Real-time Multi-process Tracing Decoder Architecture

Jun 24, 2019

Youngsoo Kim

(Electronics & Telecommunications Research Institute)

Introduction (Tracing)

- A technique used to determine how the system is behaving
  - Debugging or Monitoring purpose
  - Statically or dynamically set trace points to collect information
- Tracing information
  - typically used for debugging purposes
  - used to detect diverse programming problems

    depending on details of the information contained in the trace log
  - used both in the development cycle and after the software release
  - Include IRQ handlers, system calls, scheduling activities, network activities, and tracepoints within applications

Introduction (Tracing)

- Event logging
  - It is primarily used by system administrators
  - It records high-level information such as program installation errors
  - Event logs are clearly and concisely expressed in the standard form
- Tracing
  - It is primarily used by developers
  - It records low-level information such as thrown exceptions
  - The output format is not standardized
  - Duplicate events or information can be recorded

Introduction (Tracing)

- Since the tracer generates and decodes huge amounts of data in real time, many tracers use dedicated hardware
  - ARM Coresight and Intel® PT (Processor Trace) are typical hardware-based tracers
- Intel® PT records all information related to software execution from each hardware thread
  - When the execution of the corresponding software is completed, the accurate program flow can be indicated through the recorded trace data

Introduction (Tracing)

- (*Linux* systems) Intel® PT-based hardware trace programs are integrated into the operating system and can be used through commands such as *perf*
- (*Windows* system) tight integration with profiling and debugging mechanisms is not achieved due to kernel-related policies
- To solve this problem, many attempts have been made to implement the Intel® PT in *Windows* environment
- Both *perf* and *Windows* environments can only trace *a single process* using the Intel® PT and do not provide a way of tracing *multiple process* streams
  - If we have a method of tracing multiple process streams, we can use it in diverse fields, *specially data analysis for preventing some cyber threats*

Introduction (Tracing)

- In this paper,
  - Propose a way of extending the Intel® PT trace program to provide the ability to trace multiple process streams in *Windows* environment for various applications
  - Introduce the main features of the Intel® PT and related software
  - Describe a decoder implementation that traces multiple process streams in *Windows* environment

Intel® PT

- A hardware feature logging all the information about software execution
- Intel® PT decoder
  - Reconstruct accurate software execution flow from stored trace information
  - Store cycle count and timestamp information to synchronize with other trace log
- The trace log includes
  - Program flow information
  - Program trigger mode related information
- The debugger can use the trace log information
  - To reconstruct the code flow leading to a specific location

Intel® PT (Main Features)

- Control Flow Tracing
  - Records branch to infer program flow
  - Configures MSR (Machine State Register)s
  - Setups buffers
  - Generates a trace packet
  - Stores trace packet in a buffer or forwards it to *transport layer*
- Trace Packet
  - Generates heartbeat packet, PSB (Packet Stream Boundary), every 4K packets
  - Creates PIP (Paging Information Packet) when changing CR3 (Control Register 3)
  - TSC (Time Stamp Counter), OVF (Overflow), CBR (Core Bus Ratio) packets
  - Flow control

