Microsoft

CanSecWest

Vancouver | 15 March 2017

Microsoft Threat Intelligence Center

Microsoft

Harnessing Intel Processor Trace on Windows for Vulnerability Discovery

Andrea Allievi – Microsoft Ltd
Richard Johnson – Cisco Systems Inc

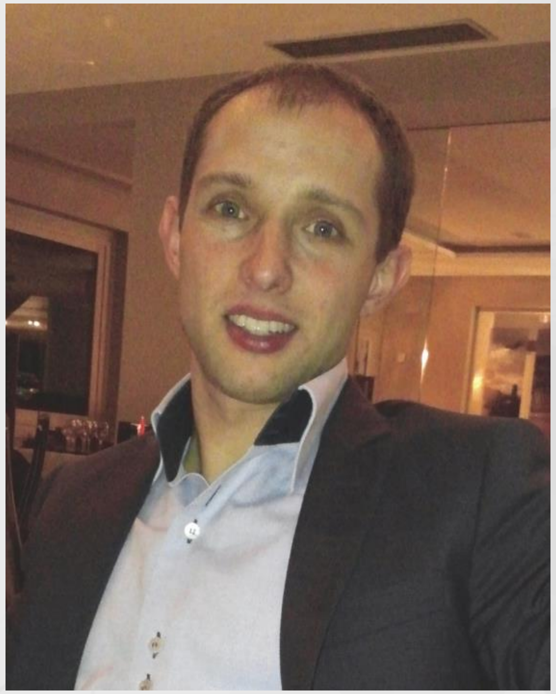
Microsoft Threat Intelligence Center

Who we are - Richard Johnson

- Research Lead
- Cisco Talos VulnDev

- Talos Vulndev
  - Third party vulnerability research
    - 170 bug finds in last 12 months
      - Microsoft
      - Apple
      - Oracle
      - Adobe
      - Google
      - IBM, HP, Intel
      - 7zip, libarchive, NTP
  - Security tool development
    - Fuzzers, Crash Triage
  - Mitigation development
    - FreeSentry

Who we are - Andrea Allievi



- Italian Security research Engineer, mainly focused on OS Security, Kernel Analysis and Malware Research
- Microsoft OSs Internals enthusiast / Kernel system level developer
- Work for the Threat Intelligence Center of Microsoft Ltd (MSTIC)
- Previously worked for Cisco Systems in the TALOS Security Research and Intelligence Group
- Previously worked for PrevX, Webroot and Saferbytes
- Original designer of the first UEFI Bootkit in 2012, Patchguard 8.1 bypass in 2014, and other research projects/analysis
- Windows Intel Pt Driver designer and developer

Introduction

- In 2014 - 2016 I have been researching high performance tracing and fuzzing
  - 2014/2015 - High Performance Fuzzing
  - 2015/2016 - Go Speed Tracer

- Ruxcon 2015 I demoed a working prototype of Intel PT for coverage fuzzing

- June 2016 we developed a prototype Intel Processor Trace driver for Windows
  - The driver has been released open-source:
    - https://github.com/intelpt

- This talk picks up where the last one left off...
  - See http://moflow.org for previous slides and talk videos

Intel Processor Trace

Intel Processor Trace

- Intel Processor Trace is a low-overhead hardware execution tracing feature
- It works by capturing information about software execution on each **hardware thread** using dedicated hardware in the CPU's Performance Monitoring Unit (PMU)
- After the execution completes software can process the captured trace data and reconstruct the exact program flow
- The trace format is highly compressed for efficient logging and requires some effort to decode

Why is this useful?

- Diagnostic code coverage
- Coverage driven fuzzing – automatically find software vulnerabilities
- Malware analysis – sandboxes can trace malware and feed it to the detection filtering platform
  - Current malware does not attempt to discover intelpt tracing*

Detecting Intel PT

- CPUID with leaf 0x7 can detect the support for Intel PT
- If supported, CPUID with leaf 0x14 can return the supported PT features
- Different CPUs implement different capabilities
- The architecture defines different MSRs to control each tracing operation

- Intel initially released Intel PT as part of Broadwell architecture
  - Limited tracing and logging modes
- Intel expanded on the functionality in Skylake
  - Multiple log buffer management modes
- Skylake architecture to be available on Xeon CPUs in 2017

Detecting Intel PT

```
INTEL_PT_CAPABILITIES ptCap = { 0 };
int cpuid_ctx[4] = { 0 };// EAX, EBX, ECX, EDX

// Processor support for Intel Processor Trace is indicated by
// CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1.
__cpuidex(cpuid_ctx, 0x07, 0);
if (!(cpuid_ctx[1] & (1 << 25))) return FALSE;

// Now enumerate the Intel Processor Trace capabilities
RtlZeroMemory(cpuid_ctx, sizeof(cpuid_ctx));
__cpuidex(cpuid_ctx, 0x14, 0);
// If the maximum valid sub-leaf index is 0 exit immediately
if (cpuid_ctx[0] == 0) return FALSE;
```

Detecting Intel PT

EAX = 0x14 - Intel Processor Trace

EBX

- Bit 00: IA32_RTIT_CTL.CR3Filter can be set to 1
  - IA32_RTIT_CR3_MATCH MSR can be accessed.
- Bit 01: Configurable PSB and Cycle-Accurate Mode.
- Bit 02: IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset.
- Bit 03: MTC timing packet and suppression of COFI-based packets.

