Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture

Sebastian Vogl and Claudia Eckert
{vogls,eckertc}@in.tum.de

Chair for IT Security
Technische Universität München
Munich, Germany

10.04.2012
1. Motivation

2. Performance Monitoring Counters (PMCs)

3. PMC-based Instruction-level Monitoring (ILM)

4. Experiments & Results

5. Summary
Outline

1. Motivation

2. Performance Monitoring Counters (PMCs)

3. PMC-based Instruction-level Monitoring (ILM)

4. Experiments & Results

5. Summary
Motivation

- Why Instructions-Level Monitoring (ILM) ?

My Research

Make use of full hardware virtualization to detect malware infections and exploitation attempts.
Motivation

Why Instructions-Level Monitoring (ILM) ?

```
main

400707: call 400584 <vulnerable>
40070c: mov 0x0, %EAX

DATA
Stack
vulnerable

400584: push %rbp
400585: mov %rsp,%rbp
400588: sub $0x20,%rsp

4006b2: leave
4006b3: ret

<vulnerable code>
```

S. Vogl and C. Eckert (TUM)
Motivation

Why Instructions-Level Monitoring (ILM)?
Motivation

- Why Instructions-Level Monitoring (ILM)?

```
main

400707: call 400584 <vulnerable>
40070c: mov 0x0, %EAX

DATA

Stack

0x40070c (RET)

RBP

vulnerable

400584: push %rbp
400585: mov %rsp,%rbp
400588: sub $0x20,%rsp

<vulnerable code>

4006b2: leave
4006b3: ret
```
Motivation

- Why Instructions-Level Monitoring (ILM)?
Motivation

Why Instructions-Level Monitoring (ILM) ?

```
main
400707: call 400584 <vulnerable>
40070c: mov 0x0, %EAX
```

```
vulnerable
400584: push %rbp
400585: mov %rsp,%rbp
400588: sub $0x20,%rsp
```

```
<vulnerable code>
4006b2: leave
4006b3: ret
```
Motivation

- Why Instructions-Level Monitoring (ILM) ?

```
main

400707: call 400584 <vulnerable>
40070c: mov 0x0, %EAX

vulnerable

400584: push %rbp
400585: mov %rsp,%rbp
400588: sub $0x20,%rsp

4006b2: leave
4006b3: ret
```

Stack

```
* /bin/bash
exit
system
DATA (EBP)
```

DATA

 `<vulnerable code>`
Motivation

Why Instructions-Level Monitoring (ILM) ?

```
400707: call 400584 <vulnerable>
40070c: mov 0x0, %EAX
```

```
400584: push %rbp
400585: mov %rsp, %rbp
400588: sub $0x20, %rsp
```

```
4006b2: leave
4006b3: ret
```
Motivation

» Why Instructions-Level Monitoring (ILM) ?

```
main
400707:  call 400584 <vulnerable>
40070c:  mov 0x0, %EAX

vulnerable
400584:  push %rbp
400585:  mov %rsp,%rbp
400588:  sub $0x20,%rsp

<vulnerable code>
4006b2:  leave
4006b3:  ret
```

Stack
* /bin/bash
exit
system
One possible Solution
Make use of a Shadow Stack to verify the target of return instructions.
Motivation

Why Instructions-Level Monitoring (ILM) ?

```
main
400707: call 400584 <vulnerable>
40070c: mov 0x0, %EAX

DATA
Stack

vulnerable
400584: push %rbp
400585: mov %rsp,%rbp
400588: sub $0x20,%rsp

<vulnerable code>
4006b2: leave
4006b3: ret

Shadow Stack

S. Vogl and C. Eckert (TUM)
Motivation

Why Instructions-Level Monitoring (ILM) ?

- main
  400707: call 400584 <vulnerable>
  40070c: mov 0x0, %EAX

- vulnerable
  400584: push %rbp
  400585: mov %rsp,%rbp
  400588: sub $0x20,%rsp
  4006b2: leave
  4006b3: ret

- <vulnerable code>

- Stack
  0x40070c (RET)

- DATA
  0x40070c (RET)

- Shadow Stack
  0x40070c (RET)
Motivation

Why Instructions-Level Monitoring (ILM) ?