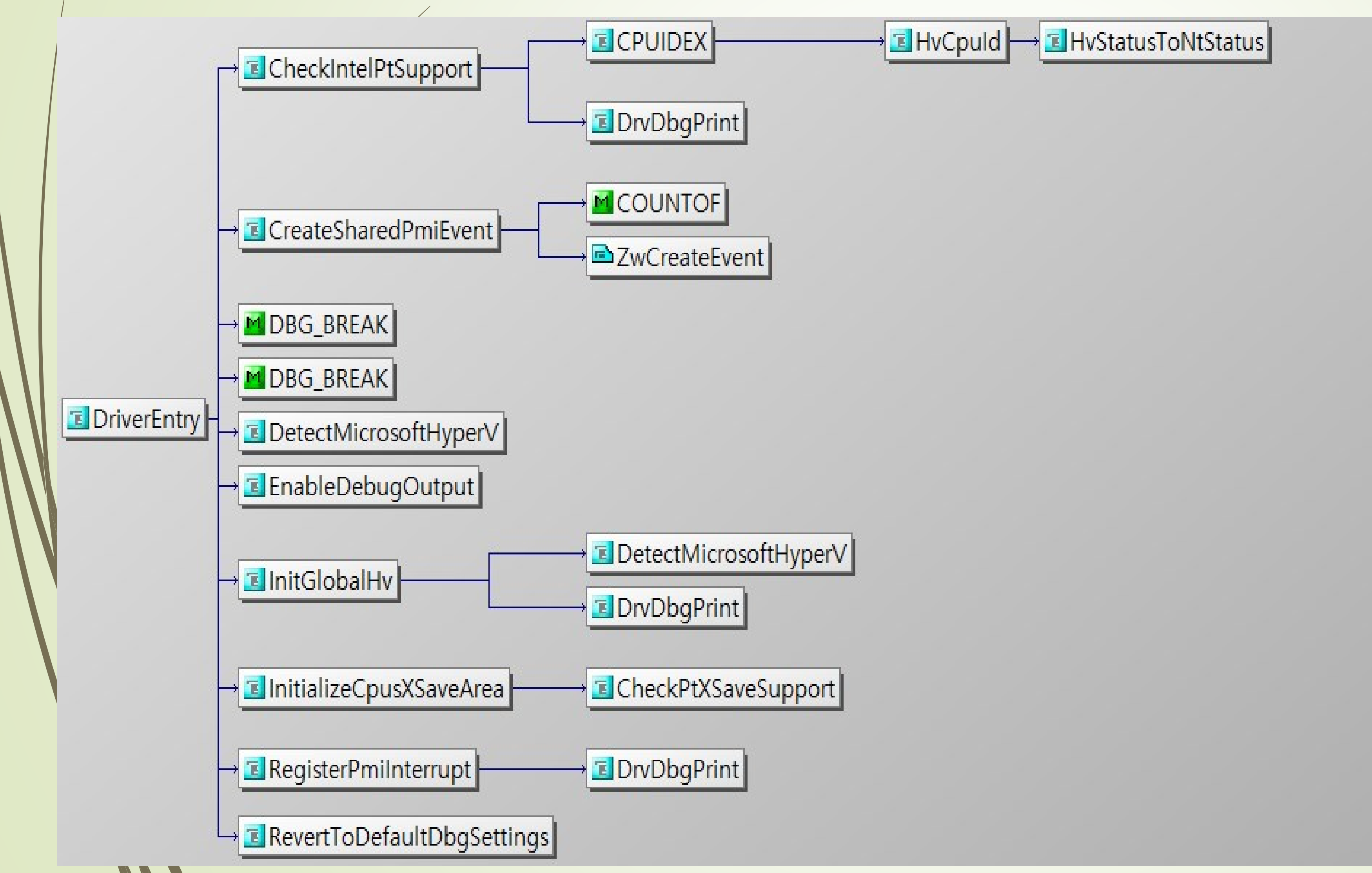
Intel® PT (Main Features)

- Cycle Accurate Mode (CYC Packet)
  - Cycle counter data related to instruction count, IPC, tracking wall-clock time
- MTC (Mini Timestamp Counter)
  - A more commonly used timer based on the crystal clock counter (CTC)
  - Used with the TSC (Time Stamp Counter) to obtain accurate timestamp values at low cost
- Perf Support
  - Perf driver can configure or control PT hardware
  - Trace data generation on perf buffer
  - Perf data decoding in userspace of perf

WindowsIntelPT driver

- Attempts to implement PT drivers and applications in the Windows environment are underway
  - *Ex. WindowsIntelPT*
- The *WindowsIntelPT* driver
  - Implements the Intel® processor trace feature of the *Skylake* architecture in a Windows environment
  - Writes trace logs directly to physical memory to prevent cache and TLB polling
  - Uses a compressed logging format suitable for long-time tracing
  - Can track all branches of the CPU core, including user space and the kernel

WindowsIntelPT driver (DriveEntry())



