Using Hardware Features for Increased Debugging Transparency

Taken from:

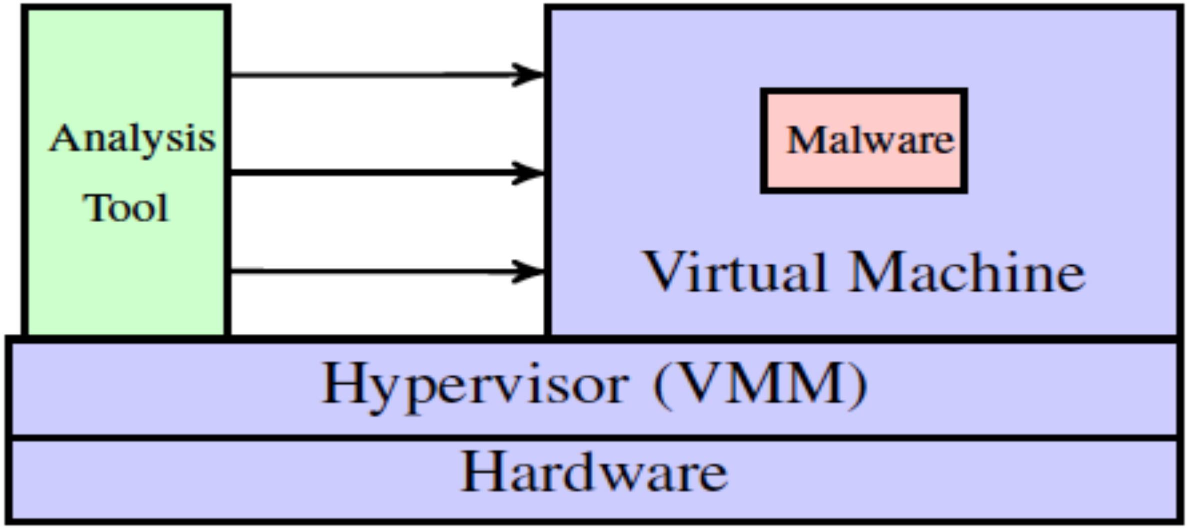
Instructor: Kun Sun, Ph.D.

College of William and Mary

Overview

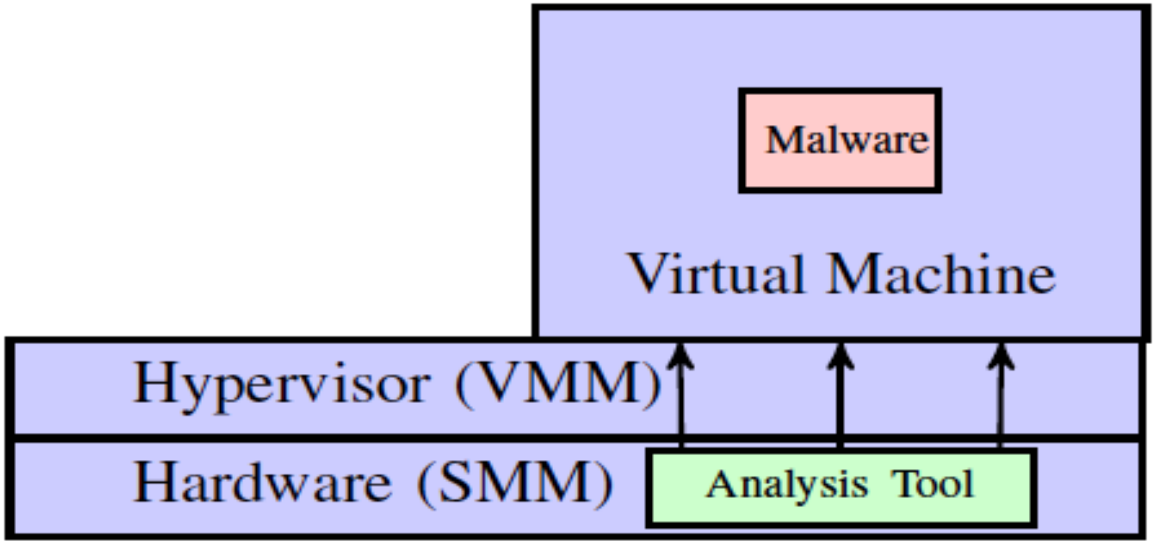
- Background
    - Traditional Malware Analysis
    - System Management Mode (SMM)
- System Architecture
- Performance Analysis
- Conclusion