```
main

400707: call 400584 <vulnerable>
40070c: mov 0x0, %EAX

vulnerable

400584: push %rbp
400585: mov %rsp,%rbp
400588: sub $0x20,%rsp

<vulnerable code>

4006b2: leave
4006b3: ret

system

Stack

* /bin/bash
exit

system

Shadow Stack

0x40070c (RET)

EIP: system
```
Motivation

- Why Instructions-Level Monitoring (ILM) ?

Observation

A Shadow Stack for return addresses can be implemented on the hypervisor-level by only trapping call and return instructions.
Motivation

- Why Instructions-Level Monitoring (ILM)?

Observation

A Shadow Stack for return addresses can be implemented on the hypervisor-level by only trapping call and return instructions.

ILM Requirements

1. Based on full hardware virtualization
Motivation

Why Instructions-Level Monitoring (ILM) ?

Observation

A Shadow Stack for return addresses can be implemented on the hypervisor-level by only trapping call and return instructions.

ILM Requirements

1. Based on full hardware virtualization
2. Secure
Motivation

Why Instructions-Level Monitoring (ILM) ?

Observation

A Shadow Stack for return addresses can be implemented on the hypervisor-level by only trapping call and return instructions.

ILM Requirements

1. Based on full hardware virtualization
2. Secure
3. Flexible
Motivation

Why a new ILM mechanism?

Existing Approaches

1. Page-Fault (PF)-based ILM
2. Debug Register (DR)-based ILM
3. Trap Flag (TF)-based ILM
Motivation

- Why a new ILM mechanism?

Existing Approaches

1. Page-Fault (PF)-based ILM
2. Debug Register (DR)-based ILM
3. Trap Flag (TF)-based ILM

⇒ None of the existing methods can provide the desired flexibility.
Motivation

Why a new ILM mechanism?

Existing Approaches

1. Page-Fault (PF)-based ILM
2. Debug Register (DR)-based ILM
3. Trap Flag (TF)-based ILM
   - Insecure
Motivation

Why a new ILM mechanism?

Existing Approaches

1. Page-Fault (PF)-based ILM
2. Debug Register (DR)-based ILM
3. Trap Flag (TF)-based ILM
   - Insecure
   - Incomplete
Motivation

Why a new ILM mechanism?

Existing Approaches

1. Page-Fault (PF)-based ILM
2. Debug Register (DR)-based ILM
3. Trap Flag (TF)-based ILM
   - Insecure
   - Incomplete
   - Inflexible
Motivation

Why a new ILM mechanism?

Existing Approaches

1. **Page-Fault (PF)-based ILM**
2. **Debug Register (DR)-based ILM**
3. **Trap Flag (TF)-based ILM**
   - Insecure
   - Incomplete
   - Inflexible

⇒ None of the existing methods can provide the desired **flexibility**.
Outline

1. Motivation

2. Performance Monitoring Counters (PMCs)

3. PMC-based Instruction-level Monitoring (ILM)

4. Experiments & Results

5. Summary
Performance Monitoring on the x86 architecture

- Performance Events
Performance Monitoring Counters (PMCs)

Overview

Performance Monitoring on the x86 architecture

- Performance Events
- PMCs that count these events
Performance Monitoring on the x86 architecture

- Performance Events
- PMCs that count these events
  - Which event is counted can be programmed.
Performance Monitoring Counters (PMCs)

Overview

Performance Monitoring on the x86 architecture

- Performance Events
- PMCs that count these events
  - Which event is counted can be programmed.
  - Can be set to raise an interrupt on overflow.
Performance Monitoring Counters (PMCs)

- Performance Events

- All instructions
- All branch instructions
- All conditional branch instructions
- All near call instructions
- All near return instructions
- All far branch instructions
PMC-based Instruction-level Monitoring (ILM)

- Trapping Performance Events

**Question**

How can we trap performance events to the hypervisor?
Question

How can we trap performance events to the hypervisor?

Challenges

1. **Interrupt Generation**: Generate an interrupt whenever the desired hardware performance event occurs.
Question

How can we trap performance events to the hypervisor?

Challenges

1. **Interrupt Generation**: Generate an interrupt whenever the desired hardware performance event occurs.

2. **Control Transfer**: The emitted interrupt must lead to a VM Exit.
Set the PMC initially to

$$\text{MAX\_PMC\_VALUE} - X + 1$$

where $X$ is the number of events that should occur before the interrupt.
Set the PMC initially to

