Intel® Trusted Execution Technology (Intel® TXT)

Software Development Guide

Measured Launched Environment Developer's Guide

March 2011

Document Number: 315168-007

Contents

Figures

Tables

Revision History

Revision Number	Description	Revision Date
-001	• Initial release.	May 2006
-002	• Established public document number • Edited throughout for clarity.	August 2006
-003	• Added launched environment consideration • Renamed LT to Intel® TXT	October 2006
-004	• Updated for production platforms • Use MLE terminology	August 2007
-005	• Updated for latest structure versions and new RLP wakeup mechanism • Added Launch Control Policy information • Removed TEP Appendix • Many miscellaneous changes and additions	June 2008
-006	• Miscellaneous errata • Added definition of LCP v2 • Multiple processor support	December 2009
-007	• Miscellaneous errata • Documented ProcessorIDList support • Described CPU Hotplug handling • Updated TXT configuration registers • Documented new TXT Heap structures • Added LCP_SBIOS_ELEMENT • Documented processor and system state after SENTER/RLP wakeup	March 2011

1 Overview

Intel’s technology for safer computing, Intel® Trusted Execution Technology (Intel® TXT), defines platform-level enhancements that provide the building blocks for creating trusted platforms.

Whenever the word trust is used, there must be a definition of who is doing the trusting and what is being trusted. This enhanced platform helps to provide the authenticity of the controlling environment such that those wishing to rely on the platform can make an appropriate trust decision. The enhanced platform determines the identity of the controlling environment by accurately measuring the controlling software (see Section 1.1).

Another aspect of the trust decision is the ability of the platform to resist attempts to change the controlling environment. The enhanced platform will resist attempts by software processes to change the controlling environment or bypass the bounds set by the controlling environment.

What is the controlling environment for this enhanced platform? The platform is a set of extensions designed to provide a measured and controlled launch of system software that will then establish a protected environment for itself and any additional software that it may execute.

These extensions enhance two areas:

- The launching of the Measured Launched Environment (MLE)
- The protection of the MLE from potential corruption

The enhanced platform provides these launch and control interfaces using Safer Mode Extensions (SMX).

The SMX interface includes the following functions:

- Measured launch of the MLE
- Mechanisms to ensure the above measurement is protected and stored in a secure location
- Protection mechanisms that allow the MLE to control attempts to modify itself

1.1 Measurement and Intel® Trusted Execution Technology

Intel TXT uses the term *measurement* frequently. Measuring software involves processing the executable such that the result (a) is unique and (b) indicates changes in the executable. A cryptographic hash algorithm meets these needs.

A cryptographic hash algorithm is sensitive to even one-bit changes to the measured entity. A cryptographic hash algorithm also produces outputs that are sufficiently large so the potential for collisions (where two hash values are the same) is extremely small. When the term measurement is used in this specification, the meaning is that the measuring process takes a cryptographic hash of the measured entity.

The controlling environment is provided by system software such as an OS kernel or VMM. The software launched using the SMX instructions is known as the Measured Launched Environment (MLE). MLEs provide different launch mechanisms and increased protection (offering protection from possible software corruption).

1.2 Dynamic Root of Trust

A central objective of the Intel TXT platform is to provide a measurement of the launched execution environment.

One measurement is made when the platform boots, using techniques defined by the Trusted Computing Group (TCG). The TCG defines a Root of Trust for Measurement (RTM) that executes on each platform reset; it creates a chain of trust from reset to the measured environment. As the measurement always executes at platform reset, the TCG defines this type of RTM as a Static RTM (SRTM).

Maintaining a chain of trust for a length of time may be challenging for an MLE meant for use in Intel TXT; this is because an MLE may operate in an environment that is constantly exposed to unknown software entities. To address this issue, the enhanced platform provides another RTM with Intel TXT instructions. The TCG terminology for this option is Dynamic Root of Trust for Measurement (DRTM). The advantage of a DRTM (also called the 'late launch' option) is that the launch of the measured environment can occur at any time without resorting to a platform reset. It is possible to launch an MLE, execute for a time, terminate the MLE, execute without virtualization, and then launch the MLE again. One possible sequence is:

1. During the BIOS load: (a) launch an MLE for use by the BIOS, (b) terminate the MLE when its work is done, (c) continue with BIOS processing and hand off to an OS.
2. Then, the OS loads and launches a different MLE.

In both instances, the platform measures each MLE and ensures the proper storage of the MLE measurement value.

1.2.1 Launch Sequence

When launching an MLE, the environment must load two code modules into memory. One module is the MLE. The other is known as an authenticated code (AC) module. The AC module is only in use during the measurement and verification process and is chipset-specific. It is digitally signed by the chipset vendor; the launch process must successfully validate the digital signature before continuing.

With the AC module and MLE in memory, the launching environment can invoke the GETSEC[SENTER] instruction provided by SMX.

GETSEC[SENTER] broadcasts messages to the chipset and other physical or logical processors in the platform. In response, other logical processors perform basic cleanup, signal readiness to proceed, and wait for messages to join the environment created by the MLE. As this sequence requires synchronization, there is an initiating logical processor (ILP) and responding logical processor(s) (RLP(s)). The ILP must be the system BSP, which is the processor with IA32_APIC_BASE MSR.BSP = 1. RLPs are also often referred to as APs.

After all logical processors signal their readiness to join and are in the wait state, the initiating logical processor loads, authenticates, and executes the AC module. The AC module tests for various chipset and processor configurations and ensures the platform has an acceptable configuration. It then measures and launches the MLE.

The MLE initialization routine completes system configuration changes (including redirecting INITs, SMIs, interrupts, etc.); it then issues a new SMX instruction that wakes up the responding logical processors (RLPs) and brings them into the measured environment. At this point, all logical processors and the chipset are correctly configured.

At some later point, it is possible for the MLE to exit and then be launched again, without issuing a system reset.

1.3 Storing the Measurement

SMX operation during the launch provides an accurate measurement of the MLE. After creating the measurement, the initiating logical processor stores that measurement in the Trusted Platform Module (TPM), defined by the TCG. An enhanced platform includes mechanisms that ensure that the measurement of the MLE (completed during the launch process) is properly reported to the TPM.

With the MLE measurement in the TPM, the MLE can use the measurement value to protect sensitive information and detect potential unauthorized changes to the MLE itself.

1.4 Controlled Take-down

Because the MLE controls the platform, exiting the MLE is a controlled process. The process includes: (a) shutting down any guest VMs if they were created; (b) and ensuring that memory previously used does not leak sensitive information.

The MLE cleans up after itself and terminates the MLE control of the environment. If a VMM was running, the MLE may choose to turn control of the platform over to the software that was running in one of the VMs.

1.5 SMX and VMX Interaction

A VM abort may occur while in SMX operation. This behavior is described in the *Intel 64 and IA-32 Software Developer Manual, Volume 3B*. Note that entering authenticated code execution mode or launching of a measured environment affects the behavior and response of the logical processors to certain external pin events.

1.6 Authenticated Code Module

To support the establishment of a measured environment, SMX enables the capability of an authenticated code execution mode. This provides the ability for a special code module, referred to as an authenticated code module (AC module), to be loaded into internal RAM (referred to as authenticated code execution area) within the processor. The AC module is first authenticated and then executed using a tamper resistant mechanism.

Authentication is achieved through the use of a digital signature in the header of the AC module. The processor calculates a hash of the AC module and uses the result to validate the signature. Using SMX, a processor will only initialize processor state or execute the AC module if it passes authentication. Since the authenticated code module is held within the internal RAM of the processor, execution of the module can occur in isolation with respect to the contents of external memory or activities on the external processor bus.

1.7 Chipset Support

One important feature the chipset provides is DMA protection via Intel® Virtualization Technology (Intel® VT) for Directed I/O (Intel® VT-d). Intel® VT-d, under control of the MLE, allows the MLE to protect itself and any other software such as guest VMs from unauthorized device access to memory. Intel VT-d blocks access to specific physical memory pages and the enforcement of the block occurs for all DMA access to the protected pages. See Chapter 1.10 for more information on DMA protection mechanisms.

The Intel TXT architecture also provides extensions that access certain chipset registers and TPM address space.

Chipset registers that interact with SMX are accessed from two regions of memory by system software using memory read/write protocols. These two memory regions, Intel TXT Public space and Intel TXT Private space, are mappings to the same set of chipset registers but with different read/write permissions depending on which space the memory access came through. The Intel TXT Private space is not accessible to system software until it is unlocked by SMX instructions.

The storage spaces accessible within a TPM device are grouped by a locality attribute and are a separate set of address ranges from the Intel TXT Public and Private spaces. The following localities are defined:

- Locality 0 : Non-trusted and legacy TPM operation
- Locality 1 : An environment for use by the Trusted Operating System
- Locality 2 : Trusted OS
- Locality 3 : Authenticated Code Module
- Locality 4 : Intel TXT hardware use only

1.8 TPM Usage

Intel TXT makes extensive use of the Trusted Platform Module (TPM) defined by the Trusted Computing Group (TCG) in the *TCG TPM Specification, Version 1.2*. The TPM provides a repository for measurements and the mechanisms to make use of the measurements. The system makes use of the measurements to both report the current platform configuration and to provide long-term protection of sensitive information.

The TPM stores measurements in Platform Configuration Registers (PCRs). PCRs provide a storage area that allows an unlimited number of measurements in a fixed amount of space. They provide this feature by an inherent property of cryptographic hashes. Outside entities never write directly to a PCR register, they “extend” PCR contents. The extend operation takes the current value of the PCR, appends the new value, performs a cryptographic hash on the combined value, and the hash result is the new PCR value. One of the properties of cryptographic hashes is that they are order dependent. This means hashing A then B produces a different result from hashing B then A. This ordering property allows the PCR contents to indicate the order of measurements.

Sending measurement values from the measuring agent to the TPM is a critical platform task. The Dynamic Root of Trust for Measurement (DRTM) requires specific messages to flow from the DRTM to the TPM. The Intel TXT DRTM is the GETSEC[SENTER] instruction and the system ensures GETSEC[SENTER] has special messages to communicate to the TPM. These special messages take advantage of TPM localities 3 and 4 to protect the messages and inform the TPM that GETSEC[SENTER] is sending the messages.

1.9 PCR Usage

As part of the measured launch, Intel TXT will extend measurements of the elements and configuration values of the dynamic root of trust into certain TPM PCRs. The constituent values of these measurements (indicated below) are provided in the SinitMleData structure described in section C.4.

While the MLE may choose to extend additional values into these PCRs, the values described below are those present immediately after the MLE receives control following the GETSEC[SENTER] instruction.

1.9.1 PCR 17

PCR 17 is initialized using the TPM_HASH_START/TPM_HASH_END sequence. The HASH_DATA provided in this sequence is the concatenation of the hash of the SINIT ACM that was used in the launch process and the 4 byte value of the SENTER parameters (in EDX and also in SinitMleData.EdxSenterFlags). As part of this sequence, PCRs 17-23 are reset to 0. The hash of SINIT is also stored in the SinitMleData.SinitHash field. If the SINIT To MLE Data Table (section C.4) version is 7 or greater, the hash of the SINIT ACM is performed using SHA-256, otherwise it uses SHA-1. If a SHA-256 hash was used, the SinitMleData.SinitHash field will contain the value of PCR 17 after the initial extend operation (see below for more details).

PCR 17 is then extended with the SHA-1 hash of the following items concatenated in this order:

BIOS ACM ID – SinitMleData.BiosAcmID (20 bytes)

STM opt-in indicator – SinitMleData.MsegValid (8 bytes)

SHA-1 hash of the STM (or all 0s if opt-out) – SinitMleData.StmHash (20 bytes)

LCP Control Field of used policy (PS or PO) – SinitMleData.PolicyControl (4 bytes)

SHA-1 hash of used policy (or all 0s if chosen not to be extended) – SinitMleData.LcpPolicyHash (20 bytes)

MLE-chosen Capabilities (or all 0s if chosen not to be extended) – OsSinitData.Capabilities (4 bytes)

If the SINIT To MLE Data Table (section C.4) version is 8 or greater, an additional 4 byte field representing processor-based S-CRTM status is concatenated. This field represents whether the S-CRTM (Static Core Root of Trust for Measurement) was implemented in the processor hardware (1) or in BIOS (0).

If SinitMleData.Version = 6, PCR 17's final value will be:

Extend (SHA-1(SinitMleData.SinitHash | SinitMleData.EdxSenterFlags))

Extend (SHA-1 (SinitMleData.BiosAcm.ID | SinitMleData.MsegValid | SinitMleData.StmHash | SinitMleData.PolicyControl | SinitMleData.LcpPolicyHash | (OsSinitData.Capabilities, 0)))

If SinitMleData.Version = 7, PCR 17's final value will be:

SHA-1 (SinitMleData.SinitHash | SHA-1 (SinitMleData.BiosAcm.ID | SinitMleData.MsegValid | SinitMleData.StmHash | SinitMleData.PolicyControl | SinitMleData.LcpPolicyHash | (OsSinitData.Capabilities, 0)))

If SinitMleData.Version >= 8, PCR 17's final value will be:

SHA-1 (SinitMleData.SinitHash | SHA-1 (SinitMleData.BiosAcm.ID | SinitMleData.MsegValid | SinitMleData.StmHash | SinitMleData.PolicyControl | SinitMleData.LcpPolicyHash | (OsSinitData.Capabilities, 0) | SinitMleData.ProcessorSCRTMStatus))

Where the Extend() operation is a SHA-1 hash of the previous value in the PCR concatenated with the value being extended (the previous value is 20 bytes of 0s in the case of the first extend to a PCR).

1.9.2 PCR 18

PCR 18 will be extended with the SHA-1 hash of the MLE, as reported in the SinitMleData.MleHash field.

Thus, PCR 18's final value will be:

Extend (SinitMleData.MleHash)

1.10 DMA Protection

This chapter briefly describes the two chipset mechanisms that can be used to protect regions of memory from DMA access by bus master devices. More details on these mechanisms can be found in the External Design Specification (EDS) of the targeted chipset family and Intel® *Virtualization Technology for Directed I/O Architecture Specification.*

1.10.1 DMA Protected Range (DPR)

The DMA Protected Range (DPR) is a region of contiguous physical memory whose last byte is the byte before the start of TSEG, and which is protected from all DMA access. The DPR size is set and locked by BIOS. This protection is applied to the final physical address after any other translations (e.g. Intel VT-d, GART, etc.).

