EPA-RIMM Prototype v1 - User Guide

EPA-RIMM Team
July 31, 2019
— Version 1.0 Release

Abstract

This User Guide provides an overview of the EPA-RIMM Prototype and UEFI-based SMM Inspector. It describes how to configure and run its options.

Contents

1 Introduction

This section provides an overview of the EPA-RIMM modules and how they relate to each other.

1.1 Abbreviations

Extensible Performance Aware Runtime Integrity Measurement Mechanism = EPA-RIMM
Diagnosis Manager = DM
Backend Manager = BEM
Frontend Manager = FEM
System Management Mode = SMM

1.2 Assumptions

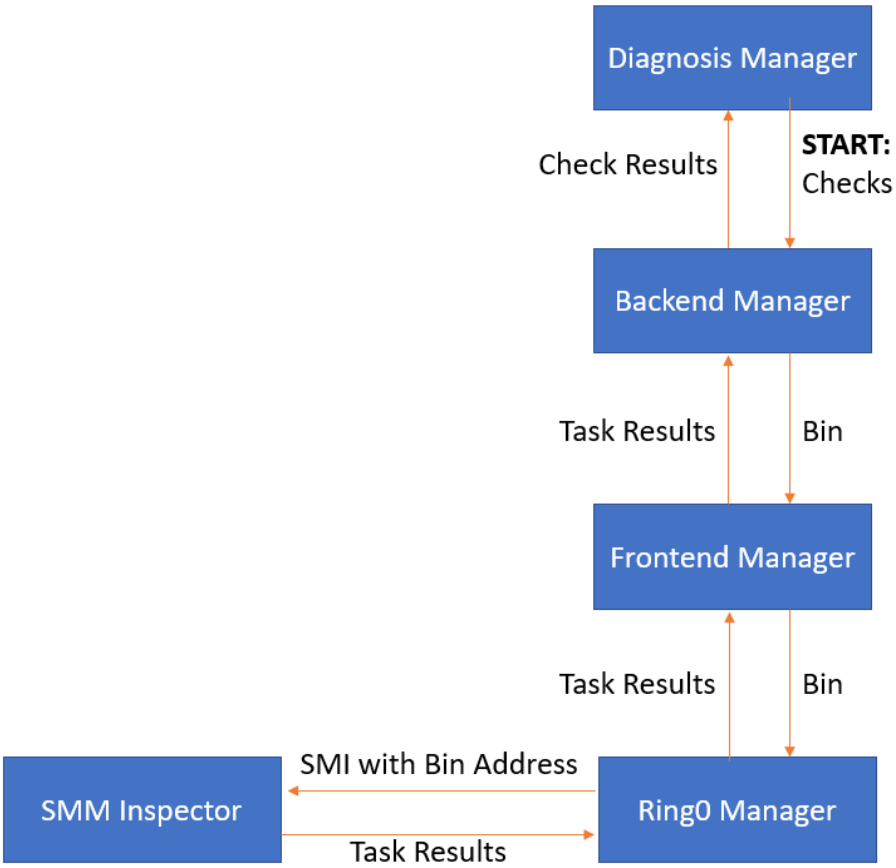
This guide assumes running Ubuntu 64bit for the Linux OS and UEFI-based SMM Inspector. This guide does not yet describe EPA-RIMM-Xen (e.g. EPA-RIMM with the STM for Xen measurements).

1.3 Overview

This prototype provides the ability for the user to configure EPA-RIMM monitoring of memory, control registers, and model-specific registers (MSRs). The prototype will demonstrate communications with the Inspector providing detections of changes to monitored resources at runtime. The prototype consists of a subset of the EPA-RIMM architecture for demonstration purposes. Note: This prototype has not undergone any security evaluation and is not designed for production usage.

1.4 Prototype Diagram

Figure 1: EPA-RIMM Prototype Architecture



2 Running the Prototype

2.1 Quick Start for Linux

1. Add the SMM Inspector to the Minnowboard firmware. Refer to Section 3.
2. (optional for memory measurements: For ease of testing, disable KASLR to avoid needing to re-provision memory Checks upon each reboot.)

