Dynamic and Transparent Analysis of Commodity Production Systems

Aristide Fattori\textsuperscript{1}  Roberto Paleari\textsuperscript{1}  Lorenzo Martignoni\textsuperscript{2}  Mattia Monga\textsuperscript{1}

\textsuperscript{1}Università degli Studi di Milano  \textsuperscript{2}University of California, Berkeley

25\textsuperscript{th} IEEE/ACM International Conference on Automated Software Engineering (ASE ’10)
How to debug a device driver?
How to debug a device driver?

Kernel debugger
How to debug a device driver?

- Kernel debugger
- VMM-based debugger
How to analyze run-time properties of a system?

Properties we would like to monitor:

- Creation of new processes (or threads)
- Execution of system calls
- Execution of kernel/user functions
- Access to hardware devices
- Memory access
- ...

Possible applications

- Profiling
- Tracing
- Debugging
- Dynamic instrumentation
Kernel-based solutions

- Require the installation of specific hooks in the kernel
- The analysis tool is implemented as a kernel module
- To analyze kernel-level code, these approaches leverage another kernel-level module...
Kernel-based solutions

- Require the installation of specific hooks in the kernel
- The analysis tool is implemented as a kernel module
- To analyze kernel-level code, these approaches leverage another kernel-level module...

...it is like a dog chasing its tail!
VMM-based solutions

- The analyzer leverages **VM-introspection techniques**
- The target system must be **already running inside a VM!**
- System-level programming inside a VM is not so easy . . .
VMM-based solutions

- The analyzer leverages VM-introspection techniques
- The target system must be already running inside a VM!
- System-level programming inside a VM is not so easy . . .

Have you ever tried to use your iPod through a VM?
Contributions

A framework to perform dynamic system-level analyses of commodity production systems

A. Fattori, R. Paleari, L. Martignoni, M. Monga
Dynamic and Transparent Analysis of Commodity Production Systems
Contributions

A framework to perform dynamic system-level analyses of commodity production systems

Features

1. Does not require any native support for the analysis (can be used on commodity or closed-source systems)

2. Supports the analysis of running systems (the target must not be rebooted)

3. User- and system-level code cannot detect nor affect the analysis infrastructure

4. Guarantees isolation of the analysis tools running on its top (a buggy tool does not cause the target system to crash)
How?

Exploit hardware support for virtualization

- A running system is migrated into a virtual machine on-the-fly
- The analysis framework runs at the hypervisor privilege level
  (it is more privileged than the OS and completely isolated)
A glimpse at hardware-assisted virtualization (Intel VT-x)

R3

App
App
App

R0

Kernel
A glimpse at hardware-assisted virtualization (Intel VT-x)
A glimpse at hardware-assisted virtualization (Intel VT-x)

- The OS needs not to be modified
- The hardware guarantees transparency & isolation
- Minimal overhead
A glimpse at hardware-assisted virtualization (Intel VT-x)

An exit/entry event causes the CPU to save the state of the guest/host inside the VMCS.
The events that trigger an exit to root mode can be configured dynamically.
Overview of the framework

User process  User process

User mode  Kernel mode

Operating system kernel
The framework is installed as the target system runs and is completely separated from the analyzed OS.
The analyzed OS needs not to be modified at all (i.e., the approach can be applied to closed-source OSes)
The analysis tool runs in an isolated execution environment (a defect in the tool does not affect the stability of the OS)
Overview of the framework

At the end of the analysis, the infrastructure can be removed on-the-fly.
Architecture

Non-root mode

Root mode

User mode

Kernel mode

1. Exit

2. Notification

Analysis tool

3. API call

6. Exception

4. API request

4a. Inspect/manipulate

5. Recover information about events

Event gate

Trap gate

Framework

Hardware

7. Interrupt

A. Fattori, R. Paleari, L. Martignoni, M. Monga
Dynamic and Transparent Analysis of Commodity Production Systems
Architecture

Non-root mode
Root mode

User mode
Kernel mode

1. Exit

2. Notification

3. API call

4. API request

4a. Inspect/manipulate

4b. Request event notification

5. Recover information about events

6. Exception

7. Interrupt

Hardware

Analysis tool

Event gate

Trap gate

Framework

API
Architecture

Non-root mode
Root mode

User mode
Kernel mode

1. Exit
2. Notification
3. API call
4. API request
4a. Inspect/manipulate
5. Recover information about events
6. Exception

