Transparent System Introspection in Support of Analyzing Stealthy Malware

Kevin Leach
PhD Dissertation
kjl2y@virginia.edu

November 30, 2016

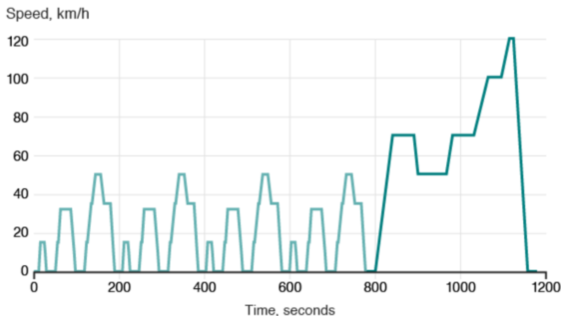
University of Virginia Engineering

Analogy: Volkswagen Scandal

- Volkswagen cheated on emissions test (over 10x EPA requirements)

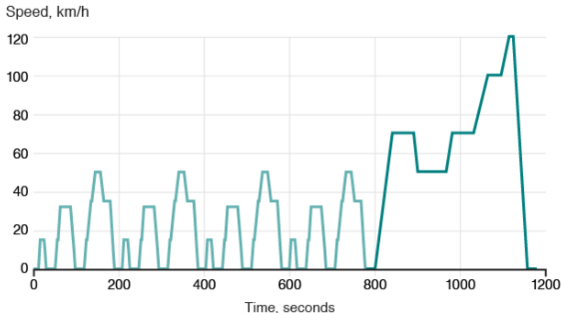
Analogy: Volkswagen Scandal

- Volkswagen cheated on emissions test (over 10x EPA requirements)
- **Car was able to detect the test**



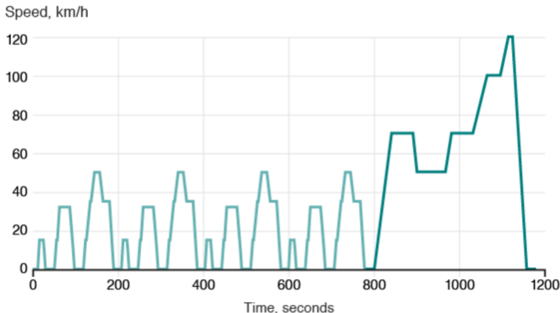
Analogy: Volkswagen Scandal

Volkswagen exploited the measurable difference between the EPA test and normal driving



Analogy: Volkswagen Scandal

Volkswagen exploited the measurable difference between the EPA test and normal driving



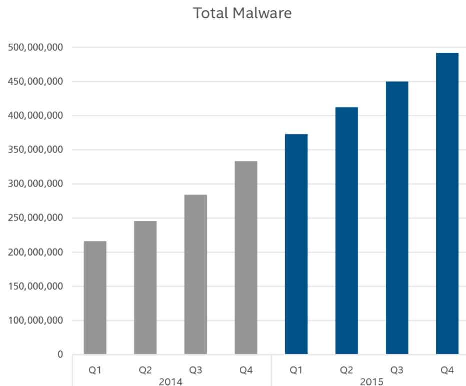
What about malware that detects analysis tools?

Overview

1. Motivation
2. Background
   - Stealthy Malware Analysis and Artifacts
   - Introspection
3. Hardware-assisted introspection and debugging
   - Transparently acquire program data in two ways:
     3.1 MALT: Using SMM for Debugging
     3.2 LO-PHI: Using DMA over PCIe for Introspection
4. Transparent program introspection
   - HOPS: Limits of transparent program introspection
5. Conclusion

Motivation

- Symantec blocked an average of 250k attacks per day during 2014
- McAfee reported 40M new malware samples during each quarter of 2015
- Kaspersky reported 320k new threats per day in 2015



Source: McAfee Labs, 2016.

Malware Analysis Challenges

- Analysts want to quickly identify malware behavior

University of Virginia Engineering

Malware Analysis Challenges

- Analysts want to quickly identify malware behavior
  - What damage does it do?

Malware Analysis Challenges

- Analysts want to quickly identify malware behavior
  - What damage does it do?
  - How does it infect a system?

University of Virginia Engineering

Malware Analysis Challenges

- Analysts want to quickly identify malware behavior
  - What damage does it do?
  - How does it infect a system?
  - How do we defend against it?

Introspection

- Understanding program behavior

Introspection

- Understanding program behavior
- Debugger *introspects* program to access raw data
  - Read variables
  - Reconstruct stack traces
  - Read disk activity

Introspection

- Understanding program behavior
- Debugger *introspects* program to access raw data
  - Read variables
  - Reconstruct stack traces
  - Read disk activity
- Analyst infers behavior of a sample from interpreting this raw data

Introspection

- Understanding program behavior
- Debugger *introspects* program to access raw data
  - Read variables
  - Reconstruct stack traces
  - Read disk activity
- Analyst infers behavior of a sample from interpreting this raw data
- Virtual Machine Introspection (VMI)
  - Plugin for a Virtual Machine Manager (slowdown)
  - Helper process inside guest VM (detectable process)

Introspection

- Understanding program behavior
- Debugger *introspects* program to access raw data
  - Read variables
  - Reconstruct stack traces
  - Read disk activity
- Analyst infers behavior of a sample from interpreting this raw data
- Virtual Machine Introspection (VMI)
  - Plugin for a Virtual Machine Manager (slowdown)
  - Helper process inside guest VM (detectable process)

But what if the program can detect our introspection tool?

