Safeguarding rootkits: Intel BootGuard

Alexander Ermolov

#whoami

Security researcher at Digital Security

a.ermolov@dsec.ru

flothrone@gmail.com

#disclaimer

1. No motherboards were harmed
2. The Intel Boot Guard implementation details given here is a result of a reverse engineering process, so it may contain some inaccuracy compared to the Intel Boot Guard specification (which is not public)

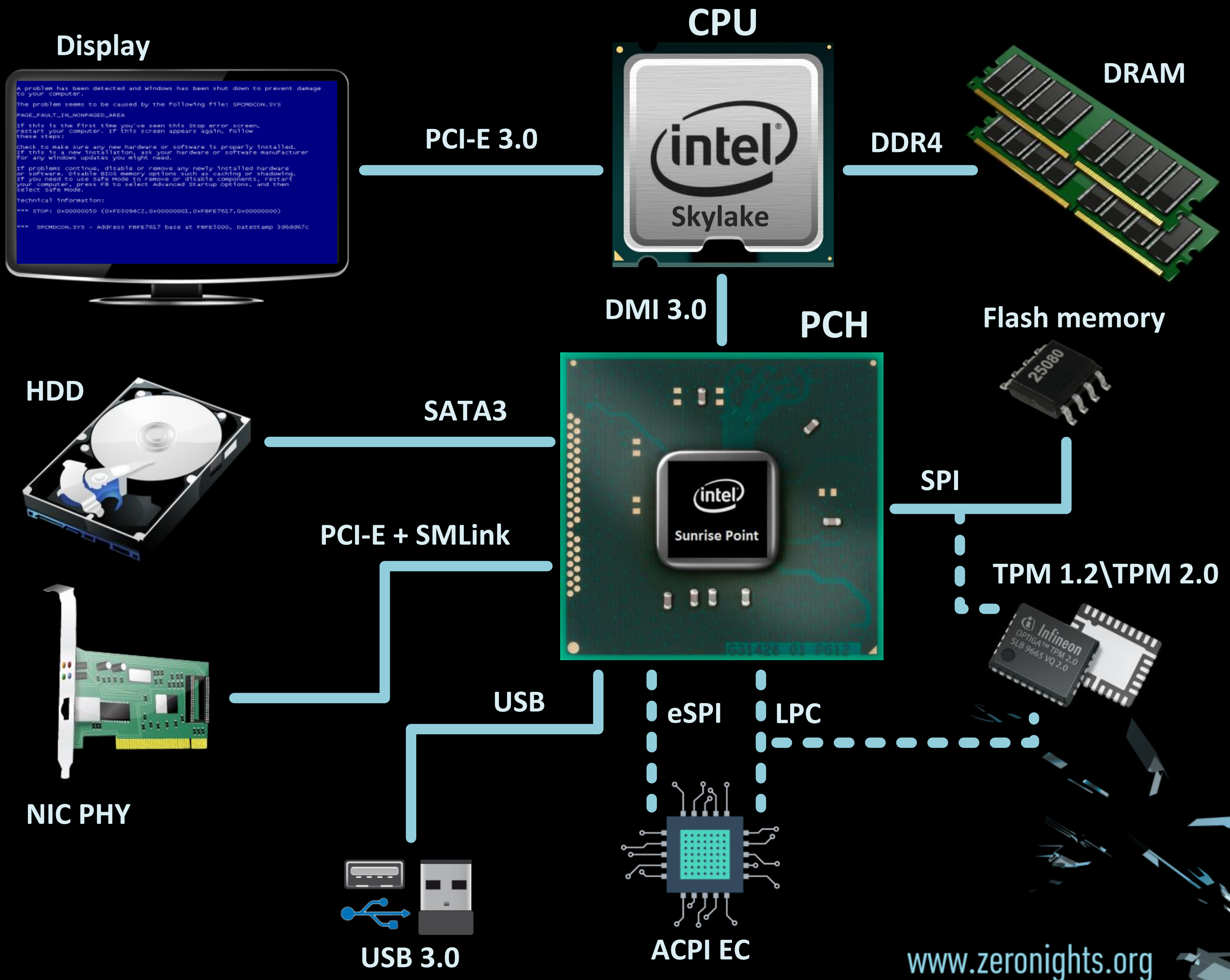
Intel x86 platform firmware

Desktop (Laptop) system overview

Execution environments:

- Intel CPU
- Intel chipset subsystems
- ACPI EC

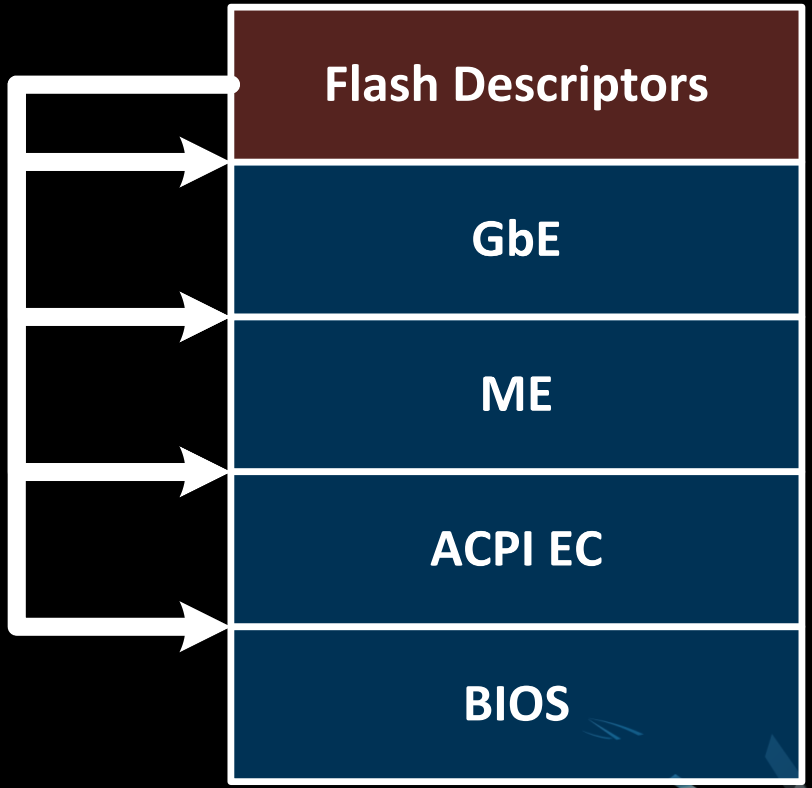
Platform firmware is stored on common SPI flash memory



Common SPI flash memory

System firmware is divided into regions:

- Flash Descriptors
  - Descriptors of other regions
  - Access permissions
  - …
- GbE
- ME
- ACPI EC (since Skylake)
- BIOS



Intel CPU

Main execution environment (BIOS\OS)

Privilege levels:

Ring 3	User Mode
…	
Ring 0	Kernel Mode
Ring -1	Hypervisor Mode
Ring -2	System Management Mode (SMM)

Intel CPU

Root of Trust

- Microcode ROM (== Boot ROM ?)
- AES key for decrypting microcode updates
- Hash of an RSA public key which verifies the microcode updates
- Hash of an RSA public key which verifies other Intel blobs (e.g. ACM…)

Intel ME

Chipset subsystem integrated into:

- Q-type chipsets since 960 series (2006)
  Intel ME 2.x – 5.x
- All chipsets since 5 series (2010)
  Intel ME 6.x – 11.x, TXE 1.x – 3.x, SPS 1.x – 4.x