Event gate
Trap gate
API

Framework

4b. Request event notification

5. Recover information about events

Hardware

7. Interrupt
Architecture

Non-root mode
Root mode
User mode
Kernel mode

1. Exit
2. Notification
3. API call
4. API request
4a. Inspect/manipulate
5. Recover information about events

Event gate
Trap gate
Analysis tool
API
Framework

Hardware

4b. Request event notification

5. Recover information about events

6. Exception

7. Interrupt

A. Fattori, R. Paleari, L. Martignoni, M. Monga
Dynamic and Transparent Analysis of Commodity Production Systems
Architecture

1. Exit
2. Notification
3. API call
4. API request
4a. Inspect/manipulate
5. Recover information about events
6. Exception

Non-root mode
Root mode
User mode
Kernel mode

Event gate
Trap gate
API
Analysis tool
Framework

4b. Request event notification
5. Recover information about events

Hardware
7. Interrupt

A. Fattori, R. Paleari, L. Martignoni, M. Monga Dynamic and Transparent Analysis of Commodity Production Systems
Architecture

Non-root mode

Root mode

User mode

Kernel mode

Event gate

Trap gate

API

Analysis tool

1. Exit

2. Notification

3. API call

4. API request

4a. Inspect/manipulate

4b. Request event notification

5. Recover information about events

6. Exception

7. Interrupt

Hardware

Framework

A. Fattori, R. Paleari, L. Martignoni, M. Monga Dynamic and Transparent Analysis of Commodity Production Systems
Architecture

Non-root mode

1. Exit
2. Notification

User mode

4a. Inspect/manipulate

4. API request
4b. Request event notification

5. Recover information about events

Kernel mode

6. Exception

Event gate

Trap gate

API

Hardware

7. Interrupt

Analysis tool

Framework

A. Fattori, R. Paleari, L. Martignoni, M. Monga Dynamic and Transparent Analysis of Commodity Production Systems
Architecture

Non-root mode
Root mode

User mode
Kernel mode

1. Exit
2. Notification
3. API call
4. API request
5. Recover information about events
4a. Inspect/manipulate

Event gate
Trap gate
API

3. Analysis tool

Framework

4b. Request event notification

Hardware

7. Interrupt

A. Fattori, R. Paleari, L. Martignoni, M. Monga
Dynamic and Transparent Analysis of Commodity Production Systems
Architecture

Non-root mode

Root mode

User mode

Kernel mode

1. Exit

2. Notification

3. API call

6. Exception

4. API request

4a. Inspect/manipulate

4b. Request event notification

5. Recover information about events

A. Fattori, R. Paleari, L. Martignoni, M. Monga Dynamic and Transparent Analysis of Commodity Production Systems
Which events can be intercepted?

- Events cause exits to root mode
- All the events exit conditionally
- Conditions are expressed as boolean conditions

\[(\text{process.name} = \text{"notepad.exe"} \land \text{syscall.name} = \text{"NtReadFile"})\]
Which events can be intercepted?

- Events cause exits to root mode
- All the events exit conditionally
- Conditions are expressed as boolean conditions
  \[(\text{process\_name} = \text{“notepad.exe”} \land \text{syscall\_name} = \text{“NtReadFile”})\]

**Native events vs high-level events**

- Traced directly through the hardware
- Very low-level operations (e.g., CPU exception)
- Traced through low-/high-level events
- High-level operations (e.g., Return from function)
### A summary of the events

<table>
<thead>
<tr>
<th>Event</th>
<th>Exit cause</th>
<th>Native exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcessSwitch</td>
<td>Change of page table address</td>
<td>✓</td>
</tr>
<tr>
<td>Exception</td>
<td>Exception</td>
<td>✓</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Interrupt</td>
<td>✓</td>
</tr>
<tr>
<td>BreakpointHit</td>
<td>Debug or page fault except.</td>
<td></td>
</tr>
<tr>
<td>WatchpointHit</td>
<td>Page fault except.</td>
<td></td>
</tr>
<tr>
<td>FunctionEntry</td>
<td>Break on function entry point</td>
<td></td>
</tr>
<tr>
<td>FunctionExit</td>
<td>Break on return address</td>
<td></td>
</tr>
<tr>
<td>SyscallEntry</td>
<td>Break on syscall entry point</td>
<td></td>
</tr>
<tr>
<td>SyscallExit</td>
<td>Break on return address</td>
<td></td>
</tr>
<tr>
<td>I0OperationPort</td>
<td>Port read/write</td>
<td>✓</td>
</tr>
<tr>
<td>I0OperationMmap</td>
<td>Watchpoint on device memory</td>
<td></td>
</tr>
</tbody>
</table>
High-Level Events

- Two main high-level events: **watchpoints** and **breakpoints**
- Other high-level events are traced through the previous ones (e.g., FunctionEntry, SyscallEntry, ...)
High-Level Events

- Two main high-level events: watchpoints and breakpoints
- Other high-level events are traced through the previous ones (e.g., FunctionEntry, SyscallEntry, ...)