UNIVERSITY of VIRGINIA ENGINEERING

Artifacts and Stealthy Malware

- Adversary achieves stealth by using *artifacts* to detect analysis tools

Artifacts and Stealthy Malware

- Adversary achieves stealth by using *artifacts* to detect analysis tools
  - Measurable "tells" introduced by analysis

Artifacts and Stealthy Malware

- Adversary achieves stealth by using *artifacts* to detect analysis tools
  - Measurable "tells" introduced by analysis
  - *Timing (nonfunctional) artifacts* — overhead incurred by analysis
    - single-stepping instructions with debugger is slow
    - imperfect VM environment does not match native speed

Artifacts and Stealthy Malware

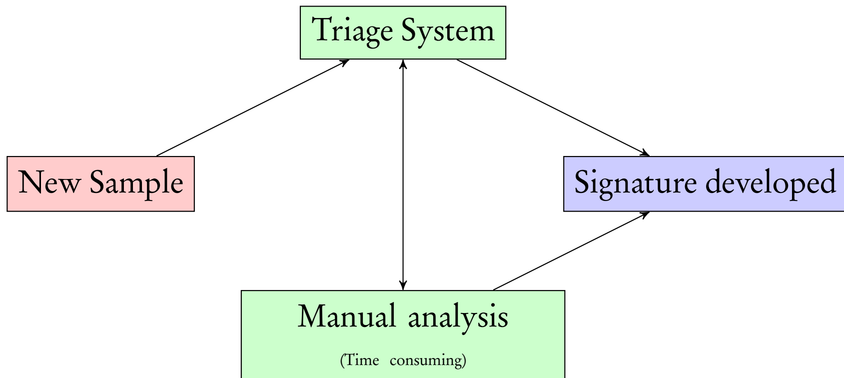
- Adversary achieves stealth by using *artifacts* to detect analysis tools
  - Measurable "tells" introduced by analysis
  - *Timing (nonfunctional) artifacts* — overhead incurred by analysis
    - single-stepping instructions with debugger is slow
    - imperfect VM environment does not match native speed
  - *Functional artifacts* — features introduced by analysis
    - `isDebuggerPresent()` — legitimate feature abused by adversaries
    - Incomplete or unfaithful emulation of some instructions by VM
    - Device names (hard disk named "VMWare disk")

Artifacts and Stealthy Malware

- Adversary achieves stealth by using *artifacts* to detect analysis tools
  - Measurable “tells” introduced by analysis
  - *Timing (nonfunctional) artifacts* — overhead incurred by analysis
    - single-stepping instructions with debugger is slow
    - imperfect VM environment does not match native speed
  - *Functional artifacts* — features introduced by analysis
    - `isDebuggerPresent()` — legitimate feature abused by adversaries
    - Incomplete or unfaithful emulation of some instructions by VM
    - Device names (hard disk named “VMWare disk”)

Significant effort to fully analyze each stealthy sample

Malware Analysis



UNIVERSITY of VIRGINIA ENGINEERING

Transparency

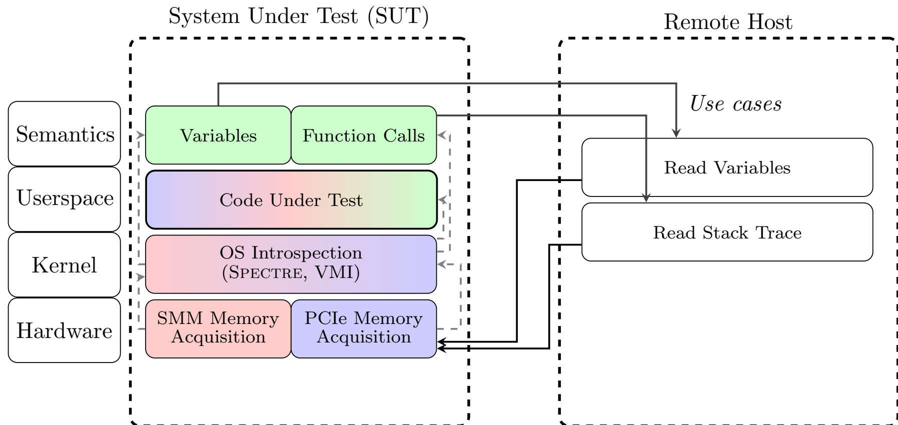
- We want accurate introspection even in the presence of stealthy malware
  - We want *transparency* — no artifacts produced by analysis

We want transparent system introspection tools to solve this 'debugging transparency problem'

Thesis Statement

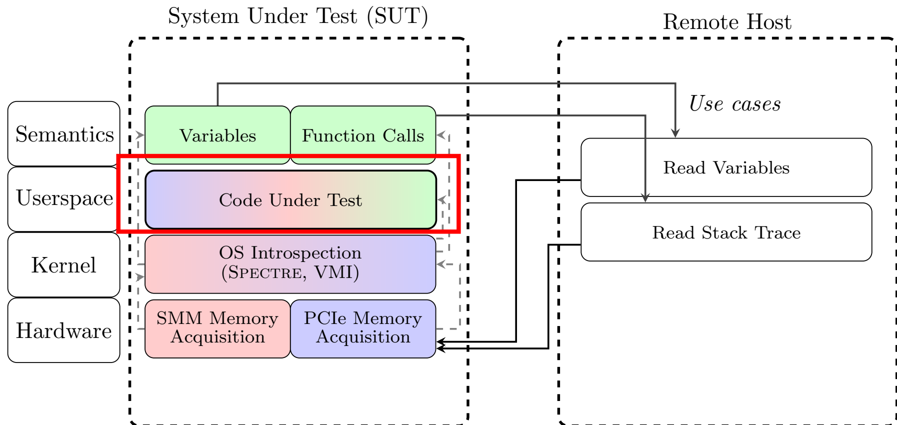
- It is possible to develop a transparent system introspection tool by independently considering timing and functional artifacts