ECX

- Bit 00: Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1 utilizing the ToPA output scheme
  - IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed.
- Bit 01: ToPA tables can hold any number of output entries
  - Maximum specified by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS.
- Bit 02: Single-Range Output scheme.
- Bit 03: Output to Trace Transport subsystem.
- Bit 31: Generated packets which contain IP payloads have LIP values
  - Includes the CS base component

Detecting Intel PT

EAX = 0x14 - Intel Processor Trace

Packet Generation (ECX = 1)

EAX

- Bits 2:0: Number of configurable Address Ranges for filtering.
- Bit 31:16: Bitmap of supported MTC period encodings

EBX

- Bits 15-0: Bitmap of supported Cycle Threshold value encodings
- Bit 31:16: Bitmap of supported Configurable PSB frequency encodings

Why is Intel PT so interesting?

- Implemented entirely in hardware
- You can trace all software that the CPU runs (except for SGX secure containers)
- Suppose you have to analyze an hypervisor or an evil SVM handler
  - With Intel PT you **can do that**!

- Performance
  - Low over-head (15% CPU perf hit for recording)
  - Logs directly to physical memory, bypassing TLB and eliminating cache pollution
  - Minimal log format takes little time to record
    - One bit per conditional branch
    - Only indirect branches log dest address

How it works - Summary

- Different kinds of trace filtering:
  1. Current Privilege Level (CPL) – used to trace all of user or kernel
  2. PML4 Page Table – used to trace a single process
  3. Instruction Pointer – used to trace a particular slice of code (or module)

- Two types of output logging:
  1. Single Range
  2. Table of Physical Addresses

Single Range

- OS should allocate a contiguous physical memory buffer (*MmAllocateContiguousMemory* is a good fit)

- This mode is best suited for
  1. Tracing of single application with sufficient size of buffer
  2. Redirect the output to a MMIO port or some JTAG controllers
  3. Always-On tracing for post-mortem or forensic analysis

- To enable:
  - Set the proper MSRs
    - `MSR_IA32_RTIT_OUTPUT_BASE` and `MSR_IA32_RTIT_OUTPUT_MASK_PTRS`
- Start the Tracing by setting the “`TraceEn`” flag in the control register
- The buffer will be filled by the processor in a circular-manner

Table of Physical Addresses

- Table of Physical Addresses (aka ToPA) is a list of tables that describes each physical address used for storing the trace
- A well-known data-structure definition PML4 (see the Intel Manual)
- This allows the processor to write data to non-contiguous memory regions
- Binary compatibility with the "MDL" data structure of Windows kernel

- Modality best suited for:

1. Tracing big code areas and/or dump the results in a user-mode file
2. Supporting pause/resume of a application and on-the-fly analysis of the dump

- Very powerful – an Interrupt could be generated by the processor at a certain point if the buffer is going to be full, or STOP signal