How to set watchpoints and breakpoints from root mode?
High-Level Events

- Two main high-level events: **watchpoints** and **breakpoints**
- Other high-level events are traced through the previous ones (e.g., FunctionEntry, SyscallEntry, ...)

**How to set watchpoints and breakpoints from **root mode**?**

**Watchpoints**

- No native support from VT-x, few **hardware watchpoints** shared with the guest
- Implemented by protecting memory pages and trapping access exceptions
High-Level Events

- Two main high-level events: **watchpoints** and **breakpoints**
- Other high-level events are traced through the previous ones (e.g., FunctionEntry, SyscallEntry, ...)

How to set watchpoints and breakpoints from **root mode**?

Breakpoints

- No native support from VT-x, few **hardware breakpoints** shared with the guest
- **Software breakpoints** are efficient, but can be detected (the byte at the breakpoint address must be modified)
- Alternatively, breakpoints can be implemented through **watchpoints** (transparent but not very efficient)
State inspection and manipulation

CPU registers

- Inspection & manipulation is trivial
- Guest registers are stored inside the VMCS

Memory

- Memory inspection & manipulation requires MMU virtualization
- We mimic the behavior of the hardware MMU to translate VA $\rightarrow$ PHY and map the physical page
OS-dependent interface

- OS-independent analysis can be uncomfortable
  (e.g., refer to a process by means of its PT base address)
- OS-dependent APIs can ease the analysis
  (e.g., refer to a process through its name)
OS-dependent interface

- OS-independent analysis can be uncomfortable
  (e.g., refer to a process by means of its PT base address)
- OS-dependent APIs can ease the analysis
  (e.g., refer to a process through its name)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetFuncAddr(n)</td>
<td>Return the address of the function $n$</td>
</tr>
<tr>
<td>GetFuncName(a)</td>
<td>Return the name of the function at address $a$</td>
</tr>
<tr>
<td>GetProcName(p)</td>
<td>Get the name of process with page directory base address $p$</td>
</tr>
<tr>
<td>GetProcPID(p)</td>
<td>Get the PID of process with page directory base address $p$</td>
</tr>
<tr>
<td>GetProcLibs(p)</td>
<td>Enumerate DLLs loaded into process $p$</td>
</tr>
<tr>
<td>GetProcStack(p)</td>
<td>Get the stack base for process $p$</td>
</tr>
<tr>
<td>GetProcHeap(p)</td>
<td>Get the heap base for process $p$</td>
</tr>
<tr>
<td>GetProcList()</td>
<td>Enumerate processes</td>
</tr>
<tr>
<td>GetDriverList()</td>
<td>Enumerate device drivers</td>
</tr>
</tbody>
</table>
OS-dependent interface

- OS-independent analysis can be uncomfortable (e.g., refer to a process by means of its PT base address)
- OS-dependent APIs can ease the analysis (e.g., refer to a process through its name)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetFuncAddr(n)</td>
<td>Return the address of the function $n$</td>
</tr>
<tr>
<td>GetFuncName(a)</td>
<td>Return the name of the function at address $a$</td>
</tr>
<tr>
<td>GetProcName(p)</td>
<td>Get the name of process with page directory base address $p$</td>
</tr>
<tr>
<td>GetProcPID(p)</td>
<td>Get the PID of process with page directory base address $p$</td>
</tr>
<tr>
<td>GetProcLibs(p)</td>
<td>Enumerate DLLs loaded into process $p$</td>
</tr>
<tr>
<td>GetProcStack(p)</td>
<td>Get the stack base for process $p$</td>
</tr>
<tr>
<td>GetProcHeap(p)</td>
<td>Get the heap base for process $p$</td>
</tr>
<tr>
<td>GetProcList()</td>
<td>Enumerate processes</td>
</tr>
<tr>
<td>GetDriverList()</td>
<td>Enumerate device drivers</td>
</tr>
</tbody>
</table>

Current implementation supports only Microsoft Windows XP
HyperDbg: The key advantages

- A kernel debugger built on top of our framework
- Offers common kernel-debugging features
  (e.g., setting breakpoints and watchpoints, single-stepping, . . . )
- OS-independent and grants complete transparency to guest OS and its applications
HyperDbg: The key advantages

- Transparent to the guest OS
- (Almost) OS independent
- Fault resistant
- Debug **any** component, even critical ones (e.g., the scheduler, interrupt handlers, ...)
- No need for a second machine (WinDbg)
HyperDbg: The key advantages

- Installed as the system *runs*
- Direct interaction with the underlying hardware
- No need to deprivilege or modify the guest OS
- Software virtualizers are not so transparent...