The DPR covers the Intel TXT heap and SINIT AC Module reserved memory (as specified in the TXT.SINIT.BASE/TXT.SINIT.SIZE registers). On current systems it is 3MB in size, and though this may change in the future it will always be large enough to cover the heap and SINIT regions.

The MLE itself may reside in the DPR as long as it does not conflict with either the SINIT or heap areas. If it does reside in the DPR then it need not be covered by the Intel VT-d Protected Memory Regions.

1.10.2 Intel® Virtualization Technology (Intel® VT) for Directed I/O (Intel® VT-d) Protected Memory Regions (PMRs)

The Intel® VT-d Protected Memory Regions (PMRs) are two ranges of physical addresses that are protected from DMA access. One region must be in the lower 4GB of memory and the other must be in the upper 4GB. Either or both may be unused.

The use of the PMRs is not mutually exclusive of DMA remapping. If the MLE enables DMA remapping, it should place the Intel VT-d page tables within the PMR region(s) in order to protect them from DMA activity prior to turning on remapping. While it is not required that PMRs be disabled once DMA remapping is enabled, if the MLE wants to manage all DMA protection through remapping tables then it must explicitly disable the PMR(s).

The MLE may reside within one of the PMR regions. If the MLE is not within the DPR region then it must be within one of the PMR regions, else SINIT will not permit the environment to be launched.

For more details of the PMRs, see the Intel® *Virtualization Technology for Directed I/O Architecture Specification.*

1.11 Intel® TXT Shutdown

1.11.1 Reset Conditions

When an Intel TXT shutdown condition occurs, the processor or software writes an error code indicating the reason for the failure to the TXT.ERRORCODE register. It then writes to the TXT.CMD.RESET command register, initiating a platform reset. After the write to TXT.CMD.RESET, the processor enters a shutdown sleep state with all external pin events, bus or error events, machine check signaling, and MONITOR/MWAIT event signaling masked. Only the assertion of reset back to the processor takes it out of this sleep state. The Intel TXT error code register is not cleared by the platform reset; this makes the error code accessible for post-reset diagnostics.

An Intel TXT shutdown can be generated by the processor during execution of certain GETSEC leaf functions (for example: ENTERACCS, EXITAC, SENTER, SEXIT), where recovery from an error condition is not considered reliable. This situation should be interpreted as an abort of authenticated execution or measured environment launch.

A legacy IA-32 triple-fault shutdown condition is also converted to an Intel TXT shutdown sequence if the triple-fault shutdown occurs during authenticated code execution mode or while the measured environment is active. The same is true for other legacy non-SMX specific fault shutdown error conditions. Legacy shutdown to Intel TXT shutdown conversions are defined as the mode of operation between:

- Execution of the GETSEC functions ENTERACCS and EXITAC
- Recognition of the message signaling the beginning of the processor rendezvous after GETSEC[SENTER] and the message signaling the completion of the processor rendezvous

Additionally, there is a special case. If the processor is in VMX operation while the measured environment is active, a triple-fault shutdown condition that causes a guest exiting event back to the Virtual Machine Monitor (VMM) supersedes conversion to the Intel TXT shutdown sequence. In this situation, the VMM remains in control after the error condition that occurred at the guest level and there is no need to abort processor execution.

Given the above situation, if the triple-fault shutdown occurs at the root level of the MLE or a VMX abort is detected, then an Intel TXT shutdown sequence is signaled. For more details on a VMX abort, see Chapter 23, “VM Exits,” in the *Intel 64 and IA-32 Software Developer Manuals, Volume 3B*.

2 Measured Launched Environment

Intel TXT can be used to launch any type of code. However, this section describes the launch, operation and teardown of a Virtual Machine Monitor (VMM) using Intel TXT; any other code would have a similar sequence.

2.1 MLE Architecture Overview

Any Measured Launched Environment (MLE) will generally consist of three main sections of code: the initialization, the dispatch routine, and the shutdown. The initialization code is run each time the Intel TXT environment is launched. This code includes code to setup the MLE on the ILP and join code to initialize the RLPs.

After initialization, the MLE behaves like the unmeasured version would have; in the case of a VMM, this is trapping various guest operations and virtualizing certain processor states.

Finally the MLE prepares for shutdown by again synchronizing the processors, clearing any state and executing the GETSEC[SEXIT] instruction.

Table 1 shows the format of the MLE Header structure which is stored within the MLE image. The MLE Header structure is used by the SINIT AC module to set up the correct initial MLE state and to find the MLE entry point. The header is part of the MLE hash.

Table 1. MLE Header structure

Field	Offset	Size (bytes)	Description
UUID	0	16	Identifies this structure
HeaderLen	16	4	Length of header in bytes
Version	20	4	Version number of this structure
EntryPoint	24	4	Linear entry point of MLE
FirstValidPage	28	4	Starting linear address of (first valid page of) MLE
MleStart	32	4	Offset within MLE binary file of first byte of MLE, as specified in page table
MleEnd	36	4	Offset within MLE binary file of last byte + 1 of MLE, as specified in page table
Capabilities	40	4	Bit vector of MLE-supported capabilities
CmdlineStart	44	4	Starting linear address of command line

Field	Offset	Size (bytes)	Description
CmdlineEnd	48	4	Ending linear address of command line

UUID: This field contains a UUID which uniquely identifies this MLE Header Structure. The UUID is defined as follows:

```
        ULONG   UUID0;      // 9082AC5A
        ULONG   UUID1;      // 74A7476F
        ULONG   UUID2;      // A2555C0F
        ULONG   UUID3;      // 42B651CB
```

This UUID value should only exist in the MLE (binary) in this field of the MLE header. This implies that this UUID should not be stored as a variable nor placed in the code to be assigned to this field. This can also be ensured by analyzing the binary.

HeaderLen: this field contains the length in bytes of the MLE Header Structure.

Version: this field contains the version of the MLE header, where the upper two bytes are the major version and the lower two bytes are the minor version. Changes in the major version indicate that an incompatible change in behavior is required of the MLE or that the format of this structure is not backwards compatible. Version 2.1 (20001H) is the currently supported version.

EntryPoint: this field is the linear address, within the MLE's linear address space, at which the ILP will begin execution upon completion of the GETSEC[SENTER] instruction.

FirstValidPage: this field is the starting linear address of the MLE. This will be verified by SINIT to match the first valid entry in the MLE page tables.

MleStart / MleEnd: these fields are intended for use by system software that needs to know which portion of an MLE binary file is the MLE, as defined by its page table. This might be useful for calculating the MLE hash when the entire binary file is not being used as the MLE.

Capabilities: this bit vector represents TXT-related capabilities that the MLE supports. It will be used by the SINIT AC module to determine whether the MLE is compatible with it and as needed for any optional capabilities. The currently defined bits for this are:

Table 2. MLE/SINIT Capabilities Field Bit Definitions

Bit position	Description
0	Support for GETSEC[WAKEUP] for RLP wakeup (1) All MLEs should support this.
1	Support for RLP wakeup using MONITOR address (SinitMleData.RlpWakeupAddr) (1) All MLEs should support this.
2	The ECX register will contain the pointer to the MLE page table on return from SINIT to the MLE EntryPoint (1).
3	STM support (1).

31:4	Reserved (must be 0)

CmdlineStart / CmdlineEnd: these fields are intended for use by system software that needs calculate the MLE hash, for MLEs that include their command lines in their identity. These are linear addresses within the MLE of the beginning and end of a buffer that will contain the command line. The buffer is padded with 0 bytes at the end. MLEs which do not include the command line in their identity should set these fields to 0.

2.2 MLE Launch

At some point system software will start an Intel TXT environment. This may be done at operating system loader time or could be done after the operating system boots. From this point on we will assume that the operating system is starting the Intel TXT environment and refer to this code as the system software.

After the measured environment startup, the application processors (RLPs) will not respond to SIPIs as they did before SENTER. Once the measured environment is launched, the RLPs cannot run the real-mode MP startup code. An alternative MP startup algorithm will need to be developed. The new MP startup algorithm would not require the RLPs to leave protected mode with paging on. The OS may also be required to detect whether a measured environment has been established and use this information to decide which MP startup algorithm is appropriate (the standard MP startup algorithm or the modified algorithm).

This section shows the pseudocode for preparing the system for the SMX measured launch. The following describes the process in a number of sub-sections:

- Intel TXT detection and processor preparation
- Detection of previous errors
- Loading the SINIT AC module
- Loading the MLE and processor rendezvous
- Performing a measured launch

2.2.1 Intel® TXT Detection and Processor Preparation

Lines 1 - 4: Before attempting to launch the measured environment, the system software should check that all logical processors support VMX and SMX (the check for VMX support is not necessary if the environment to be launched will not use VMX).

For single processor socket systems, it is sufficient if this action is only performed by the ILP. This includes physical processors containing multiple logical processors. In order to correctly handle multiple processor socket systems, this check must be performed on all logical processors. It is possible that two physical processor within the same system may differ in terms of SMX and VMX capabilities.

For details on detecting and enabling VMX see chapter 19, “Introduction to Virtual-Machine Extensions”, in the *Intel 64 and IA-32 Software Developer Manuals, Volume 3B*. For details on detecting and enabling SMX support see chapter 6, “Safer Mode Extensions Reference”, in the *Intel 64 and IA-32 Software Developer Manuals, Volume 2B* .

Lines 5 - 9: System software should check that the chipset supports Intel TXT prior to launching the measured environment. The presence of the Intel TXT chipset can be detected by executing GETSEC[CAPABILITIES] with EAX=0 & EBX=0. This instruction will return the 'Intel TXT Chipset' bit set in EAX if an Intel TXT chipset is present. The processor must enable SMX before executing the GETSEC instruction.

Lines 10 – 12: System software should also verify that the processor supports all of the GETSEC instruction leaf indices that will be needed. The minimal set of instructions required will depend on the system software and MLE, but is most likely SENTER, SEXIT, WAKEUP, SMCTRL, and PARAMETERS. The supported leaves are indicated in the EAX register after executing the GETSEC[CAPABILITIES] instruction as indicated above.

Listing 1. Intel® TXT Detection Pseudocode

```
//
// Intel TXT detection
// Execute on all logical processors for compatibility with
// multiple processor systems
//
1. CPUID(EAX=1);
2. IF (SMX not supported) OR (VMX not supported) {
3.     Fail measured environment startup;
4. }

//
// Enable SMX on ILP & check for Intel TXT chipset
//
5. CR4.SMXE = 1;
6. GETSEC[CAPABILITIES];
7. IF (Intel TXT chipset NOT present) {
8.     Fail measured environment startup;
9. }
10.  IF (All needed SMX GETSEC leaves are NOT supported) {
11.      Fail measured environment startup;
12.  }
```

2.2.2 Detection of Previous Errors

In order to prevent a cycle of failures or system resets, it is necessary for the system software to check for errors from a previous launch. Errors that are detected by system software prior to executing the GETSEC[SENTER] instruction will be specific to that software and, if persisted, will be in a manner specific to the software. Errors generated during execution of the GETSEC[SENTER] instruction result in a system reset and the error code being stored in the TXT.ERRORCODE register. Possible remediation steps are described in section 4.2.

Lines 1 - 3: The error code from an error generated during the GETSEC[SENTER] instruction is stored in the TXT.ERRORCODE register, which is persistent across soft resets. A non-zero value indicates an error. Error codes are specific to an SINIT AC module and can be found in a text file that is distributed with the module.

Lines 4 - 6: If the TXT_RESET.STS bit of the TXT.ESTS register is set, then in order to maintain TXT integrity the GETSEC[SENTER] instruction will fail. System software should detect this condition as early as possible and, after taking the appropriate remediative action, power cycle the system to clear this bit and permit a launch.

Listing 2. Error Detection Pseudocode

```
//
// Detect previous GETSEC[SENTER] failures
//
1. IF (TXT.ERRORCODE != 0) {
2.     Take remediative action;
3. }

//
// Detect previous TXT Reset
//
4. IF (TXT.ESTS[TXT_RESET.STS] != 0) {
5.     Power-cycle system;
6. }
```

2.2.3 Loading the SINIT AC Module

This action is only performed by the ILP.

BIOS may already have the correct SINIT AC module loaded into memory or system software may need to load the SINIT code from disk into memory. The system software may determine if an SINIT AC module is already loaded by examining the preferred SINIT load location (see below) for a valid SINIT AC module header.

System software should always use the most recent version of the SINIT AC module available to it. It can determine this by comparing the Date fields in the AC module headers.

System software should also match a prospective SINIT AC module to the chipset before loading and attempting to launch the module. This is described in the next two sections of this document.

System software owns the policy for deciding which SINIT module to load. It must load the previously loaded SINIT AC module in order to unseal data sealed to a previously launched environment. If an SINIT AC module is to be changed (e.g. upgraded to the latest version), the system software must allow the user to migrate secrets prior to loading the new SINIT AC module.

The BIOS reserves a region of physically contiguous memory for the SINIT AC module, which it specifies through the TXT.SINIT.BASE and TXT.SINIT.SIZE Intel TXT configuration registers. By convention, 128 KBytes of physically contiguous memory is allocated for the purpose of loading the SINIT AC module. System software must use this region for any SINIT AC module that it loads.

The SINIT AC module must be located on a 4 KByte aligned memory location. The SINIT AC module must be mapped WB using the MTRRs and all other memory must be mapped to one of the supportable memory types returned by GETSEC[PARAMETERS]. The MTRRs which map the SINIT AC module must not overlap more than 4 KBytes of memory beyond the end of the SINIT AC image. See the

GETSEC[ENTERACCS] instruction and the Authenticated Code Module Format, section A.1, for more details on these restrictions.

