UNIVERSITY of VIRGINIA ENGINEERING

Transparent System Introspection in Support of Analyzing Stealthy Malware

Kevin Leach

University of Virginia

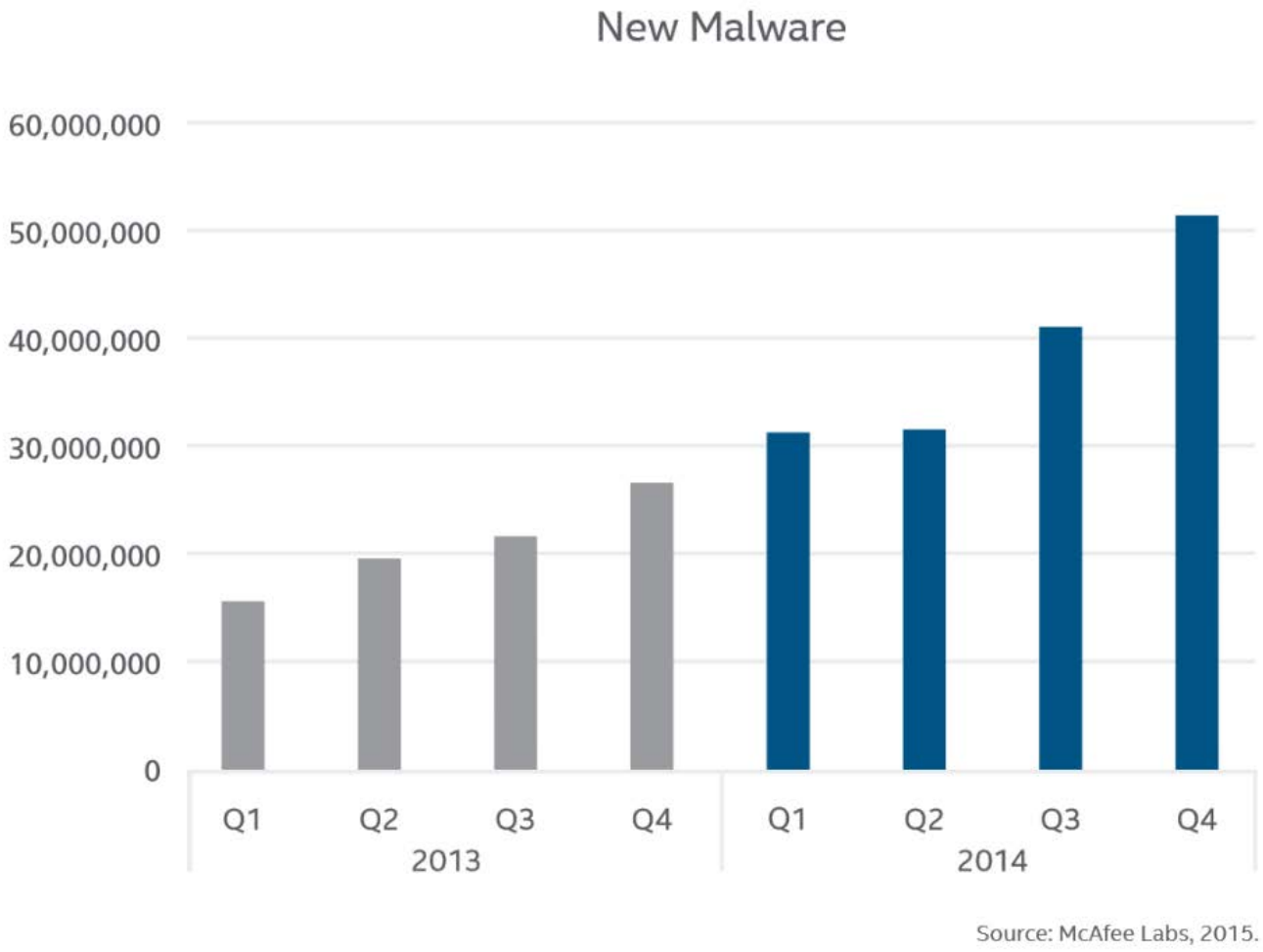
Computer Engineering

May 12, 2015

Breaking Down the Title

- Transparent System Introspection in Support of Analyzing Stealthy Malware
  1. Stealthy Malware
     - Malware that hides itself from analysis
  2. Introspection
     - Acquisition of data about malware behavior
  3. Transparency
     - Sample cannot tell if it is under analysis

Malware Proliferation



Source: McAfee Labs, 2015.

Malware Analysis

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

Stealthy Malware

- Adversary wants to hide from user and analyst
  - Prevent user from knowing
    - Don't want user to end a malicious process
  - Prevent analyst from developing a defense
    - Make zero-days last as long as possible
- Malware that hides itself is *stealthy*
  - More effort required to analyze
- Over 30% of malware exhibits stealth

Artifacts and Stealthy Malware

- To maintain secrecy, adversary uses *artifacts* to detect analysis techniques
  - *Timing (nonfunctional) artifacts* – overhead introduced 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 emulation of some instructions by VM
    - Device names (hard drive named "VMWare disk")
- **Too much effort to fully analyze each stealthy sample**
    - Automated techniques exist, but they fail against stealthy samples
    - Manual analysis is time consuming and expensive

Introspection

- Understanding program behavior
- Debugger *introspects* program to access its raw data
  - Read variables
  - Reconstruct stack traces
  - Write variables to change execution path
- Analyst infers behavior of a sample from interpreting and changing this raw data
- Virtual Machine Introspection (VMI)
  - Plugin for a Virtual Machine Manager (slowdown)
  - Helper process inside guest VM (detectable process)
- **If this sample is stealthy, the acquired introspection data may not be accurate**

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'**

Research Question

- Can we develop a transparent introspection technique that admits analyzing stealthy malware?
  1. Can we transparently acquire snapshots of process memory?
  2. Can we reconstruct the semantics of these snapshots?
  3. Can we transparently alter the execution path of the program?