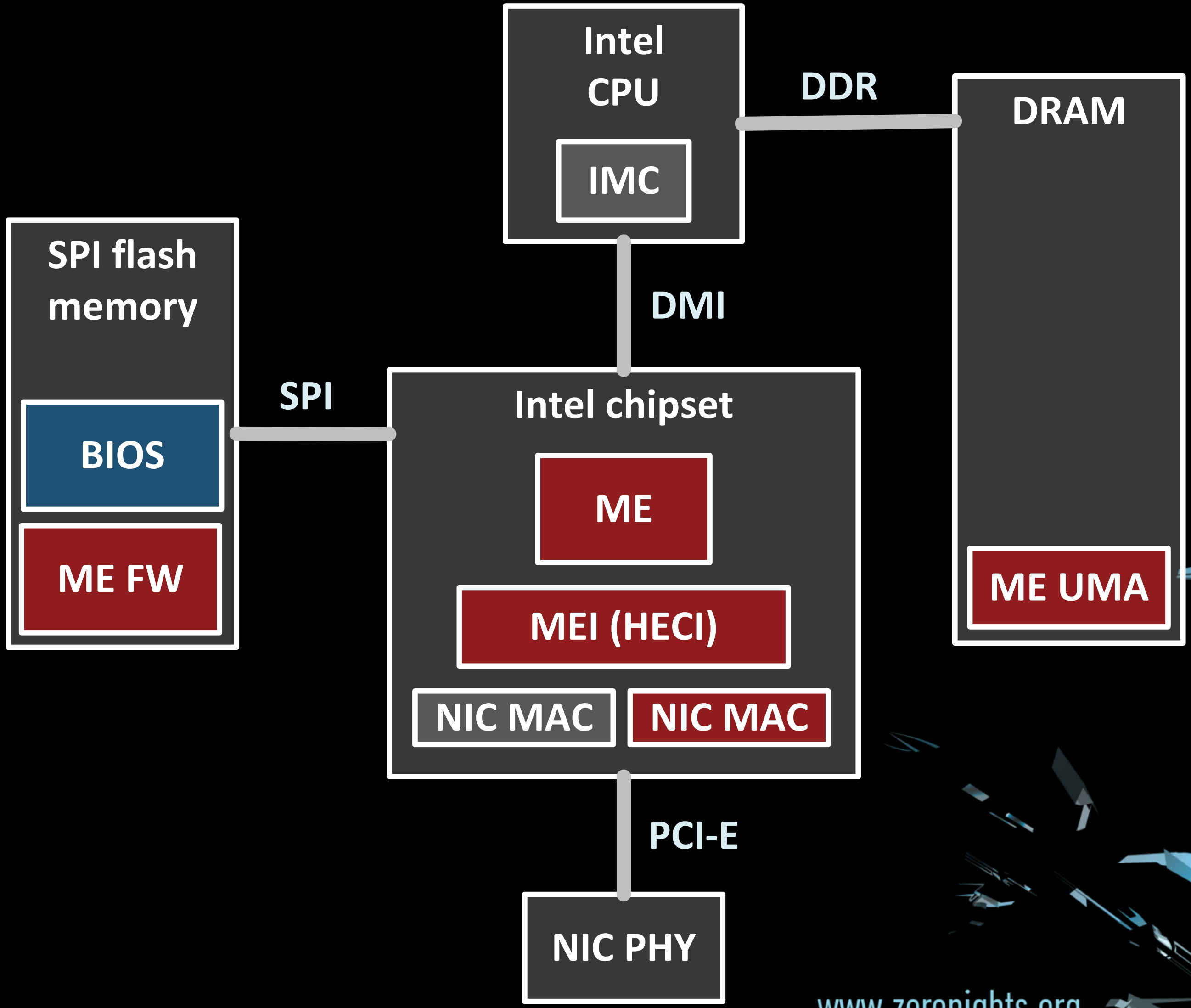
Platforms affected:

• Desktop, Laptop	Intel Management Engine (ME)
• Mobile	Intel Trusted Execution Engine (TXE)/Security Engine
• Server	Intel Server Platform Services (SPS)

Intel ME

Most privileged and hidden execution environment (Ring -3):

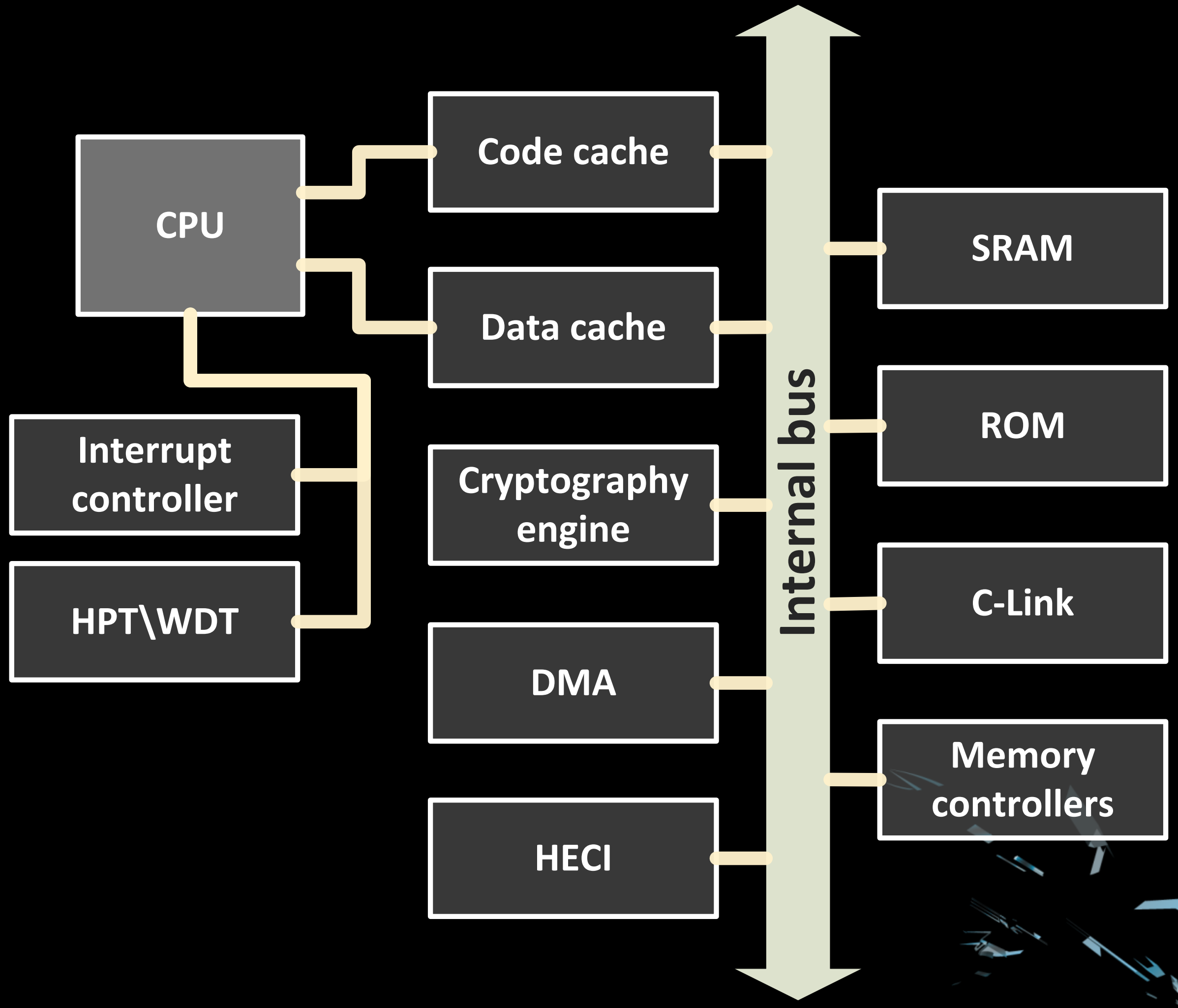
- Hidden from CPU runtime memory in DRAM
- Full access to DRAM
- Working even when CPU is in S5 (system shutdown)
- Out-of-Band (OOB) access to network interface
- Runs firmware (based on RTOS ThreadX) from common SPI flash



Intel ME

CPU architectures

- ME 2.x – 10.x, SPS 1.x – 3.x
  - ARC (ARC32/ARCompact)
- TXE 1.x – 2.x
  - SPARC
- ME 11.x, SPS 4.x, TXE 3.x
  - x86



