Transparent System Introspection in Support of Analyzing Stealthy Malware

Kevin Leach
PhD Dissertation
kjl2y@virginia.edu

November 30, 2016
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
Analysts want to quickly identify malware behavior
Malware Analysis Challenges

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

- What damage does it do?
- How does it infect a system?
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

But what if the program can detect our introspection tool?
Introspection

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

But what if the program can detect our introspection tool?
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
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?
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
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”)
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

Triage System

New Sample

Signature developed

Manual analysis
(Time consuming)
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’
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

Use cases
- Read Variables
- Read Stack Trace

System Under Test (SUT)
- Variables
- Function Calls
- Code Under Test
- OS Introspection (SPECTRE, VMI)
- SMM Memory Acquisition
- PCIe Memory Acquisition

Remote Host
Architecture

System Under Test (SUT)

Remote Host

Use cases

Read Variables

Read Stack Trace

Semantics

Userspace

Kernel

Hardware

Variables

Function Calls

Code Under Test

OS Introspection (SPECTRE, VMI)

SMM Memory Acquisition

PCLe Memory Acquisition

Component 1 – Hardware-assisted memory acquisition via PCI-e

Component 2 – Hardware-assisted memory acquisition via SMM

Component 3 – Transparent program introspection
Component 1 – Hardware-assisted memory acquisition via PCI-e
Component 2 – Hardware-assisted memory acquisition via SMM
Component 3 – Transparent program introspection
Component 1 – Hardware-assisted memory acquisition via PCI-e
Component 2 – Hardware-assisted memory acquisition via SMM
Component 3 – Transparent program introspection
The architecture includes the following components:

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

**Use cases** involve:
- **Remote Host**
  - Read Variables
  - Read Stack Trace

**System Under Test (SUT)** includes:
- Variables
- Function Calls
- Code Under Test
- OS Introspection (SPECTRE, VMI)
  - SMM Memory Acquisition
  - PCIe Memory Acquisition

**Semantics**

**Userspace**

**Kernel**

**Hardware**
Architecture

System Under Test (SUT)

Remote Host

Use cases

Read Variables

Read Stack Trace

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
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)
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

1. Find program in memory
2. Dump to remote host
3. Configure next SMI

OS/Program Code

SMI occurs

SMI Handler

Resume from SMM

Protected Mode

System Management Mode
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 12 $\mu s$ total, 8 $\mu s$ from switching)

- Measure system overhead when configuring SMIs:
  - Cause SMIs every retired instruction

- Demonstrate feasibility of approach to stealthy malware
  - Consider recent packers
## SMM Overhead