Formal Thesis Statement

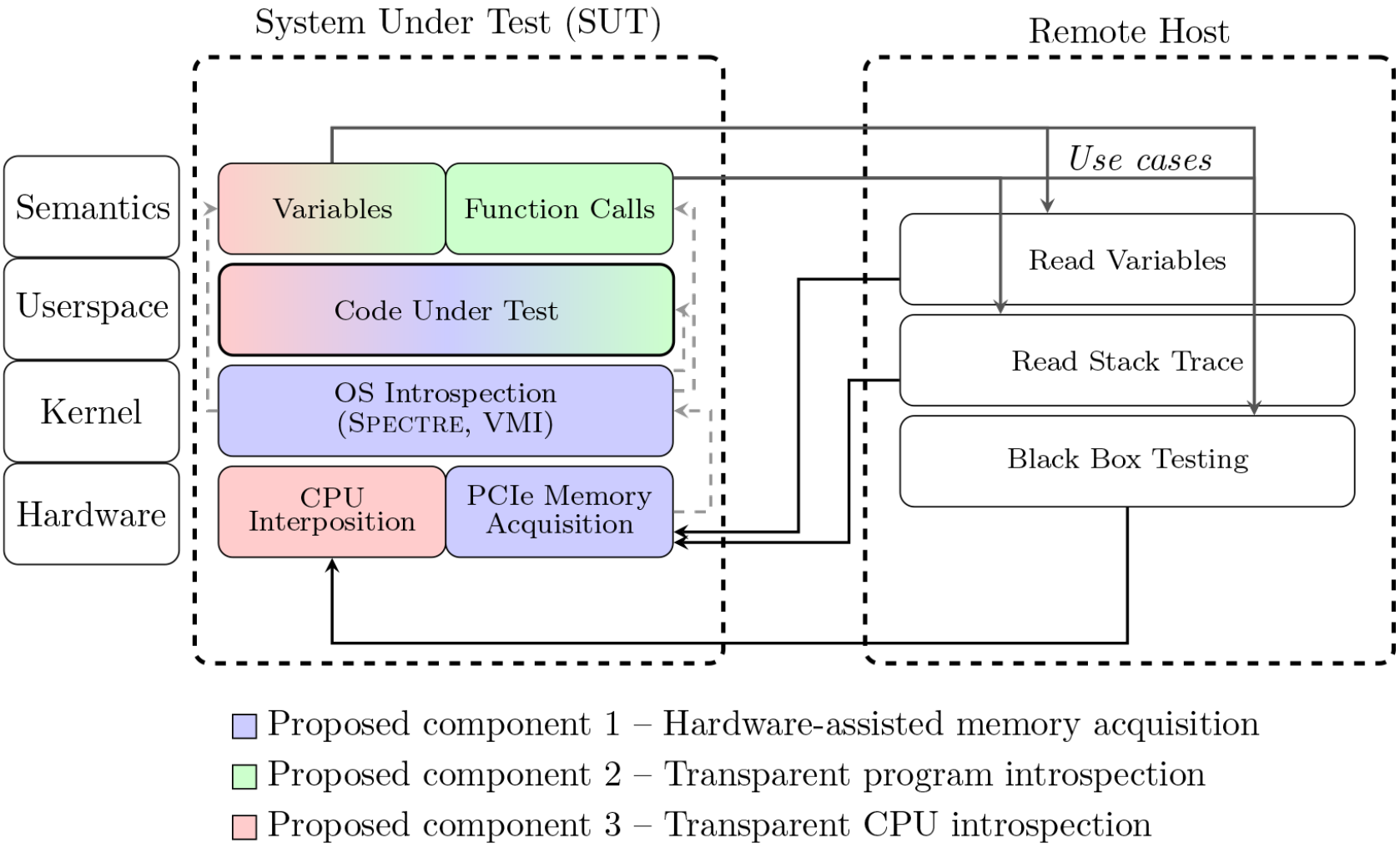
It is possible to develop a transparent debugging system capable of reading variable values, reconstructing stack traces, and writing variable values by combining hardware assisted memory acquisition, process introspection, and CPU interposition.

Proposal Overview

1. Hardware-assisted introspection
   - Transparently acquire program data
     - PCI Express
     - System Management Mode (SMM)
2. Transparent program introspection
   - Transparently reconstruct program semantics from data
3. CPU Interposition
   - Transparently change program data to affect execution
     - Black box testing of Android malware
     - Patching quadcopter firmware

We propose yes!