Intel ME

Root of Trust

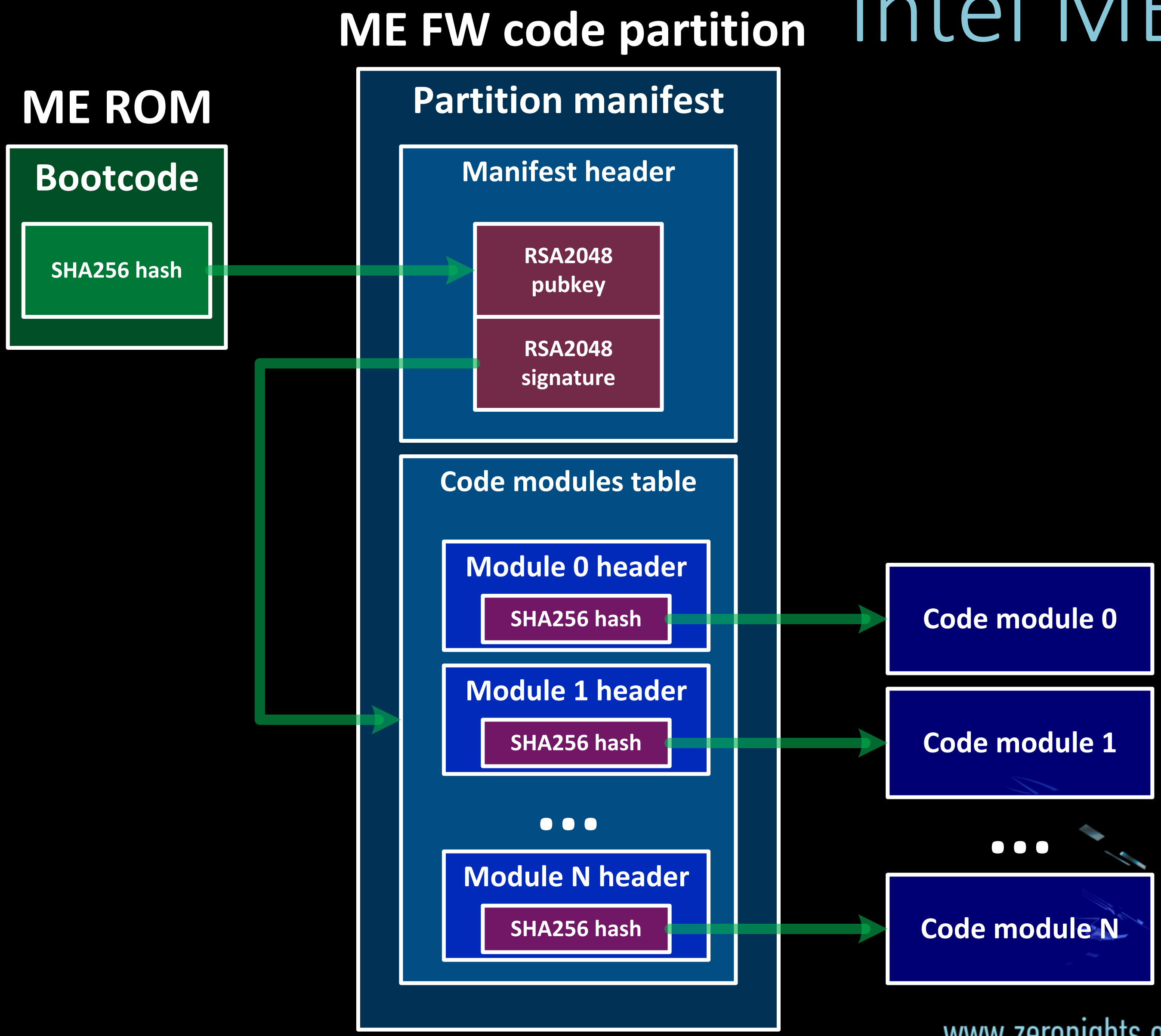
- ME ROM with the bootcode
- Hash of an RSA public key which verifies ME FW
- AES key to store sensitive data
- Field Programmable Fuses (FPFs)

Intel ME

Intel ME FW is divided into partitions of various type:

- Code
- Data
- File System
- …

Code partitions verification flow ->



Intel Integrated Sensor Hub (ISH)

Integrated in Intel SoC since ? Bay Trail ?

Seems to be truncated version of Intel ME:

- ROM with bootcode and SRAM
- Has its own HECI
- Has a DMA engine (? shares some memory with ME ?)
- Runs firmware (ISHC partition of ME FW) from common SPI flash

Firmware can be developed and signed by Intel/OEM

Advanced Control and Power Interface (ACPI) Embedded Controller (EC)

MCU, present only on laptops to make power-management and ACPI-related features:

- Fn-buttons
- Touchpad/keyboard
- Battery supply
- …

Runs firmware (generally without any protection against modifications) from:

- internal flash (can be updated by BIOS, the update binary is included into BIOS)
- common SPI flash (since Skylake)

BIOS protection mechanisms

- Hardware Write Protect jumper
- Protected Range (PR) registers
- BLE (BIOS_WE)
- SMM_BWP
- Intel BIOS Guard (PFAT)
- Intel Boot Guard

Though some vendors using a few of these, but there are always many that don’t care…

Intel Boot Guard 1.x*

* - not official version number, this is how I order it’s versions

Intel Boot Guard

Hardware-based boot integrity protection available since Haswell

RESET → Intel CPU boot ROM → Intel BIOS ACM → IBB → BIOS → OS

Operating modes:

- Measured Boot (MB)
- Verified Boot (VB)
- MB + VB

Intel BG. Measured Boot

Measured Boot uses the Trusted Platform Module (TPM) Platform Configuration Registers (PCRs) to reflect boot components integrity

```
Measure (data):
     PCR = H(PCR | H(data))
```

Some sensitive data can be sealed (`TPM_Seal`) to the PCRs state

Intel BG. Verified Boot

Verified Boot cryptographically verifies the integrity of boot components

Options, in case of a verification fail:

- Do nothing
- Immediate shutdown
- Shutdown in timeout (e.g. 1 or 30 minutes)

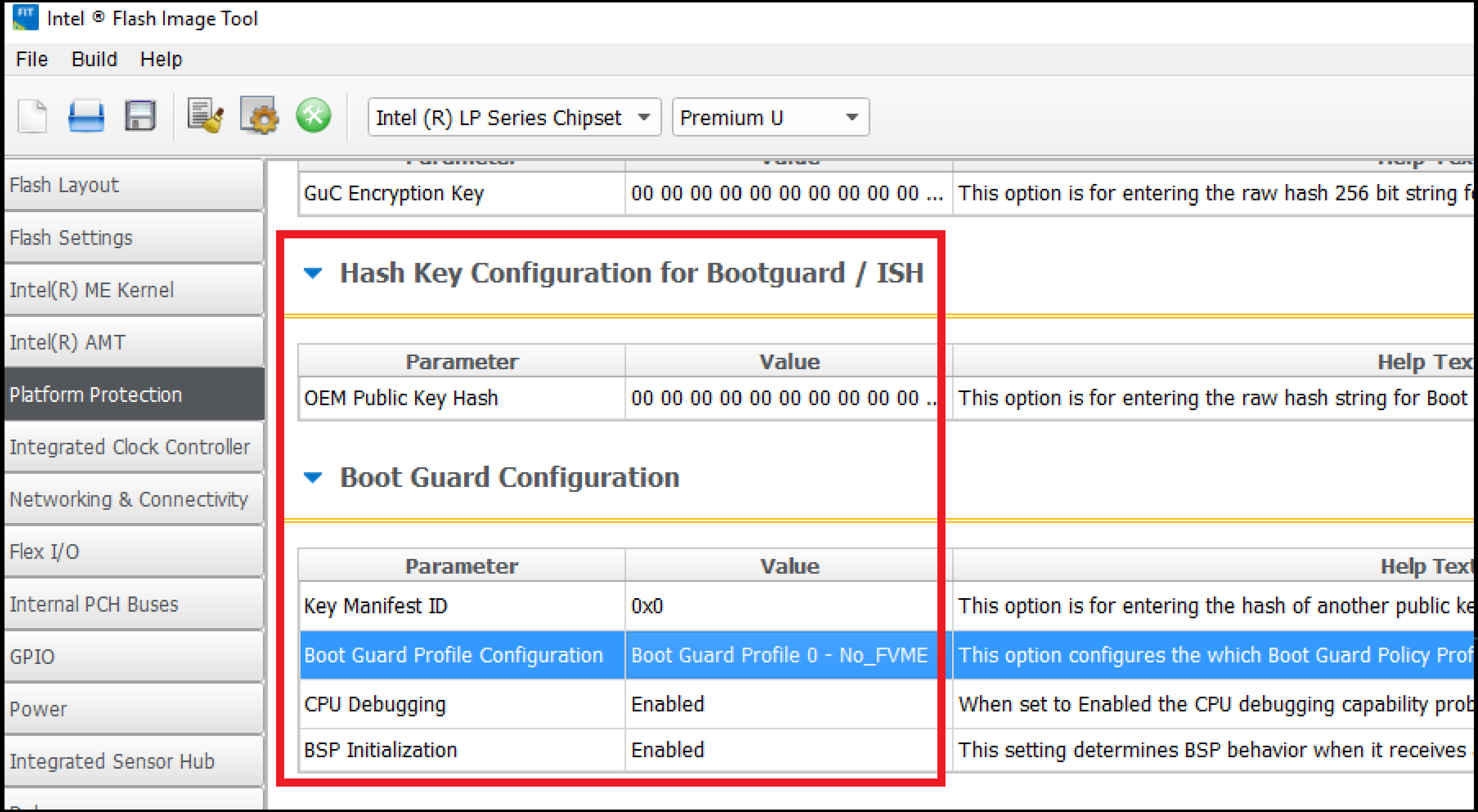
Intel BG. Configuration

Field Programmable Fuses (FPFs) are the hardware non-volatile storage inside Intel ME so only it can program and read them