\[ \text{MAX\_PMC\_VALUE} - X + 1 \]

where \( X \) is the number of events that should occur before the interrupt.

⇒ PMC will overflow after the desired number of events.
Set the PMC initially to

\[ \text{MAX\_PMC\_VALUE} - X + 1 \]

where \( X \) is the number of events that should occur before the interrupt.

⇒ PMC will overflow after the desired number of events.

⇒ An Interrupt will be generated.
The type of interrupt that is generated depends on the settings within the local Advanced Programmable Interrupt Controller (APIC).
The type of interrupt that is generated depends on the settings within the local Advanced Programmable Interrupt Controller (APIC).

It is possible to generate a Nonmaskable Interrupt (NMI).

- NMIs lead to a VM Exit if the appropriate flag is set.
**Interrupt Generation**

- The type of interrupt that is generated depends on the settings within the local Advanced Programmable Interrupt Controller (APIC).

- It is possible to generate a Nonmaskable Interrupt (NMI).
  - NMIs lead to a VM Exit if the appropriate flag is set.
  - NMIs are immediately handled by the processor.
PMC-based Instruction-level Monitoring (ILM)

- Trapping Performance Events: Control Transfer

---

**Interrupt Generation**

- The type of interrupt that is generated depends on the settings within the local Advanced Programmable Interrupt Controller (APIC).
- It is possible to generate a Nonmaskable Interrupt (NMI).
  - NMIs lead to a VM Exit if the appropriate flag is set.
  - NMIs are immediately handled by the processor.

---

**Problem: Interrupt Delivery**

- There is a gap of time between the occurrence of a performance event and the interrupt delivery.
Interrupt Generation

- The type of interrupt that is generated depends on the settings within the local Advanced Programmable Interrupt Controller (APIC).
- It is possible to generate a Nonmaskable Interrupt (NMI).
  - NMI lead to a VM Exit if the appropriate flag is set.
  - NMI are immediately handled by the processor.

Problem: Interrupt Delivery

- There is a gap of time between the occurrence of a performance event and the interrupt delivery.
- Other performance events may go unnoticed during this period of time.
PMC-based Instruction-level Monitoring (ILM)

- Trapping Performance Events: Control Transfer

### Interrupt Generation

- The type of interrupt that is generated depends on the settings within the local Advanced Programmable Interrupt Controller (APIC).
- It is possible to generate a Nonmaskable Interrupt (NMI).
  - NMIs lead to a VM Exit if the appropriate flag is set.
  - NMIs are immediately handled by the processor.

### Problem: Interrupt Delivery

- There is a gap of time between the occurrence of a performance event and the interrupt delivery.
- Other performance events may go unnoticed during this period of time.
- Problem has to be solved on a case-by-case basis.
Problem

- The number of selected instructions that are executed during interrupt delivery depend on the event that we monitor.
- If we set a PMC to count every instruction, about 6 instructions will be executed on the average before the interrupt is acknowledged.
Problem

- The number of selected instructions that are executed during interrupt delivery depend on the event that we monitor.
- If we set a PMC to count every instruction, about 6 instructions will be executed on the average before the interrupt is acknowledged.

Solution

The PMC will keep counting after an overflow occurred.

⇒ We know exactly how many instructions were executed before the interrupt was acknowledged.

⇒ Reconstruct the instruction stream and obtain the instructions that we missed.
PMC-based Instruction-level Monitoring (ILM)

- Instruction Reconstruction (IR)

**Approach**

1. Save the value of the instruction pointer on every overflow.
2. Check the value of the PMC on overflow to determine how many instructions were missed if any.
3. Disassemble every instruction starting from the last saved instruction pointer till we reach the current instruction pointer.

**Example**

1. 40f448 : mov %r12,%rdi ; <======= LAST EIP
2. 40f44b : mov $0x20,%esi
3. 40f450 : mov %rbp,%rdx