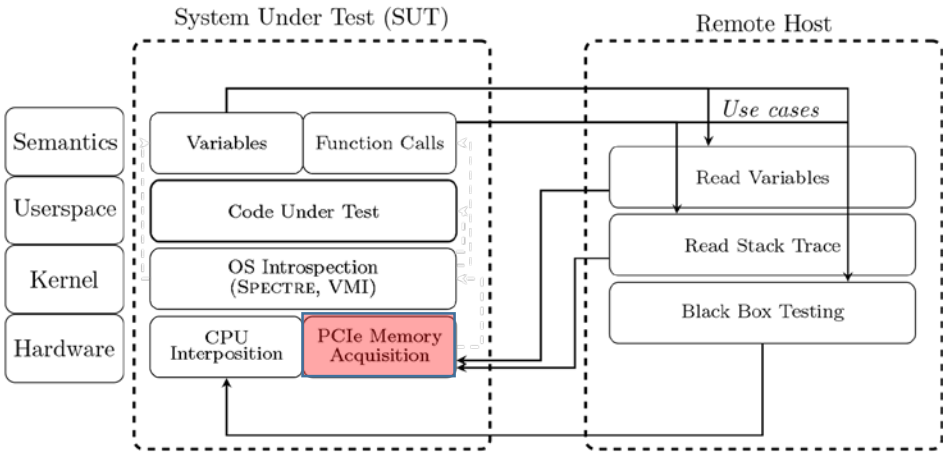
Proposed Architecture

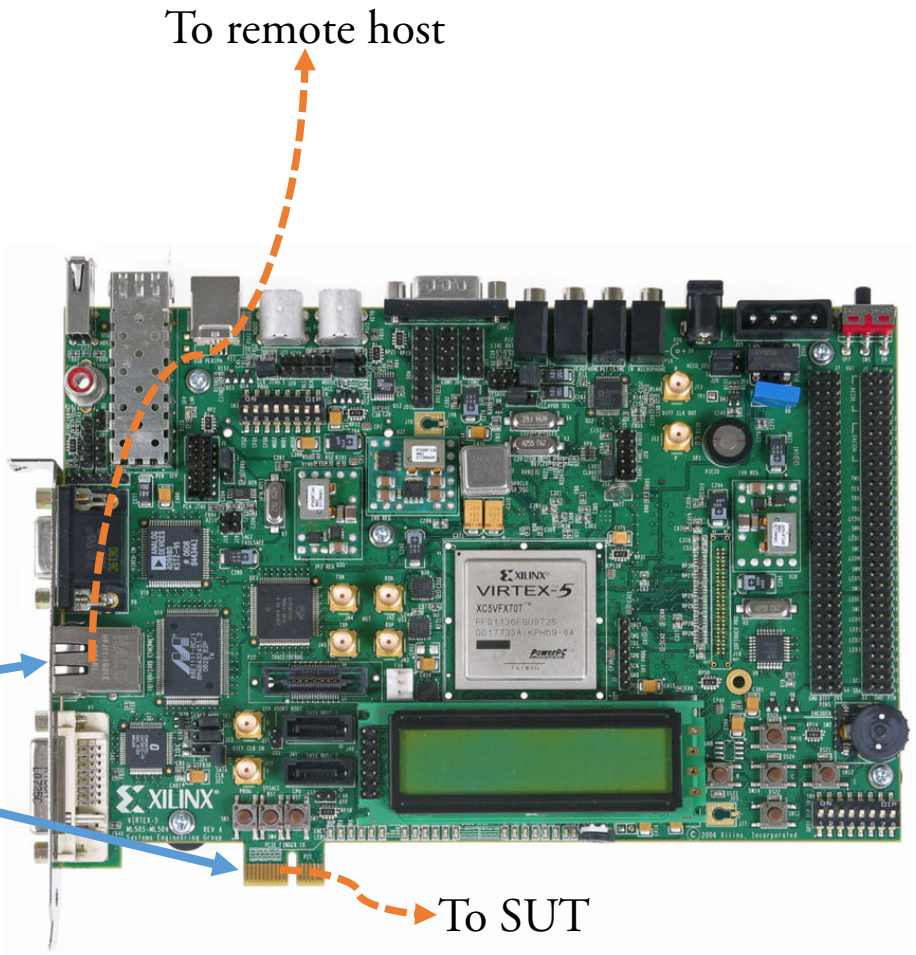


Hardware-Assisted Introspection

- Two proposed approaches
  - PCI-Express based
    - Few timing artifacts
    - Functional artifact
      (DMA access performance counter)
  - System Management Mode
    - Significant timing artifacts
    - No functional artifacts

PCI Express Memory Acquisition





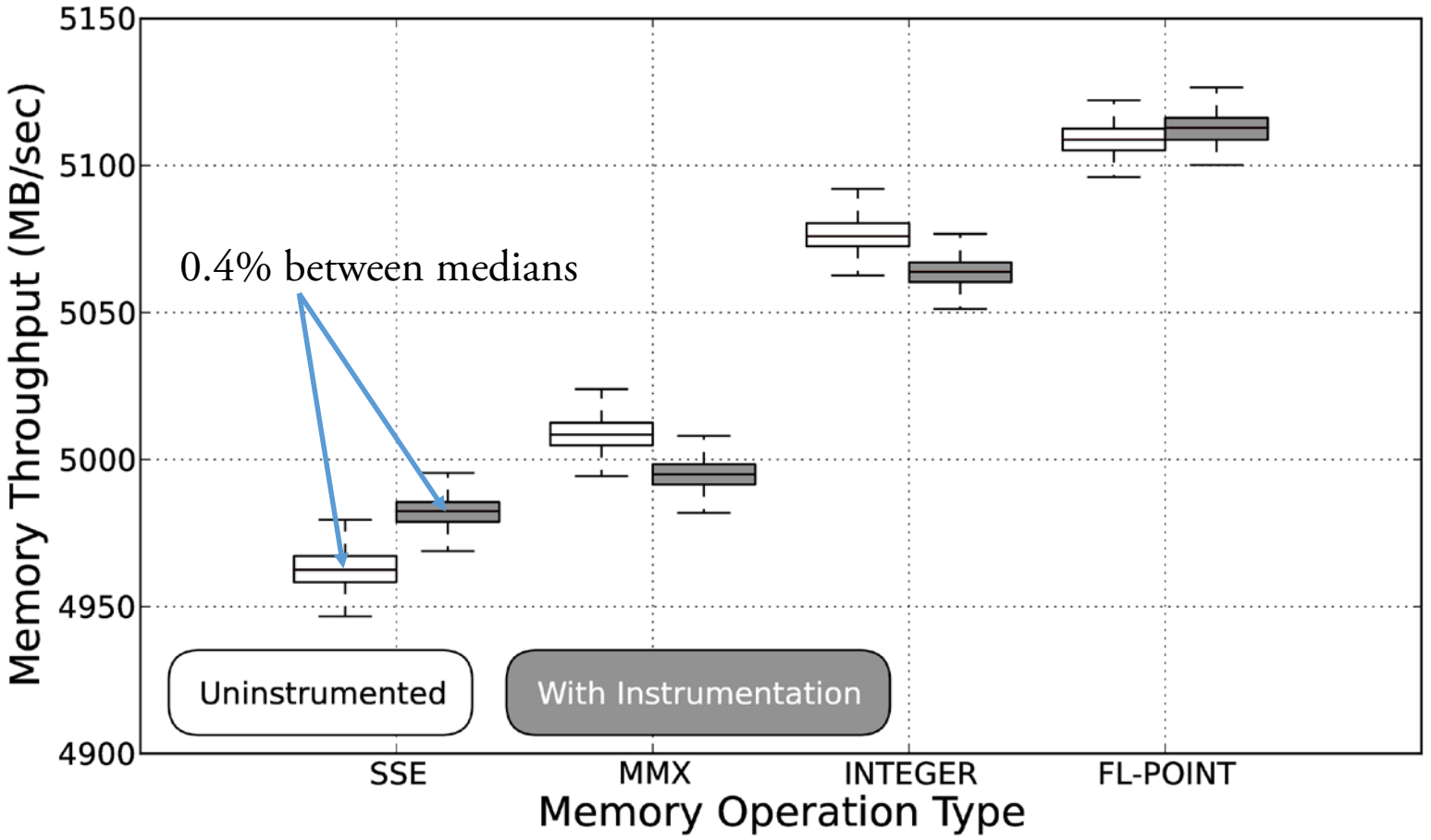
- Use Xilinx ML507
  - Gigabit Ethernet
  - PCI Express Connector