FPFs fits perfect to store the Intel BG configuration:

- Fuses can be one-time programmable
- Access only through Intel ME

Intel Boot Guard



Intel BG. Configuration

```
typedef struct BG_PROFILE
{
        unsigned long Force_Boot_Guard_ACM : 1;
        unsigned long Protect_BIOS_Environment : 1;
        unsigned long CPU_Debugging : 1;
        unsigned long BSP_Initialization : 1;
        unsigned long Measured_Boot : 1;
        unsigned long Verified_Boot : 1;
        unsigned long Key_Manifest_ID : 4;
        unsigned long Enforcement Policy : 2;   // 00b – do nothing
                                                // 01b – shutdown timeout
                                                // 11b – immediate shutdown

        unsigned long : 20;
};
```

Intel BG. Configuration

BG profiles

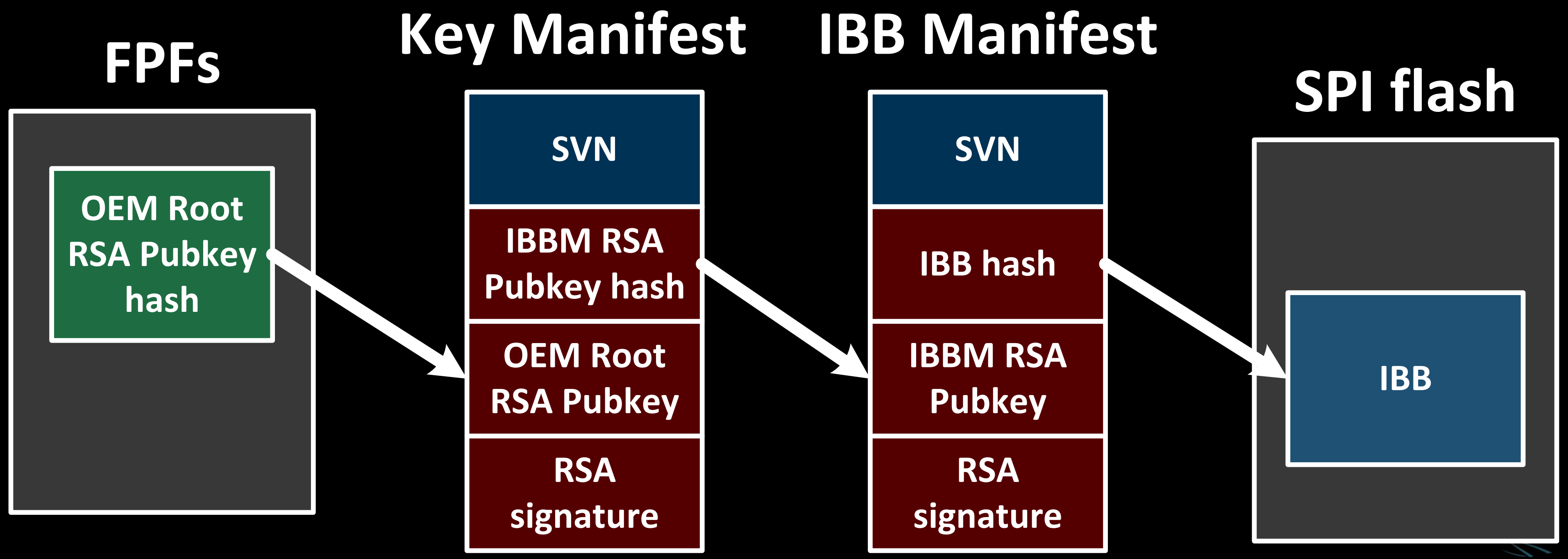
• No_FVME	Disabled
• VE	VB, shutdown timeout
• VME	VB + MB, shutdown timeout
• VM	VB + MB, do nothing
• FVE	VB, immediate shutdown
• FVME	VB + MB, immediate shutdown

Intel BG. Configuration

Intel BG configuration process

1) Prepare image with ME NVARs that should be committed to FPFs
   - Intel Flash Image Tool
2) Close the manufacturing mode and issue a global reset
   - Intel Flash Programming Tool

Intel BG. Verification flow



Researched systems

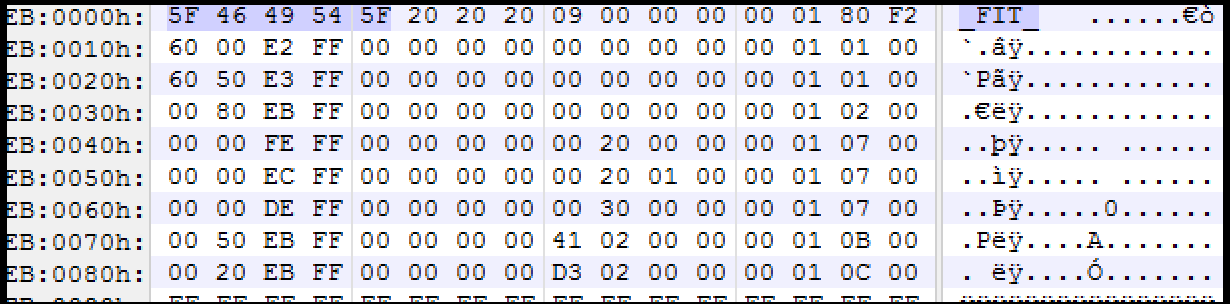
Let’s take a deeper look on BG implementation…

• Gigabyte GA-H170-D3H	BG support present
• Gigabyte GA-Q170-D3H	BG support present
• Gigabyte GA-B170-D3H	BG support present
• MSI H170A Gaming Pro	BG support not present
• Lenovo ThinkPad 460	BG support present
• Lenovo Yoga 2 Pro	BG support not present
• Lenovo U330p	BG support not present

Intel CPU boot ROM

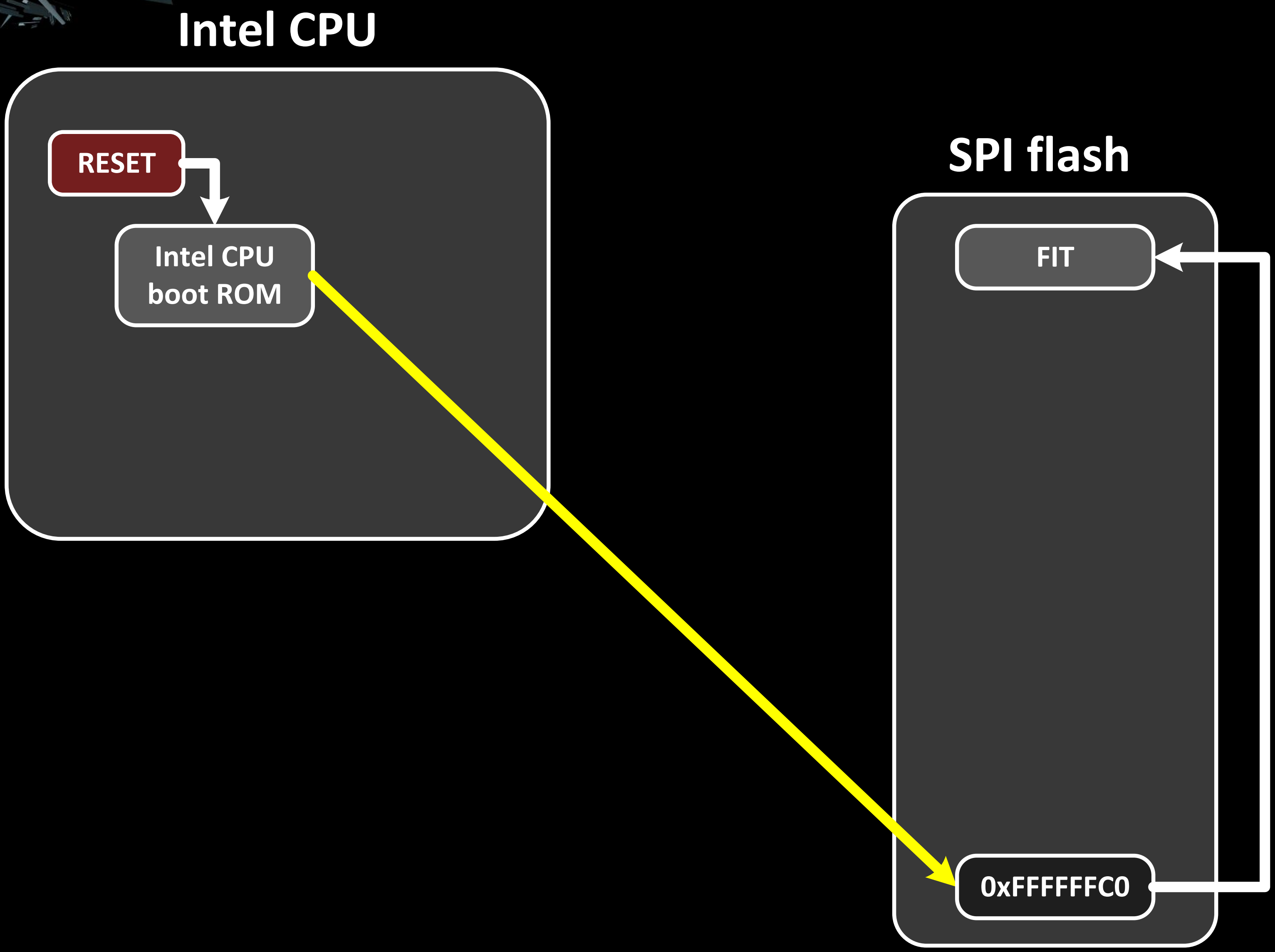
No image of it for researching, but some docs mention that it does:

1) Find the Firmware Interface Table (FIT)
   - FIT base address is located at 0xFFFFFFC0
2) Find Intel BIOS Authenticated Code Module (ACM), verify, load and execute it
   - FIT contains the base address of Intel BIOS ACM



```
EB:0000h: 5F 46 49 54 5F 20 20 20 09 00 00 00 00 01 80 F2  _FIT_   ......€ò
EB:0010h: 60 00 E2 FF 00 00 00 00 00 00 00 00 00 01 01 00  `.âÿ............
EB:0020h: 60 50 E3 FF 00 00 00 00 00 00 00 00 00 01 01 00  `Pãÿ............
EB:0030h: 00 80 EB FF 00 00 00 00 00 00 00 00 00 01 02 00  .€ëÿ............
EB:0040h: 00 00 FE FF 00 00 00 00 00 20 00 00 00 01 07 00  ..þÿ..... ......
EB:0050h: 00 00 EC FF 00 00 00 00 00 20 01 00 00 01 07 00  ..ìÿ..... ......
EB:0060h: 00 00 DE FF 00 00 00 00 00 30 00 00 00 01 07 00  ..Þÿ.....0......
EB:0070h: 00 50 EB FF 00 00 00 00 41 02 00 00 00 01 0B 00  .Pëÿ....A.......
EB:0080h: 00 20 EB FF 00 00 00 00 D3 02 00 00 00 01 0C 00  . ëÿ....Ó.......
```

Intel CPU boot ROM



Intel CPU boot ROM

The FIT is a table of few entries and the first entry is a FIT header

```
typedef struct FIT_HEADER
{
        char              Tag[8];      // '_FIT_   '
        unsigned long     NumEntries;  // including FIT header entry
        unsigned short    Version;     // 1.0
        unsigned char     EntryType;   // 0
        unsigned char     Checksum;
};
```

Intel CPU boot ROM

Other FIT entries have the same format

They describes Intel blobs that are to be parsed\executed before the BIOS, hence before the Legacy RESET-vector (0xFFFFFFF0)

```
typedef struct FIT_ENTRY
{
        unsigned long  BaseAddress;
        unsigned long  : 32;
        unsigned long  Size;
        unsigned short Version;     // 1.0
        unsigned char  EntryType;
        unsigned char  Checksum;
};
```

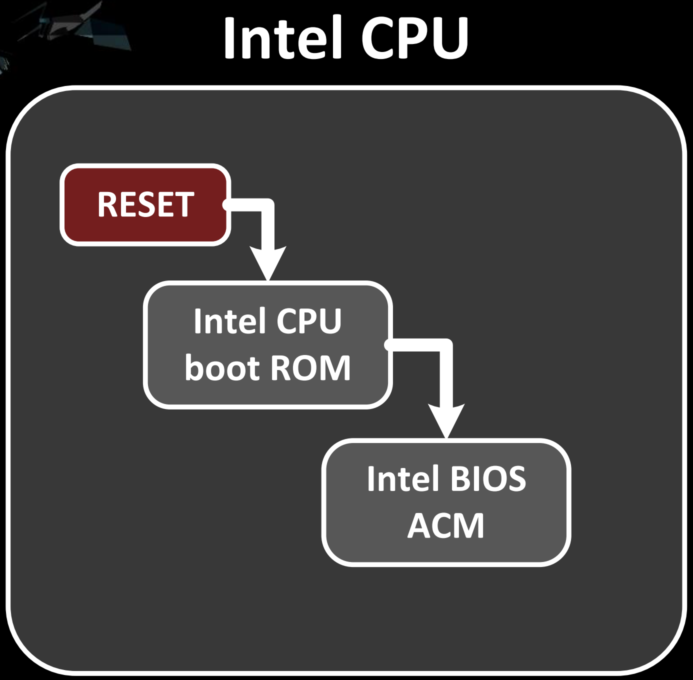
Intel CPU boot ROM

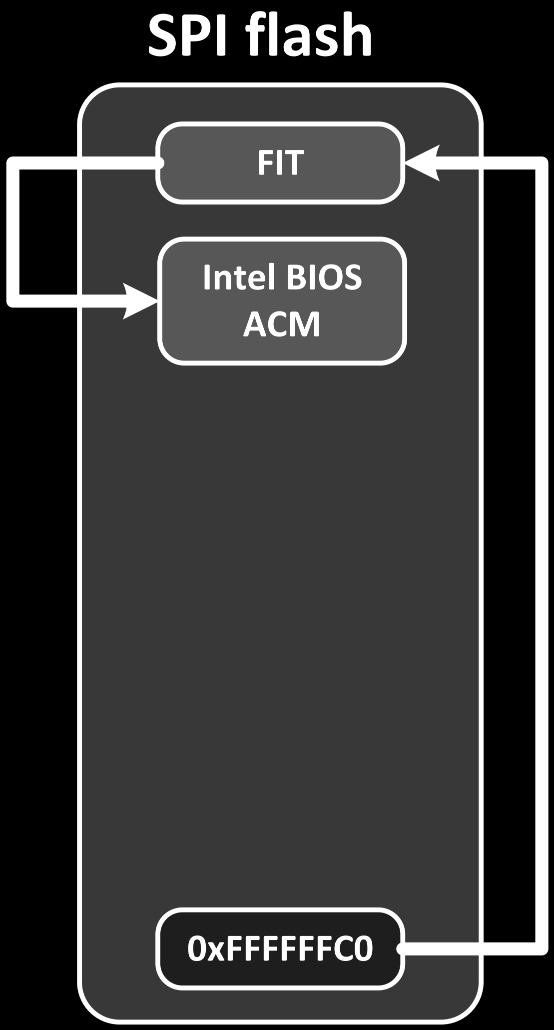
```
enum FIT_ENTRY_TYPES
{
        FIT_HEADER = 0,
        MICROCODE_UPDATE,
        BIOS_ACM,
        BIOS_INIT = 7,
        TPM_POLICY,
        BIOS_POLICY,
        TXT_POLICY,
        BG_KEYM,
        BG_IBBM
};
```

Intel CPU boot ROM

```
typedef struct BIOS_ACM_HEADER
{
            unsigned short ModuleType;      // 2
            unsigned short ModuleSubType;   // 3
            unsigned long  HeaderLength;    // in dwords
            unsigned long  : 32;
            unsigned long  : 32;
            unsigned long  ModuleVendor;    // 8086h
            unsigned long  Date;            // in BCD format
            unsigned long  TotalSize;       // in dwords
            unsigned long  unknown1[6];
            unsigned long  EntryPoint;
            unsigned long  unknown2[16];
            unsigned long  RsaKeySize;      // in dwords
            unsigned long  ScratchSize;     // in dwords
            unsigned char  RsaPubMod[256];
            unsigned long  RsaPubExp;
            unsigned char  RsaSig[256];
};
```