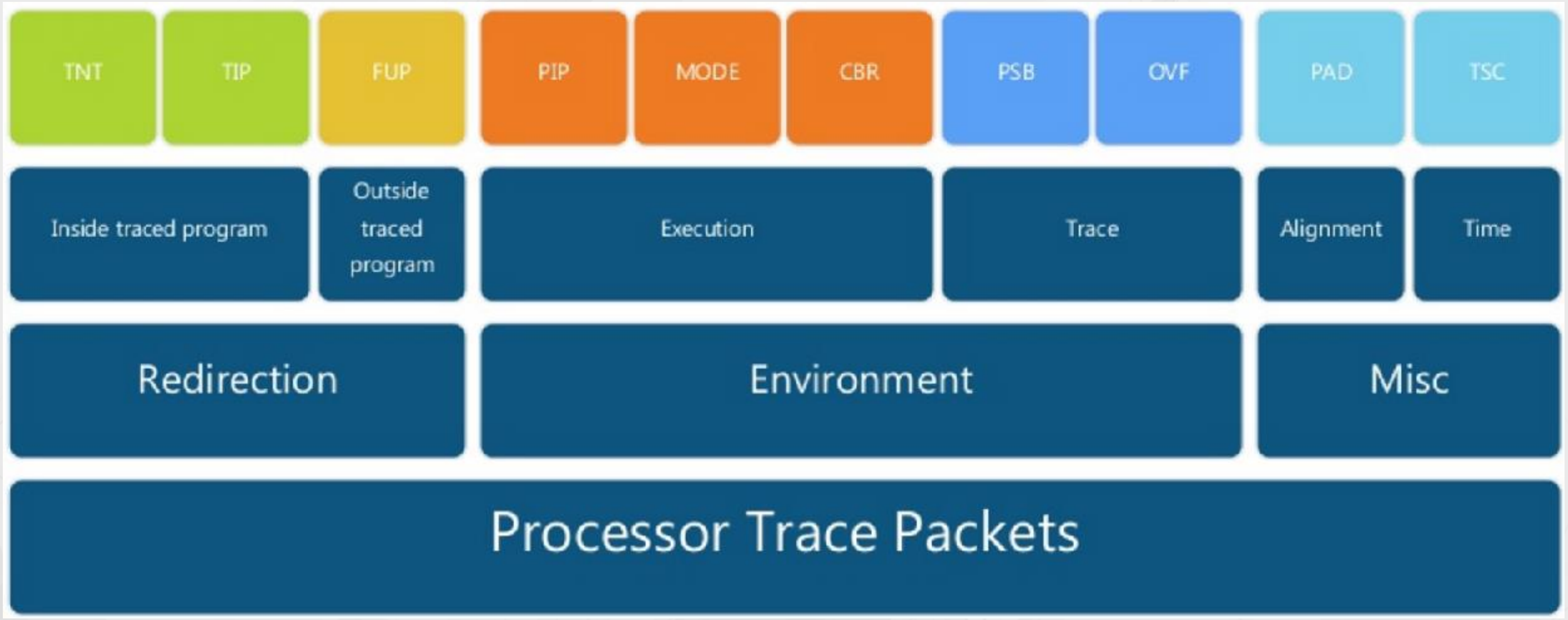
Different type of Trace Packets

1. Packet Stream Boundary (**PSB**) packets - ‘heartbeats’ that are generated at regular intervals (configurable), synchronization points for the decoder
2. Paging Information Packets (**PIP**) - record modifications made to the **CR3** register
3. Timing packets (**TSC, MTC & CYC**) packets - helps in tracking wall-clock time (related to events or not)
4. Control flow packets: taken-not-taken (TNT), target IP (TIP), Flow update (FUP), MODE packets
5. Core bus ratio packets: highlights modifications in the CPU clock
6. Overflow packets: sent when the processor encounters an internal buffer overflow

In our driver the user can decide if enable or not the generation of some kind of packets (control flow – TSC/MTC/CYC)

* Refer to the Intel’s manual for the details

Different type of Trace Packets



Windows Intel PT Driver

The Project

- We have decided to write a Windows driver, with the goal of supporting all trace and filtering modes for kernel and userspace

- At the time of this writing the driver is in version 0.5
  - Supports all the filtering mode combinations and both output modes
  - Supports multi-processors
  - Supports kernel mode code tracing and kernel mode API

- Some issues had been resolved:
  1. APIC controller programming for the PMI interrupt notification
  2. User-mode buffer mapping
  3. Multi processor issues
  4. How to trace spawned processes

The PMI Interrupt

- The ToPA output scheme supports a mode in which the CPU triggers a PMI (performance monitor interrupt) every time the buffer is full*
- We would like to enable and connect to that interrupt
- In that way we can process the trace content when buffer is full
- To control the traced process, either
  - Use a hypervisor -> VMEXIT
  - Suspend the target process from kernel, dump the trace data and resume
- Another problem here: the IRQL in which the code runs is HIGH_LEVEL
- Solved dividing the job in 3 phase: PMI Handler -> DPC -> Work Item
- Connecting the ISR and find a way to map the IoApic memory space have been not an easy task

The User mode buffer

- Processor Trace works with physical addresses - not Virtual addresses
- ToPAs describe a big buffer composed by different smaller physical chunks.
- Need a way to create a big virtual buffer composed by each chunk and map this to user-mode in a very secure manner (otherwise the driver will be subject of kernel-exploitation)
- Intel is not stupid. The ToPA and the MDL data structures are compatible
- Solution:
    - allocate physical memory using the OS facilities*
    - Convert the MDL descriptor into ToPA entries
    - Securely map the final virtual buffer using the OS

Multi-Processor and Multi Thread support

- **New** feature in version 0.5
- Each processor has an associated PT Buffer mapped in the target user-mode process (but **not in kernel-mode**)
- Only an event signaled when the PMI Interrupt fires was not enough
  - Introduced the User-mode callbacks – a smart method to manage the PT log directly from User-mode
- Still some problems in managing multi-threaded and multi process application

Kernel mode Tracing

- **New** feature in version 0.5
- The Driver is able to perform the tracing of Kernel mode code in 2 ways

1. From the user-mode application (executed with Admin privileges) -> Uses IP filtering mode
2. From another kernel-mode driver -> the driver must use the exported APIs and manage the PT buffer(s), and multi-processor stuff on its own

- In this way we have been able to perform the trace of:

1. The loading / unloading of a new Kernel module
2. Some IOCTL called by a test user application

The client code

Quite a simple setup:

1. Get an handle to the PT Device

```
hPtDev = CreateFile(L"\\\\.\\WindowsIntelPtDev", FILE_ALL_ACCESS, 0, NULL,
    OPEN_EXISTING, 0, NULL);
```