Traditional Malware Analysis



- Using virtualization technology to create an isolated execution environment for malware debugging
- Running malware inside a VM
- Running analysis tools outside a VM

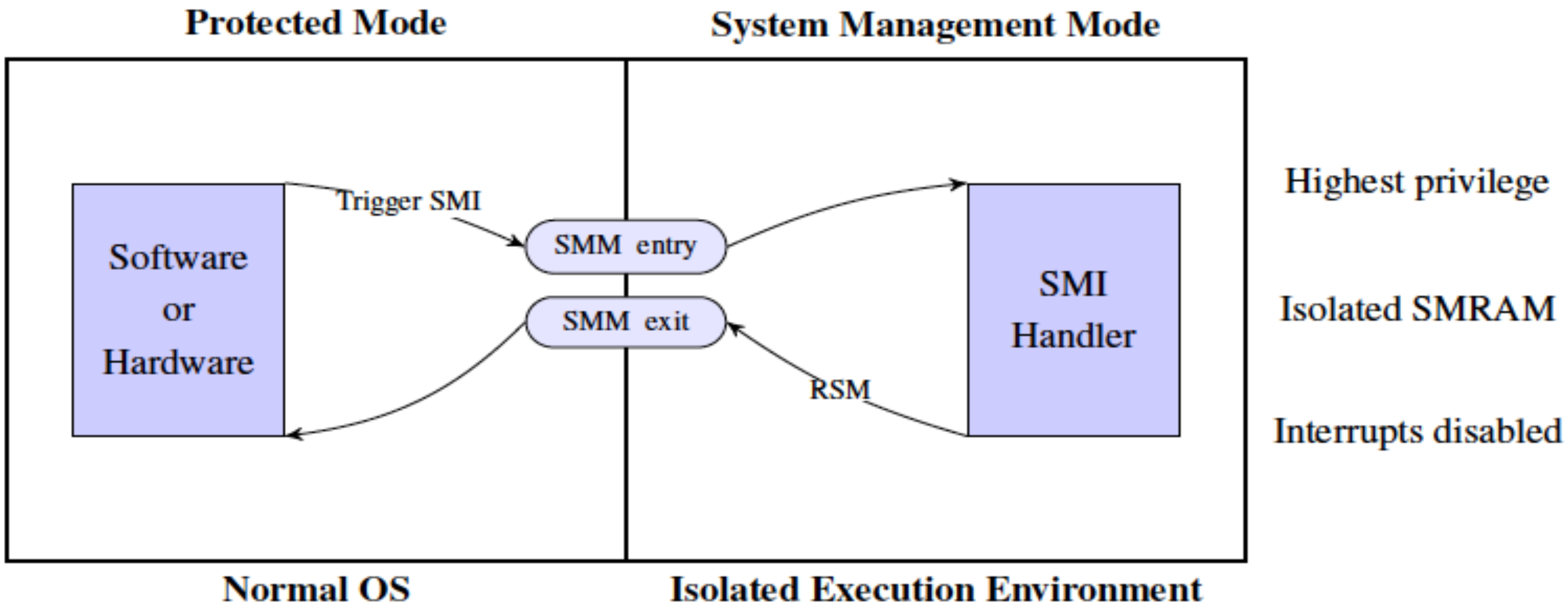
Our Approach



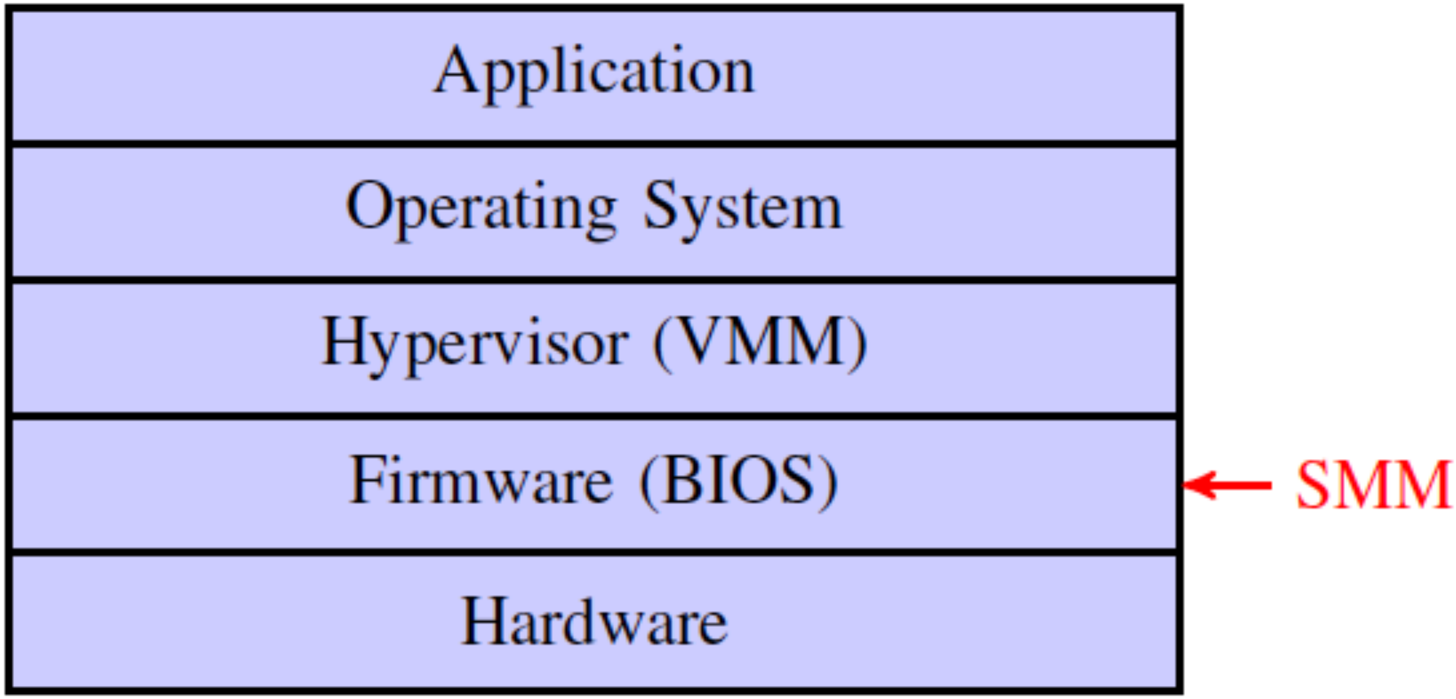
- We present a bare-metal debugging system called MalT that leverages System Management Mode for malware analysis
- Uses SMM as a hardware isolated execution environment to run analysis tools and can debug hypervisors

System Management Mode

- Approaches for Triggering a System Management Interrupt (SMI)
  - Software-based: Write to an I/O port specified by Southbridge datasheet (e.g., 0x2B for Intel)
  - Hardware-based: Network card, keyboard, hardware timers



Software Layer



Application

Operating System

Hypervisor (VMM)

Firmware (BIOS)

Hardware

SMM

System Architecture

- Traditionally malware debugging uses virtualization or emulation
- MalT debugs malware on a bare-metal machine, and remains transparent in the presence of existing anti-debugging, anti-VM, and anti-emulation techniques.