   Determine which kernel memory regions you want to measure. Refer to Section 2.3 for more details.
3. Start the Ring 0 Manager. In the Ring0-Manager directory, run ./reloadDriver.sh
4. Frontend Manager (FEM): Edit frontend/frontendConfig.py to specify appropriate hostnames. Then: python femstart.py. The FEM will stay running.
5. Backend Manager (BEM): Edit backend/backendConfig.py to specify appropriate hostnames. Then: python bemstart.py. The BEM will stay running.
6. Diagnosis Manager (DM): Edit diagmgr/dmConfig.py and specify the BEM name in SEND_HOSTS. Then: python "dmterm.py -s".

The terminal will open and commands can be entered. Refer to Section 2.1.1 for sample commands.

2.1.1 Adding Sample Checks

First, Checks can be added to the Diagnosis Manager, using "add -c...". This does not send them for processing to the BEM yet, it just tells the Diagnosis Manager what the Checks are.

The first 0 in each of the commands (e.g. add -c mem **0**...) is the monitored node ID for the given BEM. With a single monitored node for a given BEM, the value should be set to 0 for all Checks.

Unused parameters can be specified as 0, 0x0, or None. For example, the HASH_MEM_VIRT command takes no Operand so this field is set to 0 in the example below (e.g. ...HASH_MEM_VIRT **0** 0xffffffff81000000...). Similarly, the CHECK_REG or CHECK_MSR Checks take no Address field, instead use Operand to convey their argument. So the Address field should be set to 0 (e.g. ...CHECK_MSR 0x1f2 **0**...).

1. Virtual Memory Check: The following Check command measures 4096 bytes of virtual memory from 0xffffffff81000000. It is a medium priority measurement and will be performed once. The Check is now called 'mem'.

   add -c mem 0 once HASH_MEM_VIRT 0 0xffffffff81000000 4096 med

2. Control Register 0 (CR0) Check: This Check will be called 'cr0' and will measure 8 bytes of CR0. Note that the 0 following CR0_REG tells the DM to ignore the address parameter as this is not used for this Check.

   add -c cr0 0 once CHECK_REG CR0_REG 0 8 med

3. Control Register 3 (CR3) Check: This Check will be called 'cr3' and will measure 8 bytes of CR3. This Check measurement will change from time to time based on what process is running at measurement time. Thus, it can be used to demonstrate a "changed" measurement.

   add -c cr3 0 once CHECK_REG CR3_REG 0 8 med

4. Control Register 4 (CR4) Check: Similar to above control register checks, except for CR4.

   add -c cr4 0 once CHECK_REG CR4_REG 0 4 med

5. Sample MSR Check (SMRR_BASE MSR). This Check measures the SMRR_BASE MSR. Note that EPA-RIMM has a sample whitelist for MSRs (enforced in the DM and in the SMM Inspector) so some MSR indices are not measurable to avoid crashes.

   add -c SmrrBase 0 once CHECK_MSR 0x1f2 0 4 med

6. Sample dynamic MSR Check (TSC). This Check hash will change upon each measurement so it readily demonstrates a "changed" measurement.

   add -c DynamicTsc 0 once CHECK_MSR 0x10 0 8 med

2.1.2 Supported Measurements

For **memory**, SHA256 hashes can be performed over physical memory (Command = HASH_MEM_PHYS) or virtual memory (Command = HASH_MEM_VIRT). It is necessary to specify the base address and length of the desired region of memory to monitor.

For **control registers**, SHA256 hashes can be performed over the CR0, CR3, CR4, LDTR, GDTR, and IDTR registers. Command = CHECK_REG

The respective operands are:

CR0_REG
CR3_REG
CR4_REG
LDT_REG
GDT_REG
IDTR_REG

For **Model-Specific Registers (MSRs)**, SHA256 hashes can be performed over the whitelisted MSRs, consult the msr.yaml (for DM) and "msr-whitelist.c" (for SMM Inspector) to determine which MSRs are allowed to be measured on the Silvermont Atom CPU used on the Minnowboard. Command = MEASURE_MSR

Note: Performing measurements of non-supported MSRs can cause hangs within SMM so caution is advised.

2.1.3 Launching a Check

This syntax will send the 'mem' Check from the DM to the BEM for processing.

enqueue BEM_host_name mem 0 (replace BEM_host_name with your BEM host name)

2.1.4 Inspecting Check Results

From the dmterm, type in 'ls' to verify the Check results. Detailed log files are also created, e.g. diag_mgr.log, backend.log, frontend.log. The SMM Inspector, when compiled in Debug mode, also outputs results to the serial log.