The pages containing the SINIT AC module image must be present in memory before attempting to launch the measured environment. The SINIT AC module image must be loaded below 4 GBytes. System software should check that the SINIT AC module will fit within the AC execution region as specified by the GETSEC[PARAMETERS] leaf. System software should not utilize the memory immediately after the SINIT AC module up to the next 4 KByte boundary. On certain Intel TXT implementations, execution of the SINIT AC module will corrupt this region of memory.

2.2.3.1 Matching an AC Module to the Platform

As part of system software loading an SINIT AC module, the system software should first verify that the file to be loaded is really an SINIT AC module. This may be done at installation time or runtime. Lines 1 - 10 in Listing 3 below show how to do this.

Each AC module is designed for a specific chipset or set of chipsets and, optionally, processor(s). Software can examine the Chipset ID and Processor ID Lists embedded in the AC module binary to determine which chipsets and processors an AC module supports. Software should read the chipset's TXT.DIDVID register and parse the Chipset ID List to find a matching entry. If the AC module also contains a Processor ID List, then software should also match the AC module against the processor cpuid and IA32_PLATFORM_ID MSR. Attempting to execute an AC module that does not match the chipset and processor will result in a failure of the AC module to complete normal execution and an Intel TXT Shutdown.

Listing 3. AC Module Matching Pseudocode

```
TXT_ACM_HEADER               *AcmHdr;        // see Table 3
TXT_CHIPSET_ACM_INFO_TABLE   *InfoTable;     // see Table 5

//
// Find the Chipset AC Module Information Table
//
1. AcmHdr = (TXT_ACM_HEADER *)AcmImageBase;
2. UserAreaOffset = (AcmHdr->HeaderLen + AcmHdr->ScratchSize)*4;
3. InfoTable = (TXT_CHIPSET_ACM_INFO_TABLE *)(AcmBase +
                                             UserAreaOffset);

//
// Verify image is really an AC module
//
1. IF (InfoTable->UUID0 != 0x7FC03AAA) OR
2.    (InfoTable->UUID1 != 0x18DB46A7) OR
3.    (InfoTable->UUID2 != 0x8F69AC2E) OR
4.    (InfoTable->UUID3 != 0x5A7F418D) {
5.      Fail: not an AC module;
6. }

//
// Verify it is an SINIT AC module
//
7. IF (AcmHdr->ModuleType != 2) OR
```

```
8.    (InfoTable->ChipsetACMType != 1) {
9.    Fail: not an SINIT AC module;
10.  }


//
// Verify AC module and chipset production flags match
//
11.  IF (TXT.VER.FSBIF != 0xFFFFFFFF) {
12.     IF (AcmHdr->Flags[15] == TXT.VER.FSBIF[31]) {
13.        Fail: production flags mismatch;
14.     }
15.  }
16.  ELSE IF (AcmHdr->Flags[15] == TXT.VER.EMIF[31]) {
17.        Fail: production flags mismatch;
18.  }

//
// Match AC module to system chipset
//
TXT_ACM_CHIPSET_ID_LIST   *ChipsetIdList;    // see Table 6
TXT_ACM_CHIPSET_ID        *ChipsetId;        // see Table 7

19.  ChipsetIdList = (TXT_ACM_CHIPSET_ID_LIST *)
                   (AcmImageBase + InfoTable->ChipsetIdList);

//
// Search through all ChipsetId entries and check for a match.
//
20.  FOR (i = 0; i < ChipsetIdList->Count; i++) {
21.      //
22.      // Check for a match with this ChipsetId entry.
23.      //
24.      ChipsetId = ChipsetIdList->ChipsetIDs[i];
25.      IF ((TXT.DIDVID[VID] == ChipsetId->VendorId) &&
26.          (TXT.DIDVID[DID] == ChipsetId->DeviceId) &&
27.          ((((ChipsetId->Flags & 0x1) == 0) &&
28.            (TXT.DIDVID[RID] == ChipsetId->RevisionId)) ||
29.          (((ChipsetId->Flags & 0x1) == 0x1) &&
30.            (TXT.DIDVID[RID] & ChipsetId->RevisionId != 0)))) {
31.          AC module matches system chipset;
32.          GOTO CheckProcessor;
33.      }
34.  }
35.  Fail: AC module does not match system chipset;

CheckProcessor:
//
// Match AC module to processor
//
TXT_ACM_PROCESSOR_ID_LIST   *ProcessorIdList;   // see Table 8
TXT_ACM_PROCESSOR_ID        *ProcessorId;       // see Table 9
```

```
36.  ProcessorIdList = (TXT_ACM_PROCESSOR_ID_LIST *)
                   (AcmImageBase + InfoTable->ProcessorIdList);

//
// Search through all ProcessorId entries and check for a match.
//
37.  FOR (i = 0; i < ProcessorIdList->Count; i++) {
38.      //
39.      // Check for a match with this ProcessorId entry.
40.      //
41.      ProcessorId = ProcessorIdList->ProcessorIDs[i];
42.      IF (ProcessorId->FMS ==
43.           (cpuid[1].EAX & ProcessorId->FMSMask)) &&
44.          (ProcessorId->PlatformID ==
45.           (IA32_PLATFORM_ID MSR & ProcessorId->PlatformMask))
46.          AC module matches processor;
47.      }
48.  }
49.  Fail: AC module does not match processor;
```

2.2.3.2 Verifying Compatibility of SINIT with the MLE

Over time, new features and capabilities may be added to the SINIT AC module that can be utilized by an MLE that is aware of those features. Likewise, features or capabilities may be added that *require* an MLE to be aware of them in order to interoperate properly. In order to expose these features and capabilities and permit the MLE and SINIT to determine whether they support a compatible set, the MLE header contains a Capabilities field (see Table 1) that corresponds to the Capabilities field in the SINIT AC module Information Table (see Table 5).

In addition, the MinMleHeaderVer field in the AC module Information Table allows SINIT to indicate that it requires a certain minimal version of an MLE. This allows for new behaviors or features that require support from the MLE, but which older MLEs would not be aware of.

Listing 4 shows the pseudocode for the MLE to determine if it is compatible with the provided SINIT AC module.

Whiles lines 4 – 6 may be redundant with current SINIT AC modules if the MLE supports both RLP wakeup mechanisms, this permits graceful handling of future changes.

Listing 4. SINIT/MLE Compatibility Pseudocode

```
//
// Check that SINIT supports this version of the MLE
//
1. IF (InfoTable->MinMleHeaderVer > MleHeader.Version) {
2.     Fail:  SINIT requires a newer MLE
3. }

//
// Check that the known RLP wakeup mechanisms are supported
//
4. IF (MLE does NOT support at least one RLP wakeup mechanism
   specified in InfoTable->Capabilities) {
5.     Fail:  RLP wakeup mechanisms are incompatible
6. }
```

2.2.4 Loading the MLE and Processor Rendezvous

2.2.4.1 Loading the MLE

System software allocates memory for the MLE and MLE page table. The MLE is not required to be loaded into physically contiguous memory. The pages containing the MLE image must be pinned in memory and all these pages must be located in physical memory below 4 GBytes.

System software creates an MLE page table structure to map the entire MLE image. The pages containing the MLE page tables must be pinned in memory prior to launching the measured environment. The MLE page table structure must be in the format of the IA-32 Physical Address Extension (PAE) page table structure.

The MLE page table has several special requirements:

- The MLE page tables may contain only 4 KByte pages.
- A breadth-first search of page tables must produce increasing physical addresses.
- Neither the MLE nor the page tables may overlap certain regions of memory:
  - device memory (PCI, PCIe*, etc.)
  - addresses between [640k, 1M) or above Top of Memory (TOM)
  - ISA hole (if enabled)
  - the Intel TXT heap or SINIT memory regions
  - Intel VT-d DMAR tables
- There may not be any invalid (not-present) page table entries after the first valid entry (i.e. there may not be any gaps in the MLE's linear address space).
- The Page Directories must be in a lower physical address than the Page Tables.
- The Page-Directory-Pointer-Table must be in a lower physical address than the Page-Directories.
- The page table pages must be in lower physical addresses than the MLE.

Later, the SINIT AC module will check that the MLE page table matches these requirements before calculating the MLE digest. The second rule above implies that the MLE must be loaded into physical memory in an ordered fashion: a scan of MLE virtual addresses must find increasing physical addresses. The system software can order its list of physical pages before loading the MLE image into memory.

The MLE is not required to begin at linear address 0. There may be any number of invalid/not-present entries in the page table prior to the beginning of the MLE pages (i.e. first valid page). The starting linear address should be placed in the FirstValidPage field of the MLE header structure (see Section 2.1).

If the MLE will use this page table after launch then it needs to ensure that the entry point page is identity-mapped so that when it enables paging post-launch, the physical address of the instruction after paging is enabled will correspond to its linear address in the paged environment.

System software writes the physical base address of the MLE page table's page directory to the Intel TXT Heap. The size in bytes of the MLE image is also written to the Intel TXT Heap; see Appendix C.

2.2.4.2 Intel® Trusted Execution Technology Heap Initialization

Information can be passed from system software to the SINIT AC module and from system software to the MLE using the Intel TXT Heap. The SINIT AC module will also use this region to pass data to the MLE.

The system software launching the measured environment is responsible for initializing the following in the Intel TXT Heap memory (this initialization must be completed before executing GETSEC[SENTER]):

- Initialize contents of the Intel TXT Heap Memory (see Appendix C)
- Initialize contents of the OsMleData (see Appendix C) and OsMleDataSize (with the size of the OsMleData field + 8H) fields.
- Initialize contents of the OsSinitData (see section C.3) and OsSinitDataSize (with the size of the OsSinitData field + 8H) fields.

The OsMleData structure has fields for specifying regions of memory to protect from DMA (PMR Low/High Base/Size) using Intel VT-d. As described in Chapter 1.10, the MLE must be protected from DMA by being contained within either the DMA Protected Range (DPR) or one of the Intel VT-d Protected Memory Regions (PMRs). If the MLE resides within the DPR then the PMR fields of the OsMleData structure may be set to 0. Otherwise, these fields must specify a region that contains the MLE and the page tables. However, the PMR fields can specify a larger region (and separate region, since there are two ranges) than just the MLE if there is additional data that should be protected.

If the system software is using Intel VT-d DMA remapping to protect areas of memory from DMA then it must disable this before it executes GETSEC[SENTER]. In order to do this securely, system software should determine what PMR range(s) are necessary to cover all of the address range being DMA protected using the remapping tables. It should then initialize the PMR(s) appropriately and enable them before disabling remapping. The PMR values it provides in the OsSinitData PMR fields must correspond to the values that it has programmed. Once the MLE has control, it can re-enable remapping using the previous tables (after validating them).

If the MLE or subsequent code will be enabling Intel VT-d DMA remapping then the DMAR information that will be needed should be protected from malicious DMA activity until the remapping tables can be established to protect it. The SINIT AC module makes a copy of the DMAR tables in the SinitMleData region (located at an offset specified by the SinitVtdDmarTable field). Because this region is within the TXT heap, it is protected from DMA by the DPR. If the MLE or subsequent code does not use this copy of the DMAR tables, then it should protect the original tables (within the ACPI area) with the PMR range specified to SINIT. Likewise, the memory range used for the remapping tables should also be protected with the PMRs until remapping is enabled.

If the Platform Owner TXT Launch Control Policy (see Chapter 3.1.1.1 for more details about TXT Launch Control Policy) is of type POLTYPE_LIST (or POLTYPE_UNSIGNED for v1 policies) then there must be an associated data file which contains the remainder of the policy. This policy data file must be placed in memory by system software and its starting address and size specified in the LCP PO Base and LCP PO

Size fields of the OsSinitData structure. The data must be wholly contained within a DMA protected region of memory, either within the DPR (e.g. in the TXT heap) or within the bounds specified for the PMRs.

2.2.4.3 Rendezvousing Processors and Saving State

Line 1: If launching the measured environment after operating system boot, then all processors should be brought to a rendezvous point before executing GETSEC[SENTER]. At the rendezvous point each processor will set up for GETSEC[SENTER] and save any state needed to resume after the measured launch. If processors are not rendezvoused before executing SENTER, then the processors that did not rendezvous will lose their current operating state including possibly the fact that an in-service interrupt has not been acknowledged.

Lines 2 – 7: All processors check that they support SMX and enable SMX in CR4.SMXE.

Lines 8 - 10: The MLE should preserve machine check status registers if bit 6 in the TXT Extension Flags returned by GETSEC[PARAMETERS] (See 2.2.5.1for details) is set. If this bit returns 0 or parameter type '5' is not supported, the MLE must log and clear machine check status registers.

Line 11: Check that certain CR0 bits are in the required state for a successful measured environment launch.

Line 12: System software allocates memory to save its state for restoration post measured launch. The OsMleData portion of the Intel TXT Heap has been reserved for this purpose (see section C.1), though the size must be set appropriately for the memory to be available.

Line 13: The ILP saves enough state in memory to allow a return to OS execution after the measured launch and then continues launch execution. The RLPs save enough state in memory to allow return to OS execution after measured launch then execute HLT or spin waiting for transition to the measured environment.

Certain MSRs are modified by executing the GETSEC[SENTER] instruction. For example, bits within the IA32_MISC_ENABLE and IA32_DEBUGCTL MSRs are set to predetermined values. It may be desirable to restore certain bits within these MSRs to their pre-launch state after the MLE launch. If this is desired, then before executing GETSEC[SENTER], software should save the contents of these MSRs in the OsMleData area. The launched software can restore the original values into these MSRs after the GETSEC[SENTER] returns or, alternatively, the MLE can restore these MSRs with their original values during MLE initialization.

It is expected that most MLEs will want to restore the MTRR and IA32_MISC_ENABLE MSR states after the MLE launch, to provide optimal performance of the system.