Architecture



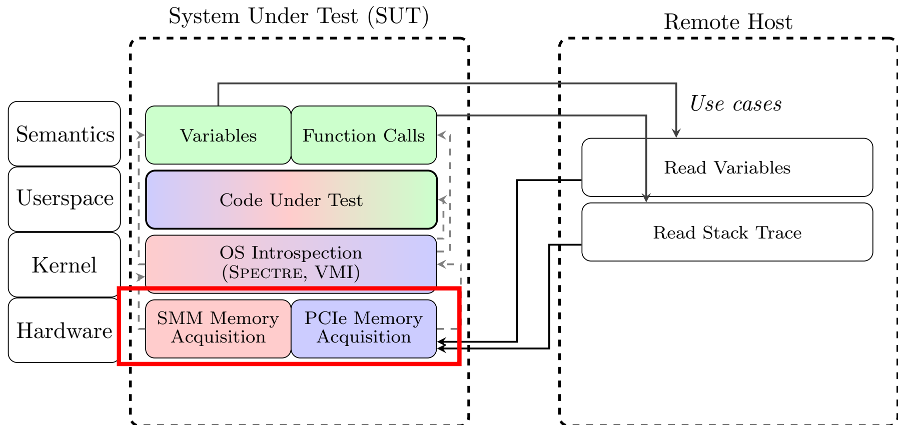
- Component 1 – Hardware-assisted memory acquisition via PCI-e
- Component 2 – Hardware-assisted memory acquisition via SMM
- Component 3 – Transparent program introspection

Architecture



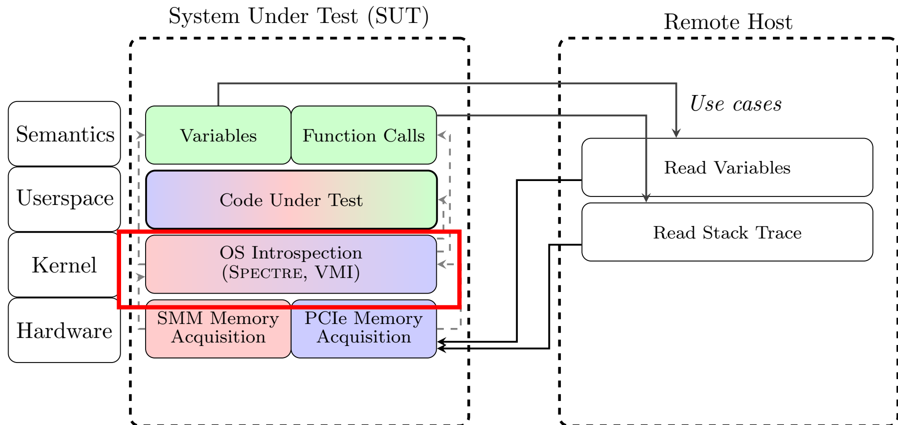
- Component 1 – Hardware-assisted memory acquisition via PCI-e
- Component 2 – Hardware-assisted memory acquisition via SMM
- Component 3 – Transparent program introspection

Architecture



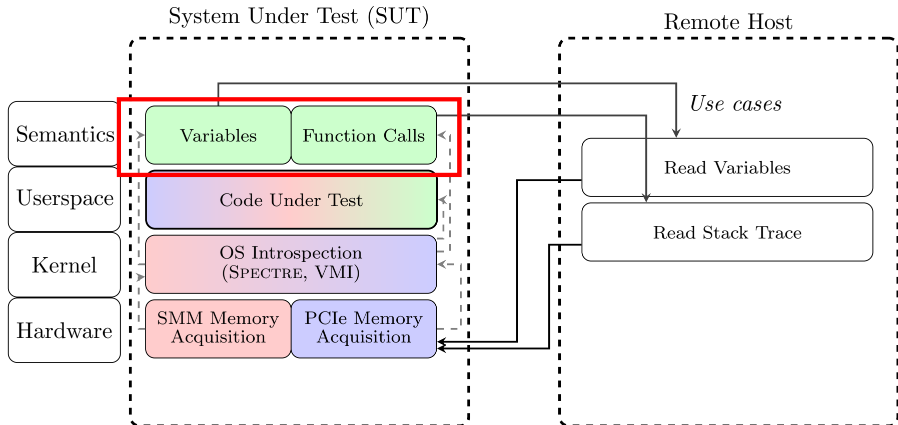
- Component 1 – Hardware-assisted memory acquisition via PCI-e
- Component 2 – Hardware-assisted memory acquisition via SMM
- Component 3 – Transparent program introspection

Architecture



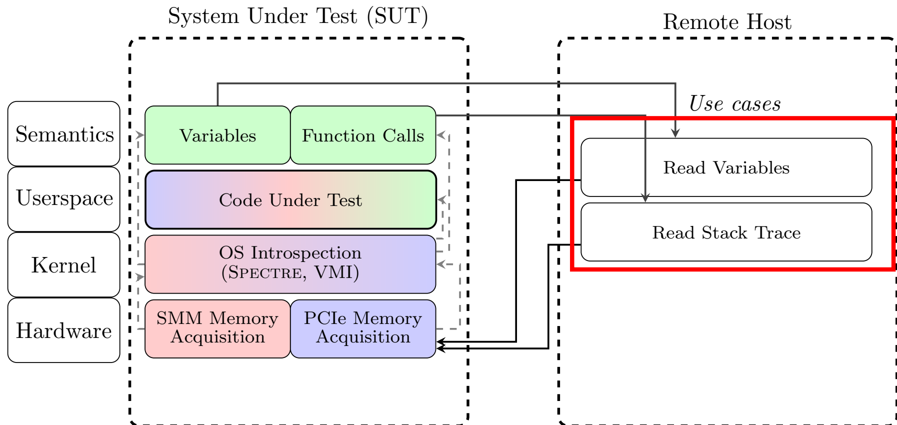
- Component 1 – Hardware-assisted memory acquisition via PCI-e
- Component 2 – Hardware-assisted memory acquisition via SMM
- Component 3 – Transparent program introspection

Architecture



- Component 1 – Hardware-assisted memory acquisition via PCI-e
- Component 2 – Hardware-assisted memory acquisition via SMM
- Component 3 – Transparent program introspection

Architecture



- Component 1 – Hardware-assisted memory acquisition via PCI-e
- Component 2 – Hardware-assisted memory acquisition via SMM
- Component 3 – Transparent program introspection

