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

Outline

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

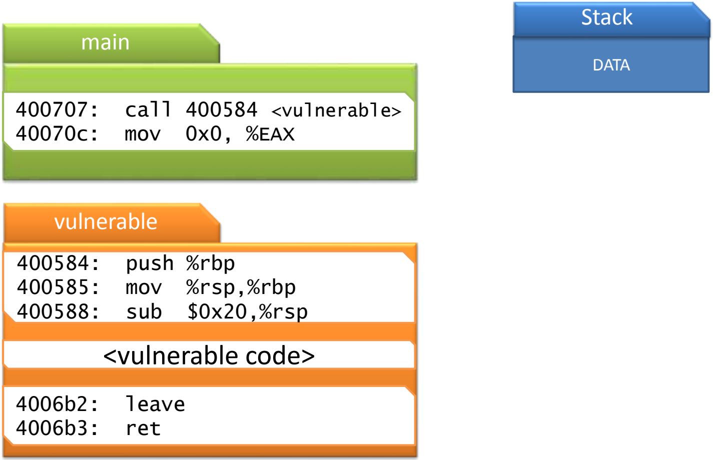
- Why Instructions-Level Monitoring (ILM) ?

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
```

vulnerable

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

<vulnerable code>

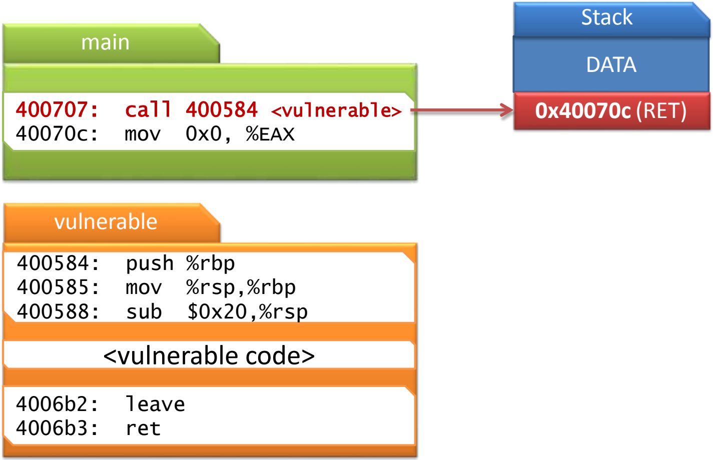
```
4006b2:   leave
4006b3:   ret
```

Stack

DATA

Motivation

▸ Why Instructions-Level Monitoring (ILM) ?



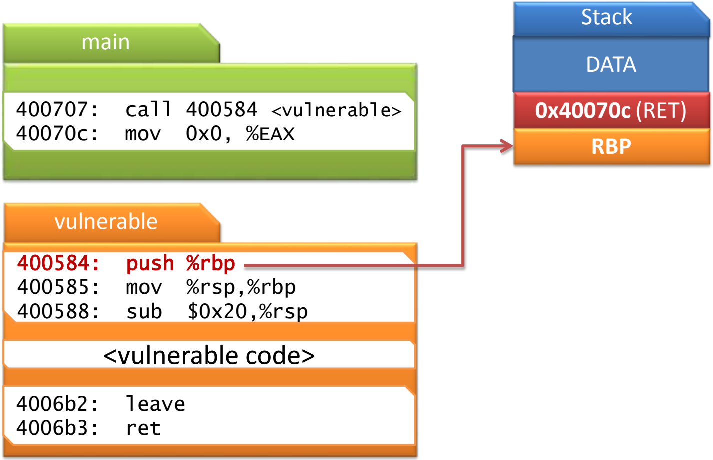
```
400707:  call 400584 <vulnerable>
40070c:  mov  0x0, %EAX
```

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

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

Motivation

▸ Why Instructions-Level Monitoring (ILM) ?



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

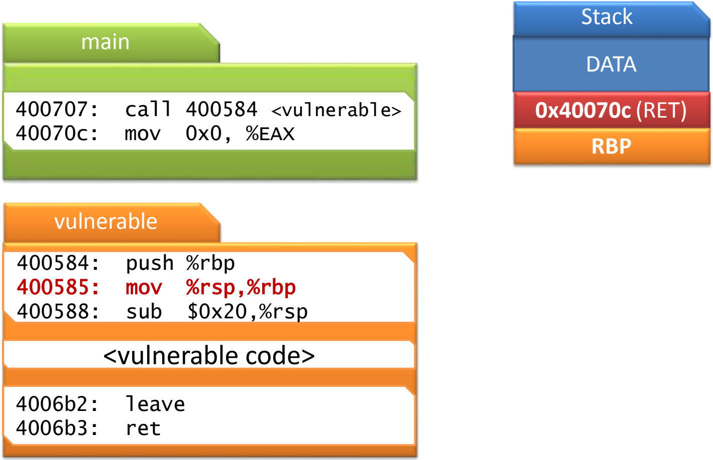
```
400584:   push %rbp
400585:   mov  %rsp,%rbp
400588:   sub  $0x20,%rsp
```

<vulnerable code>