Listing 5. Pseudocode for Rendezvousing Processors and Saving State

```
1. Rendezvous all processors;

//
// The following code is run on all processors
//
// Enable SMX
//

2. CPUID(EAX=1);
3. IF (SMX not supported) OR (VMX not supported) {
4.    Fail measured environment startup;
5. } ELSE {
6.    CR4.SMXE = 1;
7. }

8. IF (GETSEC[PARAMETERS](type=5)[6] == 1) {
9.     Clear Machine Check Status Registers;
10.  }
11.  Ensure CR0.CD=0, CR0.NW=0, and CR0.NE=1;

//
// Save current system software state in Intel TXT Heap
//

12.  Allocate memory for OsMleData;
13.  Fill in OsMleData with system software state (including MTRR
   and IA32_MISC_ENABLE MSR states);
```

2.2.5 Performing a Measured Launch

2.2.5.1 MTRR Setup Prior to GETSEC[SENTER] Execution

System software must set up the variable range MTRRs to map all of memory (except the region containing the SINIT AC module) to one of the supported memory types as returned by GETSEC[PARAMETERS], before executing GETSEC[SENTER]. System software first saves the current MTRR settings in the OsMleData area and verifies that the default memory type is one of the types returned by GETSEC[PARAMETERS] (default memory type is specified in the IA32_MTRR_DEF_TYPE MSR). Next the variable range MTRRs are set to map the SINIT AC module as WB. The SINIT AC module must be covered by the MTRRs such that no more than (4K-1) bytes after the module are mapped WB. For example, if an SINIT AC module is 11K bytes in size, an 8K and a 4K or three 4K MTRRs should be used to map it, not a single 32K MTRR. Any unused variable range MTRRs should have their valid bit cleared. The mask MTRRs covering the SINIT AC module should not set any bits beyond bit 35 (which corresponds to a 36 bit physical address space) regardless of the value of the MAXPHYADDR MSR, otherwise SINIT will treat this as an error condition.

Listing 6 shows the pseudocode for correctly setting the ILP and RLP MTRRs. This code follows the recommendation in the IA-32 Software Developer's Manual.

After MTRR setup is complete, the RLPs mask interrupts (by executing CLI), signal the ILP that they have interrupts masked, and execute halt. Before executing GETSEC[SENTER], the ILP waits for all RLPs to indicate that they have disabled their interrupts. If the ILP executed a GETSEC[SENTER] while an RLP was servicing an interrupt, the interrupt servicing would not complete, possibly leaving the interrupting device unable to generate further interrupts.

Listing 6. MTRR Setup Pseudocode

```
//
// Pre-MTRR change
//

1. Disable interrupts (via CLI);
2. Wait for all processors to reach this point;
3. Disable and flush caches (i.e. CRO.CD=1, CR0.NW=0, WBINVD);
4. Save CR4
5. IF (CR4.PGE == 1) {
6.     Clear CR4.PGE
7. }
8. Flush TLBs
9. Disable all MTRRs (i.e. IA32_MTRR_DEF_TYPE.e=0)

//
// Use MTRRs to map SINIT memory region as WB, all other regions
// are mapped to a value reported supportable by
// GETSEC[PARAMETERS]
//

10.  Set default memory type (IA32_MTRR_DEF_TYPE.type) to one
   reported by GETSEC[PARAMETERS];
11.  Disable all fixed MTRRs (IA32_MTRR_DEF_TYPE.fe=0);
12.  Disable all variable MTRRs (clear valid bit);
13.  Read SINIT size from the SINIT AC header;
14.  Program variable MTRRs to cover the AC execution region,
   memtype=WB (re-enable each one used);

//
// Post-MTRR changes
//
15.  Flush caches (WBINVD);
16.  Flush TLBs;
17.  Enable MTRRs (i.e. MTRRdefType.e=1);
18.  Enable caches (i.e. CRO.CD=0, CR0.NW=0);
19.  Restore CR4;
20.  Wait for all processors to reach this point;
21.  Enable interrupts;


//
// RLPs stop here
//
```

```
22.  IF (IA32_APIC_BASE.BSP != 1) {
23.      CLI;
24.      set bit indicating we have interrupts disabled;
25.      HLT;
26.  }

27.  Wait for all RLPs to signal that they have their interrupts
   disabled
```

2.2.5.2 Selection of Launch Capabilities

System software must select the capabilities that it wishes to use for the launch. It must choose a subset of the capabilities supported by the SINIT AC module. For mandatory capabilities, such as the RLP wakeup mechanism, one of the supported options must be chosen.

```
28.  OsSinitData.Capabilities = selected capabilities;
```

2.2.5.3 TPM Preparation

System software must ensure that the TPM is ready to accept commands and that there is no currently active locality (TPM.ACCESS_x.activeLocality bit is clear for all localities) before executing the GETSEC[SENTER] instruction.

```
29.  Read TPM Status Register until it is ready to accept a
   command
32.For all localities x, ensure that ACCESS_x.activeLocality is 0
```

2.2.5.4 Intel® Trusted Execution Technology Launch

The ILP is now ready to launch the measuring process. System software executes the GETSEC[SENTER] instruction. See chapter 6, “Safer Mode Extensions Reference”, in the *Intel 64 and IA-32 Software Developer Manuals, Volume 2B* for the details of GETSEC[SENTER] operation.

```
30.  EBX = Physical Base Address of SINIT AC Module
31.  ECX = size of the SINIT AC Module in bytes
32.  EDX = 0
33.  GETSEC[SENTER]
```

2.3 MLE Initialization

This section describes the initialization of the MLE. Listing 7 shows the pseudocode for MLE initialization.

The MLE initialization code is executed on the ILP when the SINIT AC module executes the GETSEC[EXITAC] instruction—the MLE initialization code is the first MLE code to run after GETSEC[SENTER] and within the measured environment. The SINIT AC module obtains the MLE initialization code entry point from the EntryPoint field in the MLE Header data structure whose address is specified in the OsSinitData entry in the Intel TXT Heap. The MLE initialization code is responsible for setting up the protections necessary to safely launch any additional environments or software. The initialization includes Intel TXT hardware initialization, waking and initializing the RLPs, MLE software initialization and initialization of the STM (if one is being used). Section 2.3 describes the details of MLE initialization.

During MLE initialization, the ILP executes the GETSEC[WAKEUP] instruction, bringing all the RLPs into the MLE initialization code. Each RLP gets its initial state from the MLE JOIN data structure (see the *Intel 64 and IA-32 Software Developer Manual, Volume 2B,* Table 6-11). The ILP sets up the MLE JOIN data structure and loads its physical address in the TXT.MLE.JOIN register prior to executing GETSEC[WAKEUP]. Generally the RLP initialization code will be very similar to the ILP initialization code.

If the MLE restores any state from the environment of the launching system software then it must first validate this state before committing it. This is because state from the environment prior to the GETSEC[SENTER] instruction is not considered trustworthy and could lead to loss of MLE integrity.

Lines 1 – 8: The MLE loads CR3 with the MLE page directory physical address and enables paging. The SINIT AC module has just transferred control to the MLE with paging off, now the MLE must setup its own environment. The MLE's GDT is loaded at line 3 and the MLE does a far jump to load the correct MLE CS and cause a fetch of the MLE descriptor from the GDT. At line 5 a stack is setup for the MLE initialization routine and, at line 6, the MLE segment registers are loaded. Next the MLE loads its IDT and initializes the exception handlers.

All of the instructions and data that are used before paging is enabled must reside on the same physical page as the MLE entry point and must access data with relative addressing. This is because the page tables may have been subverted by untrusted code prior to launch and so the MLE entry point's page may have been copied to a different physical address than the original. The MLE must also verify that this page is identity mapped prior to enabling paging (to ensure that the linear address of the instruction following enabling of paging is the same as its physical address).

If the MLE cannot guarantee that it was loaded at a fixed address, then it must create the identity mapping dynamically. Because the physical address of the identity page could overlap with the virtual address range that the MLE wants to use in its page tables, the MLE may need to create a trampoline page table. In such a case, the trampoline page table would consist of an identity-mapped page for the physical address of the MLE entry point and a virtual address mapping of that page which is guaranteed not to be within the desired address range (i.e. a trampoline page). That virtual address mapping would also need to be added to the page table that the MLE ultimately wants to run on. In this way the MLE can enable paging to the trampoline page table (at the identity mapped address) and then jump to the trampoline page's address and then switch page tables (CR3's) to the final table where it will begin executing at the virtual address of the trampoline page but in the final page table.

If the MLE does not enable paging then it must also validate that the physical addresses specified in the page table used for the launch are the expected ones. And as above, it must do this in code that resides on the same physical page as the MLE entry point and uses only relative addressing. The reason for this validation is that the page table could have been altered to place the MLE pages at different physical addresses than expected, without having altered the MLE measurement.

Because the MLE page table that was used for measurement does not contain pages other than those belonging to the MLE, if it wishes to continue to run in a paged environment it will need to either extend the page tables to map the additional address space needed (e.g. TXT configuration space, etc.) or to create new page tables. This should be done after it has finished establishing a safe environment. The cacheability requirements for the address space of any MLE-established page tables must follow the guidelines below.

Line 9: The MLE checks the MTRRs which were saved in the OsMleData area of the Intel TXT Heap (see Appendix C). It looks for overlapping regions with invalid memory type combinations and variable MTRRs describing non-contiguous memory regions. If either of these checks fails the MLE should fail the measured launch or correct the failure.

Before the original MTRRs are restored, the MLE must ensure that all its own pages will be mapped with defined memory types once the variable MTRRs are enabled. The MLE must ensure that the combined memory type as specified by the page table entry and variable MTRRs results in a defined memory type.

The MLE must also ensure that the TXT Device Space (0xFED20000 – 0xFED4FFFF) is mapped as UC so that accesses to these addresses will be properly handled by the chipset.

Line 10: The MLE should check that the system memory map that it will use is consistent with the memory regions and types as specified in the Memory Descriptor Records (MDRs) returned in the SinitMleData structure. Alternately, the MLE may use this table as its map of system memory. This check is necessary as the system memory map is most likely generated by untrusted software and so could contain regions that, if used for trusted code or secrets, might lead to compromise of that data. If the MLE will be using PCI Express* devices, it should verify that it is accessing their configuration space through the address range specified by the PCIE MDR type (3).

Line 11: The MLE should also verify that the Intel VT-d PMR settings that were used by SINIT to program the Intel VT-d engines, as specified in the OsSinitData structure, contain the expected values. While the MLE can only be launched if the settings cover itself and its page tables (or the pages fall within the DPR), settings beyond these regions could have been subverted by untrusted code prior to the launch.

Line 12: The ILP must re-enable SMIs which were disabled as part of the SENTER process; most systems will not function properly if SMIs are disabled for any length of time. It is recommended that the MLE enable SMIs on the ILP before enabling them on the RLPs, since some BIOS SMI handlers may hang if they receive an SMI on an AP and cannot generate one on the BSP to rendezvous all threads. Newer CPUs may automatically enable SMIs on entry to the MLE; for such CPUs there is no harm in executing GETSEC[SMCTRL].

Lines 13 – 17: If this is the ILP then the MLE does the one-time initialization, builds the MLE JOIN data structure and wakes the RLPs. This structure contains the physical addresses of the MLE entry point and the MLE GDT, along with the MLE GDT size and must be located in the lower 4GB of memory. The ILP writes the physical address of this structure to the TXT.MLE.JOIN register. An RLP will read the startup information from the MLE JOIN data structure when it is awakened. The MLE writer should ensure that the MLE JOIN data structure does not cross a page boundary between two non-contiguous pages. The MLE image must be built or loaded such that its GDT is located on a single page. Enough of the RLP entry point code must be on a single page to allow the RLPs to enable paging.

Lines 18 – 27: The MLE must look at the OsSinitData.Capabilities field to see which RLP wakeup mechanism was chosen by the pre-SENTER code and thus used by SINIT. If the MLE wants to enforce that certain capabilities or wakeup mechanism was used then it can choose to error if it finds that not to be the case. For future compatibility, MLEs should support both RLP wakeup mechanisms.

Line 30: The MLE checks several items to ensure they are consistent across all processors:

- All processors must have consistent SMM Monitor Control MSR settings. The processors must all be opt-in and have the same MSEG region or the processors must be all opt-out.
- Ensure all processors have compatible VMX features. The compatible VMX features will depend on the specific MLE implementation. For example, some implementation may require all processors support Virtual Interrupt Pending.
- Ensure all processors have compatible feature sets. Some MLE implementations may depend on certain feature being available on all processors. For example, some MLE implementation may depend on all processors supporting SSE2.

  If the MLE will use VMX then it should verify that bit 1 (VMX in SMX operation) in the IA32_FEATURE_CONTROL MSR is set. Bit 2 (VMX outside SMX operation) may also be set depending on the BIOS being used and on whether TXT has been enabled.
- Ensure all processors have a valid microcode patch loaded or all processors have the same microcode patch loaded. This check will depend on the specific MLE implementation. Some MLE implementations may require the same patch be loaded on all processors, other MLE implementations may contain a microcode patch revocation list and require all processors have a microcode patch loaded which is not on the revocation list.

Line 31: The MLE must wakeup the RLPs while the memory type for the SINIT AC module region is writeback. This is a requirement of the MONITOR mechanism for RLP wakeup. Since this is not guaranteed to be true of the original MTRRs, it is safest to wait until after the RLPs have been awakened before restoring the MTRRs to their pre-SENTER values. Alternatively, the MLE could ensure that this is the case and adjust the MTRRs if it is not. It could then restore the MTRRs before waking the RLPs. In either case, when restoring the MTRRs they should be made the same for each processor.

Line 32: The MLE should restore the IA32_MISC_ENABLE MSR to the value saved in the OsMleData structure. This MSR was set to predefined values as part of SENTER in order to provide a more consistent environment to the authenticated code module. Most MLEs should be able to safely restore the previous value without any need to verify it. The MLE should wait until the RLPs are awakened before restoring the MSR in case the original MSR did not have the Enable MONITOR FSM bit (18) set. See Appendix F for the processor state of the ILP after SENTER and the states of the RLPs after waking.

Line 33: The machine-check exceptions flag (CR4.MCE) is cleared by the GETSEC[SENTER] instruction. If the MLE supports the machine-check architecture then it should initialize the exception mechanism and enable exception reporting.

Line 34: The MLE enables SMIs on each RLP.

Line 35: The MLE enables VMX in the CR4 register. This is required before any VMX instruction can be executed.

Line 36: The MLE allocates and sets up the root controlling VMCS then executes VMXON, enabling VMX root operation.