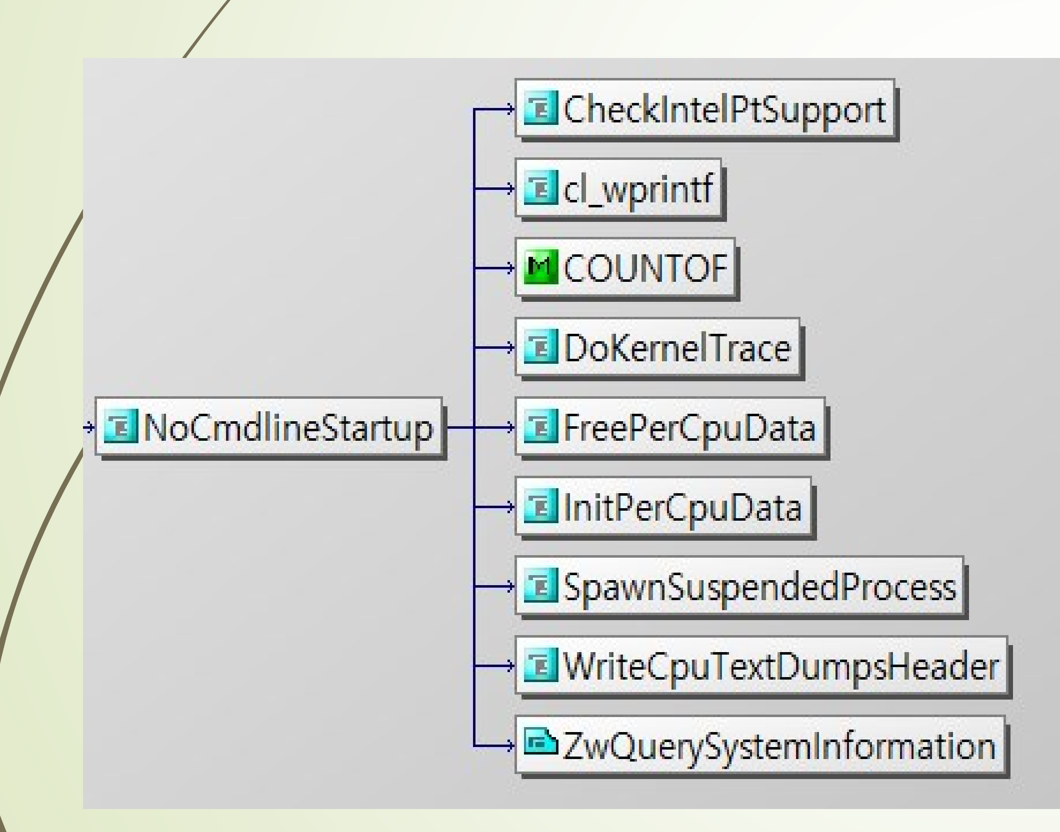
- When developing a driver, the entry function is *DriverEntry()*
- A function called by the system thread (I/O manager) at the time the driver file is loaded into memory
- Must be named *DriverEntry* when the driver is loaded into memory
  - Since the operating system first looks for a function with the name *DriverEntry*

WindowsIntelPT driver (DriveEntry() Parameters)

- The PUNICODE_STRING-> RegistryPath structure and the PDRIVER_OBJECT->DriverObject structure are required as parameters
  - The PDRIVER_OBJECT->DriverObject is a structure representing the driver
  - The PUNICODE_STRING->RegistryPath structure is the key value stored in the registry
    - \Registry\Machine\System\CurrentControlSet\Services\DriverName