2.2 Diagnosis Manager Commands

The Diagnosis Manager features a variety of commands that allow the user to interact with EPA-RIMM.

1. ls :
   Syntax: 'ls': List Check and results info
   Syntax: 'ls -c': Displays DM listening port and BEM communication port
2. add : Adds a Check to the DM. Refer also to 'enqueue' to send the Check to the BEM (Section 2.1.1.)

   Syntax: add -c (Name) (Node_ID) (sched) (command) (operand)

(address) (length) (priority)

Name = Name of the Check, for user reference.

Node_ID = The index of the monitored node from the BEM's perspective. Monitored nodes can be specified on the BEM via editing backendConfig.py and editing the DEFAULT_NODELIST or by passing a hostname:port dict as the node_addresses argument to the BEM constructor.

sched = How often to run the Check, options: 'once', 'always', 'count'.

Command: HASH_MEM_VIRT, HASH_MEM_PHYS, MEASURE_REGISTER, OR MEASURE_MSR. Refer to Section 2.1.2.

Operand: Used in MSR or Control Register measurements to specify which MSR or Control Register to measure. Refer to Section 2.1.2.

3. enqueue : Send a Check to the BEM.

   Syntax: enqueue BEM_host_name mem 0 (replace BEM_host_name with your BEM host name)

4. enqueueall : Send all Checks to the BEM.
5. help : List available DM terminal commands
6. nodelists : Sanity checks BEM hostnames from DM (Developer feature)
7. serverstart : Starts the DM's listening server.
8. serverstop : Stops the dm's listening server.
9. serverstartall : Equivalent to starting dmterm.py with "-s" argument. Starts all DM listening servers.
10. serverstopall: Stops all DM listening and sending servers.
11. ping : Checks network connectivity to hosts in the EPA-RIMM framework.

    Syntax: ping (host_name)

12. addhost : Specify a new BEM from the terminal command line.

    Syntax: Scenario: Setting up a new BEM ("epa-vm2") from the terminal command-line as opposed to dmConfig.py.

    addhost -s epa-vm2 10001 (Note: Specify the port that the DM uses to send messages to the epa-vm2 BEM)

    addhost -l epa-vm2 10003 (Note: If no DM listening server is yet configured, specify the port that the DM uses to listen to data from the epa-vm2 or any BEM)

13. exit : Close the Diagnosis Manager terminal

2.3 Kernel Memory

Measuring the Linux kernel memory requires locating the proper region(s). This section describes how to reliably locate them.

2.3.1 KASLR

Kernel Address Space Layer Randomization (KASLR) requires reprovisioning upon every boot which is possible but requires extra steps during the development phase.

For this reason, during development, it can be disabled in grub by adding 'nokaslr' in /etc/default/grub on the monitored node, e.g.:

GRUB_CMDLINE_LINUX_DEFAULT="quiet splash nokaslr"
Then do: "sudo update-grub" and reboot to make it take effect.

2.3.2 Kernel Memory

The Linux kernel operates on virtual memory addresses. The particular memory addresses can vary between systems so becoming familiar with the particular address ranges is important.

You can view the virtual addresses of your running Linux kernel from the following (note: if you don't use sudo, the addresses will all be shown as zero which is not correct):

sudo cat /proc/kallsyms >kall.txt

kall.txt will look something like this:
0000000000000000 A irq_stack_union
0000000000000000 A __per_cpu_start
0000000000004000 A cpu_debug_store
0000000000005000 A cpu_tss_rw
...
0000000000023c98 A __per_cpu_end
ffffffff81000000 T startup_64

ffffffff81000000 T _stext (Start of this kernel text section)
ffffffff81000000 T _text
ffffffff81000030 T secondary_startup_64
...
ffffffff81c03000 T _etext (End of this kernel text section)

Note that the beginning of the file will have a small set of low memory addresses. We are interested in the next set of addresses where _stext begins at ffffffff81000000 (when KASLR is disabled.) The kernel code sections we typically measure exist in this upper range. On some systems, the kernel virtual memory address space might differ (e.g. ffffffff8e000000, etc). Keep in mind you'll want to also find the end of the memory space and there may be discontiguous regions to measure.

Also note that the label (red letter) shows what type of region is described, for example, T or t = text (code) section, R or r = read-only.

2.4 Description of Code Modules

2.4.1 Python package dependencies

The python modules in the EPA-RIMM software stack have a few dependencies.

pip can be used to install them.