PCI Express Memory Acquisition

- Proposed Experiments
  - Compare performance of SUT when uninstrumented and instrumented on indicative workloads
  - Note deviation in performance
    - If medians are within margin of error, we conclude success
  - Use RAMSpeed Benchmarks
    - Runs instructions from each unit on x86 (i.e., INT, FLOAT, SSE, MMX)
    - Records throughput of RAM only (not cache)

PCI Express Preliminary Results

RAMSpeed Benchmark Results (n=500)



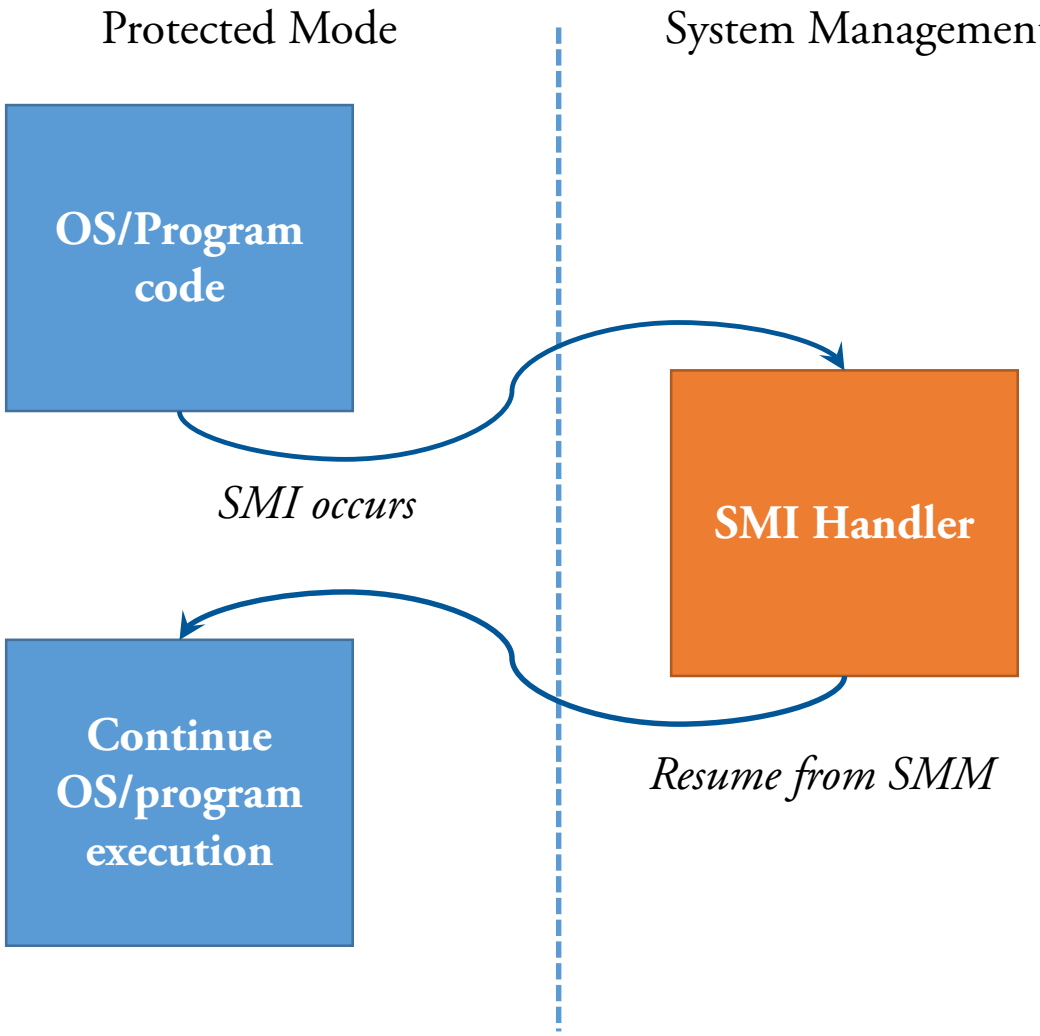
PCI Express Memory Acquisition

- Fast, low overhead
  - Under 1%, within margin of error
  - Not practically measurable by malware
- Potentially presents functional artifacts
  - DMA access performance counter

System Management Mode

- Intel x86 feature provides small, transparent, trusted computing base
- SMM akin to Protected Mode
- System Management Interrupt (SMI) to enter SMM
- SMI Handler executed in SMM
  - Code stored in System Management RAM (SMRAM)
  - Trust only the BIOS

SMM Architecture



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

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)
- Measure system overhead when configuring SMIs:
  1. Retired far control transfer instructions
  2. Retired near return instructions
  3. Retired taken branch instructions
  4. Retired instructions (i.e., per-instruction)

SMM Preliminary Results

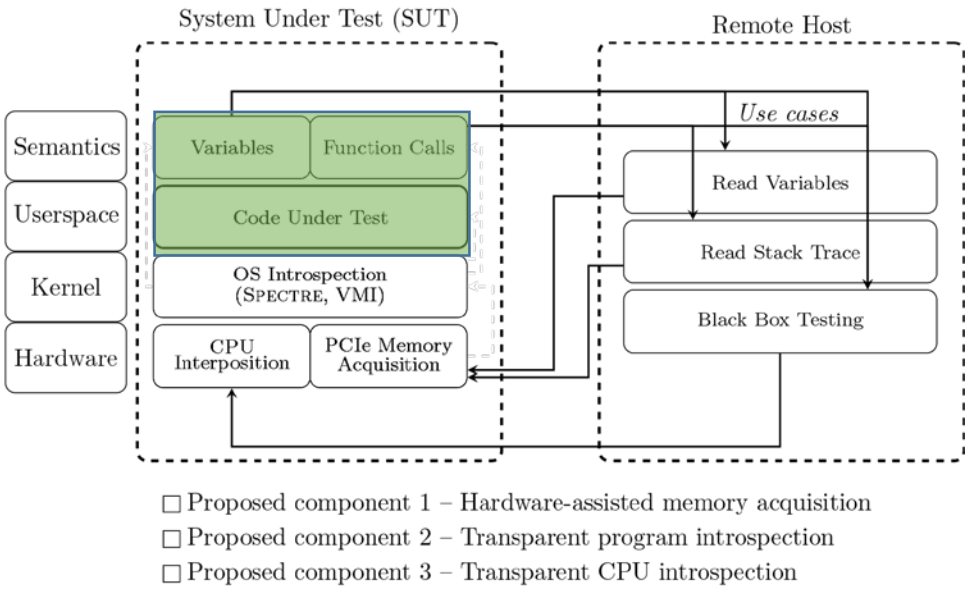
Stepping Methods	Windows				
	`pi`	`ls`	`ps`	`pwd`	`tar`
Far control	2	2	2	3	2
Near returns	30	21	22	28	29
Taken branches	565	479	527	384	245
All instructions	**973**	880	897	859	704
			Linux		
Far control	2	3	2	2	2
Near returns	26	41	28	10	15
Taken branches	192	595	483	134	159
All instructions	349	699	515	201	232