Intel CPU boot ROM





Intel BIOS ACM

```
00003BB1 Start                proc near
00003BB1                      mov     ax, ds
00003BB4                      mov     ss, ax
00003BB7                      mov     es, ax
00003BBA                      mov     fs, ax
00003BBD                      mov     gs, ax
00003BC0                      mov     esp, ebp
00003BC2                      add     esp, 1000h
00003BC8                      mov     eax, ebp
00003BCA                      add     eax, 4C8h
00003BCF                      lidt    fword ptr [eax]
00003BD2                      push    ebp
00003BD3                      call    BootGuard__
00003BD8                      mov     ebx, eax
00003BDA                      mov     edx, 0
00003BDF                      mov     eax, 3
00003BE4                      getsec
```

```
00003912 BootGuard__      proc near               ; CODE XR
00003912
00003912 var_10           = dword ptr -10h
00003912 var_C            = dword ptr -0Ch
00003912 var_8            = dword ptr -8
00003912 var_4            = dword ptr -4
00003912 arg_0            = dword ptr  8
00003912
00003912                  push    ebp
                          mov     ebp, esp
                          sub     esp, 10h
                          push    ebx
                          mov     ebx, [ebp+arg_0]
                          push    esi
                          push    edi
                          xor     eax, eax
                          lea     esi, [ebx+6000h]
                          push    ebx
                          push    esi
                          mov     [ebp+var_10], 0FFF0h
                          mov     [ebp+var_C], eax
                          mov     [ebp+var_8], eax
                          mov     [ebp+var_4], eax
                          call    PlatformInit__
                          mov     edi, eax
                          pop     ecx
00003940                  pop     ecx
00003941                  test    edi, edi
00003943                  jnz     loc_3A3E
00003949                  movzx   eax, word ptr [esi+1F0Eh]
00003950                  test    al, 3
00003952                  jz      loc_3ADA
00003958                  push    esi             ; int
00003959                  call    GetBootGuardData__
0000395E                  mov     edi, eax
00003960                  pop     ecx
00003961                  test    edi, edi
00003963                  jnz     loc_3A3E
00003969                  lea     eax, [ebp+var_C]
0000396C                  push    eax
0000396D                  push    esi
0000396E                  call    BootGuardInit__
00003973                  mov     edi, eax
```

```
00004345 BootGuardInit__ proc near                ; CODE XR
00004345
00004345 arg_0           = dword ptr  4
00004345 arg_4           = dword ptr  8
00004345
00004345                 push    edi
00004346                 mov     edi, [esp+4+arg_0]
0000434A                 call    KeyM__
0000434F                 test    eax, eax
00004351                 jnz     short loc_4384
00004353                 mov     eax, edi
00004355                 call    IbbM__
0000435A                 test    eax, eax
0000435C                 jnz     short loc_4384
0000435E                 mov     edx, [edi+1F28h]
00004364                 mov     ecx, [esp+4+arg_4]
00004368                 add     [ecx], edx
0000436A                 mov     edx, [ecx]
0000436C                 add     edx, [edi+1F30h]
00004372                 push    esi
00004373                 mov     [ecx], edx
00004375                 movzx   esi, word ptr [edi+15BEh]
0000437C                 shl     esi, 0Ch
0000437F                 add     esi, edx
00004381                 mov     [ecx], esi
00004383                 pop     esi
00004384
00004384 loc_4384:                                ; CODE XR
00004384                                          ; BootGua
00004384                 pop     edi
00004385                 retn
00004385 BootGuardInit__ endp
```

Intel BIOS ACM

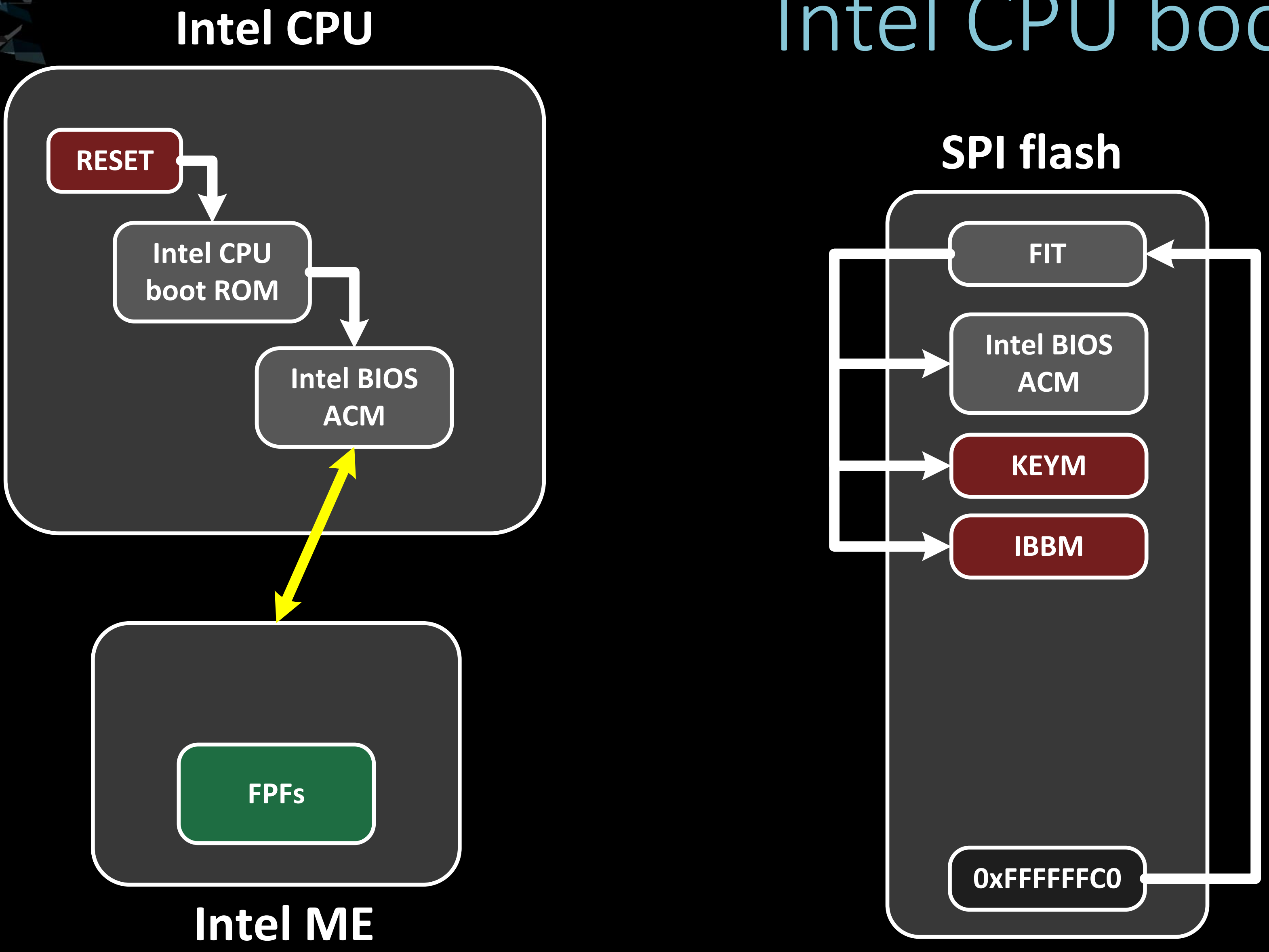
Parse FIT:

1) Retrieve hash of OEM Root Pubkey and Boot Policies from Intel ME
2) Locate Key Manifest (KEYM) and verify it
3) Locate IBB Manifest (IBBM) and verify it

Intel CPU boot ROM

```
enum FIT_ENTRY_TYPES
{
	FIT_HEADER = 0,
	MICROCODE_UPDATE,
	BIOS_ACM,
	BIOS_INIT = 7,
	TPM_POLICY,
	BIOS_POLICY,
	TXT_POLICY,
	BG_KEYM,
	BG_IBBM
};
```

Intel CPU boot ROM



Intel BIOS ACM

```
typedef struct KEY_MANIFEST
{
        char            Tag[8];          // '__KEYM__'
        unsigned char  : 8;              // 10h
        unsigned char  : 8;              // 10h
        unsigned char  : 8;              // 0
        unsigned char  : 8;              // 1
        unsigned short : 16;             // 0Bh
        unsigned short : 16;             // 20h == hash size?
        unsigned char   IbbmKeyHash[32]; // SHA256 of an IBBM public key
        BG_RSA_ENTRY    OemRootKey;
};
```

