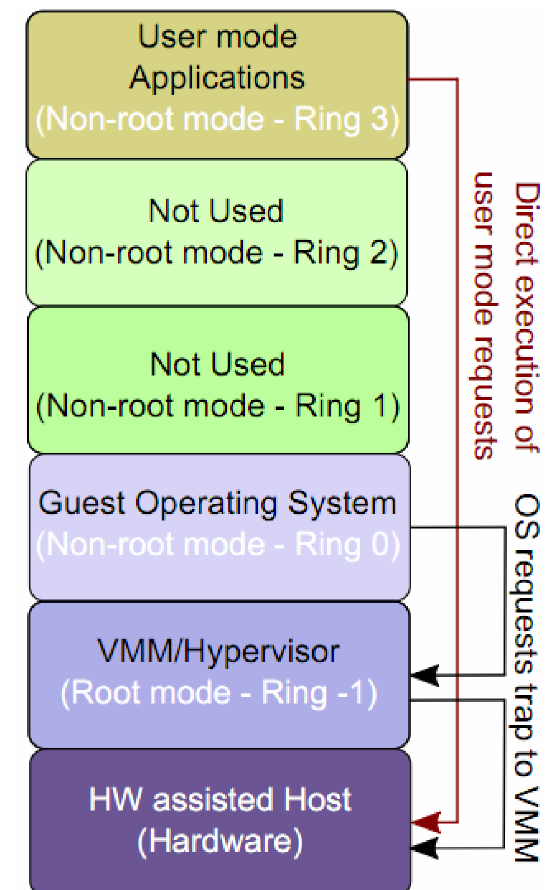
- Interfering directly with the debugger's operation
  - Using operating system's exception handling
  - Debugged application throws many exceptions intentionally
  - Debuggers try to hande all exceptions and display them to decide about them.
  - A large number of exceptions make debugging error-prone and cumbersome.
- Other methods
  - Blocking input so debugger cannot be controlled anymore
  - Requesting OS not to generate debug events
  - Modifying debug registers to disable *hardware breakpoints*
  - Locating *software breakpoints (INT03)* and overwriting them.