If pip is not installed: sudo apt-get install python-pip

pip install pycrypto pyyaml

2.4.2 Diagnosis Manager

Purpose: The Diagnosis Manager is the user interface to EPA-RIMM. It allows the user to specify particular resources to measure via Checks and ultimately receives the results.

Sends to: Backend Manager (Checks)

Receives data from: Backend Manager (Check results)

Install prereqs: See Section 2.4.1.

Language: Python

Privilege Level: Ring 3 (Application)

Files:

1. dmterm.py - This launches the Diagnosis Manager terminal, providing the main user interface to EPA-RIMM
2. dm.py - The primary DM implementation file
3. dmConfig.py - Allows user to specify host and port configurations.
4. profile.py - Implements Check profile feature which aggregates Checks that could confirm or deny presence of a rootkit - not fully developed
5. results.py - Handles Check result data

Invocation: python dmterm.py **-s**

2.4.3 Backend Manager

Purpose: Schedule task measurements / Provide received measurement results to Diagnosis Manager.

Sends to: FEM (Bins)

Receives data from: FEM (Measurement Results)

Install prereqs: See Section 2.4.1.

Language: Python 2.7x

Privilege Level: Ring 3 (Application)

Files:

1. bemstart.py - This launches the Backend Manager

2. backend.py - This is the main implementation of the Backend Manager code
3. backendConfig.py - This specifies the hostnames of the Monitored Node and Diagnosis Manager, along with network ports. It also specifies the AES and HMAC keys and Manager signature.
4. decomp.py - This performs Check to Task decomposition.

Invocation:

1. python bemstart.py

2.4.4 Oracle

Purpose: Stores golden measurements in sqlite database.

Sends data to: Backend Manager

Receives data from: Backend Manager

Language: Python

Invocation: None

2.4.5 Frontend Manager

Purpose: Serves as communication intermediary between the BEM and the Ring0 Manager Inspector. It conveys measurement requests from the BEM and results from the Ring0 Manager.

Sends data to: Ring 0 Manager (Bins)

Receives data from: BEM (Bins), Ring 0 Manager (Measurement results)

Language: Python 2.7x

Install prereqs: See Section 2.4.1.

Privilege Level: Ring 3 (Application)

Files:

1. frontend.py - This is is the main Frontend code. It receives Bins from the BEM (task by task) and provides to the Ring0 Manager by writing them to the /proc interface established by the Ring0 Manager. It then reads the results from the Inspector from the same /proc interface and provides back to the BEM.
2. frontendConfig.py - This specifies network ports for the FEM and the remote port that the FEM uses to talk to the BEM.
3. frontendLogging.ini - Specifies parameters pertaining to the FEM's logging capability.

2.4.6 Ring0 Manager

Purpose: Send measurement requests to SMM Inspector, receive measurement results from SMM Inspector and convey to FEM.

Sends data to:

SMM Inspector (Bin Address)

FEM (Measurement results)

Receives data from: SMM Inspector (Measurement results), FEM (Bins)

Language: C

Privilege Level: Ring 0 (Kernel)

Files:

1. ring0manager.c - Ring 0 Manager implementation that receives measurement requests, conveys them to SMM Inspector, and returns results/stats to the higher layers. Creates /proc/ring0manager file.
2. ring0manager.h - Task description C struct (needs to be in sync with SMM Inspector's versions), command #defines.
3. reloadDriver.sh - Single script that compiles and loads the Ring0 Manager kernel module.

2.4.7 SMM Inspector

Purpose: Measure the OS or VMM from SMM.

Sends data to: Ring0 Manager

Receives data from: Ring0 Manager

Language: C

Privilege Level: Ring 0 (SMM)

Files:

1. epa-inspector.c - Main file for Inspector, locates/parses/processes measurement bins
2. epa-inspector.h - Task data structure and command definitions
3. EPARimm.dsc - Used to compile Inspector from UEFI, specifies UEFI libraries in use and the function that initializes the Inspector during BIOS boot (inspector_init)
4. EPARimm.inf - Specifies .c, .h, .s, .asm files used by the Inspector, used for UEFI compilation
5. Integration-note.txt - Description of how to integrate the Inspector into the Minnowboard firmware so that it can be compiled in to the BIOS.
6. msr-whitelist.c - A whitelist of allowable MSRs that could be measured on an Intel Silvermont Atom CPU.
7. msr-whitelist.h - Header file for MSR Whitelist
8. Ia32 or \X64\AsmVmCall.asm and Ia32 or X64\AsmVmCall.s - EPA-RIMM-V Vmcalls to request services from the STM for EPA-RIMM-V.
9. PageWalk\PageTable.c - Converts virtual addresses to physical addresses (origin: STM code)