4. 40f453 : mov %ecx,0x28(%rsp)
5. 40f457 : mov %r8b,0x10(%rsp)
6. 40f45c : mov %r9,0x20(%rsp)
7. 40f461 : add %rbp,%r12 ; <======= CURRENT EIP
What about branches?

1. 40f24e: `pop %r12` ; <====== LAST EIP
2. 40f250: `pop %r13`
3. 40f252: `pop %r14`
4. 40f254: `pop %r15`
5. 40f256: `ret`

Problem

The target of a branch may depend on a memory operand that may have been overwritten in the meantime.
PMC-based Instruction-level Monitoring (ILM)

- The Last Branch Record (LBR) Stack

### LBR Stack

- Records the last taken **branches**
- Set of MSRs
  - A top-of-stack (TOS) pointer (MSR\_LASTBRANCH\_TOS)
  - A pair of MSRs for each branch that the stack can record:
    
    MSR\_LASTBRANCH\_x\_FROM\_IP => MSR\_LASTBRANCH\_x\_TO\_LIP

- The size of the LBR stack depends on the microarchitecture
PMC-based Instruction-level Monitoring (ILM)

The Last Branch Record (LBR) Stack

LBR Stack

- Records the last taken branches
- Set of MSRs
  - A top-of-stack (TOS) pointer (MSR_LASTBRANCH_TOS)
  - A pair of MSRs for each branch that the stack can record:
    MSR_LASTBRANCH_x_FROM_IP ⇒ MSR_LASTBRANCH_x_TO_LIP
- The size of the LBR stack depends on the microarchitecture

⇒ Save the TOS pointer on each monitoring related interrupt.
PMC-based Instruction-level Monitoring (ILM)

- The Last Branch Record (LBR) Stack

**LBR Stack**

- Records the last taken branches
- Set of MSRs
  - A top-of-stack (TOS) pointer (MSR_LASTBRANCH_TOS)
  - A pair of MSRs for each branch that the stack can record:
    MSR_LASTBRANCH_x_FROM_IP ⇒ MSR_LASTBRANCH_x_TO_LIP
- The size of the LBR stack depends on the microarchitecture

⇒ Save the TOS pointer on each monitoring related interrupt.
⇒ All taken branches are recorded between the last saved TOS and the current TOS.
PMC-based Instruction-level Monitoring (ILM)

- Instruction Reconstruction (IR)

Using the LBR Stack

1. 40f24e: pop %r12  ; <====== LAST EIP
2. 40f250: pop %r13
3. 40f252: pop %r14
4. 40f254: pop %r15
5. 40f256: ret

6

7. 40f4b3: mov %r12,%rdi  ; <====== CURRENT EIP
PMC-based Instruction-level Monitoring (ILM)

What about security?

- PMCs are MSRs
- All PMC control structures are MSRs as well
- Read/Write accesses to MSRs can be intercepted from the hypervisor

⇒ An attacker cannot disable or manipulate the PMCs.
Outline

1. Motivation
2. Performance Monitoring Counters (PMCs)
3. PMC-based Instruction-level Monitoring (ILM)
4. Experiments & Results
5. Summary
Monitored four common Linux applications at the instruction-level:

- **ls** (Argument: `/usr/bin`, 597 files)
- **tar** (Argument: `Hello World.c`, 10 LOC)
- **cat** (Argument: `Hello World.c`, 10 LOC)
- **gcc** (Argument: `Hello World.c`, 10 LOC)
Experiments & Results

Experiments

- Monitored four common Linux applications at the instruction-level:
  - `ls` (Argument: `/usr/bin`, 597 files)
  - `tar` (Argument: `Hello World.c`, 10 LOC)
  - `cat` (Argument: `Hello World.c`, 10 LOC)
  - `gcc` (Argument: `Hello World.c`, 10 LOC)

- Each application was executed multiple times using different monitoring modes:
  - PMC ALL & IR: **All** instructions & **Instruction Reconstruction**
  - TF ALL: **All** instructions
  - PMC ALL: **All** instructions without **Instruction Reconstruction**
  - PMC Branches: All **branch** instructions
  - PMC Shadow Stack: Only **call & return** instructions
Experiments & Results

- **Experiments**

- Monitored four common Linux applications at the instruction-level:
  - `ls` (Argument: `/usr/bin`, 597 files)
  - `tar` (Argument: `Hello World.c`, 10 LOC)
  - `cat` (Argument: `Hello World.c`, 10 LOC)
  - `gcc` (Argument: `Hello World.c`, 10 LOC)

- Each application was executed multiple times using different monitoring modes:
  - PMC ALL & IR: **All** instructions & **Instruction Reconstruction**
  - TF ALL: **All** instructions
  - PMC ALL: **All** instructions without **Instruction Reconstruction**
  - PMC Branches: All **branch** instructions
  - PMC Shadow Stack: Only **call** & **return** instructions

- Measured the execution time from the hypervisor for each run
- Calculated the **average slowdown factor**
## Experiments & Results

### Results

<table>
<thead>
<tr>
<th>Mode</th>
<th>ls</th>
<th>tar</th>