2. Spawn the process / decide what to trace and set the options in the `PT_USER_REQ` data structure (process ID, CPU Affinity mask, buffer size, …)

3. Start the tracing

```
DeviceIoControl(hPtDev, IOCTL_PTDRV_START_TRACE, (LPVOID)&ptStartStruct,
   sizeof(PT_USER_REQ), lpPtBuffArray, sizeof(LPVOID) * dwNumOfCpus,
   &dwBytesIo, NULL);
```

4. Stop the trace and clear the resources (important)

```
bRetVal = DeviceIoControl(hPtDev, IOCTL_PTDRV_CLEAR_TRACE,
    (LPVOID)&dwTargetCpu, sizeof(DWORD), NULL, 0, &dwBytesIo, NULL);
```

The Multiprocessor client code

1. Spawn a new thread for each CPU
2. To register the user-mode callback use the new PTDRV_REGISTER_PMI_ROUTINE IOCTL code (one call for each thread)
3. Specify an affinity mask composed by only the executing processor ID
4. Perform a wait in an alertable state

That's all!

Your User-mode callback will be called each time the CPU trace buffer will become full

Some other challenges

- CR3 physical page swappable?
  - Quick analysis shows that in Windows 10586
    - **Only the main PML4 table page** is **always** in memory
- Otherwise make use of the **PIP** packets

- The problem of the spawned processes has been resolved using the trace by IP – detect when a new process is spawned and add the new range

OR

- Use the tracing by CPL and parse the PIP packets

Demo

Vulnerability Discovery

- Now we have a fast tracing engine
- How will we utilize it for vulnerability discovery?

Evolutionary Fuzzing

- Incrementally better mutational dumb fuzzing
- Trace while fuzzing and provide feedback signal
- Evolutionary algorithms
  - Assess fitness of current input
  - Manage a pool of possible inputs
- Focused on security bugs

Evolutionary Fuzzing

- From previous research, these are the required components
  - Fast tracing engine
    - Block based granularity
  - Fast logging
    - Memory resident coverage map
  - Fast evolutionary algorithm
    - Minimum of global population map, pool diversity

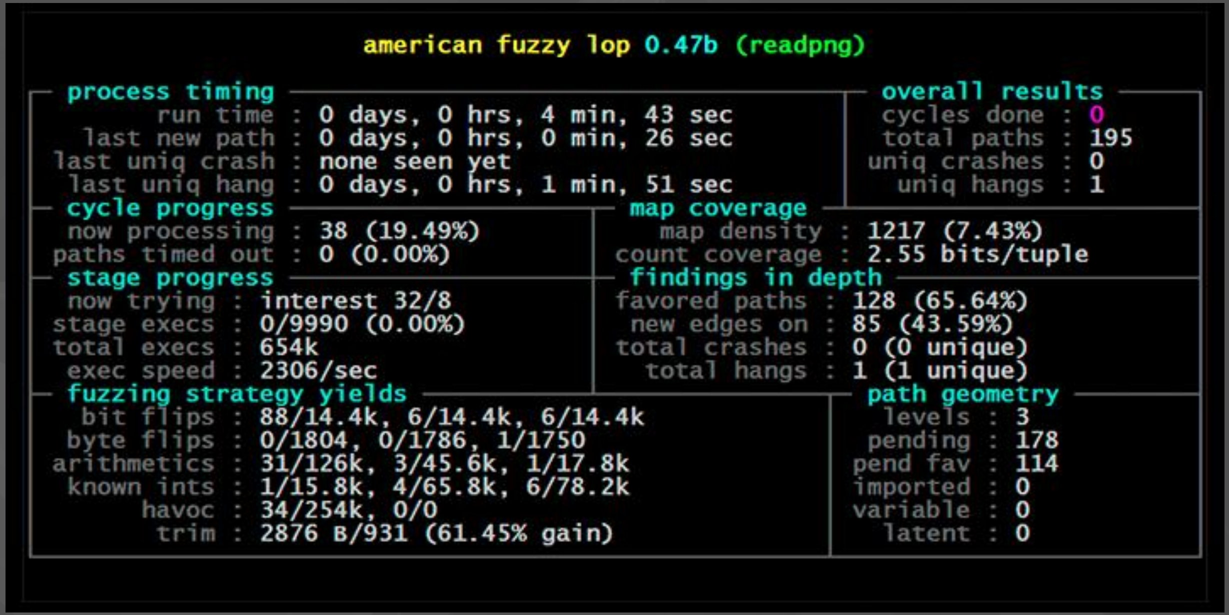
Amercian Fuzzy Lop

- Michal Zalewski 2013
  - Delivered the first performant opensource evolutionary fuzzer
- Features
  - Uses variety of traditional mutation fuzzing strategies
  - Block coverage via compile time instrumentation
  - Simplified approach to genetic algorithm
    - Edge transitions are encoded as tuple and tracked in a bloom filter
    - Includes coverage and frequency