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

Process Introspection

1. Hardware-assisted introspection
   - Transparently acquire program data
     - PCI Express
     - System Management Mode (SMM)
2. **Transparent program introspection**
   - **Transparently reconstruct program semantics from data**
3. CPU Interposition
   - Transparently change program data to affect execution
     - Black box testing of Android malware
     - Patching quadcopter firmware

Program Introspection



- First component gives us raw memory
  - Periodic snapshots from PCI-e or SMM support
- We want useful semantic information from snapshots
  - Variables
  - Activation records

Program Introspection

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

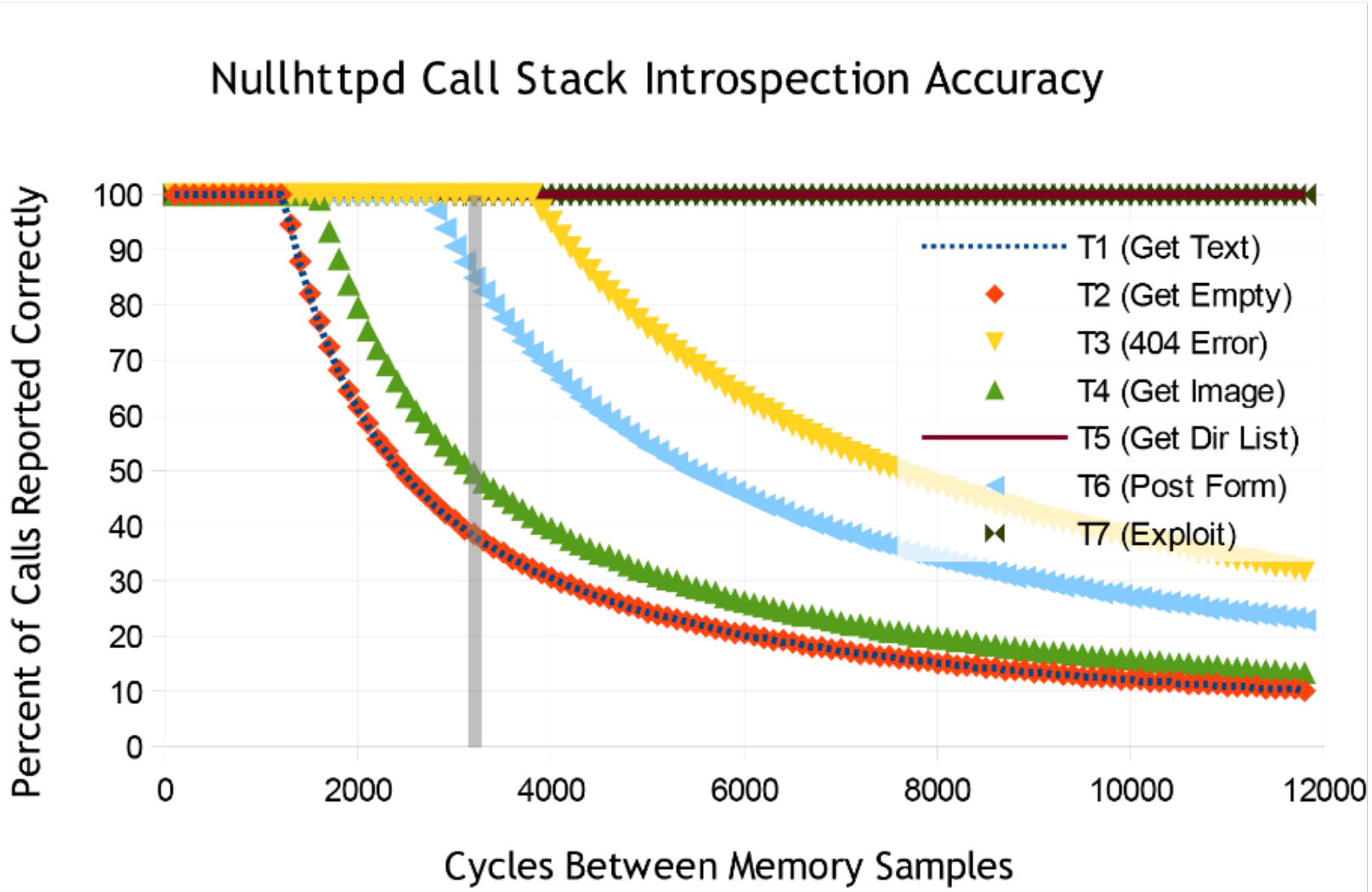
Introspection Experiments

- Consider indicative programs:
  - Wuftpd 2.6.0 (with CVE-2000-0573)
  - Nullhttpd 0.5.0 (with CVE-2002-1496)
- Run programs on indicative test cases
  - Gather ground truth on instrumented binary
  - Gather variable and stack trace information on deployed binary
    - Report fraction of variables correctly reported
    - Report stack trace as a function of sampling frequency (recall component 1 is polling-based)