Testing system virtual machines
(ISSTA ’10)
HyperDbg: Graphical User Interface

```
+==[pid: 00000004; proc: System]=+==[HyperDbg ]+===

hot-key pressed

executing command: disassemble 0x804df037

? 0x04df037: ff15c49b5580       call 0x80559bc4 <kernel32!IoCreateFile>
? 0x04df03b: 58                pop %eax
? 0x04df03c: 5a                pop %edx
? 0x04df03f: ff9530f6dfff      inc 0xfffff630
? 0x04df045: 0bh2              mov %edx, %esi
? 0x04df047: 0b5f0c            mov %ecx (%edi, %esi)
? 0x04df04a: 33e9              xor %ecx, %ecx
? 0x04df04d: 80e18             mov (%eax,%edi), %cl
? 0x04df054: 0b3f              mov (%edi, %edi)
? 0x04df05b: 81e87             mov (%edi,%eax,4), %ebx
? 0x04df064: 2be1              sub %ecx, %esp
? 0x04df066: c1e902            shr $0x2, %ecx
? 0x04df079: 0fhc              mov %esp, %edi
? 0x04df085: 3h534f55580       cmp 0x8055f534, %esi
? 0x04df091: 0f83a9010000      jae 0x64x
? 0x04df0b6: 0f35               rep movsd
? 0x04df0bc: ffd3              call %ebx
? 0x04df0d3: 0be5              mov %ebp, %esp
? 0x04df0d4: 80d24f1dffff      mov 0xfffff124, %ecx
? 0x04df0d7: 0b553c            mov 0x3c(%ebp), %edx
? 0x04df0e6: 699134010000      mov %edx, 0x14(%ecx)
? 0x04df0e7: 8a                fcall
? 0x04df0e8: 745720000000200   test $0x200000, 0x700(%ebp)
? 0x04df0e9: 7556              jnz 0x64x
? 0x04df0ec: f456c01           testb $0xl, 0x6c(%ebp)
? 0x04df0f8: 7451              jz 0x64x
? 0x04df0f8: 0b1d24f1dffff     mov 0xfffff124, %ebx
? 0x04df0f9: c6432e00           movb $0x80, 0x2c(%ebx)
? 0x04df0fa: 807b4a00           cmpb $0x80, 0x4a(%ebx)
? 0x04df0fd: 7446              izx 0x64x

end of command: disassemble 0x004df037
```

```
  
  [current] 006f58af
  [000] 000050780 f85d14dc  (i8042pport.sys)
  [001] 000050800 804d3d9f (KiInterruptDispatch+0x61)
  [002] 000050800 f85f3062  [nt32mm.sys]
  [003] 000050800 804d0047 (KiSwapProcess+0x121)
  [004] 0fafff980 fffdf980

end of command: backtrace 5
```
Information about the state of the guest (also provides OS-dependent details)
HyperDbg: Graphical User Interface

Information about what triggered HyperDbg

```c
hot-key pressed

executing command: disassemble 0x004df037
004df037: ff15c49b5580
004df038: 58
004df039: 5a
004df03a: ff0530f6dfff
004df03b: 0bh2
004df03c: 0bf0
004df03d: 33f9
004df03e: 60e18
004df03f: 0bf3f
004df040: 81e87
004df041: 2be1
004df042: e1e902
004df043: 0fh
3h3534f55580
004df044: 0bh61
004df045: f3a6
004df046: 8f7
004df047: 0bh4b
004df048: 85f9
004df049: 0bh3
004df04a: 89f7
004df04b: 89f7
004df04c: 0bh4d
004df04d: 570
004df04e: 456c01
004df04f: 7458
004df050: 61d24f1dfff
004df051: 6432e00
004df052: 8074a00
004df053: 7448
end of command: disassemble 0x004df037
```

A. Fattori, R. Paleari, L. Martignoni, M. Monga Dynamic and Transparent Analysis of Commodity Production Systems 17
HyperDbg: Graphical User Interface

Output

A. Fattori, R. Paleari, L. Martignoni, M. Monga
Dynamic and Transparent Analysis of Commodity Production Systems
HyperDbg: Graphical User Interface

Resolve symbols (OS-Dependent)
HyperDbg: Graphical User Interface

Module name (OS-Dependent)

A. Fattori, R. Paleari, L. Martignoni, M. Monga  Dynamic and Transparent Analysis of Commodity Production Systems
User interface