Lines 37 – 41: The MLE sets up the guest VM. At line 37 the MLE allocates memory for the guest VMCS. This memory must be 4K byte aligned. The MLE executes VMCLEAR with a pointer to this VMCS in order to mark this VMCS clear and allow a VMLAUNCH of the guest VM. At line 39 the MLE executes VMPTRLD so that it can initialize the VMCS at line 40. Now at line 41 the guest VM is launched for the first time.

Note: On the last extend of the TPM by the SINIT AC module, it may not wait to see if the command is complete – so the MLE needs to make sure that the TPM is ready before using it.

Listing 7: MLE Initialization Pseudocode

```
//
// MLE entry point – ILP and RLP(s) enter here
//

1. Load CR3 with MLE page table pointer (OsSinitData.MLE
   PageTableBase);
2. Enable paging;
3. Load the GDTR with the linear address of MLE GDT;
4. Long jump to force reload the new CS;
5. Load MLE SS, ESP;
6. Load MLE DS, ES, FS, GS;

7. Load the IDTR with the linear address of MLE IDT;
8. Initialize exception handlers;

//
// Validate state
//
9. Check MTRR settings from OsMleData area;
10. Validate system memory map against MDRs
11. Validate VT-d PMR settings against expected values

//
// Enable SMIs
//
12. execute GETSEC[SMCTRL];

//
// Wake RLPs
//
13. IF (ILP) {
14.     Initialize memory protection and other data
   structures;
15.     Build JOIN structure;
16.     TXT.MLE.JOIN = physical address of JOIN structure;
17.     IF (RLP exist) {
18.         IF (OsSinitData.Capabilities is set to MONITOR
   wakeup mechanism) {
19.             SinitMleData.RlpWakeupAddr = 1;
20.         }
21.         ELSE IF (OsSinitData.Capabilities is set to GETSEC
   wakeup mechanism) {
22.             GETSEC[WAKEUP];
23.         }
24.         ELSE {
25.             Fail: Unknown RLP wakeup mechanism;
26.         }
27.     }
28. }
```

```
29. Wait for all processors to reach this point;
30. Do consistency checks across processors;
31. Restore MTRR settings on all processors;
32. Restore IA32_MISC_ENABLE MSR from OsMleData
33. Enable machine-check exception handling
34. RLPs execute GETSEC[SMCTRL]


//
// Enable VMX
//
35. CR4.VMXE = 1;

//
// Start VMX operation
//
36. Allocate and setup the root controlling VMCS, execute
   VMXON(root controlling VMCS);

//
// Set up the guest container
//
37. Allocate memory for and setup guest VMCS;
38. VMCLEAR guest VMCS;
39. VMPTRLD guest VMCS;
40. Initialize guest VMCS from OsMleData area;

//
// All processors launch back into guest
//
41. VMLAUNCH guest;
```

2.4 MLE Operation

The dispatch routine is responsible for handling all VMExits from the guest. The guest VMExits are caused by various situations, operations or events occurring in the guest. The dispatch routine must handle each VMExit appropriately to maintain the measured environment. In addition, the dispatch routine may need to save and restore some of processor state not automatically saved or restored during VM transitions. The MLE must also ensure that it has an accurate view of the address space and that it restricts access to certain of the memory regions that the GETSEC[SENTER] process will have enabled. The following subsections describe various key components of the MLE dispatch routine.

2.4.1 Address Space Correctness

It is likely that most MLEs will rely on the e820 memory map to determine which regions of the address space are physical RAM and which of those are usable (e.g. not reserved by BIOS). However, as this table is created by BIOS it is not protected from tampering prior to a measured launch. An MLE, therefore, cannot rely on it to contain an accurate view of physical memory.

After a measured launch, SINIT will provide the MLE with an accurate list of the actual RAM regions as part of the SinitMleData structure of the Intel TXT Heap (see section C.4). The SinitMDR field of this data structure specifies the regions of physical memory that are valid for use by the MLE. This data structure can also be used to accurately determine SMRAM and PCIe extended configuration space, if the MLE handles these specifically.

2.4.2 Address Space Integrity

There are several regions of the address space (both physical RAM and Intel TXT chipset regions) that have special uses for Intel TXT. Some of these should be reserved for the MLE and some can be exposed to one or more guests/VMs.

2.4.3 Physical RAM Regions

There are two regions of physical RAM that are used by Intel TXT and are reserved by BIOS prior to the MLE launch. These are the SINIT AC module region and the Intel TXT Heap. Each region's base address and size are specified by Intel TXT configuration registers (e.g. TXT.SINIT.BASE and TXT.SINIT.SIZE).

The SINIT and Intel TXT Heap regions are only required for measured launch and may be used for other purposes afterwards. However, if the measured environment must be re-launched (e.g. after resuming from the S3 state), the MLE may wish to reserve and protect these regions.

2.4.4 Intel® Trusted Execution Technology Chipset Regions

There are two Intel TXT chipset regions: Intel TXT configuration register space and Intel TXT Device Space. These regions are described in Appendix B.

2.4.4.1 Intel® Trusted Execution Technology Configuration Space

The configuration register space is divided into public and private regions. The public region generally provides read only access to configuration registers and the MLE may choose to allow access to this region by guests. The private region allows write access, including to the various command registers. This region should be reserved to the MLE to ensure proper operation of the measured environment.

2.4.4.2 Intel® Trusted Execution Technology Device Space

The Intel TXT Device Space supports access to TPM localities. Localities three and four are not usable by the MLE even after the measured environment has been established, and so do not need any special treatment. Locality two is unlocked when the Intel TXT private configuration space is opened during the launch process. Locality one is not usable unless it has been explicitly unlocked (via the TXT.CMD.OPEN.LOCALITY1 command). If the MLE wants to reserve access to locality two for itself then it needs to ensure that guest/VM access to these regions behaves as an LPC abort, as defined by TCG for non-accessible localities. This behavior is that memory reads return FFh and writes are discarded. The MLE can provide this behavior by trapping guest/VM accesses to the regions and emulating the defined behavior. Instead, it could map these regions onto one of the hardware-reserved localities (three or four) and let the hardware provide the defined behavior. If the MLE does not need access to locality 2 then it can simply close this locality (TXT.CMD.CLOSE.LOCALITY2) so that neither itself nor guests will have access to it.

2.4.5 Device Assignment

If the MLE exposes devices to untrusted VMs, it must take care to completely protect itself from any affects (either intentional or otherwise) of these devices. For devices which use DMA to access memory, the MLE can protect itself through the use of Intel® VT for Directed IO (Intel® VT-d) to prevent unwanted access to memory and through VMX to manage access to device configuration space. For other types of devices, their interactions with, and affects on, the system should be fully understood before allowing an untrusted VM to access them.

2.4.6 Protecting Secrets

If there will be data in memory whose confidentiality must be maintained, then the MLE should set the Intel TXT secrets flag so that the Intel TXT hardware will maintain protections even if the measured environment is lost before performing a shutdown (e.g. hardware reset). This can be done by writing to the TXT.CMD.SECRETS configuration register. The teardown process will clear this flag once it has scrubbed memory and removed any confidential data.

2.4.7 Machine Specific Register Handling

Model Specific Registers (MSRs) pose challenges for a measured environment. Certain MSRs may directly leak information from one guest to another. For example, the Extended Machine Check State registers may contain secrets at the time a machine check is taken. Other MSRs might be used to indirectly probe trusted code. The Performance Counter MSRs, for example, could be used by the non-trusted guest to determine secrets (e.g. keys) used by the trusted code. Other MSRs can modify the MLE's operation and destroy the integrity of the measured environment.

The VMX architecture allows the MLE to trap all guest MSR accesses. Certain VMX implementations will also allow the MLE to use a bitmap to selectively trap MSR accesses. The MLE must use these VMX features to check certain guest MSR accesses, ensuring that no secrets are leaked and that MLE operation is not compromised.

An MLE might virtualize some of the MSRs. The VMX architecture provides a mechanism to automatically save selected guest MSRs and load selected MLE MSRs on VMEXIT. Selected guest MSRs may be automatically loaded on VMENTER. These features allow the MLE to virtualize MSRs, keeping a separate MSR copy for the guest and MLE. Note that using this feature will slow VMEXIT and VMENTER times. The VMX architecture provides a separate set of VMCS registers for the automatic saving and restoring of the fast system call MSRs.

There is a limit to the number of MSRs which can be swapped during a VMX transition. Bits 27:25 of the VMX_BASE_MSR+5 indicate the recommended maximum number of MSRs that can be saved or loaded in VMX transition MSR-load or MSR-store lists. Specifically, if the value of these bits is N, then 512 * (N + 1) is the recommended maximum number of MSRs referenced in each list. If the limit is exceeded, undefined processor behavior may result (including a machine check during the VMX transition).

There are certain MSRs which cannot be included in the MSR-load or MSR-store lists. In the initial VMX implementations, IA32_BIOS_UPDT_TRIG and IA32_BIOS-SIGN_ID may not be loaded as part of a VM-Entry or VM-Exit. The list of MSRs that cannot be loaded in VMX transitions is implementation specific.

The MLE must contain a built-in policy for handling guest MSR accesses. This MSR handling policy must deal with all architectural MSRs that might be accessed by guest code. The built-in MSR policy must deny access to all non-architectural MSRs.

2.4.8 Interrupts and Exceptions

To preserve the integrity of the measured environment, the MLE must be careful in how it handles exceptions and interrupts. It needs to ensure that its IDT (Interrupt Descriptor Table) has a handler for all exceptions and interrupts. The MLE should also ensure that if it uses interrupts for internal signaling that it does so securely. Likewise, it is best if exception handlers do not try and recover from the exception but instead properly terminate the environment.

2.4.9 ACPI Power Management Support

Certain ACPI power state transitions may remove or cause failure to the Intel TXT protections. The MLE must control such ACPI power state transitions. The following sections describe the various ACPI power state transitions and how the MLE must deal with these state transitions.

2.4.9.1 T-state Transitions

T-states allow reduced processor core temperature through software-controlled clock modulation. T-state transitions do not affect the Intel TXT protections, so the MLE does not need to control T-state transitions. The MLE may wish to control T-state transitions for other purposes, e.g. to enforce its own power management or performance policies.

2.4.9.2 P-State Transitions

P-state transitions allow software to change processor operating voltage and frequency to improve processor utilization and reduce processor power consumption. One some systems, where the processor does not enforce allowed combinations, the MLE must ensure software does not write an invalid combination into the GV3 MSRs.

2.4.9.3 C-State Transitions

C-states allow the processor to enter lower power state. The C0 state is the only C-state where the processor is actually executing code – in the remaining C-states the processor enters a lower power state and does not execute code. In these lower power C-states the Intel TXT protections remain intact; therefore the MLE does not need to monitor or control the C-state transitions. The MLE may wish to control C-state transitions for other purposes, e.g. to enforce its own power management or scheduling policy.

2.4.9.4 S-State Transitions

The S0 state is the system working state – the remaining S-states are low-power, system-wide sleep states. Software transitions from the S0 working state to the other S-states by writing to the PM1 control register (PM1_CNT) in the chipset. Since the Intel TXT protections are removed when the system enters the S3, S4 or S5 states, and the BIOS will gain control of the system on resume from these states, the MLE must remove secrets from memory before allowing the system to enter one of these sleeps states. Note that entering S1 does not remove Intel TXT protections and Intel chipsets do not support the S2 sleep state.

The Intel TXT chipset provides hardware to detect when the software attempts to enter a sleep state while the secrets flag is set (the TXT.SECRETS.STS bit of the TXT.E2STS register). The Intel TXT chipset will reset the system if it detects a write to the PM1_CNT register that will force the system into S3, S4 or S5 while the secrets flag is set. If the Intel TXT chipset does detect this situation and resets the system, then the BIOS AC module will scrub the memory before passing control to the BIOS. To avoid this reset and scrubbing process the MLE should remove secrets from memory and teardown the Intel TXT environment before allowing a transition to S3, S4 or S5.

Before tearing down the Intel TXT environment, the MLE may remove secrets from memory (clearing pages with secrets) or encrypt secrets for later use (e.g. for a later measured environment launch). Once this operation is complete the MLE must issue the TXT.CMD.NO-SECRETS command to clear the secrets flag. After this command is issued, the MLE may allow a transition to a S3, S4 or S5 sleep state. The MLE teardown procedure is described in more detail in Section 2.5.

2.4.9.4.1 S3

The S3 state provides special challenges for the MLE because the resume process uses the in-memory state from when S3 was entered. This means that unlike a normal boot process where trust is established as each component launches, the trust that existed at entry to S3 must be maintained/verified on resume.

Since the TXT environment must be torn down before entering S3, it will have to be re-established on resume. This part of the S3 resume process is nearly identical to the original launch. Because S3 resume should leave the platform in the same state as before S3 was entered, the PCRs should also have the same values. This means that the MLE launched on resume should be the same as the initial one launched, which means that the code/data being measured cannot include any state from before entering S3. If some state from before entering S3 is needed on resume, then it must be validated post-launch (since it is not being measured).

The MLE needs to ensure that the integrity of the TCB will be maintained across the transition. There are two possible sources for loss of integrity across S3: malicious DMA and compromise of the in-memory BIOS image. The initial, pre-S3 TXT launch process protects against DMA and so the S3 resume process should maintain such protection. Most BIOSes will execute portions of their S3 resume code from their in-memory image without first re-copying it from flash. Since this in-memory image could have been modified by any privileged software or firmware that executed as part of the original, pre-S3 boot process (e.g. option ROMs, bootloader, etc.), this too needs to be defended against.

The TXT launch process done as part of S3 resume ensures integrity and DMA protection for the measured part of the MLE itself. For the remainder of the TCB this can be accomplished by creating a memory integrity code for the TCB and sealing it to the MLE's launch-time measurements just before the TXT protections are removed prior to entering S3 (the sealed data creation attributes should include a locality that is only available to the TCB). On resume and after a successful launch, the MLE can re-calculate the value for the memory image and compare it with the sealed value to determine if the memory image has been compromised).