<table>
<thead>
<tr>
<th>Stepping method</th>
<th>Slowdown</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Windows</td>
<td>Linux</td>
</tr>
<tr>
<td></td>
<td>$\pi$</td>
<td>$\pi$</td>
</tr>
<tr>
<td></td>
<td>$gzip$</td>
<td>$gzip$</td>
</tr>
<tr>
<td>Without MALT</td>
<td>1.00x</td>
<td>1.00x</td>
</tr>
<tr>
<td>far control transfers</td>
<td>1.38x</td>
<td>1.46x</td>
</tr>
<tr>
<td>near returns</td>
<td>46.2x</td>
<td>36.1x</td>
</tr>
<tr>
<td>taken mispredicted</td>
<td>96.5x</td>
<td>77.7x</td>
</tr>
<tr>
<td>taken branches</td>
<td>634x</td>
<td>280x</td>
</tr>
<tr>
<td>mispredicted branches</td>
<td>99.6x</td>
<td>45.4x</td>
</tr>
<tr>
<td>branches</td>
<td>745x</td>
<td>290x</td>
</tr>
<tr>
<td>instructions</td>
<td>1021x</td>
<td>492x</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Stepping method</th>
<th>Slowdown</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Windows</td>
<td></td>
<td>Linux</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\pi$</td>
<td>gzip $\pi$</td>
<td>$\pi$ gzip</td>
<td></td>
</tr>
<tr>
<td>Without MALT</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
</tr>
<tr>
<td>far control transfers</td>
<td>1.38x</td>
<td>1.36x</td>
<td>1.46x</td>
<td>1.42x</td>
</tr>
<tr>
<td>near returns</td>
<td>46.2x</td>
<td>39.1x</td>
<td>36.1x</td>
<td>34.7x</td>
</tr>
<tr>
<td>taken mispredicted</td>
<td>96.5x</td>
<td>40.2x</td>
<td>77.7x</td>
<td>81.2x</td>
</tr>
<tr>
<td>taken branches</td>
<td>634x</td>
<td>935x</td>
<td>280x</td>
<td>903x</td>
</tr>
<tr>
<td>mispredicted branches</td>
<td>99.6x</td>
<td>149x</td>
<td>45.4x</td>
<td>138x</td>
</tr>
<tr>
<td>branches</td>
<td>745x</td>
<td>1196x</td>
<td>290x</td>
<td>1033x</td>
</tr>
<tr>
<td>instructions</td>
<td><strong>1021x</strong></td>
<td><strong>1519x</strong></td>
<td><strong>492x</strong></td>
<td><strong>1369x</strong></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Packing Tool</th>
<th>MALT</th>
<th>OllyDbg</th>
<th>DynamoRIO</th>
<th>VMware Fusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPX v3.08</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Obsidium v1.4</td>
<td>✓</td>
<td>x (access violation)</td>
<td>x (segfault)</td>
<td>✓</td>
</tr>
<tr>
<td>ASPack v2.29</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Armadillo v2.01</td>
<td>✓</td>
<td>x (access violation)</td>
<td>x (crash)</td>
<td>x (crash)</td>
</tr>
<tr>
<td>Themida v2.2.3.0</td>
<td>✓</td>
<td>x (exception)</td>
<td>x (exception)</td>
<td>x (no VM)</td>
</tr>
<tr>
<td>RLPack v1.21</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PELock v1.0694</td>
<td>✓</td>
<td>x</td>
<td>x (segfault)</td>
<td>✓</td>
</tr>
<tr>
<td>VMPProtect v2.13.5</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x (crash)</td>
</tr>
<tr>
<td>eXPressor v1.8.0.1</td>
<td>✓</td>
<td>x</td>
<td>x (segfault)</td>
<td>x (crash)</td>
</tr>
<tr>
<td>PECompact v3.02.2</td>
<td>✓</td>
<td>x (access violation)</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
## SMM vs. Packers

<table>
<thead>
<tr>
<th>Packing Tool</th>
<th>MALT</th>
<th>OllyDbg</th>
<th>DynamoRIO</th>
<th>VMware Fusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPX v3.08</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Obsidium v1.4</td>
<td>✓</td>
<td>✗ (access violation)</td>
<td>✗ (segfault)</td>
<td>✓</td>
</tr>
<tr>
<td>ASPack v2.29</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Armadillo v2.01</td>
<td>✓</td>
<td>✗ (access violation)</td>
<td>✗ (crash)</td>
<td>✗ (crash)</td>
</tr>
<tr>
<td>Themida v2.2.3.0</td>
<td>✓</td>
<td>✗ (exception)</td>
<td>✗ (exception)</td>
<td>✗ (no VM)</td>
</tr>
<tr>
<td>RLPack v1.21</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PELock v1.0694</td>
<td>✓</td>
<td>✗</td>
<td>✗ (segfault)</td>
<td>✓</td>
</tr>
<tr>
<td>VMPProtect v2.13.5</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗ (crash)</td>
</tr>
<tr>
<td>eXPressor v1.8.0.1</td>
<td>✓</td>
<td>✗</td>
<td>✗ (segfault)</td>
<td>✗ (crash)</td>
</tr>
<tr>
<td>PECompact v3.02.2</td>
<td>✓</td>
<td>✗ (access violation)</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
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

SUT
- Host Memory
- Host SATA

LO-PHI FPGA
- DMA
- PCIe connector
- Ethernet

SATA
- SATA 1
- SATA 2

Hard drive

DMA to Remote System
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 I0Zone benchmarks
LO-PHI Memory Overhead

<table>
<thead>
<tr>
<th>Memory Operation Type</th>
<th>Uninstrumented</th>
<th>With Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE</td>
<td>4900</td>
<td>4950</td>
</tr>
<tr>
<td>MMX</td>
<td>5000</td>
<td>5050</td>
</tr>
<tr>
<td>INTEGER</td>
<td>5100</td>
<td>5150</td>
</tr>
<tr>
<td>FL-POINT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LO-PHI Disk Overhead