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

Motivation

- Why Instructions-Level Monitoring (ILM) ?



main

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

vulnerable

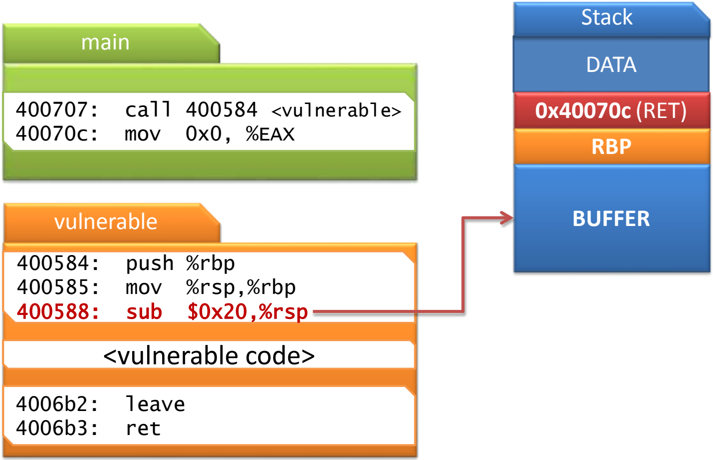
```
400584:   push %rbp
400585:   mov  %rsp,%rbp
400588:   sub  $0x20,%rsp
```

<vulnerable code>

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

Motivation

▸ Why Instructions-Level Monitoring (ILM) ?



main

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

vulnerable

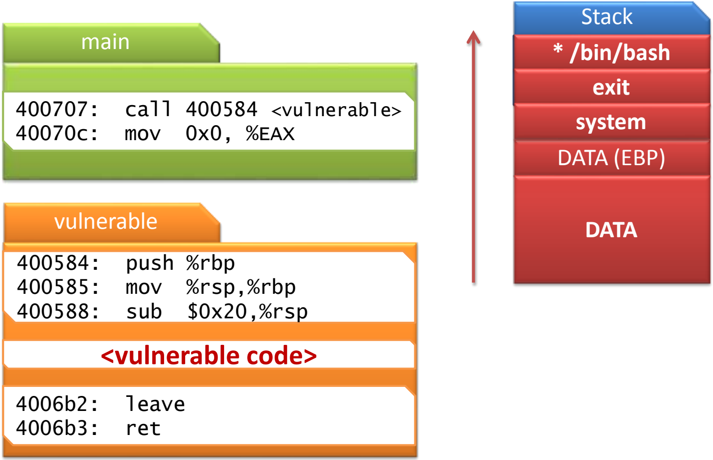
```
400584:   push %rbp
400585:   mov  %rsp,%rbp
400588:   sub  $0x20,%rsp
```

<vulnerable code>

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

Motivation

▸ Why Instructions-Level Monitoring (ILM) ?



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

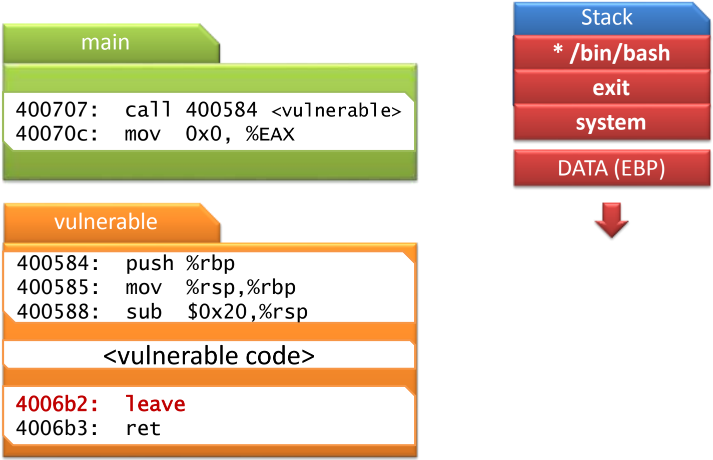
```
400584:   push %rbp
400585:   mov  %rsp,%rbp
400588:   sub  $0x20,%rsp
```

<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
```