10. PageWalk\pagewalk.h - (origin: STM code)

2.5 Frontend Manager Configuration Options

The frontendConfig.py file has the following configuration options:

FRONTEND_PORT = 10005 – EPA-Frontend listens on this port

BACKEND_PORT = 10001 – EPA-Backend listens on this port (needs to match Backend's setting)

PROCFILE_NAME = "/proc/ring0manager" – Use SMM-based inspector

BACKEND_SERVER = "dataserver" – Specify the Backend Manager's host name

THIS_HOST = "epa-vm" – Specify the monitored node's name

LEGACY_BACKEND = False – Keep this set to False for this code release.

2.6 Ring0 Manager Configuration Options

The Ring0 Manager has a number of parameters that govern its operation.

It also configures the SMM Inspector to allow changing the SMM Inspector's settings without requiring a BIOS flash.

The Ring 0 Manager defaults to Linux measurements, to specify Xen measurements for EPA-RIMM-Xen, XEN_VIRT needs to be set to 0x01 in ring0manager.h.

Ring0 Manager Configuration
In ring0Manager.c:

#define VERBOSE 1 – Print all debug output in Linux system log

#define SHOW_BIN_PERF – Just print out total bin times and nothing else during runtime. These can be viewed by 'dmesg'. This avoids a large number of prints in the system log and just logs the total time spent in SMM for a given Bin.

#define DEBUG_EARLY_EXIT – For debugging EPA-Frontend to Ring 0 Manager communication. This allows the Ring0 Manager to receive input from the FEM but exits without processing it.

#define DISABLE_INT – For EPA-RIMM-V, temporarily disable interrupts. This allows benchmarking the cost of the SMM/STM without device interrupts impacting the measurements. Note: This option does not seem to make a performance difference in Linux but does on Xen.

#define STMPERF – Tell the Ring0 Manager to request the performance stats from the STM.

SMM Inspector Configuration via epaConfig
File/function:(ring0manager.c ring0manager_init() epaConfig.) The Ring0 Manager populates different enable/disables in epaConfig along

with signatures and keys. This is a debug capability to allow dynamically changing these parameters without a BIOS re-flash.

epaConfig.HmacCreateEnable = OFF; – Inspector creates HMAC over each Task in the Bin when set to ON.

epaConfig.HmacCheckEnable = OFF; – Inspector creates HMACs over each Task in the Bin when set to ON.

epaConfig.AesEncryptEnable = ON; – Inspector encrypts the results it sends back when set to ON.

epaConfig.AesDecryptEnable = ON; – Inspector decrypts the incoming bins when set to ON.

epaConfig.CheckSmrrOverlapEnable = ON; – Inspector checks to make sure Bins don't reside within the protected SMM memory region when set to ON. This is a security check to make sure that an attacker isn't trying to trick the Inspector to overwrite SMM's own memory.

epaConfig.MaxMemoryHashSize = 0x1000000; – Limit the maximum size of the memory hash to be performed, this avoids a denial of service by an attacker providing a huge memory hash to be performed.

epaConfig.MaxTasks = 14; – Specify the maximum number of Tasks in a Bin. This needs to match the BEM and Ring0 Manager

memcpy(&epaConfig.InspectorSig, "INSPECTOR12345678901"...) – Inspector "signature"

memcpy(&epaConfig.ManagerSig, "MANAGER1234567890123"...) – BEM "signature"

memcpy(&epaConfig.Aes256CbcKey, Aes256CbcKey, ...) – Encryption key used for AES encryption. Needs to match BEM.

memcpy(&epaConfig.HmacKey, HmacKey, ...) – Specify crypto key used by HMAC. Needs to match BEM.

...

sendEpaConfig(&epaConfig); – Generate an SMI to transmit the epaConfig to the Inspector

2.7 SMM Inspector Configuration Options

The SMM Inspector has a hard-coded set of configuration knobs, however, the Ring0 Manager can re-configure these at any time. In the prototype, the Ring0 Manager re-configures the SMM Inspector when the Ring0 Manager loads.