Intel BIOS ACM

```
typedef struct BG_RSA_ENTRY
{
        unsigned char  : 8;             // 10h
        unsigned short : 16;            // 1
        unsigned char  : 8;             // 10h
        unsigned short RsaPubKeySize;   // 800h
        unsigned long  RsaPubExp;
        unsigned char  RsaPubKey[256];
        unsigned short : 16;            // 14
        unsigned char  : 8;             // 10h
        unsigned short RsaSigSize;      // 800h
        unsigned short : 16;            // 0Bh
        unsigned char  RsaSig[256];
};
```

Intel BIOS ACM

```
typedef struct IBB_MANIFEST
{
        ACBP Acbp;              // Boot policies

        IBBS Ibbs;              // IBB description
        IBB_DESCRIPTORS[];

        PMSG Pmsg;              // IBBM signature
};
```

Intel BIOS ACM

```
typedef struct ACBP
{
        char            Tag[8];         // '__ACBP__'
        unsigned char  : 8;             // 10h
        unsigned char  : 8;             // 1
        unsigned char  : 8;             // 10h
        unsigned char  : 8;             // 0
        unsigned short : 16;            // x & F0h = 0
        unsigned short : 16;            // 0 < x <= 400h
};
```

Intel BIOS ACM

```
typedef struct IBBS
{
        char            Tag[8];             // '__IBBS__'
        unsigned char  : 8;                 // 10h
        unsigned char  : 8;                 // 0
        unsigned char  : 8;                 // 0
        unsigned char  : 8;                 // x <= 0Fh
        unsigned long  : 32;                // x & FFFFFFF8h = 0
        unsigned long  Unknown[20];
        unsigned short : 16;                // 0Bh
        unsigned short : 16;                // 20h == hash size ?
        unsigned char  IbbHash[32];         // SHA256 of an IBB
        unsigned char  NumIbbDescriptors;
};
```

Intel BIOS ACM

Initial Boot Block (IBB) content is described in IBB_DESCRIPTORS

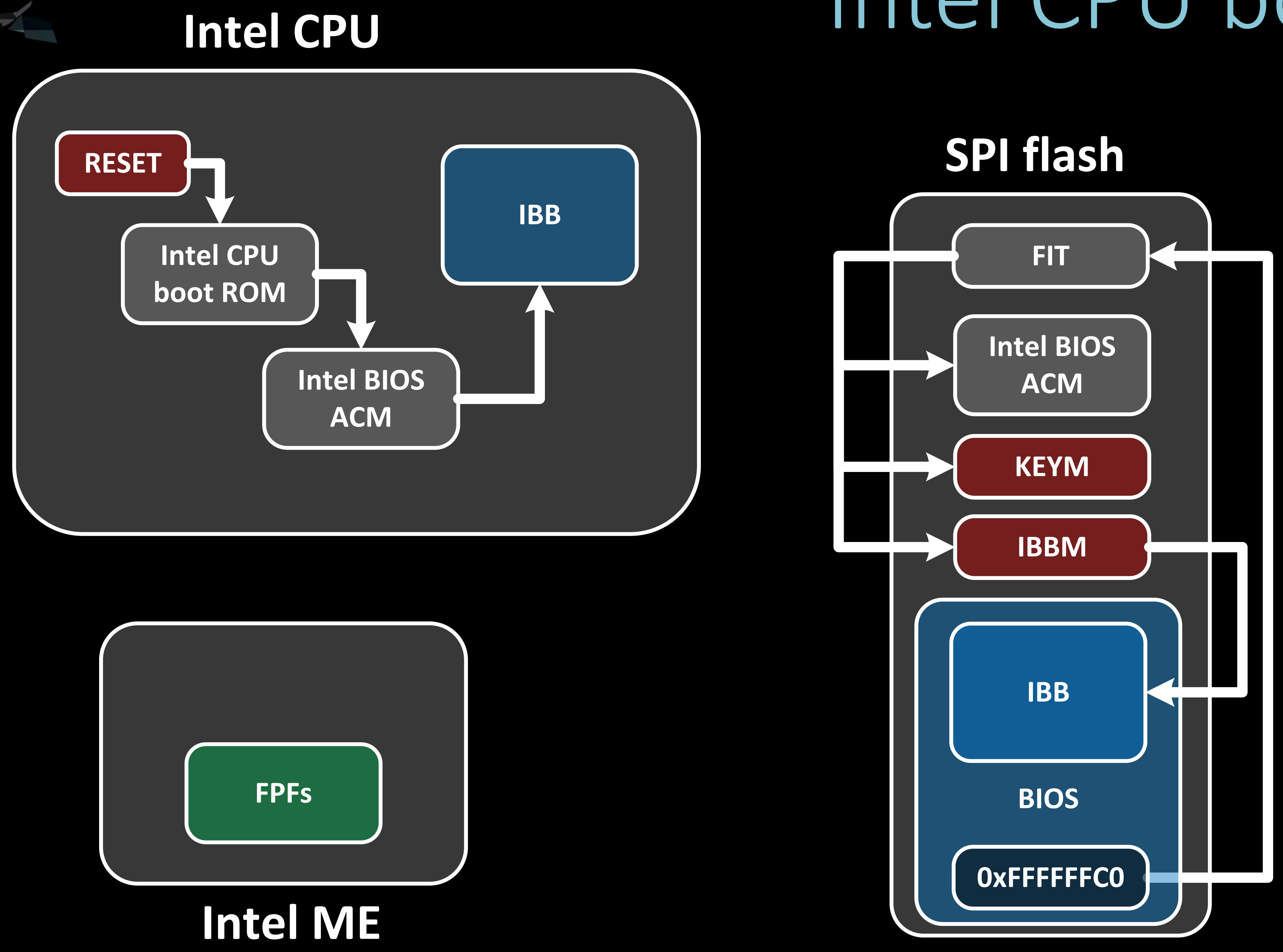
```
typedef struct IBB_DESCRIPTOR
{
        unsigned long  : 32;
        unsigned long  BaseAddress;
        unsigned long  Size;
};
```

So the concatenation of blocks (usually all SEC/PEI modules in UEFI image) that are pointed by IBB descriptors forms the IBB

Intel BIOS ACM

```
typedef struct PMSG
{
        char              Tag[8];                     // '__PMSG__'
        unsigned char     : 8;                        // 10h
        BG_RSA_ENTRY      IbbKey;
};
```

Intel CPU boot ROM



IBB

Hence, the SEC/PEI code is verified before the CPU starts executing from the RESET vector (FFFFFFF0h)

Then the BootGuard supporting code in PEI must verify the DXE volumes

Such PEI module is developed by OEM, e.g.:

- Lenovo
  LenovoVerifiedBootPei {B9F2AC77-54C7-4075-B42E-C36325A9468D}
- Gigabyte
  BootGuardPei {B41956E1-7CA2-42DB-9562-168389F0F066}

IBB

This BootGuard PEI module does:

- Find the hash table by the GUID
- Verify the DXE code pointed by this hash table

LenovoVerifiedBootPei

```
if (EFI_PEI_SERVICES->GetBootMode() != BOOT_ON_S3_RESUME)
{
        if (!FindHashTable())
                return EFI_NOT_FOUND;

        if (!VerifyDxe())
                return EFI_SECURITY_VIOLATION;
}
```

LenovoVerifiedBootPei

Hash table PEI module {389CC6F2-1EA8-467B-AB8A-78E769AE2A15}

```
typedef struct HASH_TABLE
{
        char           Tag[8];            // '$HASHTBL'
        unsigned long NumDxeDescriptors;

        DXE_DESCRIPTORS[];
};

typedef struct DXE_DESCRIPTOR
{
        unsigned char BlockHash[32];      // SHA256
        unsigned long Offset;
        unsigned long Size;
};
```

BootGuardPei

```
int bootMode = EFI_PEI_SERVICES->GetBootMode();

if (bootMode != BOOT_ON_S3_RESUME &&
    bootMode != BOOT_ON_FLASH_UPDATE &&
    bootMode != BOOT_IN_RECOVERY_MODE)
{
      if (!FindHashTable())
            return EFI_NOT_FOUND;

      if (!VerifyDxe())
            return EFI_SECURITY_VIOLATION;
}
```

BootGuardPei

Hash table PEI module {389CC6F2-1EA8-467B-AB8A-78E769AE2A15}

```
typedef HASH_TABLE DXE_DESCRIPTORS[];

typedef struct DXE_DESCRIPTOR
{
	unsigned char BlockHash[32];     // SHA256
	unsigned long BaseAddress;
	unsigned long Size;
};
```

Safeguarding rootkits

The issue

One day I found out that some systems have the SPI flash regions unlocked and the BootGuard configuration not set (nor enabled, nor disabled):

- All Gigabyte systems
- All MSI systems
- 21 Lenovo branded notebook machine types and 4 ThinkServer machine types
- other few vendors I cannot mention at the moment

That's because of the close manufacturing fuse was not set at the end of the manufacturing line.