<vulnerable code>

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

Stack

* /bin/bash

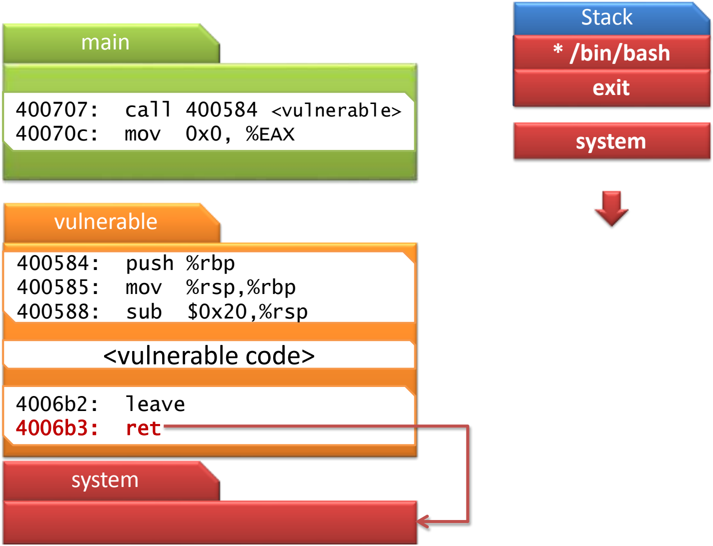
exit

system

DATA (EBP)