Refer to Section 2.6 (Ring0 Manager) for the details on SMM Inspector configuration (epaConfig).

3 Building the EPA-RIMM Software Stack

New Minnowboard Max BIOS releases and their accompanying documentation can be found at: https://firmware.intel.com/projects/minnowboard-max .

The release notes (e.g. https://firmware.intel.com/sites/default/files/minnowboard_max-rel_0_99-releasenotes.txt) provide the necessary steps to build a standard Minnowboard BIOS. It will be necessary to integrate the SMM Inspector source manually.

We typically build the BIOS on Windows but this isn't a requirement. For Windows, Visual Studio 2015 is recommended. We also don't use the FSP option.

3.1 Preparing the standard Minnowboard BIOS Sources and Binaries

First, we will make sure that we can build a standard Minnowboard BIOS before we try adding in the SMM Inspector.

Sample steps for building the Minnowboard 0.99 BIOS on Windows:

Prerequisites:
-Visual Studio 2015,
-Python 2.7.10 (https://www.python.org/downloads/release/python-2710)
-Git for Windows.

1. Setup IASL... Go to https://acpica.org/downloads and download iasl-win-20160527.zip. Extract and copy iasl.exe to c:\ASL
2. Download NASM* 2.12.02 binaries from http://www.nasm.us/pub/nasm/releasebuilds/2.12.02/win64/nasm-2.12.02-win64.zip
   Create C:\NASM and put nasm.exe there.
3. Download http://wiki.overbyte.eu/arch/openssl-1.1.0g-win32.zip
   Create c:\openssl and put openssl.exe there

   Add the path of openssl.exe ("C:\openssl") to system environment variable OPENSSL_PATH.
4. Make a "c\fw\max2-99" directory. Note: It's best to keep the directory names short as long path names can break the BIOS build.
5. "cd c:\fw\max2-99"
6. Check out the edk2 sources: "git clone https://github.com/tianocore/edk2.git -b UDK2017"
7. "cd c:\fw\max2-99\edk2"
8. "git checkout vUDK2017"
9. "cd c:\fw\max2-99"
10. "git clone https://github.com/tianocore/edk2-platforms.git -b devel-MinnowBoardMax-UDK2017"
11. "cd c:\fw\max2-99\edk2-platforms"
12. "git checkout 423105b15de6dfd769eed56026fa3bc28eb349ef"
13. "cd c:\fw\max2-99"

14. "git clone https://github.com/tianocore/edk2-BaseTools-win32.git"
15. "cd c:\fw\max2-99\edk2-BaseTools-win32"
16. "git checkout 0e088c19ab31fccd1d2f55d9e4fe0314b57c0097"
17. Copy the contents of c:\fw\max2-99\edk2-BaseTools-win32 to C:\fw\max2-99\edk2\BaseTools\Bin\win32 .
18. Download MinnowBoard MAX 0.99 Binary Object Modules from http://firmware.intel.com/projects/minnowboard-max

    After extracting the download, make a directory called c:\fw\max2-99\silicon and put the three directories in there.
19. Add OpenSSL to the BIOS tree.

    cd c:\fw\max2-99\edk2\CryptoPkg\Library\Openssllib

    git clone -b OpenSSL_1_1_0e https://github.com/openssl/openssl openssl

3.2 Build a standard BIOS w/o EPA's SMM Inspector

1. "cd c:\fw\max2-99\edk2-platforms\Vlv2TbltDevicePkg"
2. Build either a Debug or Release BIOS.

   Debug BIOSes give you a lot of serial output which is helpful for debugging issues. The Debug BIOS can be built by:
   "Build_IFWI.bat MNW2 Debug"

   However, if you are doing benchmarking, make sure you use a Release bios. There will be no serial output with a Release BIOS and SMI performance will be higher. This can be created by instead doing:

   "Build_IFWI.bat MNW2 Release"
3. If all goes well, you should see "The final IFWI file is located in c:\fw\max2-99\edk2-platforms\Vlv2TbltDevicePkg\Stitch\"

   The BIOS image will be called something like this: MNW2MAX1.X64.0099.D01.1811151453.bin

   Note that the "D" in "D01...." indicates that this is a Debug BIOS. A Release BIOS will have an "R".

3.3 Adding in the SMM Inspector

1. Check out the EPA code from Github. The SMM-Inspector directory contents will be needed for integration into the Minnowboard firmware sources.