Overview

1. Motivation
2. Background
   - Stealthy Malware Analysis and Artifacts
   - Introspection
   - Architecture
3. Hardware-assisted introspection and debugging
   - Transparently acquire program data in two ways:
     - 3.1 MALT: Using SMM for Debugging
     - 3.2 LO-PHI: Using DMA over PCIe for Introspection
4. Transparent program introspection
   - HOPS: Limits of transparent program introspection
5. Conclusion

UNIVERSITY of VIRGINIA ENGINEERING

Hardware-Assisted Introspection

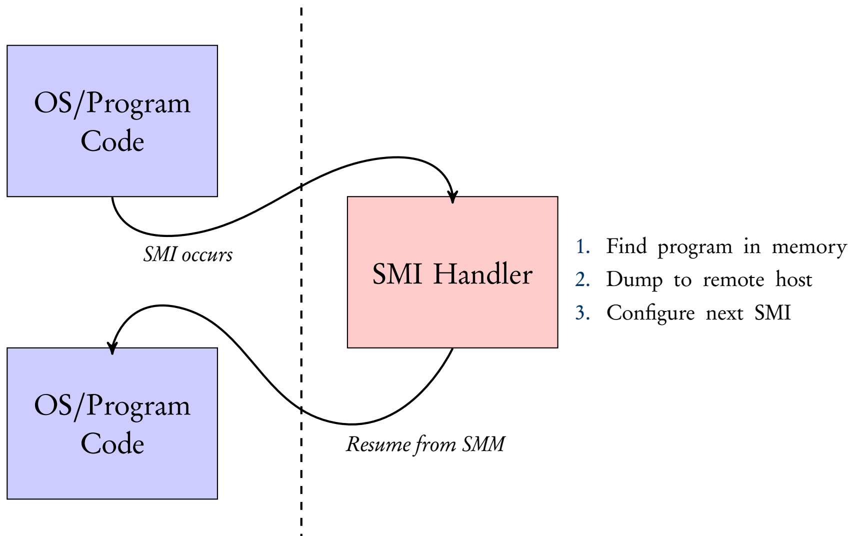
- Two approaches
  1. MALT, using System Management Mode (SMM)
     - Significant timing artifacts
     - No functional artifacts
  2. LO-PHI, FPGA-based custom circuit
     - Few timing artifacts
     - Increased functional artifacts (e.g., DMA access performance counter)

UNIVERSITY of VIRGINIA ENGINEERING

SMM-based Memory Acquisition

- Intel x86 feature provides small, OS-transparent and -agnostic, trusted computing base
- Custom SMI Handler executed in SMM
  - Code stored in System Management RAM (SMRAM)
  - Trust only the BIOS
  - Logically atomically executed transparently from OS

SMM Architecture



SMM Experiments

- Measure time elapsed during each SMM-related operation
  1. SMM Switch after SMI
  2. Find target program
  3. Configure next SMI
  4. Switch back from SMM

  (Under 12μs total, 8μs from switching)
- Measure system overhead when configuring SMIs:
  - Cause SMIs every retired instruction
- Demonstrate feasibility of approach to stealthy malware
  - Consider recent packers

University of Virginia Engineering

SMM Overhead

Stepping method	Slowdown			
	Windows		Linux	
	π	*gzip*	π	*gzip*
Without MALT	1.00x	1.00x	1.00x	1.00x
far control transfers	1.38x	1.36x	1.46x	1.42x
near returns	46.2x	39.1x	36.1x	34.7x
taken mispredicted	96.5x	40.2x	77.7x	81.2x
taken branches	634x	935x	280x	903x
mispredicted branches	99.6x	149x	45.4x	138x
branches	745x	1196x	290x	1033x
instructions	1021x	1519x	492x	1369x

For reference, the state-of-the-art Ether yields an overhead of 3000x for a similar operation.

University of Virginia Engineering

SMM Overhead

Stepping method	Slowdown			
	Windows		Linux	
	π	*gzip*	π	*gzip*
Without MALT	1.00x	1.00x	1.00x	1.00x
far control transfers	1.38x	1.36x	1.46x	1.42x
near returns	46.2x	39.1x	36.1x	34.7x
taken mispredicted	96.5x	40.2x	77.7x	81.2x
taken branches	634x	935x	280x	903x
mispredicted branches	99.6x	149x	45.4x	138x
branches	745x	1196x	290x	1033x
instructions	1021x	1519x	492x	1369x

For reference, the state-of-the-art Ether yields an overhead of 3000x for a similar operation.

SMM vs. Packers

Packing Tool	*MALT*	OllyDbg	DynamoRIO	VMware Fusion
UPX v3.08	✓	✓	✓	✓
Obsidium v1.4	✓	✗ (access violation)	✗ (segfault)	✓
ASPack v2.29	✓	✓	✓	✓
Armadillo v2.01	✓	✗ (access violation)	✗ (crash)	✗(crash)
Themida v2.2.3.0	✓	✗ (exception)	✗ (exception)	✗(no VM)
RLPack v1.21	✓	✓	✓	✓
PELock v1.0694	✓	✗	✗ (segfault)	✓
VMProtect v2.13.5	✓	✗	✓	✗ (crash)
eXPressor v1.8.0.1	✓	✗	✗ (segfault)	✗ (crash)
PECompact v3.02.2	✓	✗ (access violation)	✓	✓

SMM vs. Packers

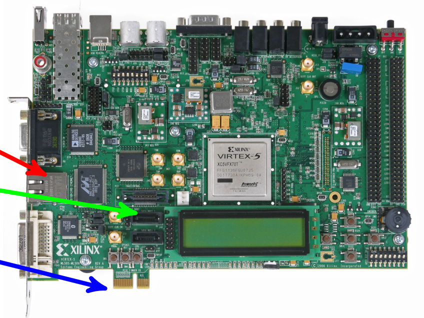
Packing Tool	*MALT*	OllyDbg	DynamoRIO	VMware Fusion
UPX v3.08	✓	✓	✓	✓
Obsidium v1.4	✓	✗ (access violation)	✗ (segfault)	✓
ASPack v2.29	✓	✓	✓	✓
Armadillo v2.01	✓	✗ (access violation)	✗ (crash)	✗(crash)
Themida v2.2.3.0	✓	✗ (exception)	✗ (exception)	✗(no VM)
RLPack v1.21	✓	✓	✓	✓
PELock v1.0694	✓	✗	✗ (segfault)	✓
VMProtect v2.13.5	✓	✗	✓	✗ (crash)
eXPressor v1.8.0.1	✓	✗	✗ (segfault)	✗ (crash)
PECompact v3.02.2	✓	✗ (access violation)	✓	✓