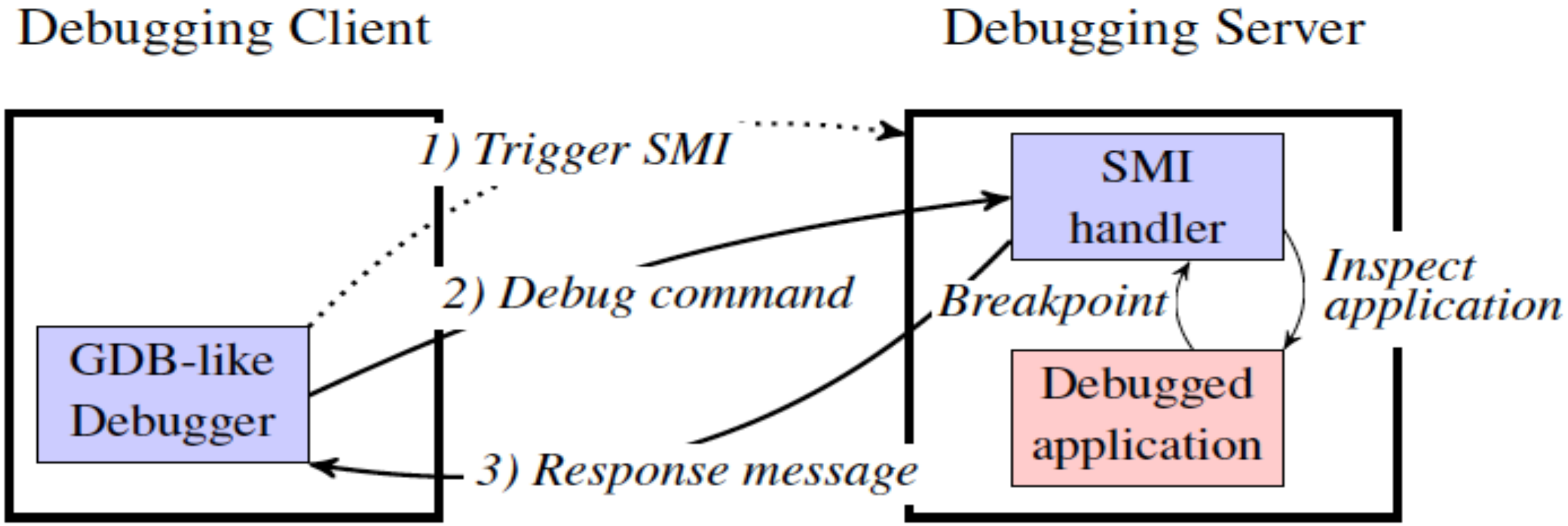


Figure : Architecture of MalT

Basic Debug Commands

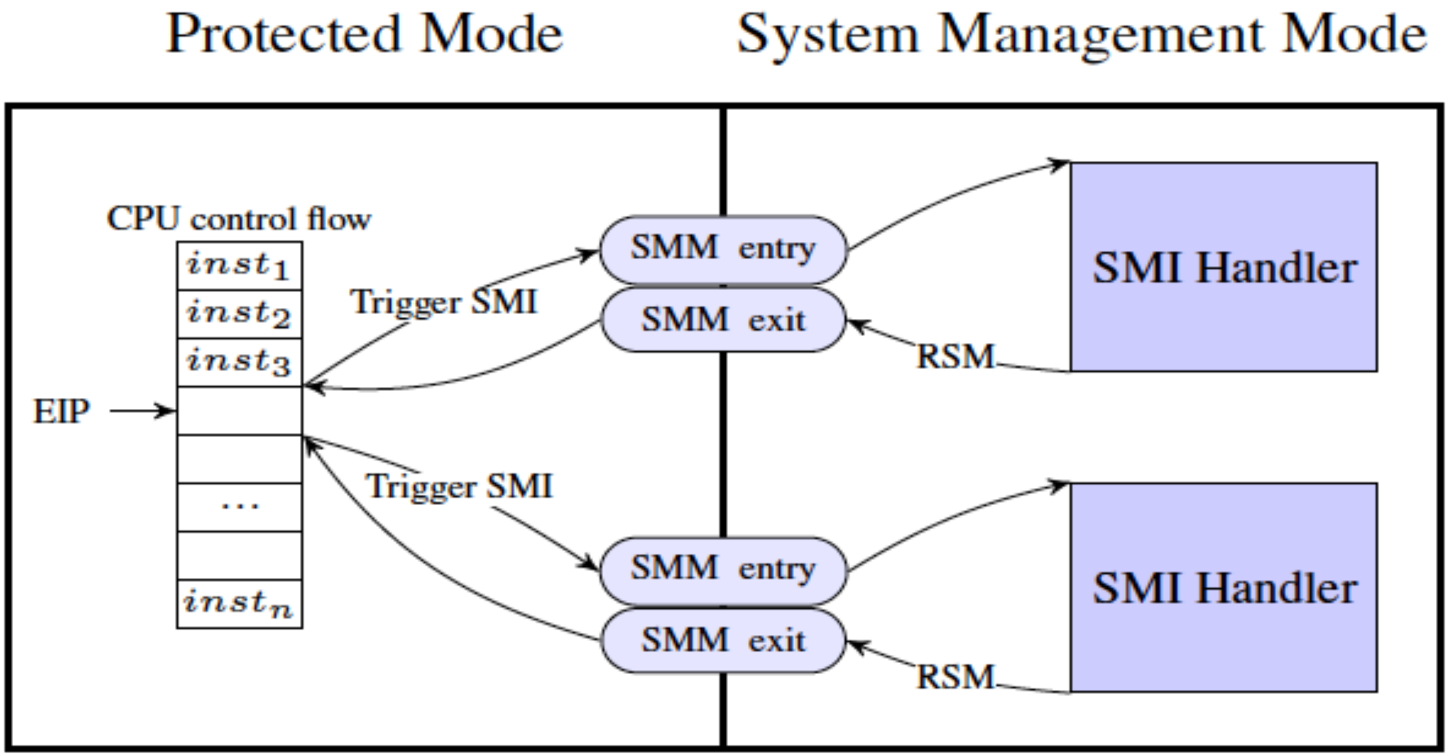
- R: read all registers
- Wr1r2…rn: write to certain register
- mAAAALLLL: read from particular memory address
- sAAAALLLL: write to particular memory address
- BAAAA: set a new breakpoint
- KAAAA: remove a breakpoint
- X: clear all breakpoints
- C: continue execution after a breakpoint
- SI, SB, SF, SN: stepping command