   Create c:\github-epa
   cd c:\github-epa
   git clone https://github.com/PPerfLab/EPARIMM-Release .
2. Create c:\fw\edk2\EPARimm
3. Copy the contents of the c:\github-epa \epastack \SMM-Inspector directory into c:\fw\edk2\EPARimm

4. We need to adjust the BIOS build files to enable the compilation of EPA-RIMM (via the .dsc file) and its addition into the BIOS image (via the .fdf file).

   Edit c:\fw\max2-99\edk2-platforms\Vlv2TbltDevicePkg\PlatformPkg.fdf, add the highlight EPARimm.inf line below:

   (DXE_ARCHITECTURE)(*TARGET*)/(DXE_ARCHITECTURE)/PnpDxe.inf
   #
   # SMM
   #
   INF MdeModulePkg/Core/PiSmmCore/PiSmmIpl.inf
   INF MdeModulePkg/Core/PiSmmCore/PiSmmCore.inf
   INF UefiCpuPkg/PiSmmCpuDxeSmm/PiSmmCpuDxeSmm.inf
   INF UefiCpuPkg/CpuIo2Smm/CpuIo2Smm.inf
   INF EPARimm/EPARimm.inf <– Can add here
   INF MdeModulePkg/Universal/LockBox/SmmLockBox/SmmLock-Box.inf
   INF UefiCpuPkg/PiSmmCommunication/PiSmmCommunication-Smm.inf

5. Edit c:\fw\max2-99\edk2-platforms\Vlv2TbltDevicePkg\PlatformPkgX64.dsc

   #
   # SMM
   #
   MdeModulePkg/Core/PiSmmCore/PiSmmIpl.inf
   MdeModulePkg/Core/PiSmmCore/PiSmmCore.inf
   UefiCpuPkg/PiSmmCpuDxeSmm/PiSmmCpuDxeSmm.inf
   UefiCpuPkg/CpuIo2Smm/CpuIo2Smm.inf
   EPARimm/EPARimm.inf <– Can add here
   MdeModulePkg/Universal/LockBox/SmmLockBox/SmmLockBox.inf
   UefiCpuPkg/CpuS3DataDxe/CpuS3DataDxe.inf

6. Enable the SMM Inspector to read host memory and not be required to use the CommBuffer (which isn't currently supported in the Inspector)

   In c:\fw\edk2\UefiCpuPkg\PiSmmCpuDxeSmm\SmmCpuMemoryManagement.c, need to adjust permissions on host memory for SMM. Comment out the highlighted section below as shown, note: some lines below have been reformatted to fit in this document.

```
MemoryMapEntryCount = mUefiMemoryMapSize/mUefiDescriptorSize;
MemoryMap = mUefiMemoryMap;
for (Index = 0; Index < MemoryMapEntryCount; Index++) {
  if (IsUefiPageNotPresent(MemoryMap)) {
  /* <-- Here
  DEBUG ((DEBUG\_INFO, "UefiMemory protection: 0x%lx
  - 0x%lx\n", MemoryMap->PhysicalStart,
        MemoryMap->PhysicalStart +
        (UINT64)EFI_PAGES_TO_SIZE((UINTN)MemoryMap->
                NumberOfPages)));
        SmmSetMemoryAttributes (
        MemoryMap->PhysicalStart,
        EFI\_PAGES\_TO\_SIZE((UINTN)MemoryMap->
```

```
        NumberOfPages),
        EFI\_MEMORY\_RO
        );
    */ <-- End
    }
    MemoryMap = NEXT\_MEMORY\_DESCRIPTOR(
    MemoryMap, mUefiDescriptorSize);
  }
```