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

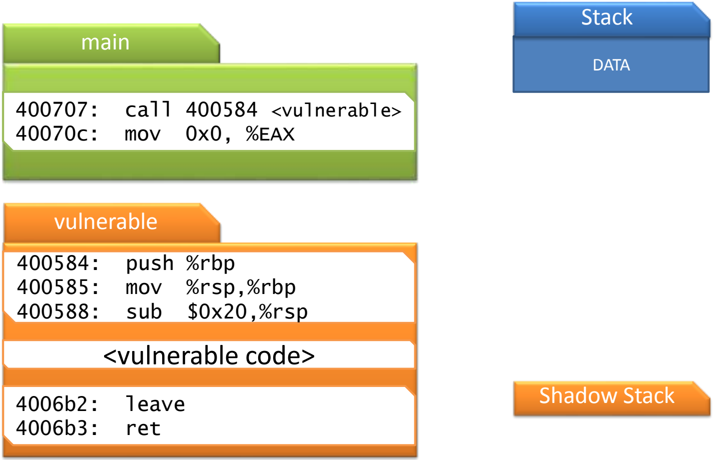
system

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
```

vulnerable

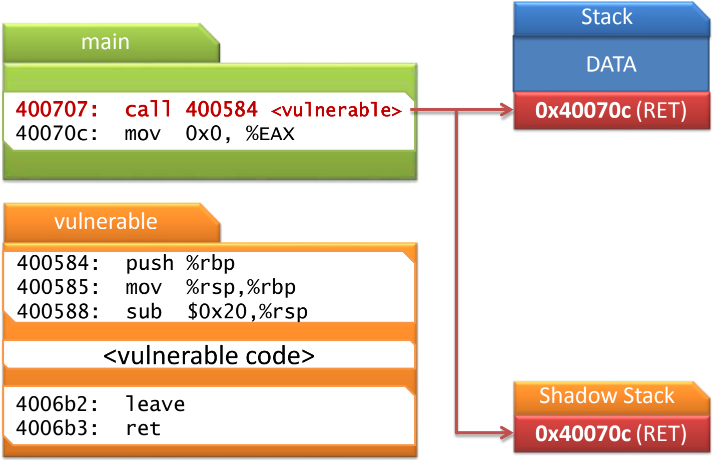
```
400584:   push %rbp
400585:   mov  %rsp,%rbp
400588:   sub  $0x20,%rsp
```

<vulnerable code>

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

Motivation

▸ Why Instructions-Level Monitoring (ILM) ?



main

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

vulnerable

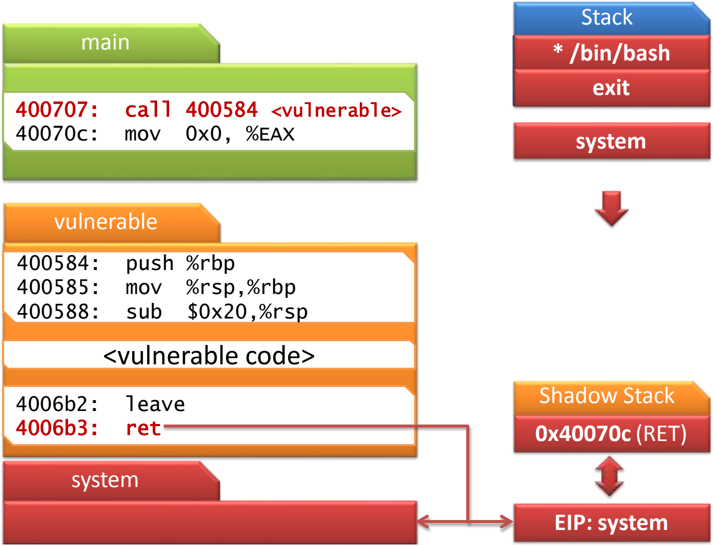
```
400584:  push %rbp
400585:  mov  %rsp,%rbp
400588:  sub  $0x20,%rsp
```

<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
```

<vulnerable code>

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

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.

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

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

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

Existing Approaches

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

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 **flexbility**.

Outline

1 Motivation

2 Performance Monitoring Counters (PMCs)

3 PMC-based Instruction-level Monitoring (ILM)

4 Experiments & Results

5 Summary

Performance Monitoring Counters (PMCs)

▸ Overview

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 Counters (PMCs)

Performance Monitoring on the x86 architecture

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

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

Outline

1. Motivation
2. Performance Monitoring Counters (PMCs)
3. PMC-based Instruction-level Monitoring (ILM)
4. Experiments & Results
5. Summary

PMC-based Instruction-level Monitoring (ILM)

▸ Trapping Performance Events

Question

How can we trap performance events to the hypervisor?

PMC-based Instruction-level Monitoring (ILM)

▸ Trapping Performance Events

Question

How can we trap performance events to the hypervisor?

Challenges

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

PMC-based Instruction-level Monitoring (ILM)

▸ Trapping Performance Events

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.

PMC-based Instruction-level Monitoring (ILM)

▸ Trapping Performance Events: Signal Generation

Set the PMC initially to

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

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

Set the PMC initially to

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

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

⇒ PMC will overflow after the desired number of events.