  - Uses portable* Posix API for shared memory, process creation

Amercian Fuzzy Lop

- Contributions
  - Tracks edge transitions
    - Not just block entry
  - Global coverage map
    - Generation tracking
  - Fork server
    - Reduce fuzz target initialization
  - Persistent mode fuzzing
  - Builds corpus of unique inputs reusable in other workflows



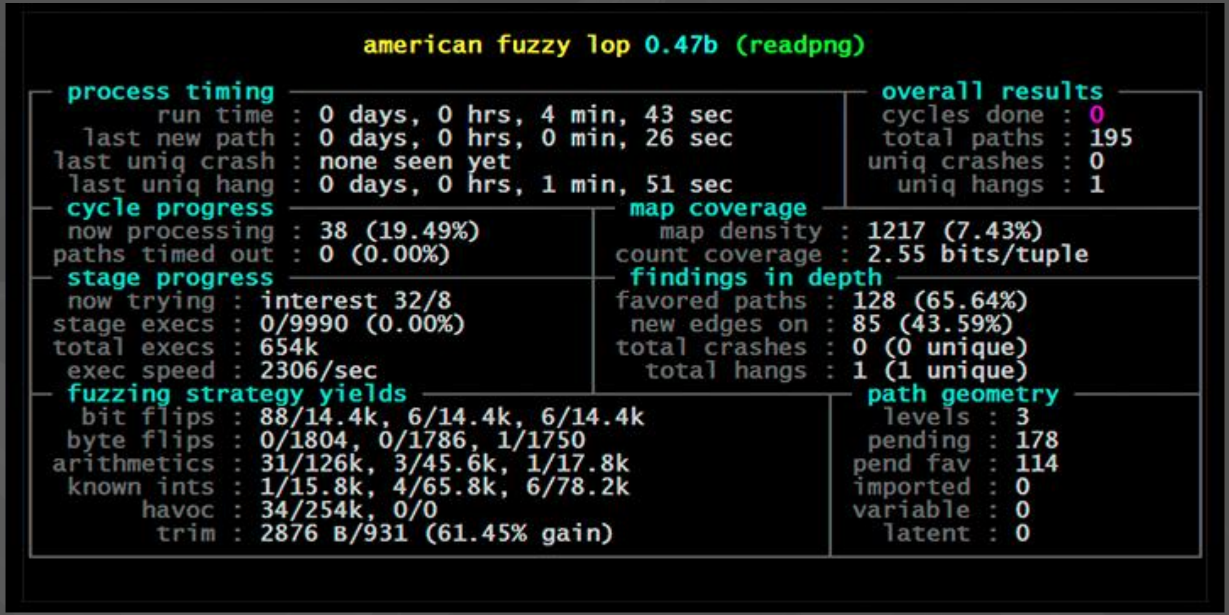
```
                    american fuzzy lop 0.47b (readpng)

 process timing                                        overall results
        run time : 0 days, 0 hrs, 4 min, 43 sec        cycles done : 0
   last new path : 0 days, 0 hrs, 0 min, 26 sec        total paths : 195
 last uniq crash : none seen yet                      uniq crashes : 0
  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec         uniq hangs : 1
 cycle progress                           map coverage
  now processing : 38 (19.49%)              map density : 1217 (7.43%)
 paths timed out : 0 (0.00%)             count coverage : 2.55 bits/tuple
 stage progress                           findings in depth
  now trying : interest 32/8             favored paths : 128 (65.64%)
 stage execs : 0/9990 (0.00%)             new edges on : 85 (43.59%)
 total execs : 654k                      total crashes : 0 (0 unique)
  exec speed : 2306/sec                    total hangs : 1 (1 unique)
 fuzzing strategy yields                                path geometry
   bit flips : 88/14.4k, 6/14.4k, 6/14.4k                levels : 3
  byte flips : 0/1804, 0/1786, 1/1750                   pending : 178
 arithmetics : 31/126k, 3/45.6k, 1/17.8k               pend fav : 114
  known ints : 1/15.8k, 4/65.8k, 6/78.2k               imported : 0
       havoc : 34/254k, 0/0                            variable : 0
        trim : 2876 B/931 (61.45% gain)                  latent : 0
```

Amercian Fuzzy Lop

- Trace Logging
  - Each block gets a unique ID
  - Traversed edges are indexed into a bloom filter map
  - Create a hash from the src and dst block IDs
  - Increment map for each time an edge is traversed

  - Each trace is easily comparable to the entire session history



```
                 american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec        │  total paths : 195    │
│ last uniq crash : none seen yet                       │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec        │   uniq hangs : 1      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 38 (19.49%)       │    map density : 1217 (7.43%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.55 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 32/8         │ favored paths : 128 (65.64%)           │
│ stage execs : 0/9990 (0.00%)        │  new edges on : 85 (43.59%)            │
│ total execs : 654k                  │ total crashes : 0 (0 unique)           │
│  exec speed : 2306/sec              │   total hangs : 1 (1 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k            │    levels : 3          │
│  byte flips : 0/1804, 0/1786, 1/1750                │   pending : 178        │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k             │  pend fav : 114        │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k             │  imported : 0          │
│       havoc : 34/254k, 0/0                          │  variable : 0          │
│        trim : 2876 B/931 (61.45% gain)              │    latent : 0          │
└─────────────────────────────────────────────────────┴────────────────────────┘
```