7. Relax CommBuffer requirement In MdePkg/Library/SmmMemLib/SmmMemLib.c, need to work around CommBuffer check. Note: When EPA-RIMM switches to using CommBuffer to receive Ring0Manager bins, we don't need to do this.

```
      DEBUG ((
        EFI_D_ERROR,
        "CpuStart (0x%lx) - PhysicalSize (0x%lx)\n",
        mSmmMemLibInternalSmramRanges[Index].CpuStart,
        mSmmMemLibInternalSmramRanges[Index].PhysicalSize
        ));
      return FALSE;
    }
  }
```

```
// Temporary workaround for EPA-RIMM
/* <-- Here!
  //
  // Check override for Valid Communication Region
  //
  if (mSmmReadyToLock) {
    EFI_MEMORY_DESCRIPTOR          *MemoryMap;
    BOOLEAN                InValidCommunicationRegion;

    InValidCommunicationRegion = FALSE;
    MemoryMap = mMemoryMap;
    for (Index = 0; Index < mMemoryMapEntryCount; Index++) {
      if ((Buffer >= MemoryMap->PhysicalStart) &&
          (Buffer + Length <= MemoryMap->PhysicalStart +
           LShiftU64 (MemoryMap->NumberOfPages,
            EFI_PAGE_SHIFT))) {
        InValidCommunicationRegion = TRUE;
      }
      MemoryMap = NEXT_MEMORY_DESCRIPTOR(MemoryMap,
       mDescriptorSize);
    }

    if (!InValidCommunicationRegion) {
      DEBUG ((
        EFI_D_ERROR,
        "SmmIsBufferOutsideSmmValid: Not in
         ValidCommunicationRegion: Buffer (0x%lx) - Length
         (0x%lx), ",
        Buffer,
        Length
```

```
        ));
      ASSERT (FALSE);
      return FALSE;
    }
  }
  */ <-- End!

  return TRUE;
}
```

3.4 Compiling

Once the EPA-RIMM SMM Inspector has been integrated into the firmware source, refer to Section 3.2 to build the BIOS.

4 Task Description

This section describes a key data structure used by the Inspector (Task Description). This data structures explains what should be measured in any given measurement session.

The Results description re-uses the same data structure and populates certain fields in it.

4.1 Task Description - C struct

The following is the Task structure definition. The BEM has its own Python class for this data (class Task()), however, the struct packing/unpacking capability of Python allows converting the Python class representation to a C struct.

struct task_t {

(The ivec1 and ivec2 fields are not encrypted but are HMAC'd)

uint64_t **ivec1**; – *The first two fields are in plaintext as the initialization vector ("IV") for the AES encryption can be in plaintext. Both fields are aggregated to form the IV.*
uint64_t **ivec2**;

(The fields from cmd all the way to Stat6 are encrypted and HMAC'd)

uint64_t **cmd**; – *Measurement command (e.g. MSR, mem, register)*
uint64_t **operand**; – *For MSRs/Registers, specify which one to measure*
uint64_t **virt_addr**; – *For virtual mem measurements*
uint64_t **phys_addr**; – *For phys mem measurements*
uint64_t **len**; – *Size to hash*
uint64_t **result**; – *Task measurement result, e.g. Init, Changed, Unchanged, Error*
uint64_t **nonce**; – *Nonce value to help prevent replay attacks*
uint64_t **cost**; – *Task measurement cost in clocks*
uint64_t **priority**; – *Unused*
uint64_t **last_checked**; – *RDTSC time of last check*
uint64_t **task_uuid**; – *Unique identifier for task*
uint64_t **reserved1**; – *Reserved field, should be 0.*
uint32_t **hash[SHA256_INTS]**; – *32 byte SHA256 hash*

Note: The "signatures" below are not true signatures in the RSA sense. Currently, they are a simple string comparison

uint32_t **manager_sig[SIGN_INTS]**; – *20 bytes Backend Manager "signature"*
uint32_t **inspector_sig[SIGN_INTS]**; – *20 bytes Inspector "signature"*

There are seven holders for performance stats, Stat0..Stat6. Each of these holders is a C Union and thus can hold two 32bit stats (SmallStat[0] and SmallStat[1] or one 64bit stat, e.g. BigStat).

The 32bit stats are typically used with calculations, e.g. to measure the cost of an operation:

TimeStamp1 = AsmReadTsc(); – *Get a before timestamp*
SomeOperation(); – *Perform operation*
TimeStamp2 = AsmReadTsc(); – *Get the after timestamp*

bin[i].Stat0.SmallStat[0] = TimeStamp2 - TimeStamp1; – *Save the cost of SomeOperation into Stat0.SmallStat[0]. Stat0.SmallStat[1] remains available but writing to Stat0.BigStat would clobber both SmallStat[0] and SmallStat[1] so use carefully.*

union Stat Stat0;
union Stat Stat1;
union Stat Stat2;
union Stat Stat3;
union Stat Stat4;
union Stat Stat5;
union Stat Stat6;

uint8_t Hmac[HMAC_SIZE]; – *An HMAC over the Task/Results*
};

typedef union Stat
uint32_t smallstat[2];
uint64_t bigstat;
};