![Graph showing disk throughput vs file size with and without instrumentation.](image-url)
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

<table>
<thead>
<tr>
<th>Technique Employed</th>
<th># Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait for keyboard</td>
<td>3</td>
</tr>
<tr>
<td>BIOS-based</td>
<td>6</td>
</tr>
<tr>
<td>Hardware id-based</td>
<td>28</td>
</tr>
<tr>
<td>Processor feature-based</td>
<td>62</td>
</tr>
<tr>
<td>Exception-based</td>
<td>79</td>
</tr>
<tr>
<td>Timing-based</td>
<td>251</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Observed Behavior</th>
<th>Number of Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Created new process(es)</td>
<td>765</td>
</tr>
<tr>
<td>Opened socket(s)</td>
<td>210</td>
</tr>
<tr>
<td>Started service(s)</td>
<td>300</td>
</tr>
<tr>
<td>Loaded kernel modules</td>
<td>20</td>
</tr>
<tr>
<td>Modified GDT</td>
<td>58</td>
</tr>
<tr>
<td>Modified IDT</td>
<td>10</td>
</tr>
</tbody>
</table>
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
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
MALT and LO-PHI both provide raw introspection data transparently
- Periodic snapshots of memory (and potentially disk) via SMM or PCIe

- Raw memory snapshots
- HOPS: Transparent Introspection
- Variables
- Stack Trace
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
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
Example Dynamic Stack Trace

Stack Trace

Time $t$ (cycles)
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
Nullhttpd Call Stack Introspection Accuracy

Percent of Calls Reported Correctly vs. Cycles Between Memory Samples

- T1 (Get Text)
- T2 (Get Empty)
- T3 (404 Error)
- T4 (Get Image)
- T5 (Get Dir List)
- T6 (Post Form)
- T7 (Exploit)
# Variable Accuracy

<table>
<thead>
<tr>
<th></th>
<th>nullhttpd</th>
<th>wuftpdp</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Locals</strong></td>
<td>43% 133/306</td>
<td>46% 202/436</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td>65% 168/260</td>
<td>56% 119/214</td>
</tr>
<tr>
<td><strong>Globals</strong></td>
<td>100% 77/77</td>
<td>92% 4218/4580</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>59% 378/643</td>
<td>90% 4539/5230</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Conference/Year</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE S&amp;P 2015</td>
<td>Using Hardware Features for Increased Debugging Transparency</td>
</tr>
<tr>
<td>TDSC 2016</td>
<td>Towards Transparent Debugging</td>
</tr>
<tr>
<td>NDSS 2016</td>
<td>LO-PHI: Low-Observable Physical Host Instrumentation</td>
</tr>
<tr>
<td>SANER 2016</td>
<td>Towards Transparent Introspection</td>
</tr>
</tbody>
</table>

## Other systems security publications

<table>
<thead>
<tr>
<th>Conference/Year</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsiaCCS 2015</td>
<td>TrustLogin: Securing Password-Login on Commodity Operating Systems</td>
</tr>
<tr>
<td>ESORICS 2014</td>
<td>A Framework to Secure Peripherals at Runtime</td>
</tr>
<tr>
<td>DSN 2013</td>
<td>Spectre: A Dependable System Introspection Framework</td>
</tr>
<tr>
<td>DSN 2013</td>
<td>Barley: Combining Control Flow with Resource Consumption to Detect Jump-based ROP Attacks</td>
</tr>
</tbody>
</table>