Windows Evolutionary Fuzzing

- Started research into this area in 2015
  - High Performance Fuzzing
  - Go Speed Tracer
- Windows Software primarily distributed as binaries
  - High speed binary code coverage required
- Seemed like a good opportunity to use Intel Processor Trace
  - First prototyped on Linux using simple-pt
  - Demoed Linux afl-intelpt at Ruxcon 2015
- Lack of a usable driver for Windows lead to partnership with Andrea

WinAFL

- Ivan Fratric July 2016
  - First performant windows evolutionary fuzzer
- Features
  - Its American Fuzzy Lop! For Windows!
  - Windows API port for memory and process creation
  - DynamoRIO based code coverage
  - Filter based on module
  - Block and Edge tracing modes
    - Block tracing by default due to issues with multi-threading
  - Persistent execution mode

WinAFL

- Ivan Fratric July 2016
  - First performant windows evolutionary fuzzer
- Persistence
  - Multiple inputs can be parsed without exiting the process
  - DynamoRIO allows hooking of target function
    - User specifies address and number of arguments
    - On function exit, WinAFL repopulates args and loops function
    - User specifies number of loops before process restart

WinAFL

- Ivan Fratric July 2016
  - First performant windows evolutionary fuzzer
- Persistence is key
  - Restart process each time (disable persistence) ~2.3 exec/s
  - Persist 100 iterations before restart ~72 exec/s
  - Persist 1000 iterations ~123 exec/s
  - Persist 10000 iterations ~133 exec/s

WinAFL IntelPT

- Richard Johnson 2016
  - Windows hardware driven evolutionary fuzzer
- Key problems to solve
  - The IntelPT log does not contain Block IDs or all branch targets
  - Parsing large compressed logs is time consuming
  - Native persistence mode is not yet implemented
    - *Work in progress using Avrf as hooking engine
  - We can filter up to 4 address ranges or whole process