Overview

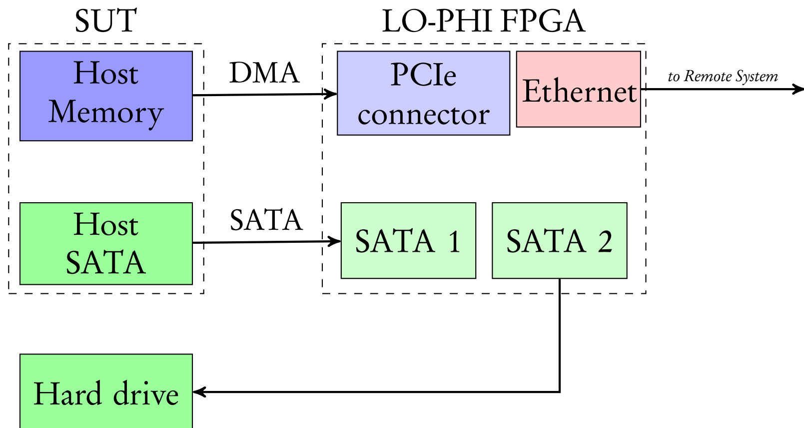
1. Motivation
2. Background
   - Stealthy Malware Analysis and Artifacts
   - Introspection
   - Architecture
3. Hardware-assisted introspection and debugging
   - Transparently acquire program data in two ways:
     - 3.1 MALT: Using SMM for Debugging
     - 3.2 LO-PHI: Using DMA over PCIe for Introspection
4. Transparent program introspection
   - HOPS: Limits of transparent program introspection
5. Conclusion

FPGA Memory and Disk Acquisition

- Use Xilinx ML507
  - Gigabit Ethernet
  - 2x SATA connectors
  - PCI Express connector



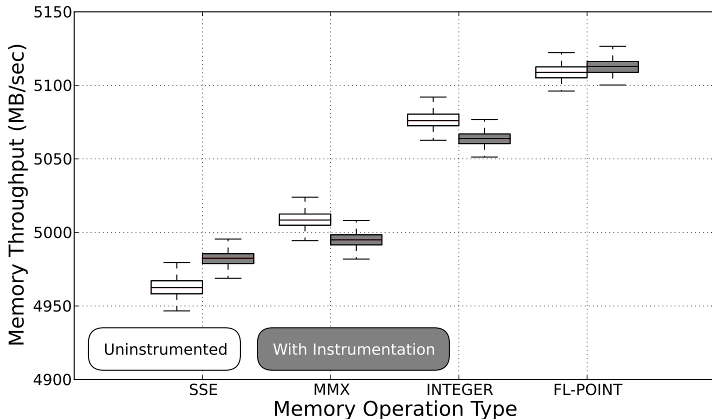
LO-PHI Architecture



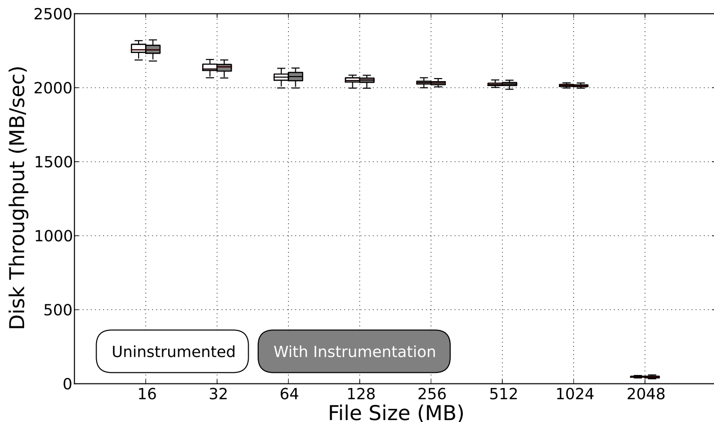
LO-PHI Experimentation

- Compare performance of SUT when LO-PHI is present vs. absent on indicative workloads
- Memory throughput: use `RAMSpeed` benchmarks
- Disk throughput: Use `IOZone` benchmarks

LO-PHI Memory Overhead



LO-PHI Disk Overhead



University of Virginia Engineering

LO-PHI Case Studies

- **Paranoid Fish** (stealthy malware proof-of-concept)
  - Failed to detect LO-PHI
  - **Comparison:** State-of-the-art Anubis and Cuckoo were both detected via virtualization artifacts
- **Labeled Malware** (429 coarsely-labeled samples)
  - LO-PHI correctly matched labels

Technique Employed	# Samples
Wait for keyboard	3
BIOS-based	6
Hardware id-based	28
Processor feature-based	62
Exception-based	79
Timing-based	251

University of Virginia Engineering

LO-PHI Case Studies (2)

- **More Labeled Malware** (213 well-labeled samples)
  - Blind analysis identified various behaviors, all of which were confirmed by ground truth
- **Unlabeled Malware** (1091 samples)
  - Used LO-PHI to study behavior of samples

Observed Behavior	Number of Samples
Created new process(es)	765
Opened socket(s)	210
Started service(s)	300
Loaded kernel modules	20
Modified GDT	58
Modified IDT	10

MALT and LO-PHI Summary

- Two alternatives to hardware-assisted introspection
  - MALT uses SMM to achieve low functional artifacts (but causes overhead)
  - LO-PHI uses custom FPGA hardware to achieve low overhead
    (but exposes minimal functional artifacts)
- Implemented and demonstrated the feasibility of prototypes based on both alternatives
- MALT and LO-PHI both provide useful raw introspection data *transparently*