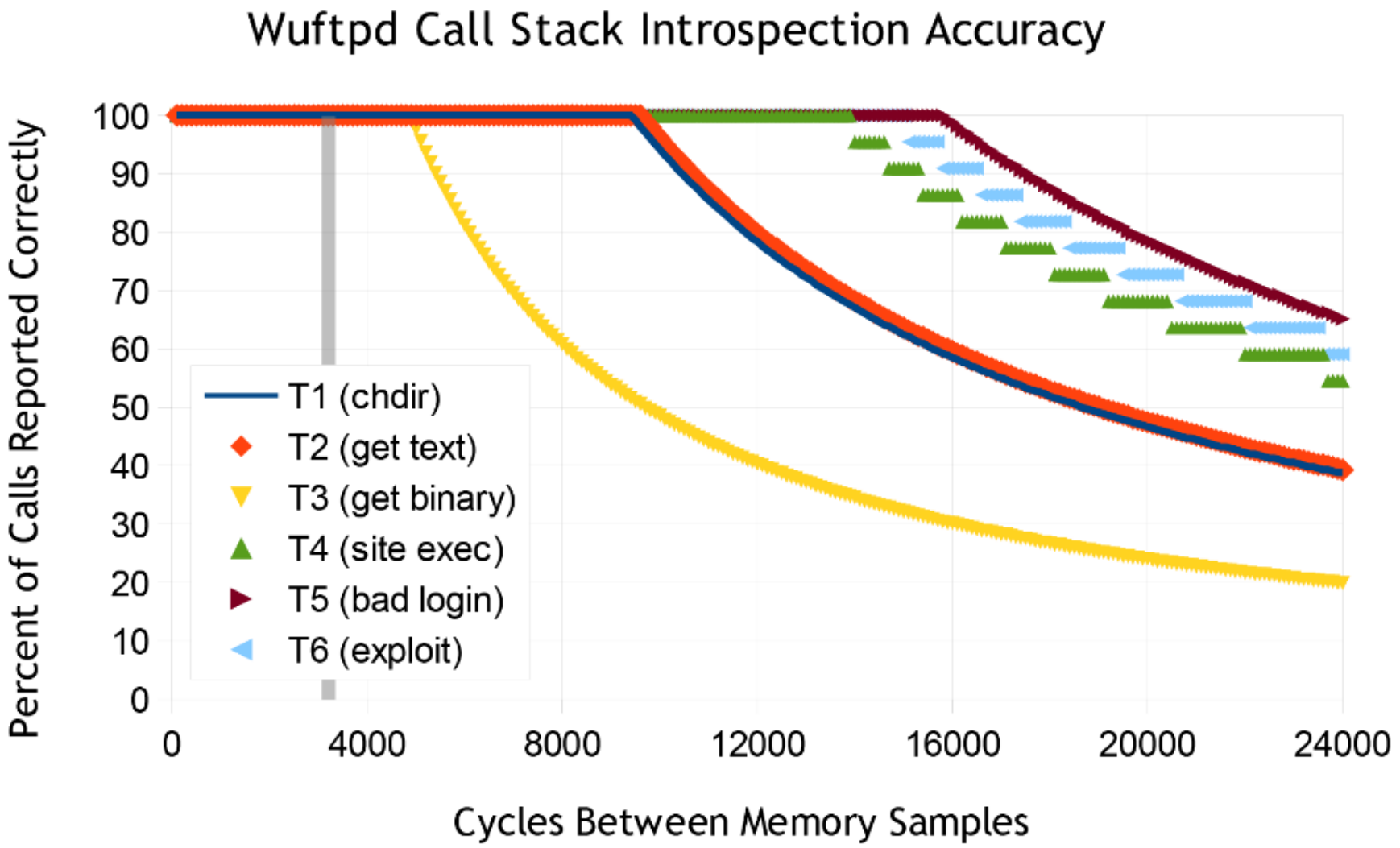
Introspection Preliminary Results

	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

Introspection Preliminary Results



Introspection Preliminary Results



1. Hardware-assisted introspection
   - Transparently acquire program data
     - PCI Express
     - System Management Mode (SMM)
2. Transparent program introspection
   - Transparently reconstruct program semantics from data
3. **CPU Interposition**
   - **Transparently change program data to affect execution**
     - **Black box testing of Android malware**
     - **Patching quadcopter firmware**

CPU Interposition

- Interposer sits between platform mainboard and CPU
  - Allows snooping and changing all signals to CPU
- In practice, far too slow for full scale debugging
- We propose using this hardware to *insert* instructions
  - Component 2 permits reading values, but we also want to write values
  - Alternatively, we can insert instructions to change software

Risk Mitigation: 2 Options

- Propose two potential avenues to explore CPU interposition
  - Support black box testing of Android Malware
  - Support patching of ARM-based firmware in autonomous aerial vehicles
- Propose fail early experiments, pick one option to take to completion
  - This component is speculative
  - Fail-early to help us ensure timely completion of satisfactory component

Option 1: Android Black Boxing

- CPU Interposition allows inserting instructions to change variable values
- Assume we know location of a variable
  - Component 2 gives us the location
  - Also must consider variables only in registers
- Specifically, find locations implicated in branches
  - Stealthy malware may have a branch instruction to decide the presence of artifacts
    - if (artifact) then x; else y;
      - We want to change 'artifact' so we can exercise a different branch in the program
- Fail early experiment: retarget component 2 for ARM

Option 1: Proposed Experiments

- Overarching question: what fraction of the outcomes of branches can we successfully change?
- Consider every branch in program reachable via given test cases
- Every branch has an associated register it checks
  - Attempt inserting instruction to change value of variable before branch
  - If we can change the outcome of a branch 80% of the time, we are successful
    - Can repeat if we fail: 5 repeats yields 99.97% success (3 sigma rule)

Option 2: Save the Quadcopters

- Program repair permits automatically patching software
- Quadcopters are realtime, resource-constrained platforms that fail
  - Cannot change source code or reflash EEPROM while running

Option 2: Quadcopter Firmware

- CPU interposition allows inserting instructions, but not removing them
- We want to convert a given patch to a sequence of instructions for insertion while running
  - We can undo some instructions
    - `add rax, rax, 5` can be undone with `sub rax, rax, 5`
- We must also deploy the patch transparently
  - After current program counter to ensure execution
  - Before target location of insertion