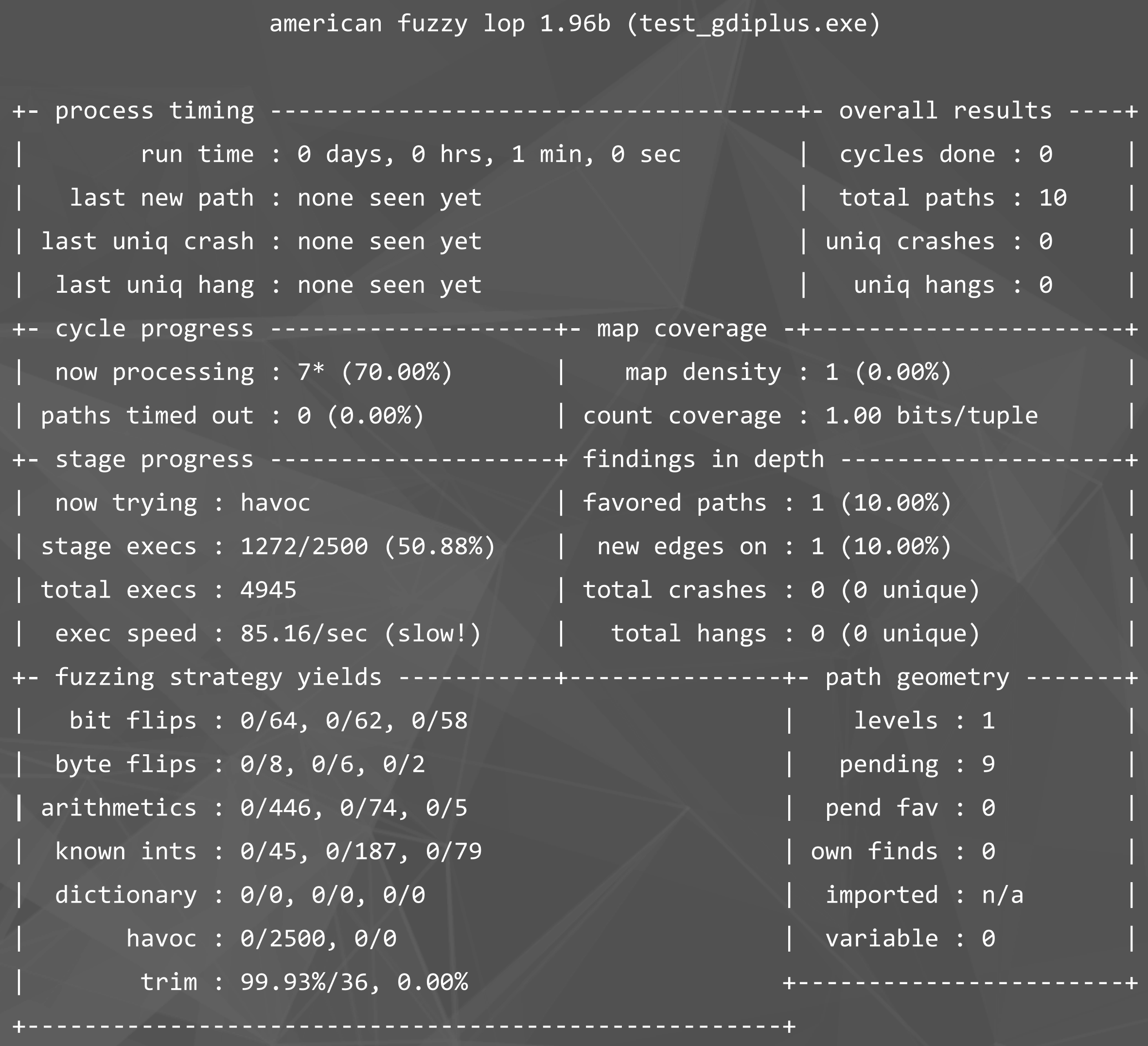
WinAFL IntelPT

- Richard Johnson 2016
  - Windows hardware driven evolutionary fuzzer
- Current status
  - WinAFL IntelPT now accurately decodes full trace
  - The TIP packet of IntelPT holds target addresses
    - Generated for indirect branches and return
  - The TNT packets are conditional branch states
    - We must disassemble from last known IP to recover conditional branch target
    - We use a discovered branch cache to reduce disassembly time (needs persist to disk*)
  - Edge src/dst encoded into AFL bloom filter

  - We currently use CreateProcess and WaitForSingleObject
    - See Go Speed Tracer for experiments in Windows fork()

WinAFL IntelPT

- Performance
  - Dummy loop benchmark
  - CreateProcess / Wait
  - 85 exec/sec



```
                 american fuzzy lop 1.96b (test_gdiplus.exe)

+- process timing -------------------------------------+- overall results ----+
|        run time : 0 days, 0 hrs, 1 min, 0 sec         |  cycles done : 0     |
|   last new path : none seen yet                       |  total paths : 10    |
| last uniq crash : none seen yet                       | uniq crashes : 0     |
|  last uniq hang : none seen yet                       |   uniq hangs : 0     |
+- cycle progress --------------------+- map coverage -+----------------------+
|  now processing : 7* (70.00%)       |    map density : 1 (0.00%)            |
| paths timed out : 0 (0.00%)         | count coverage : 1.00 bits/tuple      |
+- stage progress --------------------+ findings in depth --------------------+
|  now trying : havoc                 | favored paths : 1 (10.00%)            |
| stage execs : 1272/2500 (50.88%)    |  new edges on : 1 (10.00%)            |
| total execs : 4945                  | total crashes : 0 (0 unique)          |
|  exec speed : 85.16/sec (slow!)     |   total hangs : 0 (0 unique)          |
+- fuzzing strategy yields -----------+---------------+- path geometry -------+
|   bit flips : 0/64, 0/62, 0/58                      |    levels : 1         |
|  byte flips : 0/8, 0/6, 0/2                         |   pending : 9         |
| arithmetics : 0/446, 0/74, 0/5                      |  pend fav : 0         |
|  known ints : 0/45, 0/187, 0/79                     | own finds : 0         |
|  dictionary : 0/0, 0/0, 0/0                         |  imported : n/a       |
|       havoc : 0/2500, 0/0                           |  variable : 0         |
|        trim : 99.93%/36, 0.00%                      +-----------------------+
+-----------------------------------------------------+
```

WinAFL IntelPT

- Performance
  - Trace enabled
  - No log parsing
  - 72 exec/sec
  - 15% tracing overhead

```
                  american fuzzy lop 1.96b (test_gdiplus.exe)

+- process timing -------------------------------------+- overall results ----+
|        run time : 0 days, 0 hrs, 1 min, 0 sec        |  cycles done : 0     |
|   last new path : none seen yet                      |  total paths : 10    |
| last uniq crash : none seen yet                      | uniq crashes : 0     |
|  last uniq hang : none seen yet                      |   uniq hangs : 0     |
+- cycle progress --------------------+- map coverage -+----------------------+
|  now processing : 7* (70.00%)       |    map density : 1 (0.00%)            |
| paths timed out : 0 (0.00%)         | count coverage : 1.00 bits/tuple      |
+- stage progress --------------------+ findings in depth --------------------+
|  now trying : havoc                 | favored paths : 1 (10.00%)            |
| stage execs : 763/2500 (30.52%)     |  new edges on : 1 (10.00%)            |
| total execs : 4436                  | total crashes : 0 (0 unique)          |
|  exec speed : 72.19/sec (slow!)     |   total hangs : 0 (0 unique)          |
+- fuzzing strategy yields -----------+---------------+- path geometry -------+
|   bit flips : 0/64, 0/62, 0/58                     |    levels : 1         |
|  byte flips : 0/8, 0/6, 0/2                        |   pending : 9         |
| arithmetics : 0/446, 0/74, 0/5                     |  pend fav : 0         |
|  known ints : 0/45, 0/187, 0/79                    | own finds : 0         |
|  dictionary : 0/0, 0/0, 0/0                        |  imported : n/a       |
|       havoc : 0/2500, 0/0                          |  variable : 0         |
|        trim : 99.93%/36, 0.00%                     +-----------------------+
+----------------------------------------------------+
```