## Other publications

<table>
<thead>
<tr>
<th>Conference/Year</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigComp 2016</td>
<td>A MapReduce Framework to Improve Template Matching Uncertainty</td>
</tr>
<tr>
<td>Ubicomp 2016</td>
<td>Assessing Social Anxiety Using GPS Trajectories and Point-Of-Interest Data</td>
</tr>
<tr>
<td>DSN 2016</td>
<td>An Uncrewed Aerial Vehicle Attack Scenario and Trustworthy Repair Architecture</td>
</tr>
<tr>
<td>TIST'2015</td>
<td>DAEHR: A Discriminant Analysis Framework for Electronic Health Record Data and an Application to Early Detection of Mental Health Disorders</td>
</tr>
<tr>
<td>IEEE Big Data 2016</td>
<td>M-SEQ: Early Detection of Anxiety and Depression via Temporal Orders of Diagnoses in Electronic Health Data</td>
</tr>
</tbody>
</table>
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  
kernell32!DebugActiveProcess to prevent other debuggers from attaching to a process |
| **PEB Field** | PEB!IsDebuggered 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 |
## 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         | `cpu_id` 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
## SMM Attacks and Solutions

<table>
<thead>
<tr>
<th>SMM Attacks</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlocked SMRAM</td>
<td>Set D_LCK bit</td>
</tr>
<tr>
<td>SMRAM reclaiming</td>
<td>Lock remapping and TOLUD registers</td>
</tr>
<tr>
<td>Cache poisoning</td>
<td>SMRR</td>
</tr>
<tr>
<td>Graphics aperture</td>
<td>Lock TOLUD</td>
</tr>
<tr>
<td>TSEG location</td>
<td>Lock TSEG base</td>
</tr>
<tr>
<td>Call/fetch outside of SMRAM</td>
<td>No call/fetch outside of SMRAM</td>
</tr>
</tbody>
</table>
Bonus: Caching Issues

- **LO-PHI**: DMA accesses are cache coherent by default
  - When disabled, accuracy and overhead are not influenced
  - Compute $\pi$, query memory, results are the same with/without LO-PHI

- **MALT**: Instruction caching potentially influences system overhead
  - The 12$\mu$s cost is fixed
  - Depending on workload, the OS may switch contexts more, causing more overhead
**Bonus: MALT Overhead**

CPU flow of execution

Clock

Device

Interrupt handler, switch to **Task 2**

Task 1

Task 2

Interrupt handler, return to **Task 1**

Task 1

Time

Overhead while executing handler

Raise interrupt \( 1 \)

Interrupt completes \( 2 \)

Raise interrupt \( 3 \)

Interrupt completes \( 4 \)
Bonus: LO-PHI Disk Writes

![Box plot showing disk throughput for different file sizes with and without instrumentation. The x-axis represents file size in MB, and the y-axis represents disk throughput in MB/sec. The plot compares uninstrumented and instrumented conditions, with error bars indicating variability.]
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?
VMM decides to bill a guest:

1 2 1 2 1 2 1

CPU time

0ms 30ms 60ms 90ms 120ms 150ms 180ms
Applications in Cloud Security

VMM decides to bill a guest:

1 2 1 2 1 2 1

CPU time
0ms 30ms 60ms 90ms 120ms 150ms 180ms

VMM decides to bill a guest:

1 1 1 1

CPU time
0ms 30ms 60ms 90ms 120ms 150ms 180ms
Applications in Cloud Security

Protected System

1. VMM (e.g., Xen)
2. VM1
3. SMI Handler
4. VM2
5. VM3
6. Data
7. Remote Monitor
8. VM1 data
9. VM2 data
10. VM3 data
11. True timer

Diagram showing the flow of data and interactions between VMs and the SMI Handler within a protected system.
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
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
Can HOPS be used to determine Pafish’s stealth mechanism?

Pafish Stack Trace

<table>
<thead>
<tr>
<th>Time t (cycles)</th>
<th>Sample x</th>
<th>Stack Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>main</td>
<td>main</td>
</tr>
<tr>
<td>12821779</td>
<td>gs_m_a</td>
<td>main</td>
</tr>
<tr>
<td>13089882</td>
<td>gs_m_a</td>
<td>main</td>
</tr>
<tr>
<td>14157321</td>
<td>gs_m_a</td>
<td>main</td>
</tr>
<tr>
<td>3879031005</td>
<td>gs_m_a</td>
<td>main</td>
</tr>
</tbody>
</table>

Code around sample 1

```c
int gensandbox_mouse_act()
{
    POINT p1, p2;
    GetCursorPos(&p1);
    Sleep(2000);
    GetCursorPos(&p2);
    if (p1.x==p2.x && ...) traced("found");
    else nottraced("not found");
}```