<th>cat</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMC ALL &amp; IR</td>
<td>755  (18s)</td>
<td>1002 (3.0s)</td>
<td>334  (0.6s)</td>
<td>1263 (92s)</td>
</tr>
<tr>
<td>TF ALL</td>
<td>310  (7.0s)</td>
<td>415 (1.2s)</td>
<td>142  (0.3s)</td>
<td>545 (40s)</td>
</tr>
<tr>
<td>PMC ALL</td>
<td>273  (6.5s)</td>
<td>403 (1.2s)</td>
<td>126  (0.3s)</td>
<td>435 (32s)</td>
</tr>
<tr>
<td>PMC Branches</td>
<td>163  (4.0s)</td>
<td>259 (0.8s)</td>
<td>81   (0.2s)</td>
<td>281 (21s)</td>
</tr>
<tr>
<td>PMC Shadow Stack</td>
<td>95   (2.0s)</td>
<td>196 (0.6s)</td>
<td>31   (0.1s)</td>
<td>212 (15s)</td>
</tr>
</tbody>
</table>
Experiments & Results

- Improving the Performance

Improving the Performance

- The performance of the approach heavily depends on the number of the VM Exits.

Possible Approaches

- Precise Event Based Sampling (PEBS)
- Branch Trace Store (BTS)

Security

The overall security of the mechanisms will decrease if the VM Exits are reduced.
Improving the Performance

- The performance of the approach heavily depends on the number of the VM Exits.
- The performance will increase by almost the same factor as the VM Exits are decreased.
Improving the Performance

- The performance of the approach heavily depends on the number of the VM Exits.
- The performance will increase by almost the same factor as the VM Exits are decreased.
- Possible Approaches
  - Precise Event Based Sampling (PEBS)
  - Branch Trace Store (BTS)
Experiments & Results

- Improving the Performance

Improving the Performance

- The performance of the approach heavily depends on the number of the VM Exits.
- The performance will increase by almost the same factor as the VM Exits are decreased.
- Possible Approaches
  - Precise Event Based Sampling (PEBS)
  - Branch Trace Store (BTS)

Security

The overall security of the mechanisms will decrease if the VM Exits are reduced.
Summary

Contributions

- PMC-based trapping
- A flexible and secure ILM mechanism
- Instruction Reconstruction

The proposed ILM mechanism still leads to significant overhead. However, the mechanism can be significantly faster than existing hardwared-based mechanism on the x86 architecture. There is still a lot of room for improvements. More detailed experiments are needed.

⇒ We encourage other researchers to explore the possibilities of PMC-based trapping as well as PMC-based ILM.
Summary

Contributions

- PMC-based trapping
- A flexible and secure ILM mechanism
- Instruction Reconstruction

Performance

- The proposed ILM mechanism still leads to significant overhead.
- However, the mechanism can be significantly faster than existing hardware-based mechanism on the x86 architecture.
- There is still a lot of room for improvements.
- More detailed experiments are needed.

We encourage other researchers to explore the possibilities of PMC-based trapping as well as PMC-based ILM.
Summary

Contributions

- PMC-based trapping
- A flexible and secure ILM mechanism
- Instruction Reconstruction

Performance

- The proposed ILM mechanism still leads to significant overhead.
- However, the mechanism can be significantly faster than existing hardwared-based mechanism on the x86 architecture.
- There is still a lot of room for improvements.
- More detailed experiments are needed.

⇒ We encourage other researchers to explore the possibilities of PMC-based trapping as well as PMC-based ILM.
Questions?
Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 
Ropdefender: a detection tool to defend against return-oriented programming attacks. 

Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 
Ether: Malware Analysis via Hardware Virtualization Extensions. 

Tal Garfinkel, Keith Adams, and Andrew Warfield. 
Compatibility is not transparency. 
Intel Corporation.  

Corey Malone, Mohamed Zahran, and Ramesh Karri.  
Are hardware performance counters a cost effective way for integrity checking of programs.  

Jonas Pfoh, Christian Schneider, and Claudia Eckert.  
Exploiting the x86 Architecture to Derive Virtual Machine State Information.  
Jonas Pfoh, Christian Schneider, and Claudia Eckert.  
Nitro: Hardware-based System Call Tracing for Virtual Machines.  