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegPath)
{
    UNREFERENCED_PARAMETER(pRegPath);
    NTSTATUS ntStatus = STATUS_SUCCESS;
    KAFFINITY activeProcessorsMask = 0;                    // The active processors mask
    DWORD dwNumOfProcs = 0;                                // Number of system processors
    DWORD dwBuffSize = 0;                                  // The global driver data size in bytes
    UNICODE_STRING devNameString = { 0 };                  // The I/O device name
    UNICODE_STRING dosDevNameString = { 0 };               // The DOS device name (Usermode access)
    PDEVICE_OBJECT pDevObj = NULL;                         // The device object
    INTEL_PT_CAPABILITIES ptCap = { 0 };                   // The Intel PT Capabilities for this processor
```

WindowsIntelPT driver (DriveEntry() Functions in order)



① Identify the number of processors
② Allocate all data space for the driver
③ Check whether virtualization Hyper V is supported
④ Check whether PT is supported
⑤ Create a PMI (Performance Monitoring Interrupt) event and register interrupt
⑥ Initialize user mode callback function list
⑦ Initialize Unicode conversion
⑧ Create and control IOCTL
⑨ Create and control DriverObject
⑩ Create and control DeviceIoControl
⑪ Create and control PT_USER_REQ
⑫ Start/stop PT

Multi-process Tracer Extension

- We analyzed the *NoCmdlineStartup* function for multi-process analysis and studied the extension scheme
  - A function tracing without a command line argument
- For multi-process analysis, it is necessary to execute several device controls
  - To do this, it is necessary to declare array variables additionally
    - Ex) PT device handle value variable and target process path
- First, the dump file should be created and initialized for analyzing multi-processes
- Then, the PT device handles the value of the global variable of the initialized application information

Multi-process Tracer Extension

- It shows that when tracing for multiple processes, it determines the number of processors to run in the process, and scheduling should be added between processes and the available processors

```
if (sysInfo.dwNumberOfProcessors > 1) {
    // Ask how many processor to use
    wprintf(L"On how many processors would you like to run the process? [1/%i] ",
    wscanf_s(L"%i", &dwCpusCount);

    if (dwCpusCount > sysInfo.dwNumberOfProcessors) {
        wprintf(L"Invalid value, assuming all the processors as valid.\r\n");
        cpuAffinity = sysInfo.dwActiveProcessorMask;
        dwCpusCount = sysInfo.dwNumberOfProcessors;
    } else
        cpuAffinity = ((DWORD_PTR)(-1i64) >> ((sizeof(DWORD_PTR) * 8) - dwCpus
    if (FALSE)
        // If you would like to test the different affinities:
        cpuAffinity = 0xd;
    _ASSERT((sysInfo.dwActiveProcessorMask | cpuAffinity) == sysInfo.dwActiveProc
```

Multi-process Tracer Extension

- It shows the part responsible for creating and registering the performance measurement threads
  - When multiple analyses are concurrently performed, multi-thread generation is required

```
// Create the PMI threads (1 per target CPU)
for (int i = 0; i < (int)dwCpusCount; i++) {
    PT_PMI_USER_CALLBACK pmiDesc = { 0 };
    HANDLE hNewThr = NULL;
    DWORD newThrId = 0;

    hNewThr = CreateThread(NULL, 0, PmiThreadProc, (LPVOID)i, CREATE
    // Register this thread and its callback
    pmiDesc.dwThrId = newThrId;
    pmiDesc.kCpuAffinity = (1i64 << i);
    pmiDesc.lpAddress = PmiCallback;
    bRetVal = DeviceIoControl(hPtDev, IOCTL_PTDRV_REGISTER_PMI_ROUTI
    if (bRetVal) {
        pCpuDescArray[i].dwPmiThrId = newThrId;
        pCpuDescArray[i].hPmiThread = hNewThr;
        ResumeThread(hNewThr);
    }
}
#pragma endregion
```

Multi-process Tracer Extension

- It shows the kernel test mode
- It should be modified to find multiple addresses if multiple traces are running concurrently

```
else {
    // Grab the target module base address
    SYSTEM_ALL_MODULES * pSysAllModules = NULL;
    NTSTATUS ntStatus = 0;
    CHAR modNameAnsi[0x80] = { 0 };
    sprintf_s(modNameAnsi, COUNTOF(modNameAnsi), "%S", procPath);
    ntStatus = ZwQuerySystemInformation(11, pSysAllModules, 0, &dwBytesIo);
    if (ntStatus == STATUS_INFO_LENGTH_MISMATCH) {
        pSysAllModules = (SYSTEM_ALL_MODULES*)VirtualAlloc(NULL, dwBytesIo + 64
        RtlZeroMemory(pSysAllModules, dwBytesIo);