PMC-based Instruction-level Monitoring (ILM)

Set the PMC initially to

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

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.

Interrupt Generation

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

PMC-based Instruction-level Monitoring (ILM)

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.

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.

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.

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.

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.

PMC-based Instruction-level Monitoring (ILM)

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.

PMC-based Instruction-level Monitoring (ILM)

▸ Instruction Reconstruction (IR)

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

```
40f448:  mov    %r12,%rdi              ; <====== LAST EIP
40f44b:  mov    $0x20,%esi
40f450:  mov    %rbp,%rdx
40f453:  mov    %ecx,0x28(%rsp)
40f457:  mov    %r8b,0x10(%rsp)
40f45c:  mov    %r9,0x20(%rsp)
40f461:  add    %rbp,%r12              ; <====== CURRENT EIP
```

PMC-based Instruction-level Monitoring (ILM)

Instruction Reconstruction (IR)

What about branches?

```
40f24e:  pop    %r12            ; <====== LAST EIP
40f250:  pop    %r13
40f252:  pop    %r14
40f254:  pop    %r15
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

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.

LBR Stack

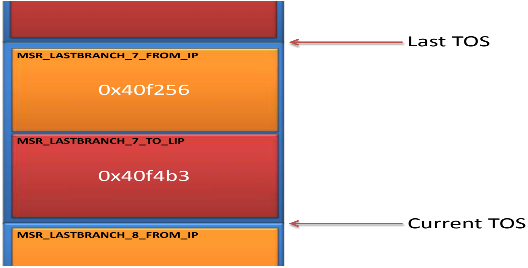
- Records the last taken **branches**
- Set of MSRs
  - A top-of-stack (TOS) pointer (`MSR_LASTBRANCH_TOS`)
  - A pair of MSRs for each branch that the stack can record: `MSR_LASTBRANCH_x_FROM_IP` ⇒ `MSR_LASTBRANCH_x_TO_LIP`
- The size of the LBR stack depends on the microarchitecture

⇒ 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)



MSR_LASTBRANCH_7_FROM_IP

0x40f256

MSR_LASTBRANCH_7_TO_LIP

0x40f4b3

MSR_LASTBRANCH_8_FROM_IP

Last TOS

Current TOS

Using the LBR Stack

```
40f24e:  pop     %r12                    ; <====== LAST EIP
40f250:  pop     %r13
40f252:  pop     %r14
40f254:  pop     %r15
40f256:  ret

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

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)

- 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**

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

Improving the Performance

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

Improving the Performance

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

Improving the Performance

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

Outline

1 Motivation

2 Performance Monitoring Counters (PMCs)

3 PMC-based Instruction-level Monitoring (ILM)

4 Experiments & Results

5 Summary

Summary

Contributions

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

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.

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**.

Summary

▸ Questions?

References I

- Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy.
Ropdefender: a detection tool to defend against return-oriented programming attacks.
In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

- Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee.
Ether: Malware Analysis via Hardware Virtualization Extensions.
In *Proceedings of the 15th ACM conference on Computer and Communications Security*, 2008.

- Tal Garfinkel, Keith Adams, and Andrew Warfield.
Compatibility is not transparency.
In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, 2007.

References II

- Intel Corporation.
  Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide, 2011.

- Corey Malone, Mohamed Zahran, and Ramesh Karri.
  Are hardware performance counters a cost effective way for integrity checking of programs.
  In *Proceedings of the sixth ACM workshop on Scalable Trusted Computing*, 2011.

- Jonas Pfoh, Christian Schneider, and Claudia Eckert.
  Exploiting the x86 Architecture to Derive Virtual Machine State Information.
  In *Proceedings of the fourth international conference on Emerging Security Information, Systems and Technologies*, 2010.

References III

Jonas Pfoh, Christian Schneider, and Claudia Eckert.
Nitro: Hardware-based System Call Tracing for Virtual Machines.
Advances in Information and Computer Security, 2011.