WinAFL IntelPT

- Performance
  - Full tracing and parsing
  - 55 exec/sec
  - 22% parsing overhead

  - Total of ~35% overhead

```
                 american fuzzy lop 1.96b (test_gdiplus.exe)

+- process timing -------------------------------------+- overall results ----+
|        run time : 0 days, 0 hrs, 1 min, 0 sec        |  cycles done : 0     |
|   last new path : 0 days, 0 hrs, 0 min, 0 sec        |  total paths : 48    |
| last uniq crash : none seen yet                      | uniq crashes : 0     |
|  last uniq hang : none seen yet                      |   uniq hangs : 0     |
+- cycle progress --------------------+- map coverage -+----------------------+
|  now processing : 0 (0.00%)         |    map density : 2594 (3.96%)         |
| paths timed out : 0 (0.00%)         | count coverage : 1.49 bits/tuple      |
+- stage progress --------------------+ findings in depth --------------------+
|  now trying : calibration           | favored paths : 9 (18.75%)            |
| stage execs : 20/40 (50.00%)        |  new edges on : 47 (97.92%)           |
| total execs : 3359                  | total crashes : 0 (0 unique)          |
|  exec speed : 55.81/sec (slow!)     |   total hangs : 0 (0 unique)          |
+- fuzzing strategy yields -----------+---------------+- path geometry -------+
|   bit flips : 0/0, 0/0, 0/0                         |    levels : 2         |
|  byte flips : 0/0, 0/0, 0/0                         |   pending : 48        |
| arithmetics : 0/0, 0/0, 0/0                         |  pend fav : 9         |
|  known ints : 0/0, 0/0, 0/0                         | own finds : 37        |
|  dictionary : 0/0, 0/0, 0/0                         |  imported : n/a       |
|       havoc : 0/0, 0/0                              |  variable : 47        |
|        trim : 0.00%/1341, n/a                       +-----------------------+
+-----------------------------------------------------+
```

Demo

WinAFL + IntelPT

Conclusions

- Tracing is used very often in fuzzing and dynamic analysis
- Intel Processor Trace is a promising mechanism for hardware tracing
- Intel is dedicated to producing high performance trace features

- TODO List:
  1. Implement thread context switch tracing in a reliable way (ETW)
  2. Modify a Hypervisor to be able to use Intel PT inside a Guest VM
  3. Understand how to trace VMM, SMM code and test with SGX software
  4. Enable persistent mode in native apps with Intel PT

Microsoft

Thank you!

https://github.com/intelpt

@richinseattle / rjohnson@moflow.org
@aall86

Questions?

Multi-Threaded and Multi-Process applications

- Always increasing in their number (think about AppContainer or Browsers for example)
- A simple solution resides in the log parser:
  - Make use of the PIP (Page information packets) to identify each process
- Big drawbacks: the size of the log is HUGE – the time needed to parse it is even MORE
- Register a Process / Thread Creation callback in Kernel mode and trace one process per time
  - Simple solution, log size still acceptable
  - Some malware or complex applications requires process interactions

BUT …

In the beginning was a PUSHAD …

- Do you remember the old glorious PUSAHD instructions?
- From the Intel manuals: "Pushes the contents of the general-purpose registers onto the stack."
- No equivalence for X64 registers or Kernel MSR
- I was studying how to trace only a single thread, intercepting the Windows Thread Context Switcher
- Someone has pinpoint to me the existence of another very-cool instruction in the AMD64 architecture, but no so known by the research community

Special thanks to Xinyang Ge of Microsoft Research for signaling this

… and now it is XSAVE

- Saves some processor state components to the XSAVE area
- MMX, SSE, AVX, AVX-512 user mode registers (What a heck is AVX-512?)
- … and even the **new CPU registers that belongs to Intel PT and Intel MPX**
- New CPUID leaf functions for compatibility verification, new CPUs opcodes
- Basically is a **very fast way** to save even X64 Kernel-accessible Register in a particular memory buffer
- To use this feature in user-mode you have to fill the XCR0 register with XSETBV instruction
- Instead for kernel mode staff, you have to fill a special MSR register: IA32_XSS (number 0x0DA0)
- Finally a call to the XSAVE (or XSAVES if in Kernel mode) fills the buffer with the needed information *

Thread tracing

- Originally I planned to manual save each Intel PT MSRs after intercepting the thread context switcher
- While analyzing the Windows 10 Context Switcher, I realized that it already supports the XSAVE feature
- 2 solutions -> We conclude that it was not feasible in a very stable manner:
  1. Find a way to hook or divert the KeSwapContext routine -> No public-available method -> Patchguard become angry
  2. Use ETW

Research still in progress!