        ntStatus = ZwQuerySystemInformation(11, pSysAllModules, dwBytesIo, &dwB
        if (ntStatus == 0) {
            // Search for the SimplePt
            for (unsigned i = 0; i < pSysAllModules->dwNumOfModules; i++) {
                SYSTEM_MODULE_INFORMATION curMod = pSysAllModules->modules[i];
                LPSTR lpTargetModName = curMod.ImageName + curMod.ModuleNameOff
                if (_stricmp(modNameAnsi, lpTargetModName) == 0) {
                    // Target module found
                    wprintf(L"Found \"%S\" kernel driver in memory.\r\n", lpTar
                    remoteModInfo.lpBaseOfDll = curMod.Base;
                    remoteModInfo.SizeOfImage = curMod.Size;
                    break;
                }
            }
        }
    } « end if ntStatus==STATUS_INFO... »
    if (pSysAllModules) VirtualFree((LPVOID)pSysAllModules, 0, MEM_RELEASE);
    ptStartStruct.bTraceKernel = TRUE;
    ptStartStruct.bTraceUser = FALSE;
} « end else »
```

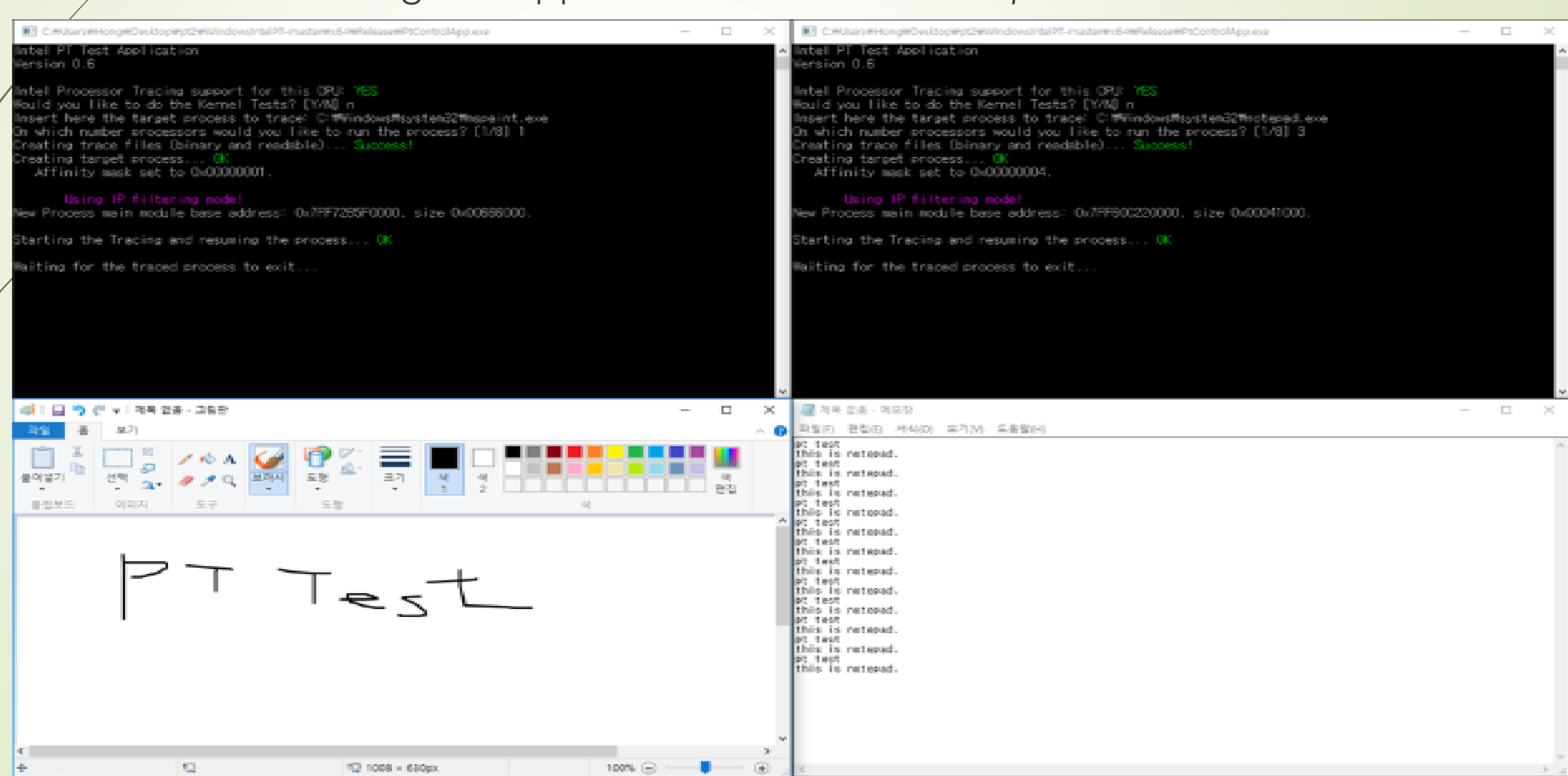
Multi-process Tracer Extension