If there were additional measurements extended to the DRTM PCRs as part of the original boot process, these will need to be re-established since these PCRs are cleared when the MLE is re-launched. The entity that makes the measurements should seal the measurement values (not the resulting PCR values) to the PCRs values in effect just before it extends the measurements into the PCRs (the sealed data creation attributes should include a locality that is only available to software trusted by the entity). On resume from S3, when that entity is resumed it will unseal the values and re-extend them into the appropriate PCRs.

It may be necessary for the MLE to seal additional information that is required to securely re-establish the trusted environment. For instance, the portion of the TCB needed to re-establish final DMA protections (e.g. with VT-d DMA remapping) will need to be DMA protected by the MLE as part of the post-resume launch. The MLE may need to save the bounds of this region prior to entering S3 (both in plain text and sealed). It would then use the plain text saved bounds to determine the PMR values to specify as part of the re-launch. Post-launch the MLE would unseal the bounds information and verify it against the bounds specified in the launch.

Because the boot process on resuming from an S3 state does not re-measure the elements of the SRTM, software prior to entering the S3 state must execute the TPM_SaveState command to inform the TPM to preserve the state of its PCRs. Upon resume from S3, the BIOS must provide a flag to TPM_Startup to indicate that the TPM is to restore the saved state. If the TPM's state is not saved prior to entering S3, then the TPM will be non-functional after resuming. Normally an OS TPM driver would perform the TPM_SaveState command when the OS indicated that it was entering S3. However, if the MLE cannot be sure that the environment it establishes will perform this command, it may wish to do so itself prior to entering S3. If the MLE alters the TPM state (e.g. extending to PCRs, etc.) after TPM_SaveState has been performed then the TPM may invalidate the previously saved state. In such cases, the MLE must also perform this command and it should be the last TPM command that is executed, in order to ensure that the state is not changed afterwards. There is no harm if this command is executed multiple times prior to S3.

2.4.10 Processor Capacity Addition (aka CPU Hotplug)

TXT unaware VMMs and OS kernels assume control of application processors during boot using INIT-SIPI-SIPI mechanism. Upon receipt of a SIPI, the processor resumes execution at the specified SIPI vector.

On the other hand, an MLE issues GETSEC[WAKEUP] (or a write to the wakeup address) to assume control of application processors (RLPs) following SENTER. The RLPs begin execution at an address pointed to by the MLE JOIN data structure. TXT for multiprocessors enables processor capacity addition (also known as CPU hotplug or hotadd) after the TXT environment has been launched. Processor capacity addition can be a result of physical addition of a processor package to a running system or on-lining of a processor package that was previously inactive. The TXT processor capacity addition flow makes use of INIT-SIPI-SIPI as the RLP wakeup mechanism, but the hot added logical processors use the MLE JOIN data structure to determine their entry point.

The processor capacity addition flow for an MLE is documented below:

1. New processors are released from reset. They execute measured and verified BIOS code.

2. BIOS configures the new processors. At the end of configuration, BIOS clears the BSP flag in the APICBASE register of new processors and leaves them in a CLI/HLT loop.
3. The MLE is notified of this event via the standard ACPI mechanism.
4. Some MLEs may choose to allow write access to the TXT public configuration region by guests. However, during the processor capacity addition flow, the MLE must prevent guests from writing to the public region in order to prevent them from modifying the TXT.MLE.JOIN register in the middle of this flow.
5. The MLE must prepare the MLE JOIN data structure for the new processors. It may choose to use the same values as it did for the initial RLPs or it may use different ones. In either case, they are subject to the same restrictions as for the initial RLP wakeup. The MLE then writes the physical address of the JOIN data structure into the TXT.MLE.JOIN register.
6. The MLE issues the INIT–SIPI-SIPI sequence to each newly added logical processor to take control of these processors.
7. In response to the SIPI, each new processor will detect that this is a capacity addition to an existing measured environment and resume execution at the entry point specified in the MLE JOIN data structure. The processor will ignore the SIPI vector that may have been supplied. The new processor gets its initial state from the MLE JOIN data structure just like an RLP would during the MLE launch process.

2.5 MLE Teardown

This section describes an orderly measured environment teardown. This occurs when the guest OS or the MLE decides to teardown the measured environment (for example prior to entering an ACPI sleep state such as S3). The listing below shows the pseudocode for teardown of the measured environment.

Line 1: Rendezvous all processors at “exiting Intel TXT environment” point in guest. No need for the guests to save their state as their state will be stored in a VMCS on VMEXIT to the monitor.

Lines 2 and 3: After all processors in the guest rendezvous, all processors execute a VMCALL to the teardown routine in the MLE. Once in the MLE, each processor increments a counter in trusted memory. All processors except the BSP/ILP (the processor with IA32_APIC_BASE MSR.BSP=1) wait on a memory barrier. The ILP waits for all other processors to enter MLE teardown routine then signals the other processors to resume with teardown.

If not all processors reach the rendezvous in the guest, the ILP may timeout and VMCALL to the MLE teardown routine. If not all processors arrive in the MLE teardown routine, the ILP forces all other processors into the MLE with an NMI IPI. Both these conditions are treated as errors – the ILP should proceed with the measured environment teardown but log an error.

At line 4, each processor reads all guest state from its VMCS and stores this data in memory, since after VMXOFF the processors will no longer be able to access data in their VMCS. This state will be needed to restore the guest execution after teardown.

The MLE automatically saves certain guest state (general purpose registers which are not part of the VMCS guest area) on VM Exit. The MLE may need to restore this state when it reenters the guest after the GETSEC[SEXIT].

Line 5: Once all processors are in the MLE and have saved guest state from the VMCS, all processors clear their appropriate registers to remove secrets from these registers.

Lines 6: All processors flush VMCS contents to memory using VMCLEAR. The MLE must flush any VMCS which might contain secrets – this would include all guest VMCSes in a multi-VM environment.

Line 7: The processors wait until all processors have reached this point before resuming execution. This allows all the VMCS flushes to complete before the ILP encrypts or scrubs secrets. Processors should execute an SFENCE to ensure all writes are completed before continuing.

Line 9: The ILP encrypts and stores exposed secrets from all trusted VMs. Note that encrypted secrets will have to be stored in memory until the OS can put them to disk. This will require extra memory above and beyond the memory holding secrets. This step assumes that the RLPs do not have secrets that are not visible to the ILP. Therefore when the ILP scrubs/encrypts all secrets, this will deal with secrets in the RLP caches also.

Line 10: The ILP again clears appropriate registers to remove any secrets from those registers.

Line 11: The ILP scrubs all trusted memory (except the teardown routine itself and encrypted memory). Note that the scrub itself clears secrets still held in the cache.

Line 12: The ILP executes WBINVD to invalidate its caches (to ensure last few pages of zeros actually get to memory).

Lines 13 - 16: If the MLE is going to enter the S3 state, the ILP calculates a memory integrity code and seals it.

Line 17: The ILP caps, or extends, the dynamic PCRs with some value. This prevents an attacker from unsealing the secrets after the teardown using the same PCRs, since the dynamic PCRs are not reset after GETSEC[SEXIT].

Line 18: The ILP writes the NoSecrets in memory command. (TXT.CMD.NO-SECRETS)

Line 19: The ILP should unlock the system memory configuration (TXT.CMD.UNLOCK-MEM-CONFIG) (that was locked by SINIT) once secrets have been removed from memory. This will facilitate re-launching the MLE and may be necessary for a graceful shutdown of the system.

Line 20: The ILP closes Intel TXT private configuration space.

Line 23: The RLPs wait while the ILP encrypts and scrubs secrets from memory.

Line 25: Each processor then disables processor virtualization. If an STM was launched then it must be torn down before VMX is disabled. See section 25.15.7 of the *Intel 64 and IA-32 Software Developer Manual, Volume 3B* for more information.

Lines 26 - 31: The RLPs wait on a memory barrier while the ILP executes the GETSEC[SEXIT] instruction to initiate the teardown of SMX operation.

At end of GETSEC[SEXIT], the ILP simply continues to the next instruction (still running in monitor's context – paging on). The ILP signals the RLPs to continue.

Lines 32 and 33: The former monitor code now restores guest state left behind when the guest executed the VMCALL to enter the MLE teardown routine. All processors perform the transition to guest OS, now operating as normal environment rather than guest.

The guest MSRs must be restored when restarting the guest OS. The MLE can restore the MSRs with information in the VMCS (VM-exit MSR store count) and the VM-exit MSR store area, or the guest OS could save important MSR settings before calling the teardown routine and restore its own MSR settings after resuming after teardown.

If the MLE is going to return control to a designated guest after tearing down then the MLE must ensure that no interrupts are left pending or unserviced before returning control to the designated guest. Any interrupts left pending or unserviced may prevent further interrupt servicing once the designated guest is restarted.

Listing 8. Measured Environment Teardown Pseudocode

```
1. Rendezvous processors in guest OS;
2. All processors VMCALL teardown in MLE;
3. Rendezvous all processors in MLE teardown routine;
4. All processors read guest state from VMCS, store values in
   memory;

//
// Remove and encrypt all secrets from registers and memory
//
5. All processors clear their appropriate registers;
6. All processors flush VMCS contents to memory using VMCLEAR;
7. Wait for all processors to reach this point;
8. IF (ILP) {
9.    Encrypt and store secrets in memory;
10.    Again clear appropriate registers to remove secrets;
11.    Scrub all trusted memory;
12.    WBINVD caches;
13.    IF (S3) {
14.       Create memory integrity code
15.       Seal memory integrity code
16.    }
17.    “cap” dynamic TPM PCRs;
18.    Write to TXT.CMD.NO-SECRETS;
19.    Unlock memory configuration
20.    Close private Intel TXT configuration space;
21.    Signal RLPs that scrub is complete;
22. } ELSE { // RLP
23.    Wait for ILP to signal completion of memory scrub;
24. }

//
// Stop VMX operation
//
25. VMXOFF;


//
```

```
// RLPs wait while ILP executes SEXIT
//
26. IF (ILP) {
27.    GETSEC[SEXIT];
28.    signal completion of SEXIT;
29. } ELSE {
30.    wait for ILP to signal completion of SEXIT;
31. }

//
// Transition back to the guest OS
//

32. Restore guest OS state from device memory;
33. Transition back to guest OS context;
```

2.6 Other Considerations

2.6.1 Saving MSR State across a Measured Launch

Execution of the GETSEC[SENTER] instruction loads certain MSRs with pre-defined values. For example, GETSEC[SENTER] will load IA32_DEBUGCTL MSR with 0H and will load the GV3 MSR with a predetermined value. The software can deal with this in several different ways. The launching software may save the state of these MSRs before measured launch and restore the state after the launch returns. In this case the MLE will need to check the values that are restored. Another approach is to have the launch software save the desired state and have the MLE restore the values before resuming the guest. The software could also leave these MSRs in the state established by GETSEC[SENTER].

The IA32_MISC_ENABLE MSR should be saved and restored around measured launch and teardown.

3 Verifying Measured Launched Environments

Launch Control Policy (LCP) is the verification mechanism for the Intel TXT verified launch process. LCP is used determine whether the current platform configuration or the environment to be launched meets a specified criteria. Policies may be defined by the Platform Owner, and/or, as a default set by the Platform Supplier.

The policy described in this section applies to TXT-capable platforms produced in 2009 and later. A description of the data structures, in a pseudo-code format, can be found in Appendix D.

The policy definition for previous platforms can be found in Appendix D.

3.1 Overview

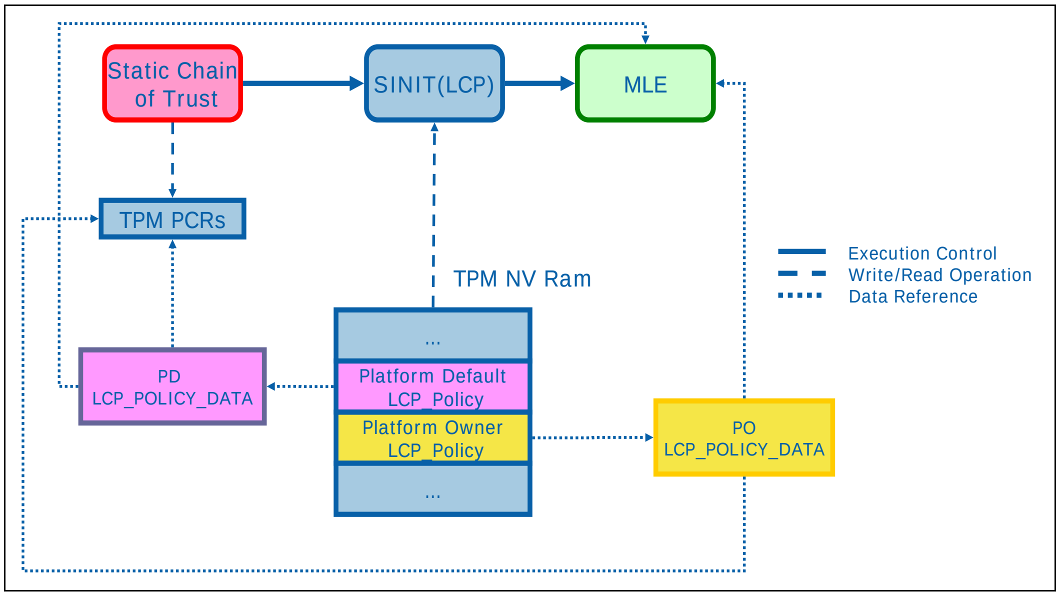
The Launch Control Policy architecture consists of the following components:

- LCP Policy Engine – part of the SINIT ACM and enforces the policies stored on the platform
- LCP Policies – stored in the TPM, they specify the policies SINIT ACM will enforce.
- LCP PolicyData objects – referenced by the Policy structures in the TPM; each contains a list of valid policy elements, such as measurements of MLEs or platform configurations.

Figure 1. Launch Control Policy Components shows how these components relate to each other. The figure also shows that there are two possible policies available on the platform: the Platform Supplier policy, as established by the manufacturer, OEM, ODM, etc., and the Platform Owner's policy.