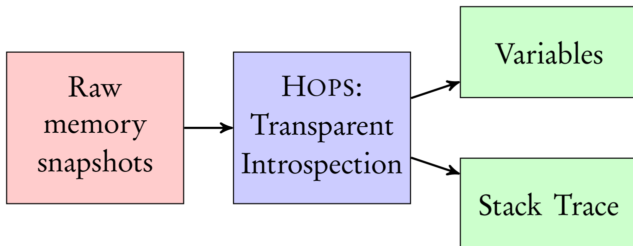
University of Virginia Engineering

Overview

1. Motivation
2. Background
   - Stealthy Malware Analysis and Artifacts
   - Introspection
   - Architecture
3. Hardware-assisted introspection and debugging
   - Transparently acquire program data in two ways:
     - 3.1 MALT: Using SMM for Debugging
     - 3.2 LO-PHI: Using DMA over PCIe for Introspection
4. Transparent program introspection
   - HOPS: Limits of transparent program introspection
5. Conclusion

Transparency and the Semantic Gap

- MALT and LO-PHI both provide raw introspection data transparently
  - Periodic snapshots of memory (and potentially disk) via SMM or PCIe



Transparent Introspection

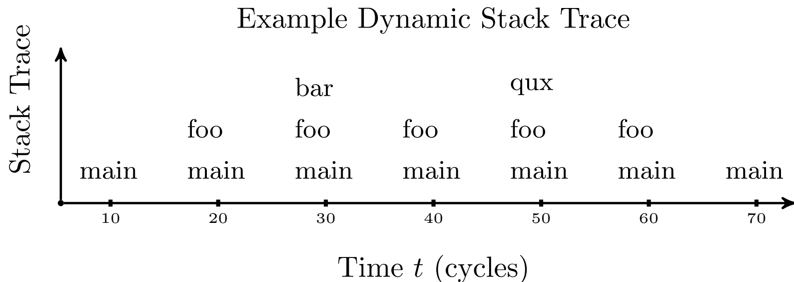
- Assume access to source code for ground truth
  - Two versions of binary
    - "Deployed" version represents sample being analyzed
    - "Instrumented" versions helps us hypothesize locations of semantic information
- Report fraction of variables correctly identified in the Deployed binary
- Report fraction of function call correctly identified in runtime stack trace

Transparent Introspection

- Assume access to source code for ground truth
  - Two versions of binary
    - "Deployed" version represents sample being analyzed
    - "Instrumented" versions helps us hypothesize locations of semantic information
- Report fraction of variables correctly identified in the Deployed binary
- Report fraction of function call correctly identified in runtime stack trace

- **What are the tradeoffs between maintaining transparency vs. fidelity of introspection**

Stack Trace Example



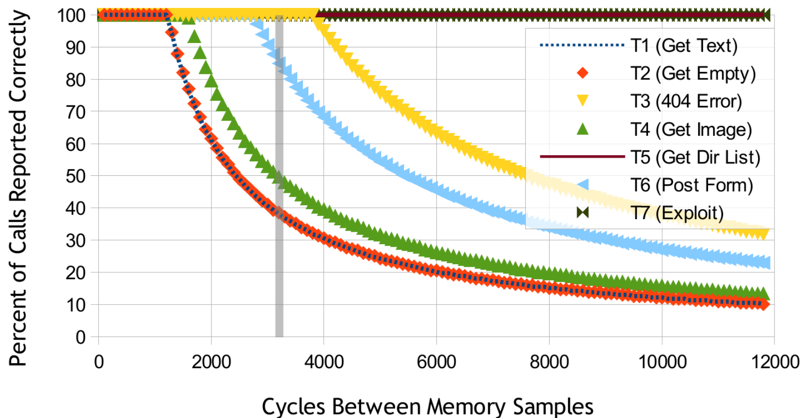
UNIVERSITY of VIRGINIA ENGINEERING

Introspection Experiments

- Consider indicative programs:
  - Wuftpd 2.6.0
  - Nullhttpd 0.5.0
- Run programs on indicative test cases
  1. Gather ground truth from instrumented binary
  2. Gather variable and stack trace information on deployed binary
     - Report fraction of variables correctly reported
     - Report stack trace as a function of sampling frequency[1]

[1] Recall we assume access to periodic snapshots of memory

Call Stack Experiment



University of Virginia Engineering

Variable Accuracy

	nullhttpd		wuftpd	
Locals	43%	133/306	46%	202/436
Stack	65%	168/260	56%	119/214
Globals	100%	77/77	92%	4218/4580
Overall	59%	378/643	90%	4539/5230

Human Study

- 30 participants
- 30 C code snippets
- 3 Program understanding questions (from Sillito et al.)

- Half given HOPS data (treatment)
- Half given gdb data (control)

- treatment group performed the same as control with significance
  - HOPS provides no worse information than gdb wrt code understanding **with the added transparency property**

HOPS Summary

- HOPS explores the tradeoff space between transparency and fidelity of output

Overview

1. Motivation
2. Background
   - Stealthy Malware Analysis and Artifacts
   - Introspection
   - Architecture
3. Hardware-assisted introspection and debugging
   - Transparently acquire program data in two ways:
     3.1 MALT: Using SMM for Debugging
     3.2 LO-PHI: Using DMA over PCIe for Introspection
4. Transparent program introspection
   - HOPS: Limits of transparent program introspection
5. Conclusion