```
#include <TlHelp32.h>
#include <tchar.h>
#include <psapi.h>
#include <process.h>

PROCESS_LIST pList = { {0},0 };
TCHAR ProcessPath[255];
TCHAR ProcessName[255];



DWORD WINAPI SearchProcessListThread(LPVOID lpParam) {
    HANDLE hProcess = NULL;
    //PROCESS_LIST * processlist = ((PROCESS_LIST *)lpParam);
    PROCESSENTRY32 pe32 = { 0 };
    PROCESS_INFORMATION pi = { 0 };
    int index = 0;
    int cnt = 0;
    char buf[260] = { NULL };
    TCHAR szImagePath[MAX_PATH] = { 0, };

    /* ... */

    ZeroMemory(szImagePath, sizeof(szImagePath));

    printf("[%25s\t%5s]\n", "System Process", "PID");
    pe32.dwSize = sizeof(PROCESSENTRY32);

    while (1) {
        hProcess = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
        if (Process32First(hProcess, &pe32)) {
            while (Process32Next(hProcess, &pe32)) {
                cnt = 0;
                if (wcsstr(pe32.szExeFile, DEFAULT_PROCESS_NAME)) {
                    for (int j = 0; j < pList.numOfProcess; j++) {
                        if (pList.dwProcessIdList[j] == pe32.th32ProcessID) {
                            cnt = 1;
                            break;
```

- It is part of the code that tests multiple streams of traces using the extended *WindowsIntelPT* driver
- The test program calls the *createProcess* function and passes the path *ptcontrolapp.exe* to *processPath*
- It also provides functions
  - To detect and pass new processes
  - To retrieve a list of currently running processes for selecting the traced processes

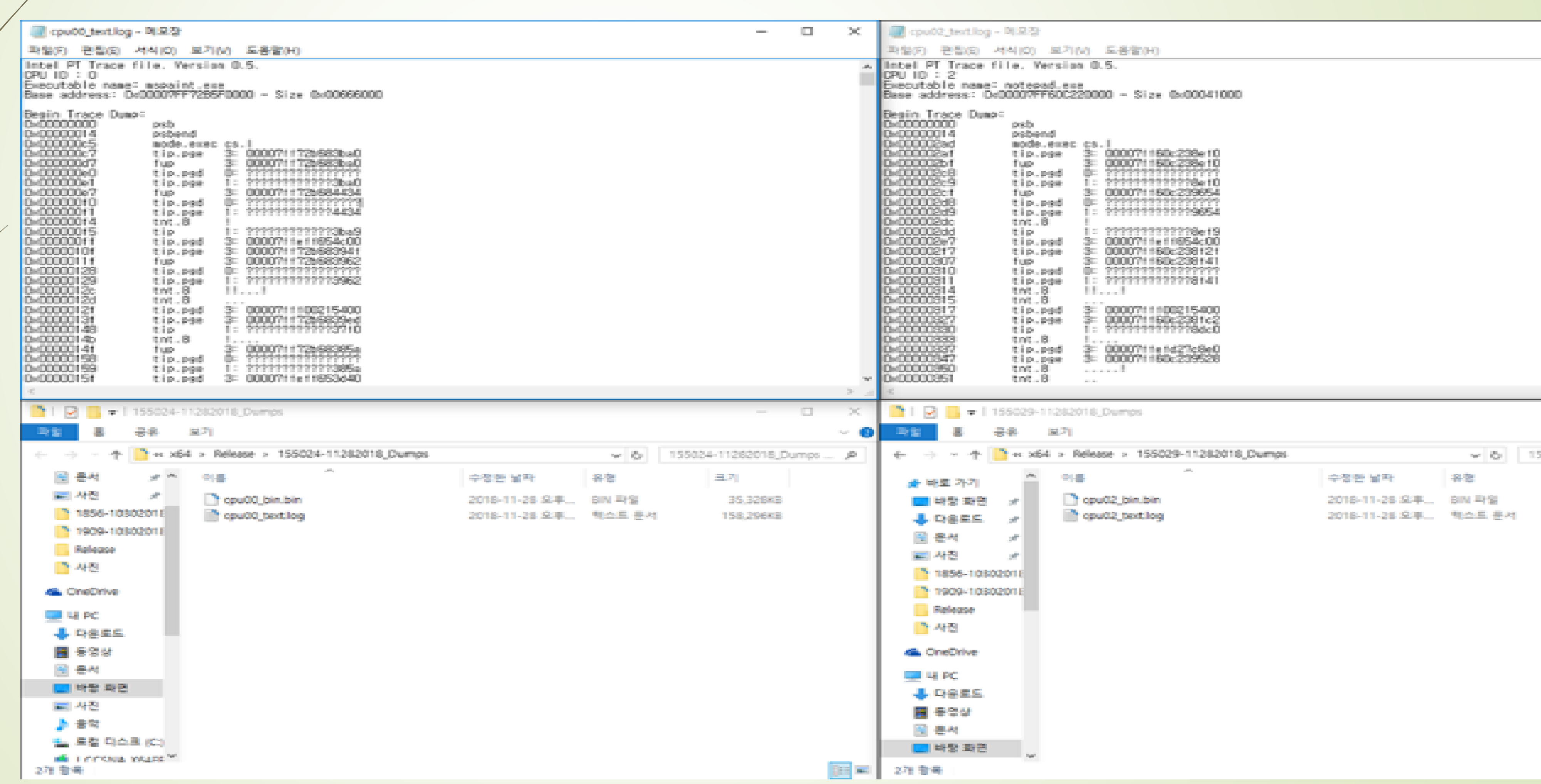
Multi-process Tracer Extension

- It shows tracing two applications, *Paint* and *Notepad*



Multi-process Tracer Extension

- It shows a dump of the results of tracing two applications simultaneously



Future Work (Dynamic Analysis for malwares)

- 2 types of malware analysis
  - (Static or Code Analysis) Performed by dissecting the different resources of the binary file *without executing it* and studying each component