When the platform boots, its state is measured and recorded by the Static Root of Trust for Measurement (SRTM) and the other components which make up the static chain of trust; these events occur from when the platform is powered on until the Intel TXT measured launch (or until some component breaks the static chain). At this point GETSEC[SENTER] is invoked, and control is passed through the authenticated code execution area to the SINIT authenticated code module. The LCP engine in SINIT reads the LCP Policy Indices in the TPM NV, decides which policy to use, and checks the Platform Configuration and the Measured Launched Environment as require by the chosen policy. The measured environment is then launched.

Figure 1. Launch Control Policy Components



3.1.1 LCP Components

The description of the policy that the SINIT AC module implements, consists of two policies: one policy is set by the Platform Supplier, known as the Platform Supplier (PS), and the other belongs to the Platform Owner (PO). Both policies are stored in the non-volatile store of the Trusted Platform Module (TPM NV). By storing the policy in the TPM NV, access controls can be applied to it; it also enables the policy to persist across platform power cycles.

3.1.1.1 LCP Policy

The *LCP_POLICY* structure (for a full listing see section E.1) is used for both the Platform Supplier and the Platform Owner policies. The size of the structure currently needs to be kept to a minimum in order to preserve the scarce resources of the TPM NV storage, which is why additional structures for both Supplier and Owner policies (*LCP_POLICY_DATA*) that can be persisted elsewhere are provided to handle additional information.

Figure 2. LCP_POLICY Structure

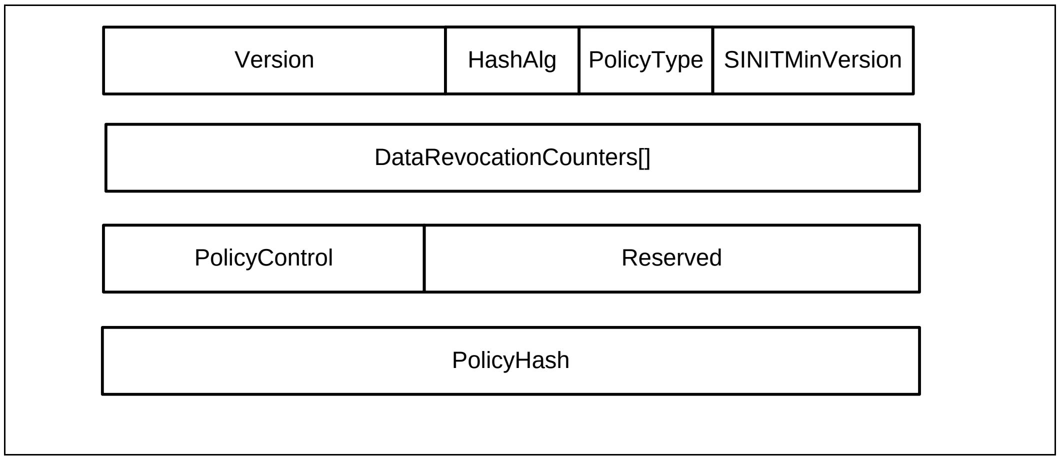


Figure 2 diagrammatically illustrates the LCP_POLICY structure (fields not to scale):

Version specifies the version of the LCP_POLICY structure and, implicitly, of the policy engine semantics. It is of the format <major>.<minor> where the major version is the MSB of the field and the minor version is the LSB. All minor versions of a given major version will be backwards compatible. If new fields are added they will be at the end and the semantics of all previous minor versions are maintained (though they can be extended). Major version are not guaranteed to be backwards compatible with each other and so SINIT will fail to launch if it finds a major version that it is not compatible with. The version of the LCP_POLICY structure defined here is 2.2 (202H).

HashAlg identifies the hashing algorithm used for the *PolicyHash* field. If the algorithm type is not supported by ACM processing the policy, then it shall stop processing the policy and fail.

PolicyType indicates whether an additional LCP_POLICY_DATA structure is required.

- If the *PolicyType* field is *LCP_POLTYPE_ANY* then the value in the *PolicyHash* field is ignored and the environment to be launched is simply measured before execution control is passed to it. No corresponding *LCP_POLICY_DATA* is expected.
- If the *PolicyType* field is *LCP_POLTYPE_LIST* then the value of *PolicyHash* is the result of computing a hash over the LCP_POLICY_DATA structure per the rules below.

If the type specified is not supported by the ACM processing the policy then it shall stop processing the policy and fail.

SINITMinVersion specifies the minimum version of SINIT that can be used. This value corresponds to the AcmVersion field in the AC module Information Table (see Table 5). This value must be less than or equal to the value of AcmVersion in the executing SINIT image for that SINIT to continue; otherwise SINIT will fail the launch. There is no revocation mechanism for other (than SINIT) ACMs. If there is a LCP_MLE_ELEMENT element in a policy then the *SINITMinVersion* in that element will be combined with the value in the LCP_POLICY, per the description in section E.4.1. If there is no LCP_MLE_ELEMENT element in the policy then the value in the LCP_POLICY will be used.

The *DataRevocationCounters* field specifies, for each LCP_POLICY_LIST, the minimum counter value, from the *RevocationCounter* field of that list, which will be accepted. If the value in the *RevocationCounter* field is less then this value then SINIT will fail the launch. For LCP_POLICY_LISTs that are not signed, the corresponding *DataRevocationCounters* index will be ignored. For each LCP_POLICY_LIST that is signed, it must set the *DataRevocationCounters* element at the index corresponding to its own index in *PolicyLists[]*. E.g. if *PolicyLists[0]* is signed, *PolicyLists[1]* unsigned, and *PolicyLists[2]* signed, then the revocation counter for *PolicyLists[0]* will be *DataRevocationCounters[0]*, the counter for *PolicyLists[2]* will be *DataRevocationCounters[2]*, and *DataRevocationCounters[1]* will be ignored. Values in indices greater than the number of lists will be ignored.

The *PolicyControl* field provides a number of control bits which are defined as:

- Bits31:4 Reserved and should be set to zero
- Bit 3 Signifies whether an Owner policy is required by SINIT. Setting it to 1 will cause SINIT to fail the launch if there is no Platform Owner policy. If there are both Supplier and Owner policies, they will be evaluated according to the rules below (this bit has no affect on that). This bit will be ignored in the Platform Owner policy.
- Bit 2 Identifies whether the OsSinitData.Capabilities field will be extended into PCR 17 (if set to 1 then it will be extended).
- Bit 1 Identifies whether the platform will allow AC Modules which have been marked as pre-production to be used to launch the MLE. If this bit is 0 and a pre-production AC Module has been invoked, it will cause a TXT reset during GETSEC[SENTER].
  Note: The use of any pre-production AC Module will result in PCRs 17 and 18 being capped with random values.
- Bit 0 Is reserved and must be set to 0

3.1.1.2 PolicyHash Field for LCP_POLTYPE_LIST

For policies of type LCP_POLTYPE_LIST, the LCP_POLICY_DATA may contain multiple lists, some of which are signed and some which are not. In order to realize the value of signed policies, *PolicyHash* can't be a simple hash over the entire LCP_POLICY_DATA or even changes to signed policy lists would cause a change in the measurement of the policy.

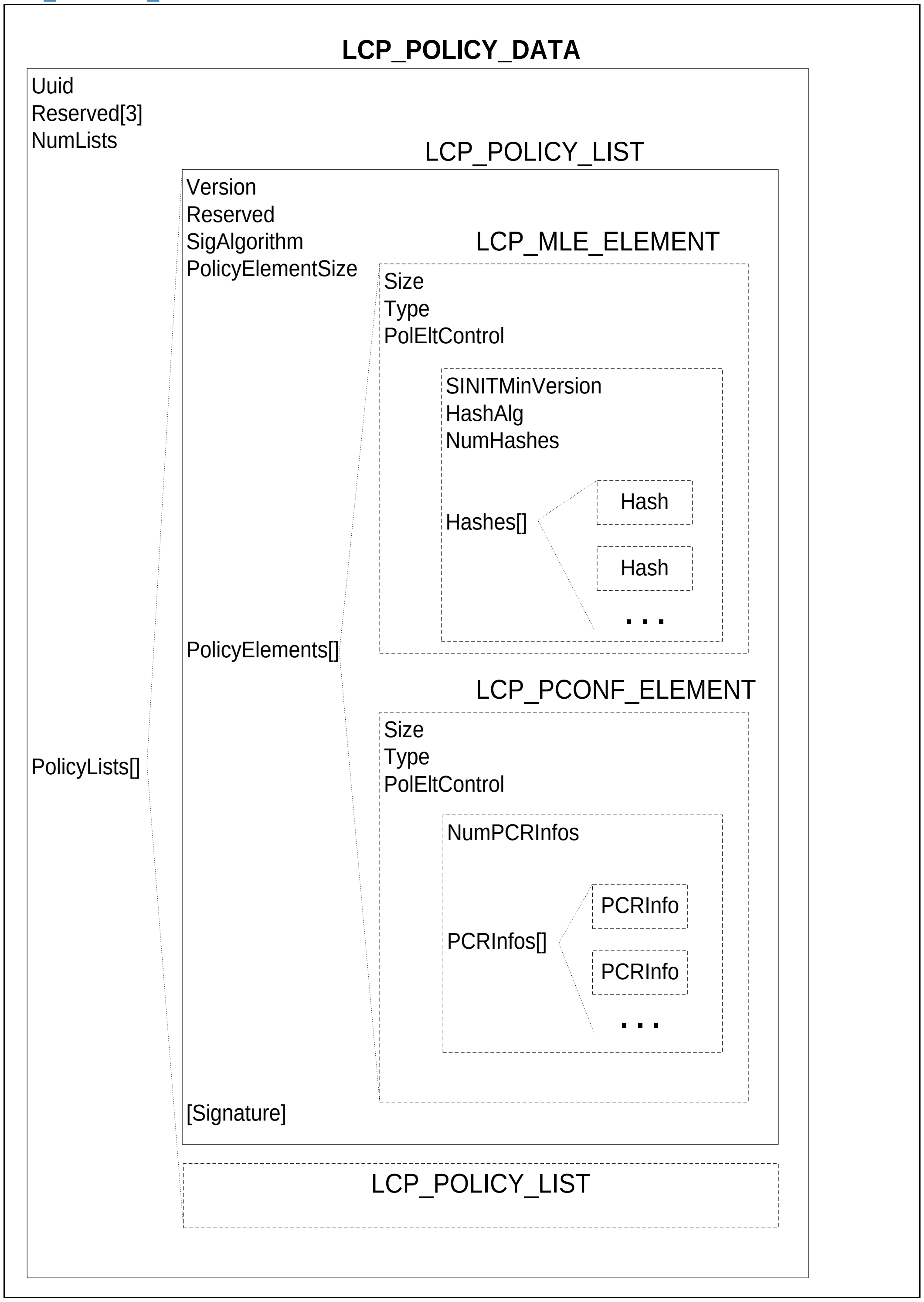
The measurement of a policy list depends on whether the list is signed. For a signed list (LCP_POLSALG_RSA_PKCS_15), the measurement is the hash (as specified by the *HashAlg* field of the corresponding LCP_POLICY) of the public (verification) key in the *PubkeyValue* member of the *Signature* field. For unsigned lists (LCP_POLSALG_NONE), the measurement is the hash (also as specified by the *HashAlg* field of the corresponding LCP_POLICY) of the entire list (LCP_POLICY_LIST).

The value of the *PolicyHash* field will be the hash of all of the policy list measurements concatenated (there is no end padding if the number of lists present is less than LCP_MAX_LISTS). For example, if there is only a single list then the value of *PolicyHash* will be SHA-1(SHA-1(list)).

3.1.1.3 LCP Policy Data

The purpose of the *LCP_POLICY_DATA* structure is to provide the additional data needed to enforce the policy but in a separate entity that doesn't have to consume TPM NV space. A full description of *LCP_POLICY_DATA* can be found in section E.2.

Figure 3. LCP_POLICY_DATA Structure



The *PolicyLists[]* field allows the object to contain a number of lists. A list may be either signed or unsigned and can contain any type of LCP_POLICY_ELEMENT structures (e.g. for MLE policy, platform configuration policy, etc.).

3.1.1.4 LCP Policy Element

Policy elements are the self-describing entities that contain the actual policy conditions. Since they are self-describing, policy engines can ignore the elements that they don't understand or support. This allows for adding new element types without breaking backwards compatibility.

Size is the size (in bytes) of the entire LCP_POLICY_ELEMENT structure, including the type-specific *Data* and the *Size* field itself. A policy engine can use this to skip over an element that it does not understand or support.

The *PolEltControl* provides a number of control bits which are divided into two groups of 16 bits each, one that is specific to the element type and one that applies to all element types and is defined as:

- Bits31:16 Reserved for element type –specific uses and should set to zero
- Bits15:1 Reserved and should set to zero
- Bit 0 If set to 1 specifies that this policy element type in the Owner policy unconditionally overrides (i.e. ignores) any policy elements of the same type in the Supplier policy (when both policies are present). This Supplier policy element type is overridden if this bit is set in any elements of this type in the Owner policy. In order to keep the same behavior as that of the previous version of LCP, and to ensure that if the Supplier policy is of type LCP_POLTYPE_ANY that the Owner policy will have control, the default setting of the bit in Owner policies should be 1. This bit will be ignored in the Platform Supplier policy.

The contents of the *Data[]* field are dependent of the type of the element and are described for each type in the subsections of section E.4.

3.1.2 Signed Policies

The purpose of signed policies is to provide a mechanism that allows policy authors to update the list of permissible environments without having to update the TPM NV (note that if revocation is used that the TPM NV must be updated to increment the revocation counter). This allows updates to be simple file pushes rather than physical or remote platform touches. It also facilitates sealing against the policy, as sealed data does not have to be migrated when the policy is updated.

The use of this mechanism places certain responsibilities on policy authors:

- The private signature key needs to be kept secure and under the control of the key owner at all times.
- The private signature key needs to be strong enough for the full lifetime of the policy [for the Platform Supplier we have estimated up to seven years].

3.1.3 Supported Cryptographic Algorithms

The following algorithms are defined for the current version of the Launch Control Policy:

Hashing – SHA-1

Signature – RSA PKCS V1.5

It is the responsibility of the policy author to ensure that their policy uses an algorithm supported by the version of the AC module being used. If the policy specifies an unsupported algorithm, the policy will fail and, depending on the ACM evaluating the policy, the environment will not be permitted to launch or the processor will not boot.