Publications

	Supporting this dissertation
IEEE S&P 2015	Using Hardware Features for Increased Debugging Transparency
TDSC 2016	Towards Transparent Debugging
NDSS 2016	LO-PHI: Low-Observable Physical Host Instrumentation
SANER 2016	Towards Transparent Introspection
	Other systems security publications
AsiaCCS 2015	TrustLogin: Securing Password-Login on Commodity Operating Systems
ESORICS 2014	A Framework to Secure Peripherals at Runtime
DSN 2013	Spectre: A Dependable System Introspection Framework
DSN 2013	Barley: Combining Control Flow with Resource Consumption to Detect Jump-based ROP Attacks
	Other publications
BigComp 2016	A MapReduce Framework to Improve Template Matching Uncertainty
Ubicomp 2016	Assessing Social Anxiety Using GPS Trajectories and Point-Of-Interest Data
DSN 2016	An Uncrewed Aerial Vehicle Attack Scenario and Trustworthy Repair Architecture
TIST 2015	DAEHR: A Discriminant Analysis Framework for Electronic Health Record Data and an Application to Early Detection of Mental Health Disorders
IEEE Big Data 2016	M-SEQ: Early Detection of Anxiety and Depression via Temporal Orders of Diagnoses in Electronic Health Data

University of Virginia Engineering

Summary

- **Stealthy malware** uses **artifacts** to detect analysis environments
- **Transparent introspection** prevents malware from subverting analysis
- SMM-based ***MALT*** acquires memory snapshots with no functional artifacts
- FGPA-based **LO-PHI** acquires memory and disk snapshots with no timing artifacts
- ***HOPS*** computes useful semantic information from periodic snapshots

Bonus: Artifacts (1)

Anti-debugging	
API Call	Kernel32!IsDebuggerPresent returns 1 if a target process is being debugged ntdll!NtQueryInformationProcess: ProcessInformation field set to -1 if the process is being debugged kernel32!CheckRemoteDebuggerPresent returns 1 in debugger process NtSetInformationThread with ThreadInformationClass set to 0x11 will detach some debuggers kernel32!DebugActiveProcess to prevent other debuggers from attaching to a process
PEB Field	PEB!IsDebugged is set by the system when a process is debugged PEB!NtGlobalFlags is set if the process was created by a debugger
Detection	ForceFlag field in heap header (+0x10) can be used to detect some debuggers UnhandledExceptionFilter calls a user-defined filter function, but terminates in a debugging process TEB of a debugged process contains a NULL pointer if no debugger is attached; valid pointer if some debuggers are attached Ctrl-C raises an exception in a debugged process, but the signal handler is called without debugging Inserting a Rogue INT3 opcode can masquerade as breakpoints Trap flag register manipulation to thwart tracers If entryPoint RVA is set to 0, the magic MZ value in PE files is erased ZwClose system call with invalid parameters can raise an exception in an attached debugger Direct context modification to confuse a debugger 0x2D interrupt causes debugged program to stop raising exceptions Some In-circuit Emulators (ICEs) can be detected by observing the behavior of the undocumented 0xF1 instruction Searching for 0xCC instructions in program memory to detect software breakpoints TLS-callback to perform checks

UNIVERSITY of VIRGINIA ENGINEERING

Bonus: Known Artifacts (2)

Anti-virtualization	
VMWare	Virtualized device identifiers contain well-known strings *checkvm* software can search for VMWare hooks in memory Well-known locations/strings associated with VMWare tools
Xen	Checking the VMX bit by executing CPUID with EAX as 1 CPU errata: AH4 erratum
Other	LDTR register IDTR register (Red Pill) Magic I/O port (`0x5658`, 'VX') Invalid instruction behavior Using memory deduplication to detect various hypervisors including VMware ESX server, Xen, and Linux KVM
Anti-emulation	
Bochs	Visible debug port
QEMU	`cpuid` returns less specific information Accessing reserved MSR registers raises a General Protection (GP) exception in real hardware; QEMU does not Attempting to execute an instruction longer than 15 bytes raises a GP exception in real hardware; QEMU does not Undocumented `icebp` instruction hangs in QEMU, while real hardware raises an exception Unaligned memory references raise exceptions in real hardware; unsupported by QEMU Bit 3 of FPU Control World register is always 1 in real hardware, while QEMU contains a `0`
Other	Using CPU bugs or errata to create CPU fingerprints via public chipset documentation

University of Virginia Engineering

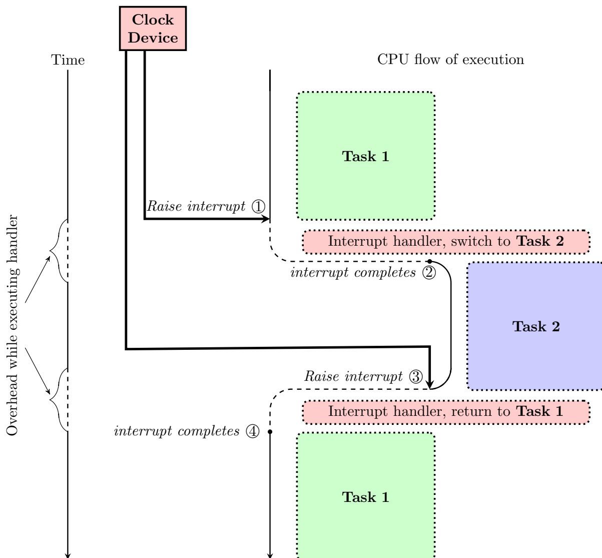
Bonus Slide: SMM-related Attacks

SMM Attacks	Solutions
Unlocked SMRAM	Set D_LCK bit
SMRAM reclaiming	Lock remapping and TOLUD registers
Cache poisoning	SMRR
Graphics aperture	Lock TOLUD
TSEG location	Lock TSEG base
Call/fetch outside of SMRAM	No call/fetch outside of SMRAM