    - The binary file can also be disassembled (or reverse engineered) using a disassembler
  - (Dynamic or Behavioral Analysis) Performed by observing the behavior of the malware while it is actually running on a host system
    - This form of analysis is often performed in a sandbox environment to prevent the malware from actually infecting production systems
- We're focucing dynamic analysis in an actual environment not sandbox
  - Using tracing logs from multi-stream Intel-PT decoder

Future Work (Dynamic Analysis for malwares)

- A Malicious Code Analyzing System
  - Multi-stream PT decoder in Windows environment
  - A storage for malwares (input)
  - A storage for accumulating tracing logs (output)
  - An AI-based Analysis Server
    - Preprocessing tracing logs
    - Generating and applying diverse learning models
  - Trace log dumper
    - It dumps trace logs of processes starting/stoping for *N* minutes from running a specific malware
  - Automatic rebooter
    - It reboots this system repeatedly after *N+1* minutes from running a malware
  - …

Summary

- Intel® PT uses proprietary hardware to record all information about software execution on each hardware thread
- When the software execution is completed, PT can process the trace data of the software and reconstruct the correct program flow
- (*Windows* systems) There is no close integration with the profiling and debugging mechanism due to problems such as kernel opening
- To this end, some individuals/organizations are implementing PTs in *Windows* environments
  - However, the *perf* and the *WindowIntelPT* can trace only a single process using PT
- We propose a method to extend the existing PT trace program in order to overcome this shortcoming by supporting multi-process stream tracing in *Windows* environment
- As a future plan, we're designing a malicious code analyzing system using multi-stream PT decoder

Thank You