# Dynamic Analysis/Platforms

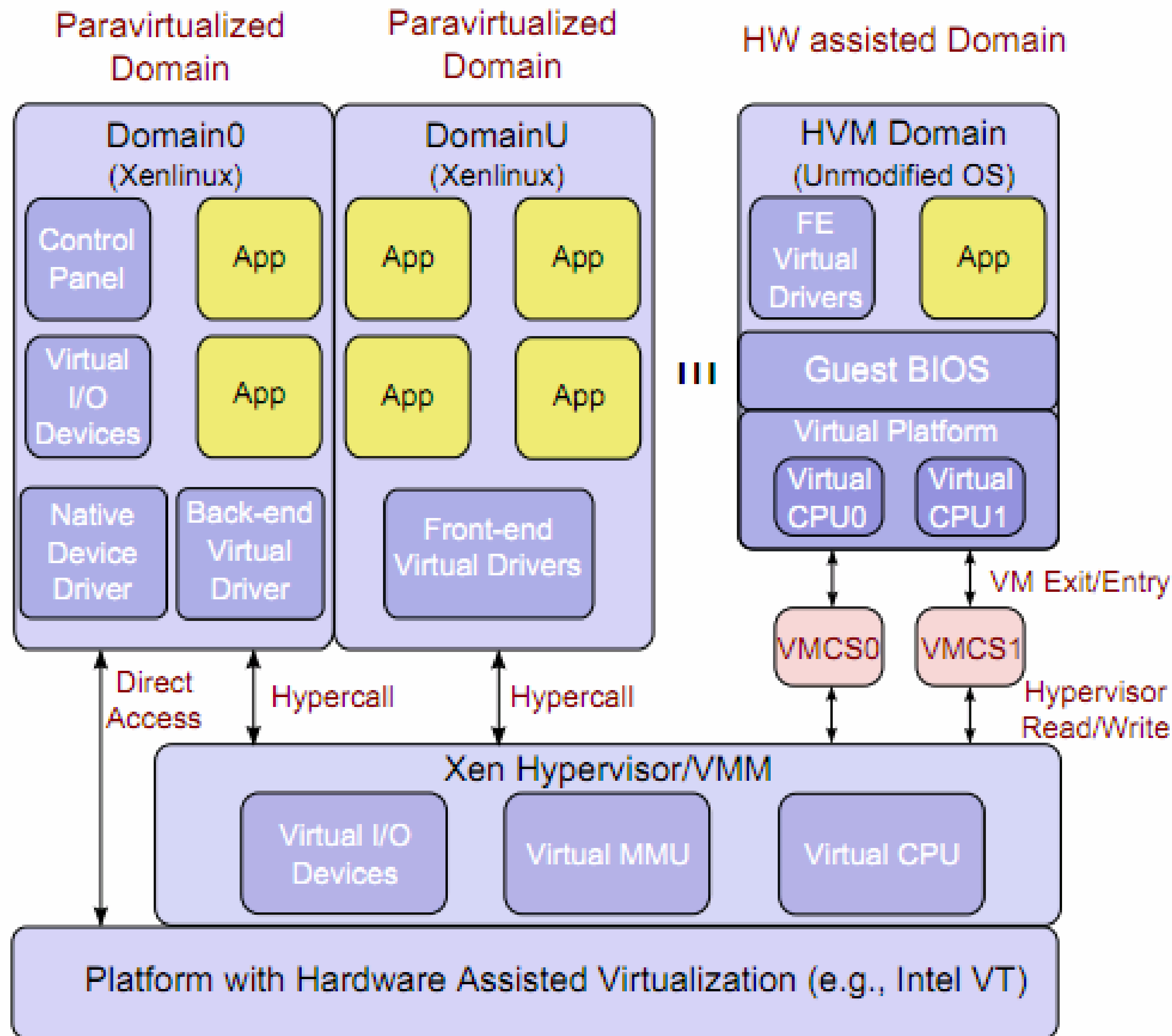
- Most of the virtualization based solutions can be evaded
  - The analyser runs at the same privilege level as the hostile code
  - Shortcomings from inaccurate/incomplete system emulation
    - Examples include : QEMU based platforms: Anubis, CWSandbox, Norman Sandbox
  - In-memory presence
  - Presence in CPU registers
  - Non-native measurement of time, etc
- Novel out-of-the-guest approaches mitigate these problems
  - Based on hardware assisted virtualization (e.g., Intel VT)
  - Make heavy use of extended feature set
    - Instruction set constraints for guests
    - Various way of instruction handling, etc.
  - Examples: Ether, Azure

# Hardware Assisted Virtualization

- AMD and Intel introduced new architecture in 2006
  - AMD-V and Intel VT (Virtualization Technology)
- New and higher privilege level is defined (protection ring -1)
- Native instruction execution
- Intel VT
  - VMX root mode for VMM / Hypervisor
  - VMX non-root mode for guest OS
  - VMX transitions VM Exit/ VM Entry
  - Saving the state of the guest by VMCSs
    - VMCS (Virtual Machine Control Structure)
- Hardware assisted virtualization based technologies: Xen, KVM