Debugging memory needs to fill the semantic gaps, since all the addresses are virtual addresses.

Breakpoints in MalT

- Software-based breakpoints are not stealthy.
- We use hardware-based breakpoints
- For each Protected Mode instruction, the SMI handler takes the following steps:
  1. Check if the target application is the running thread when the SMI is triggered
  2. Check if the current EIP equals to a stored breakpoint address
  3. Start to count instructions in the performance counter, and set the corresponding performance counter to the maximum value
  4. Configure LAPIC so that the performance counter overflow generates an SMI.

Step-by-step Debugging in MalT



- Debugging program instruction-by-instruction
- Using performance counters to trigger an SMI for each instruction

Performance Analysis

- Testbed Specification
  - Motherboard: ASUS M2V-MX SE
  - CPU: 2.2 GHz AMD LE-1250
  - Chipsets: AMD K8 Northbridge + VIA VT8237r Southbridge
  - BIOS: Coreboot + SeaBIOS

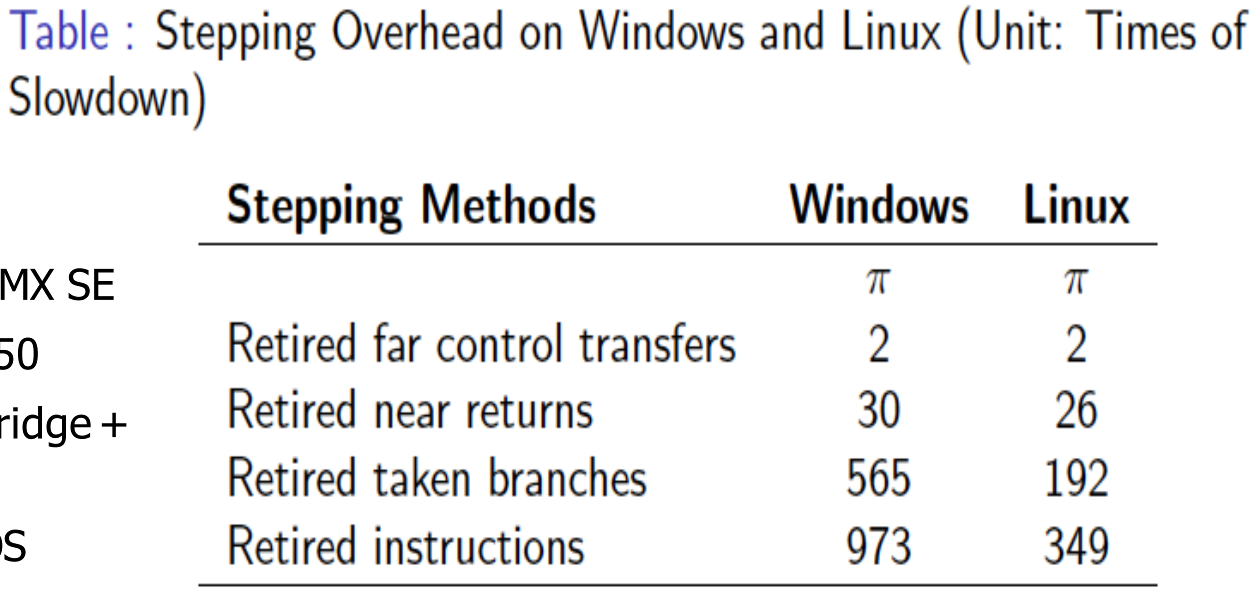


Table : Stepping Overhead on Windows and Linux (Unit: Times of Slowdown)

Stepping Methods	**Windows**	**Linux**
	π	π
Retired far control transfers	2	2
Retired near returns	30	26
Retired taken branches	565	192
Retired instructions	973	349

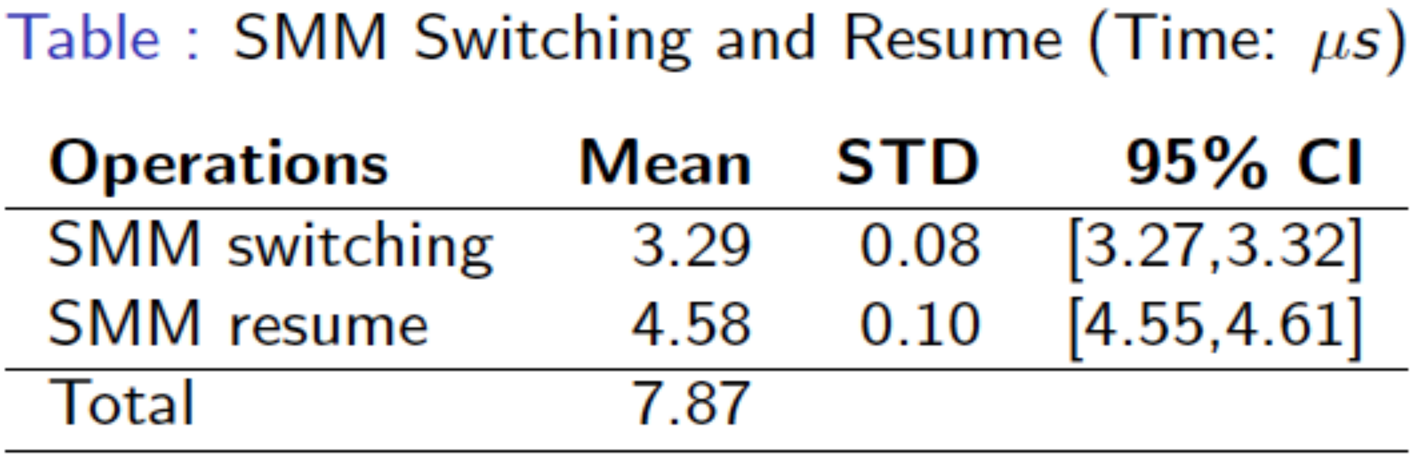


Table : SMM Switching and Resume (Time: μs)

Operations	**Mean**	**STD**	**95% CI**
SMM switching	3.29	0.08	[3.27,3.32]
SMM resume	4.58	0.10	[4.55,4.61]
Total	7.87		

Conclusion

- Benefit: Able to mitigate existing anti-debugging, anti-VM, and anti-emulation techniques
- Limitations of MaLT
  - SMM-based / ring-2 based rootkits
  - External timer based timing
  - Kernel exploits that can inspect LAPCI
  - Kernel exploits that mutate data structures to confuse MaLT