Option 2: Proposed Experiments

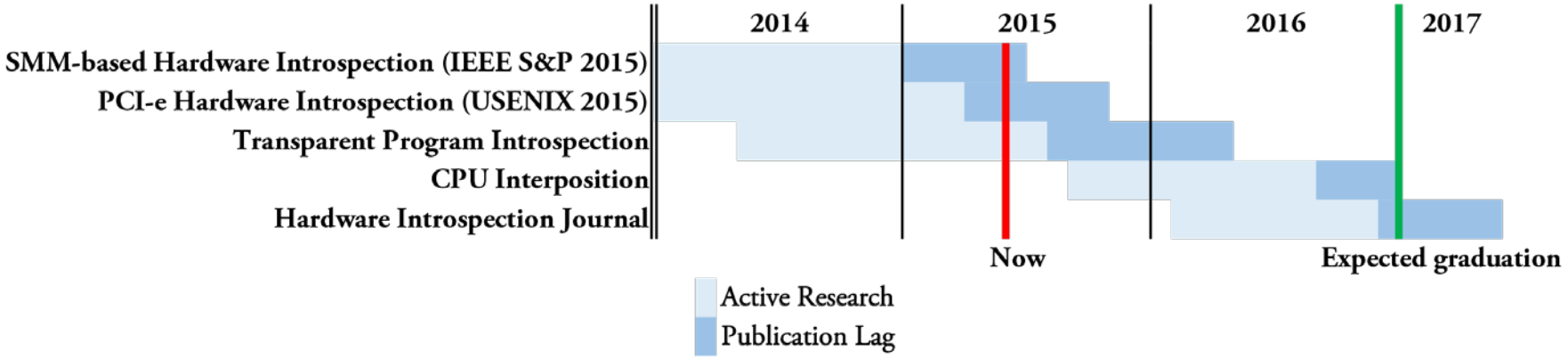
- Early failure: classify indicative patches according to general operations
  - We hypothesize that patches consist of categories of general 'building block' operations (e.g., 'remove call to function')
  - This approach is feasible if we can take 100 indicative patches and manually convert 80 of them into fewer than 10 building block operations
- Assume we have a patch and desired deployment location
  - Periodically check program counter and compare to desired location
  - Determine whether to deploy the patch based on comparison
  - We are successful if we insert and exercise 80% of patches (3 sigma rule as before)

Publications Supporting Research

1. **K. Leach**, W. Weimer. HOPS: Towards Transparent Introspection. In preparation.
2. **K. Leach**. LO-PHI: Low Observable Physical Host Instrumentation. Under review at USENIX 2015. February 2015.
3. F. Zhang, **K. Leach**, A. Stavrou, H. Wang. Using Hardware Features for Increased Debugging Transparency. In *the 36th IEEE Symposium on Security and Privacy* (Oakland 2015). To appear.
4. F. Zhang, **K. Leach**, H. Wang, A. Stavrou. TrustLogin: Securing Password-Login on Commodity Operating Systems. In *Proceedings of the 10th ACM Symposium on Information, Computer, and Communications Security* (ASIACCS2015). Singapore. Acceptance rate: 17.8%.
5. F. Zhang, H. Wang, **K. Leach**, A. Stavrou. A Framework to Secure Peripherals at Runtime. *Proceedings of the 19th European Symposium on Research in Computer Security* (ESORICS 2014). September 2014. Wroclaw, Poland. Acceptance rate: 24.7%.
6. F. Zhang, **K. Leach**, K. Sun, A. Stavrou. Spectre: A Dependable System Introspection Framework. In *the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN 2013). June 24-27, Budapest, Hungary. Acceptance rate: 19%.
7. K. Leach. Barley: Combining Control Flow with Resource Consumption to Detect Jump-based ROP Attacks. In *the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN 2013). June 24-27, Budapest, Hungary. Acceptance rate: 33%

Proposed Timeline and Venues

- Typical venues:
  - USENIX (January/February)
  - RAID (April/May)
  - ACSAC (May/June)
  - ACM CCS (May)
  - ASPLOS (July)
  - NDSS (August)
  - IEE S&P (November/December)



Summary

- Three components to provide debugging transparency
  - Hardware-assisted memory acquisition
    - PCI-e presents low overhead
    - SMM exposes few functional artifacts
  - Program Introspection
    - Use snapshots to reconstruct useful semantic information
  - CPU Interposition
    - Allow writing variable values to help exercise different paths of execution in a sample
- Questions?

Why ARM vs. x86?

- Preliminary investigation supports CPU interposition for ARM platforms
  - x86 challenges
    - CPU interposition is not precise enough for proposed use cases
    - Hardware is prohibitively expensive (>$35,000)
  - ARM benefits
    - CPU interposition is better supported by community
    - Hardware is much cheaper (~$3,000)

Known Artifacts (1)

Anti-debugging	
API Calls	isDebuggerPresent
	NtQueryInformationProcess.ProcessInformation
	CheckRemoteDebuggerPresent
	NtSetInformationThread with ThreadInformationClass = 0x11
	DebugActiveProcess prevents attaching other debuggers
PEB Field	IsDebugged flag
	NtGlobalFlags
Detection	ForceFlag in heap header
	UnhandledExceptionFilter calls user-defined function, but terminates in a debugging process
	TEB contains valid pointer under debug
	Ctrl-C raises exception under debug, signal handler not scoped by debugger
	Rogue INT-3 instructions
	Trap flag register manipulation thwarts tracers
	entryPoint RVA = 0 erases magic MZ value in PE files
	ZwClose system call with invalid paramters raises exception in attached debugger
	Direct context modification confuses debuggers
	0x2D interrupt causes program to stop raising exceptions
	Undocumented 0xF1 instruction
	Searching for CC instructions
	TLS-callback to perform checks

Known Artifacts (2)