# Intel VT and Xen Overview

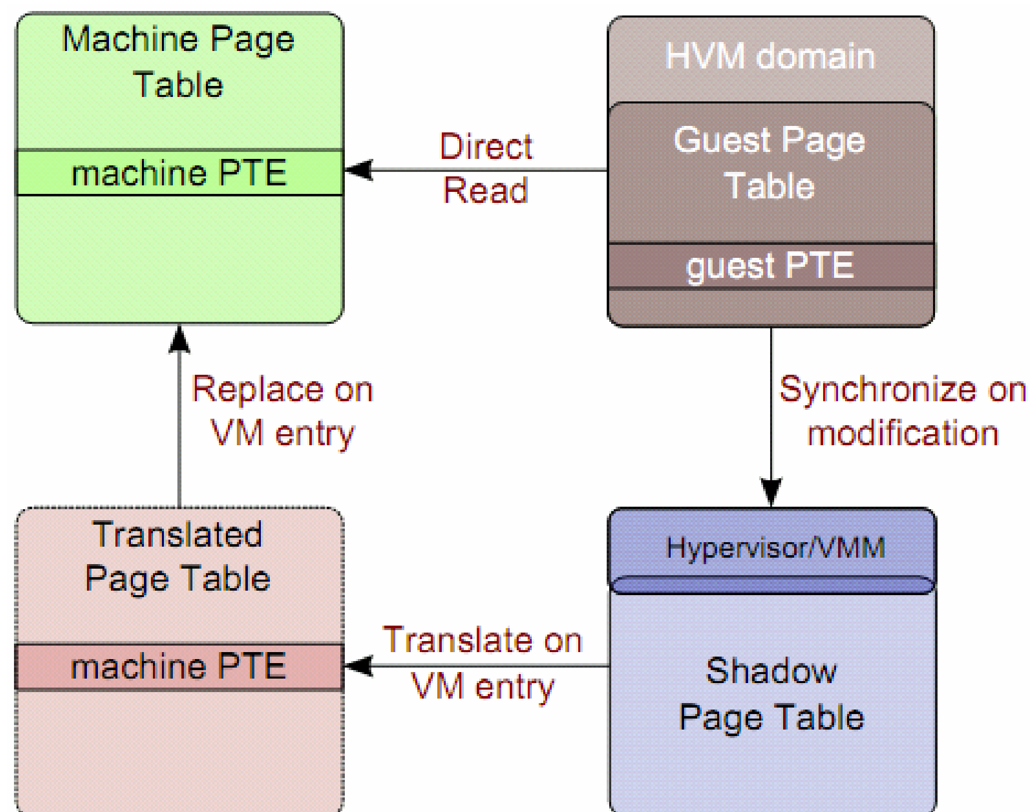


# About Xen

- Basically paravirtualization technology, however, being extended by Intel to support Intel VT and hardware assisted virtualization
- Virtual Machine Monitor (VMM) is called as Hypervisor
  - Specialized small kernel
  - Virtualizes essential hw resources, e.g., CPU, memory, etc.
- Each virtual machine is called as *Domain*
  - Domain 0: Driver domain → direct access to all I/O devices, administers domains (create, delete, reset, etc.)
  - Domain U: Hardware assisted or paravirtualized domains

# Xen’s Memory Management

- Shadow page tables
  - in-memory copies of real page tables
  - The hypervisor manages and synchonizes them with the guest’s page tables as well as controlling their memory allocations
  - Used in VMX root mode as guests cannot do their own translation



# Dynamic Analysis/Ether

- Out-of-the-guest malware analysis platform based on Xen and Intel VT
- Transparency requirements
  - Higher privilege of analyzer environment
  - No non-privileged side effects
  - Same instruction execution semantics
  - Identical exception handling
  - Identical notion of time
- Features of Ether
  - No in-guest memory presence
  - Hide of changes made on CPU registers
  - Memory protection: modifies only shadow page tables
  - Privileged instruction handling
  - No instruction emulation
  - Controlling timing (e.g., RDTSC instruction)

# Dynamic Analysis/Summary

- Resistant to obfuscation techniques such as
  - Self-modifying code
  - Runtime-packing
  - Encryption
  - Anti-disassembly tricks
- However, only one execution path is covered
- Hardware assisted based solutions mitigate many previous problems

# What we have learnt

- The basics of malware and malware analysis
- The advantages and disadvantages of static and dynamic analysis
- Weaknesses of current pure software virtualization solutions
- The basics of hardware assisted virtualization
- Features of out-of-the-guest malware analyzers