- We cannot rely on the guest OS graphic libraries
- A small VGA driver to interact with the system’s video card
- The driver is neither OS nor hardware dependent
HyperDbg: Implementation

User interface

- We cannot rely on the guest OS graphic libraries
- A small VGA driver to interact with the system’s video card
- The driver is neither OS nor hardware dependent

User interaction

- An user can activate HyperDbg by pressing an hot-key
- In non-root mode keystrokes are intercepted by leveraging VT-x functionalities (i.e., IOOperationPort events)
- In root mode a simple driver reads the keystrokes
In summary
In summary

Contributions

A framework to perform dynamic system-level analyses of commodity production systems

Features

1. Does not require any native support for the analysis
   (can be used on commodity or closed-source systems)
2. Supports the analysis of running systems
   (the target must not be rebooted)
3. User- and system-level code cannot detect nor affect the analysis infrastructure
4. Guarantees isolation of the analysis tools running on its top
   (a buggy tool does not cause the target system to crash)
In summary

**Contributions**

A framework to perform dynamic system-level analyses of commodity production systems

**Features**

1. Does not require any native support for the analysis (can be used on commodity or closed-source systems)
2. Supports the analysis of running systems (the target must not be rebooted)
3. User- and system-level code cannot detect nor affect the analysis infrastructure
4. Guarantees isolation of the analysis tools running on its top (a buggy tool does not cause the target system to crash)

**Architecture**

[Diagram showing the architecture with key components and steps labeled 1 to 7.]

A. Fattori, R. Paleari, L. Martignoni, M. Monga Dynamic and Transparent Analysis of Commodity Production Systems
In summary

Contributions

A framework to perform dynamic system-level analyses of commodity production systems

Features

1. Does not require any native support for the analysis (can be used on commodity or closed-source systems)
2. Supports the analysis of running systems (the target must not be rebooted)
3. User- and system-level code cannot detect nor affect the analysis infrastructure
4. Guarantees isolation of the analysis tools running on its top (a buggy tool does not cause the target system to crash)

Architecture

Non-root mode
Root mode
User mode
Kernel mode

1. Event
2. Notification
3. API call
4. Exception

Hardware

Event gate
Trap gate

4a. Request event notification
5. Recover information about events
6. Interrupt
7. Interrupt

Analysis tool
API

A. Fattori, R. Paleari, L. Martignoni, M. Monga Dynamic and Transparent Analysis of Commodity Production Systems
In summary

Contributions

A framework to perform dynamic system-level analyses of commodity production systems

Features

1. Does not require any native support for the analysis (can be used on commodity or closed-source systems)
2. Supports the analysis of running systems (the target must not be rebooted)
3. User- and system-level code cannot detect nor affect the analysis infrastructure
4. Guarantees isolation of the analysis tools running on its top (a buggy tool does not cause the target system to crash)

Architecture

http://code.google.com/p/hyperdbg/
Dynamic and Transparent Analysis of Commodity Production Systems

http://code.google.com/p/hyperdbg

Thank you!
Any questions?

Aristide Fattori
aristide@security.dico.unimi.it
Backup slides
Watchpoints: Details

- Interrupt execution of memory access (read/write)
- Implemented by protecting memory pages and trapping access exceptions
Watchpoints: Details

- Interrupt execution of memory access (read/write)
- Implemented by protecting memory pages and trapping access exceptions

Monitor any access to a given memory address
Watchpoints: Details

- Interrupt execution of memory access (read/write)
- Implemented by protecting memory pages and trapping access exceptions

Remove any permission from the target page
Watchpoints: Details

- Interrupt execution of memory access (read/write)
- Implemented by protecting memory pages and trapping access exceptions

Further accesses trigger a CPU exception
Watchpoints: Details

- Interrupt execution of memory access (read/write)
- Implemented by protecting memory pages and trapping access exceptions

If the faulty addr. matches a watchpoint, dispatch the event
**Watchpoints: Details**

- **Interrupt execution of memory access (read/write)**
- **Implemented by protecting memory pages and trapping access exceptions**

![Diagram of physical memory and processes](attachment:image.png)

> Restore the original permissions to resume the execution
Watchpoints: Details

- Interrupt execution of memory access (read/write)
- Implemented by protecting memory pages and trapping access exceptions

To hide watchpoints we modify the entry in which the page table is mapped
(i.e.: we install a Shadow Page Table into the guest operating system with stricter permission than the original PT)
Late launching

- The target system becomes the guest of a virtual machine.
- The VMCS is configured to reflect the current state of the guest.
- When the framework installation is over, the control is returned to the guest.
- The CPU restores the guest state from the VMCS (so that the guest execution is resumed just as nothing happened.)