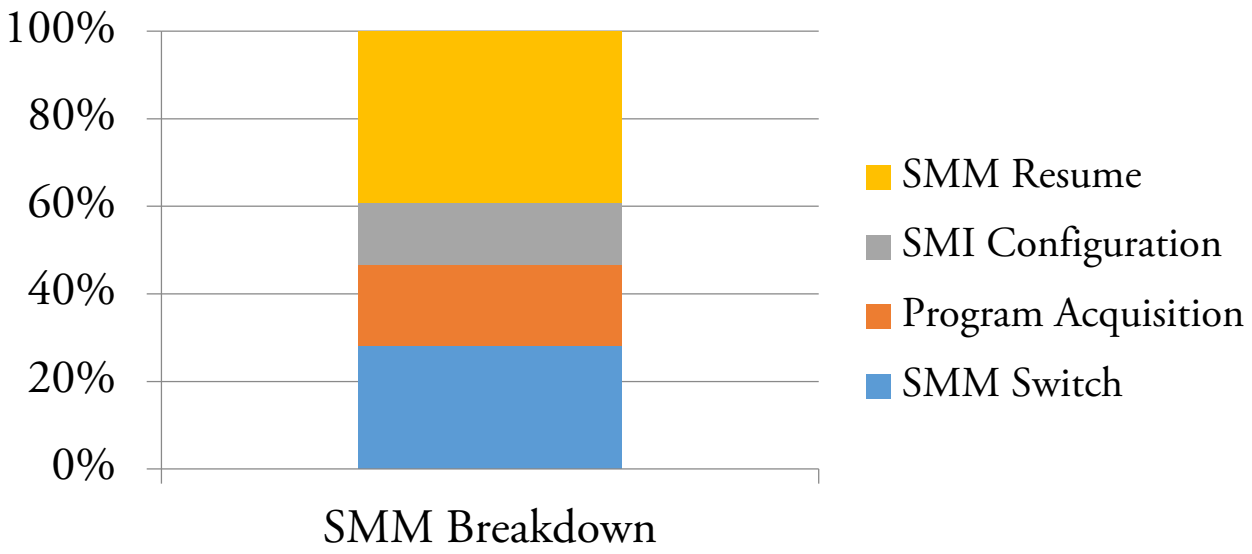
Anti-Virtualization	
VMWare	Virtualized device identifiers contain well-known strings
	Checkvm software searches for VMWare hooks in memory
	Well-known locations/strings associated with VMWare Tools
Other	LDTR register
	IDTR register (Red Pill)
	Magic I/O port (0x5658, 'VX')
	Invalid instruction behavior
	Memory deduplication to detect hypervisors
Anti-emulation	
Bochs	Visible debug port
QEMU	Cpuid returns less specific information
	Accessing reserved MSRs raises General Protection (GP) exception on bare metal
	Execution instruction longer than 15 bytes raises GP exception on bare metal
	Undocumented icebp instructions hangs QEMU (bare metal raises exception)
	Unaligned memory references raise exceptions in bare metal; unsupported by QEMU
	Bit 3 of FPU Control World register always 1 on bare metal, QEMU contains a 0
Other	Use CPU bugs or errata to create CPU fingerprints via public chipset documentation

Remarks on SMM

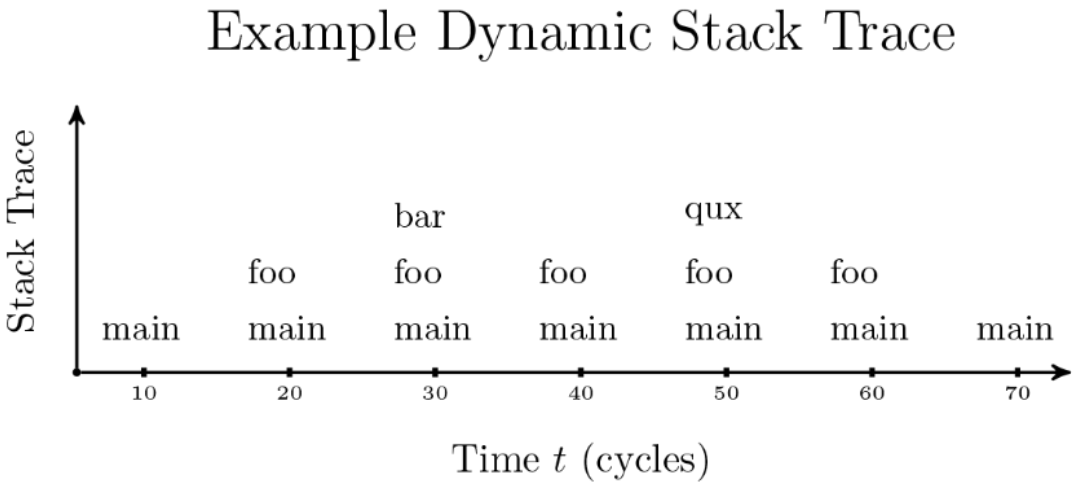
- SMM is generalizable
  - Every Intel platform since 386 includes SMM
- Relatively small trusted code base (TCB)
  - TCB restricted to BIOS
    - Thousands of lines of code
  - VMI systems require trust to extend to kernel and VMM
    - Millions of lines of code
- Configuration of subsequent SMI necessary for periodicity
  - Also ensures transparency (DOS attacks from kernel space)
  - Configured according to available performance counters

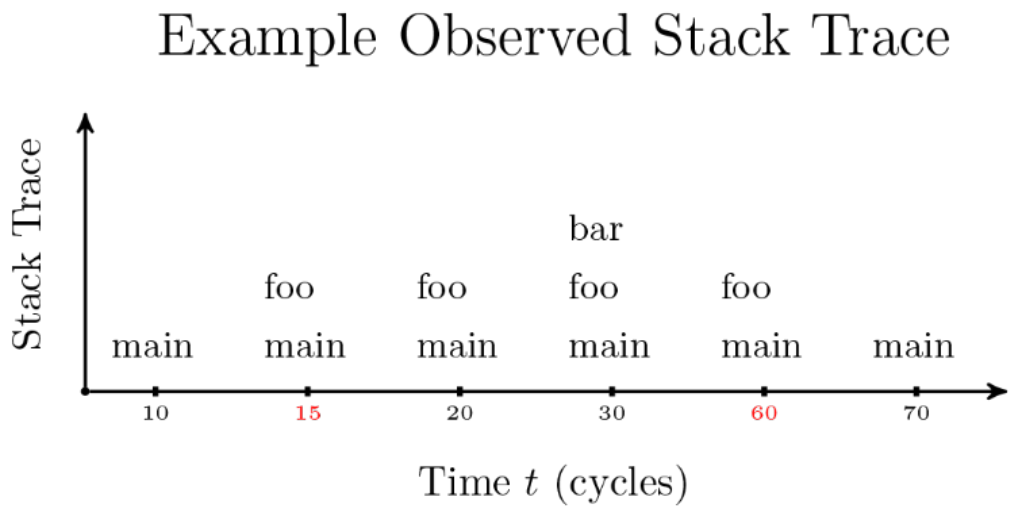
SMM Preliminary Results

Operation	Mean (μs)	STD (μs)
SMM Switch	3.29	0.08
Program Acquisition	2.19	0.09
SMI Configuration	1.66	0.06
SMM Resume	4.58	0.10
Total	**11.72**	



Clarifying Stack Traces





More on ARM Branches

- ARM branching instructions
  - B, BL, BX, BLX, BXJ (one register operand)
  - IT (x, y, z, cond)
  - CBZ CBNZ (one register operand)
  - TBB TBH (two register operands)

Relating Quadcopters to Malware

- Technical challenges posed by patching quadcopter firmware are indicative of challenges in stealthy malware analysis
- If we insert the patch too early, self-verifying code could prevent patch from executing
- Failure to path requires repeated attempt
  - This is the heart of *transparent breakpointing*