Bonus: Caching Issues

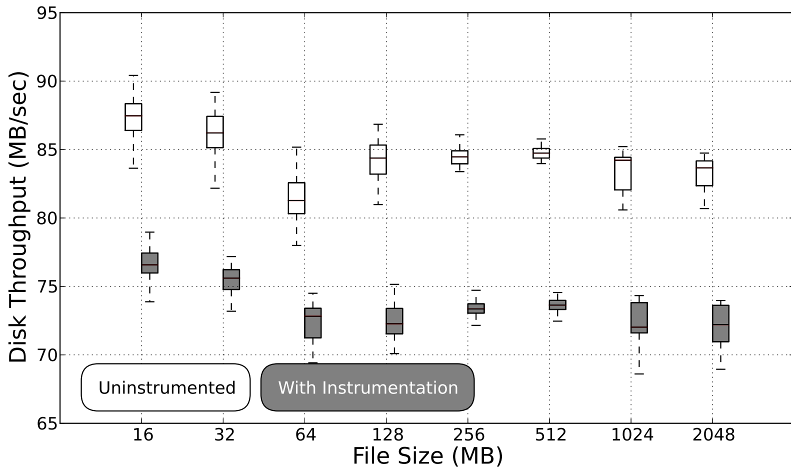
- LO-PHI: DMA accesses are cache coherent by default
  - When disabled, accuracy and overhead are not influenced
  - Compute π, query memory, results are the same with/without LO-PHI
- MALT: Instruction caching potentially influences system overhead
  - The 12μs cost is fixed
  - Depending on workload, the OS may switch contexts more, causing more overhead

Bonus: MALT Overhead



UNIVERSITY of VIRGINIA ENGINEERING

Bonus: LO-PHI Disk Writes



UNIVERSITY of VIRGINIA ENGINEERING

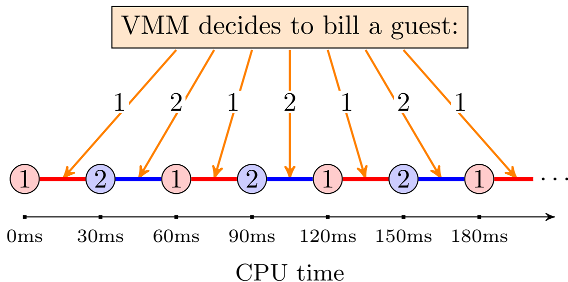
Bonus: Future Directions

1. Transparent program control via CPU interposition
   - Can we change the program's execution without changing the code in memory?
   - Place programmable device between motherboard and CPU
2. Applications in Cloud Security
   - Use transparent introspection to prevent resource stealing attacks in cloud environments
3. Generalization of stealth
   - Models for human typing and mouse movement
   - In mobile devices, models for human eye movement

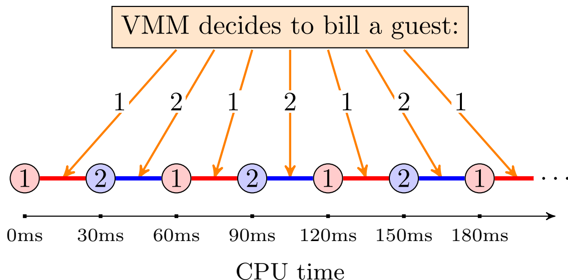
CPU Interposition

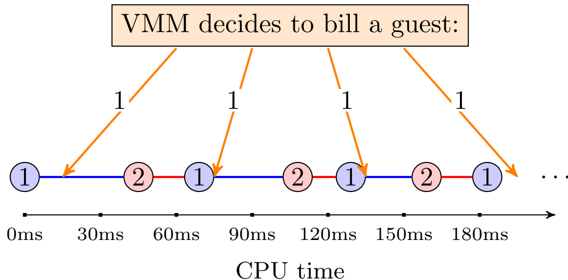
- We interposed SATA traffic with LO-PHI
- CPU interposers exist for Intel and ARM platforms
    - Can we transparently alter instructions on the fly?

Applications in Cloud Security

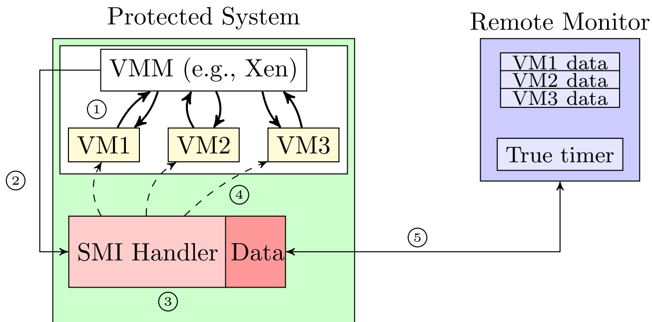


Applications in Cloud Security





Applications in Cloud Security



UNIVERSITY of VIRGINIA ENGINEERING

Generalizing Stealth

- We want to improve automated analysis of stealthy samples

Generalizing Stealth

- We want to improve automated analysis of stealthy samples
- What if the malware engages the GUI? or measures keyboard/mouse usage?

Generalizing Stealth

- We want to improve automated analysis of stealthy samples
- What if the malware engages the GUI? or measures keyboard/mouse usage?
  - Ultimately, we want to dynamically explore malware state

Generalizing Stealth

- We want to improve automated analysis of stealthy samples
- What if the malware engages the GUI? or measures keyboard/mouse usage?
  - Ultimately, we want to dynamically explore malware state
  - But what if the malware detects our automatic actuation?

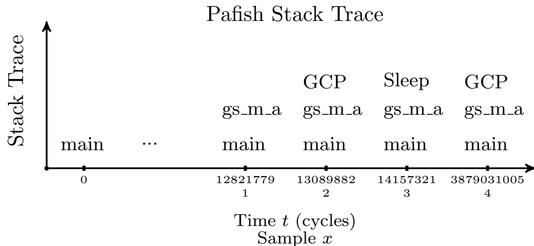
Generalizing Stealth

- We want to improve automated analysis of stealthy samples
- What if the malware engages the GUI? or measures keyboard/mouse usage?
  - Ultimately, we want to dynamically explore malware state
  - But what if the malware detects our automatic actuation?

- We need to explore approaches to modeling how humans engage malicious processes

Bonus: Pafish Case Study

- Can HOPS be used to determine Pafish's stealth mechanism?



Code around sample 1

```
t = ...       int gensandbox_mouse_act(){
12821779        POINT p1, p2;
13089882        GetCursorPos(&p1);
14157321        Sleep(2000);
3879031005      GetCursorPos(&p2);
              if (p1.x==p2.x && ...)
                traced("found");
              else
3879559528      nottraced("not found");
```