Lenovo statement

«Lenovo has released fixes for the affected products, which can be found at https://support.lenovo.com/solutions/LEN_9903 or via our security advisory website, https://support.lenovo.com/product_security, and we have adjusted manufacturing processes, where necessary, to prevent reoccurrence of this issue in the future. We sincerely appreciate Mr. Ermolov's responsible disclosure and partnership in this matter.»

Intel statement

«Intel’s guidance to our business partners is to close manufacturing mode at the end of production in order to maximize the security of the platform.»

Safeguarding rootkits

So any user could configure the Intel BG instead of OEM:

- Load into OS
- Modify BIOS
- Write proper BG configuration and verification entities (KEYM, IBBM) using Intel Flash Image Tool
- Set the closemnf fuse using the Intel Flash Programming Tool

This will permanently enable Intel BG on the system and will protect modified BIOS

DEMO

Safeguarding rootkits

The rootkit can be an SMM driver with the following capabilities:

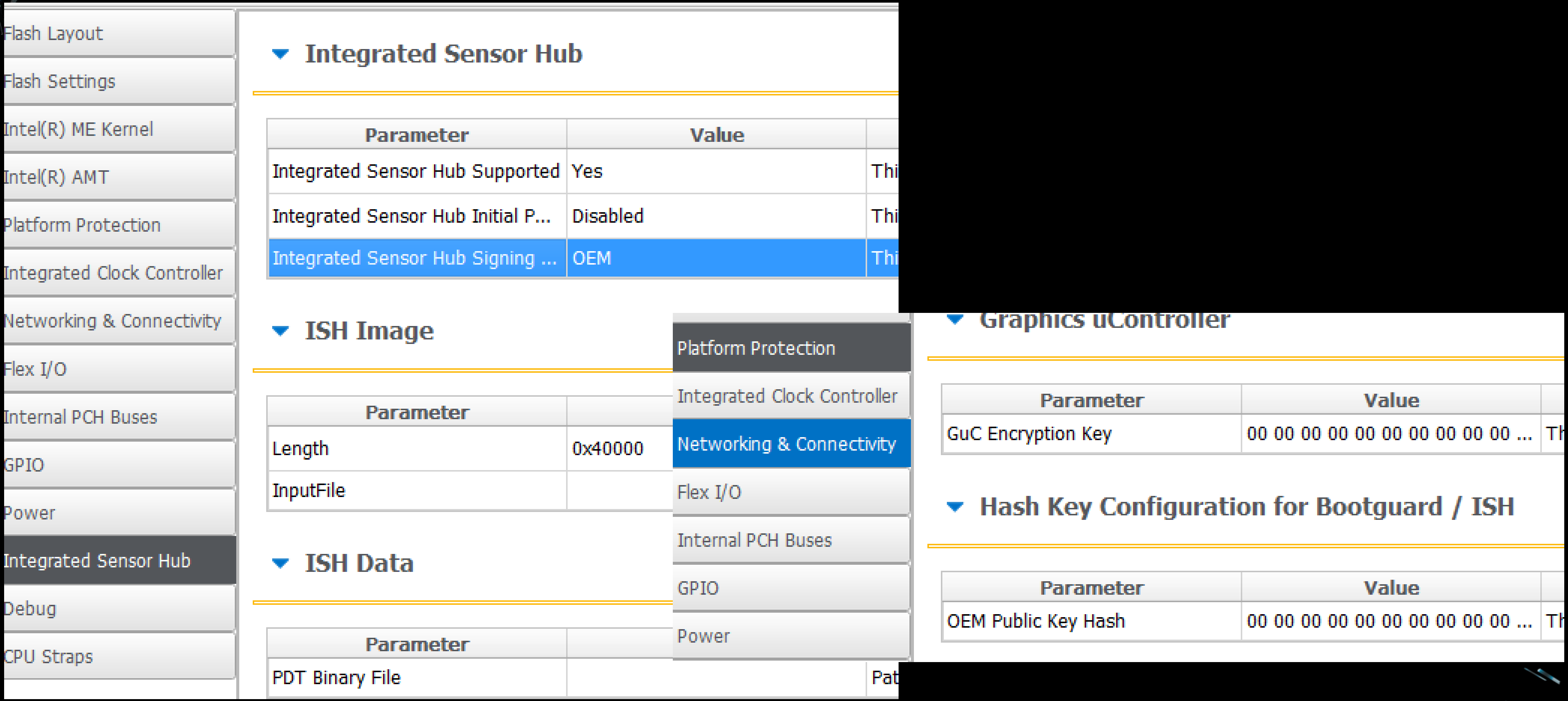
1) Executed during OS
    - Registers a SMI ISR and configure a timer to generate SMI events
2) Full (except ME UMA) access to CPU physical address space and complete isolation from OS
    - SMRAM
3) An encrypted blob which self-decrypts itself during upon each execution

Safeguarding rootkits

Hence, the issue allows:

- to create hidden, black box and irremovable (even with SPI flash programmer) rootkit on a platform
- to modify the ISH firmware on the platform which opens a new attack surface

Safeguarding rootkits



Conclusion

* - not official version number, this is how I order it's versions

Conclusion

- Description of Intel BootGuard implementation
- There are so many proprietary Intel blobs executing before RESET-vector
- The number of execution environments is increasing (CPU x86_64, ME x86, ISH x86, …)
- A scenario to make any past BIOS modification permanent and updatable only from BG Root Key owner

Mitigation

- Vendors that intentionally left the closemnf fuse unset in servicing purposes should find another way
- Vendors that left the closmnf fuse by mistake should roll out a fix (Lenovo have already done this)

- Users can disable the Intel BG technology manually:

  Just run the MEinfo to make sure the Intel BG in not configured on the platform and run the FPT with –closemnf argument

Mitigation

```
OEM Public Key Hash FPF                     Not set
OEM Public Key Hash ME
0000000000000000000000000000000000000000000000000000000000000000
ACM SVN FPF                                 0x0
KM SVN FPF                                  0x0
BSMM SVN FPF                                0x0
GuC Encryption Key FPF                      Not set
GuC Encryption Key ME
0000000000000000000000000000000000000000000000000000000000000000

                                            FPF                     ME
                                            ---                     --
Force Boot Guard ACM                        Not set                 Disabled
Protect BIOS Environment                    Not set                 Disabled
CPU Debugging                               Not set                 Enabled
BSP Initialization                          Not set                 Enabled
Measured Boot                               Not set                 Disabled
Verified Boot                               Not set                 Disabled
Key Manifest ID                             Not set                 0x0
Enforcement Policy                          Not set                 0x0
PTT                                         Not set                 Enabled
EK Revoke State                             Not Revoked
PTT RTC Clear Detection FPF                 Not set
```

Mitigation

```
OEM Public Key Hash FPF
0000000000000000000000000000000000000000000000000000000000000000
OEM Public Key Hash ME
0000000000000000000000000000000000000000000000000000000000000000
ACM SVN FPF                                     0x0
KM SVN FPF                                      0x0
BSMM SVN FPF                                    0x0
GuC Encryption Key FPF
0000000000000000000000000000000000000000000000000000000000000000
GuC Encryption Key ME
0000000000000000000000000000000000000000000000000000000000000000

                                                FPF                      ME
                                                ---                      --
Force Boot Guard ACM                            Disabled                 Disabled
Protect BIOS Environment                        Disabled                 Disabled
CPU Debugging                                   Enabled                  Enabled
BSP Initialization                              Enabled                  Enabled
Measured Boot                                   Disabled                 Disabled
Verified Boot                                   Disabled                 Disabled
Key Manifest ID                                 0x0                      0x0
Enforcement Policy                              0x0                      0x0
PTT                                             Enabled                  Enabled
PTT Lockout Override Counter                    0x0
EK Revoke State                                 Not Revoked
```

Thank you