3.2 Policy Engine Logic

3.2.1 Policies

Before evaluating a policy, the policy engine must first verify the policy's integrity. For policy of type LCP_POLTYPE_LIST, the engine must verify each LCP_POLICY_LIST in the LCP_POLICY_DATA. In the case that a list is signed, that means that the signature must be verified. For an unsigned list, the hash of the list must be calculated. The hash of the LCP_POLICY_DATA structure, as calculated per section 3.1.1.2 above, is then compared with the hash in the LCP_POLICY that was read from the TPM NVRAM.

The policy engine must scan the policy for each policy element that it supports. When a policy contains multiple lists in its LCP_POLICY_DATA, the policy engine will evaluate each list sequentially. As soon as it finds a match that satisfies the policy element being evaluated (e.g. MLE, platform configuration, etc.) it will stop evaluating further elements and lists.

For a policy of type LCP_POLTYPE_ANY, the policy engine will treat that policy as successfully evaluating every policy element type.

For a policy of type LCP_POLTYPE_LIST, for every policy element type supported by the ACM evaluating the policy that is present in any of the lists, at least one instance of that element type must evaluate successfully in order for the policy to succeed. If a particular policy element type is not in any of the lists then that condition is not evaluated and any state is accepted. For instance:

> If SINIT is processing a policy that contains two lists, the first containing only an LCP_MLE_ELEMENT and the second containing only a LCP_PCONF_ELEMENT, then the MLE being launched must appear in the first list's LCP_MLE_ELEMENT and the current platform configuration must satisfy the second list's LCP_PCONF_ELEMENT; otherwise the launch will fail.
>
> If SINIT is processing a policy that contains two lists, each containing only an LCP_MLE_ELEMENT element, then the MLE being launched must appear in at least one list's LCP_MLE_ELEMENT. Since no other element types are present, any other platform condition or state is acceptable (e.g. any PCR values).

3.2.2 Combining Policies

When both the Platform Supplier and the Platform Owner have established policies on the platform, the two policies are combined to give a resultant policy which the LCP policy engine will enforce.

For every policy element that the policy engine evaluates, the policy engine will consider the evaluation successful if either the Supplier or Owner policy evaluates that element successfully (evaluated per the rules above), unless *any* of the Owner policy element instances has set bit 0 of the *PolEltControl* field of its element instance. If bit 0 is set then only the Owner policy for that policy element will be evaluated. This permits the Platform Owner to prevent rollback of signed Platform Supplier policies and also allows the Platform Owner final control of the platform's policy. As such, policy engines should evaluate the Owner policy first, if it exists.

For policy elements that only appear in one policy, that policy will be used.

If there is no Owner policy then only the Supplier policy is evaluated and it is evaluated per the rules of section 3.2.1. However, the policy engine must permit any MLE to launch, regardless of whether it is in the Supplier policy. This is to permit the Platform Supplier to include an LCP_MLE_ELEMENT (in order to allow launching of an MLE provided by the Platform Supplier even if an Owner policy did not include that MLE on its own list) but still permit the Platform Owner to launch any MLE without having to provision an Owner policy. This preserves the same Owner-visible behavior as the case where the Platform Supplier did not provide an LCP_MLE_ELEMENT. Note that all other element types are evaluated as expected.

The following table shows which policy will be evaluated for all combinations of policies and types. This is not the same as which policy is actually executed, since a policy may not contain any elements which are understood by the policy engine and thus that policy would not actually be executed (same as None).

	No PS	**PS type ANY**	**PS type LIST**
No PO	None[1]	PS	PS[2]
PO type ANY	PO	PO	PO
PO type LIST	PO	PO	Both[3]

[1] If no policy is evaluated then all platform and software configurations are permitted.
[2] Per above, any MLE will be allowed to launch regardless of the Supplier policy.
[3] This is effectively a union of the two policies (caveat bit 0 of *PolEltControl*).

As can be seen from this table, if the Owner has installed a policy of type ANY then it will override all policy elements in the Supplier policy, including any PCONF or SBIOS elements intended to restrict the BIOS. If the owner is only interested in making sure that any MLE can launch, then there is no need for a PO policy, as that is already the behavior (see above).

3.3 Measuring the Enforced Policy

The LCP engine in SINIT will extend to PCR 17 a hash value which represents the policy(ies) against which the environment was launched. This hash value is determined by the rules in the following sections.

It is important that for all policy cases that a measurement will always be extended to PCR 17 in order to prevent the MLE from later extending a value of a policy that was not evaluated. This is an issue because PCR 17 is open to locality 2 extends and the MLE executes with locality 2 access. If this were not done, such an MLE could get access to data which were sealed against some known policy by another MLE.

As a matter of integrity, the *LCP_POLICY::PolicyControl* field will always be extended into PCR 17. If an Owner policy exists, its *PolicyControl* field will be extended; otherwise the Supplier policy's will be. If there are no policies, 32 bits of 0s will be extended.

Other ACMs' policy engines do not extend to the DRTM PCRs.

3.3.1 No Policy Data

When no policy is executed (includes all ACMs per above; there may be Supplier and/or Owner policies but none of their policy elements are understood by the ACMs' policy engines or they contain no policy elements), 20 bytes of 0s will be extended to PCR 17.

3.3.2 Allow Any Policy

If an Owner policy exists and is of type LCP_POLTYPE_ANY, or no Owner policy exists and the Supplier policy is of type LCP_POLTYPE_ANY, then 20 bytes of 0s will extended to PCR 17.

3.3.3 Policy with LCP_POLICY_DATA

Because the measurement may contain the measurements of more than one policy list, it is important that the SINIT ACMs for all platforms order the list measurements in the same way so that identical policy evaluations will extend PCR 17 with the same value.

The policy engine will order the policy list measurements according to the order in which it evaluates policy elements. For this version of the specification, the following policy element types are evaluated in this order: LCP_POLELT_TYPE_MLE, LCP_POLELT_TYPE_PCONF, LCP_POLELT_TYPE_SBIOS. If additional policy element types are supported in the future, their evaluation order will be specified.

The policy engine will not measure duplicate policy lists, so if the same list is evaluated for more than one policy element then it will only be measured the first time. This is the case even if the same signature key is used for multiple lists.

The value that will be extended to PCR 17 will be the hash of the concatenation of all policy list measurements used.

If an Owner policy exists and is of type LCP_POLTYPE_LIST, no Owner policy exists and the Supplier policy is of type LCP_POLTYPE_LIST, or both the Owner and Supplier policies exist and are of type LCP_POLTYPE_LIST, then the value extended to PCR 17 is calculated as described above.

3.3.4 Force Platform Owner Policy

Whether the Platform Supplier policy indicates that the Platform Owner policy must be present (via bit 3 of its *PolicyControl* field) or not does not affect the measurement of the enforcing policy. The measurement will be calculated as indicated above.

3.4 Revocation

3.4.1 SINIT Revocation

LCP also enables a limited self-revocation mechanism for SINIT. SINIT itself enforces this on launch and a failure (i.e. the executing SINIT was revoked) results in a TXT reset with an error code stored in the TXT.ERRORCODE register. The algorithm or process that SINIT uses to determine whether it has been revoked is described in text on the *SINITMinVersion* field in section 3.1.1.1.

3.5 Platform Owner Index

The Platform Owner policy index is intended to represent the policy defined by the owner of the platform, and as such should be provisioned with access control permissions that enforce that control over the policy remains with the platform owner.

The following attribute settings are required:

Index Value: 0x40000001

Size: 54 bytes

Read Locality: 3 (can be others as well)

Read Auth: None

PCR Read: None

The simplest access control setting that maintains platform owner control is to set the Write Auth to Owner. However, this also means that updating the policy requires the owner authorization. As the owner authorization can be used for almost all TPM management operations, it may be desirable to limit its use.

Another possibility would be to use a separate authorization just for this index. That would eliminate having to provide the owner authorization just to update the policy.

If trusted software, running in the context of the MLE or with access to TPM locality 2, needs to be able to update the policy, then it would be possible to have no Write Auth but set the Write Locality to 2. This would permit whatever software was able to successfully launch to update the policy.

Note that it is not advisable to use PCR Write controls, since it would mean that the specified PCR could not change over time (e.g. if the software measured into it was upgraded). This is because the index attributes cannot be changed once the index is created.

4 Development and Deployment Considerations

4.1 Launch Control Policy Creation

Depending on the usage model, it may be desirable to create a Launch Control Policy at the time the MLE is built. This would apply in the cases where only one MLE is expected to run on the system. In such cases, the policy can be pre-created and provisioned during the installation of the MLE.

If multiple MLEs are expected to run on the system, or if there is to be a platform configuration policy, then it is likely that the policy will need to be created at the time of deployment.

In either case, it is advisable that the policy should contain an SINITMinVersion value that corresponds to the lowest versioned SINIT that is required. In the case of a system that supports multiple MLEs, a different SINITMinVersion may be specified with each MLE's policy element. The effective SINITMinVersion value will be the highest of the values in the PS policy, PO policy, and matching MLE element (for each one that exists). This effective SINITMinVersion will be compared to the AcmVersion of the SINIT being used. Setting the SINITMinVersion value in the policy prevents an attacker from substituting an older version of SINIT (if there is one for a given platform) that may have security issues.

4.2 Launch Errors and Remediation

If there is an error during a measured launch, the platform will be reset and an error code left in the TXT.ERRORCODE register. It is important for MLE vendors to consider how their software will handle such errors and allow users or administrators to remediate them.

If an MLE is launched automatically either as part of the boot process or as part of an operating system's launch process, it needs to be able to detect a previous failure in order to prevent a continuous cycle of boot failures. Such failures may occur as part of the loading and preparation for the GETSEC[SENTER] instruction or they may occur during the processing of that instruction before the MLE is given control.

In the former case, it is the MLE launching software that is detecting the error condition (e.g. a mismatched SINIT ACM, TXT not being enabled, etc.) and that software can use whatever mechanism it chooses to persist the error or to handle it at that time. In the latter case, the system will be reset before the MLE launching software can handle the error and so that software should be able to detect the error in the TXT.ERRORCODE register and take appropriate action.

The particular remediation action needed will depend on the error itself. If the MLE launch happens early in the boot process, the launching software may need a way of booting into a remediation operating system. If the launch happens within an operating system environment, the software may be able to remediate in that environment.

4.3 Determining Trust

While a TXT Launch Control Policy can be used to prevent software use of TPM locality 2 and access to TXT private space, it is not a general mechanism to prevent unwanted software from executing on a system. Consequently, an MLE cannot itself determine whether it is running in a TXT measured environment (it could be running on an emulator that spoofs PCR values, chipset registers, etc.).

In order to gain trust in an MLE (or rather, in software that uses TXT with the intention of being an MLE), the PCR values for the MLE (PCRs 17 and 18) must be used to make the trust "decision". The trust "decision" must either be made by a party external to the MLE's system (i.e. remote attestation) or by the release of some data that is not available to untrusted software (i.e. local attestation).

Remote attestation involves a remote party requesting the MLE to provide a TPM quote of the PCRs needed to determine trust (at least 17 and 18) and that remote party verifying the quote and making a trust determination of the PCR values. The remote party can then act on the trust level in various ways (disconnect the MLE system from the network, not provide it with network credentials, etc.). The details involved in the remote attestation process can be found at the Trusted Computing Group's (TCG) website (http://www.trustedcomputinggroup.org/).

Local attestation, also known as SEALing, uses the TPM to encrypt some data bound to certain PCR values (and/or locality). The data must be SEALed by the MLE system when the system is in a trusted state (there are multiple ways to establish an initial trusted state) and the PCR/binding made to values that represent the desired trusted state (the initial and final states don't have to be the same as long as they are both trusted). The SEALed data is then persisted, to be retrieved and UNSEALed when the trusted MLE is running. The specifics of SEALing can also be found on the TCG website.

4.3.1 Migration of SEALed data

If SEALing/local attestation is used to protect data, then the MLE must be able to accommodate the upgrading/changing of components whose measurements are in the PCRs being SEALed to. Since PCRs 17 and 18 contain the TCB for the MLE, at a minimum this would include SINIT, LCP, the MLE, etc. (see section 1.9 for the complete contents of these PCRs). If data is SEALed to additional PCRs then changes to the entities which are measured into these other PCRs must also be handled. While data may be sealed to PCRs, locality, or an auth value, or any combination thereof, migration is only an issue when PCRs are sealed to.

When one of the elements of the SEALed data's PCRs is changed, the TPM will no longer UNSEAL that data. So if no migration or backup of the plaintext data is made, then after the next TXT launch that data will not be available to the new MLE. And unless the original MLE can be re-launched, the data will be lost. Thus, some provision for making the data available to the new MLE must be made while the data is still available in plaintext form (UNSEALed), i.e. in the original MLE.

The most seamless and secure method for migrating the data to the new environment is for the original environment to re-SEAL the data to the new environment. This requires the original environment to calculate what the PCR values will be in the new environment. For PCRs 17 and 18, this can be done using the information in section 1.9, coupled with knowing or being able to calculate the constituent values for the new components. Alternately, if the new environment were run on a trusted system (so that nothing would tamper with the measurements), the PCR values could then be collected from that system and used directly as the new values without having to calculate them from the components.

Other methods, such as password-based recovery, key escrow, etc. would be the same as for any other encrypted data and have similar tradeoffs.

4.4 Deployment

4.4.1 LCP Provisioning

4.4.1.1 TPM Ownership

Because creation and writing to the Platform Owner policy TPM NV index requires the TPM owner authorization credential, the installation program should accommodate differing IT policies for how and when TPM ownership is established. In some enterprises, IT may take ownership before a system is deployed to an end user. In others, the TPM may be un-owned until the first application that requires ownership establishes it.

Since the TPM owner authorization credential will be required to modify the Platform Owner policy, if the installation program creates the credential it should provide a mechanism for securely saving that credential either locally or remotely.

4.4.1.2 Policy Provisioning

The Platform Owner policy TPM NV index will need to be created by the MLE installation program (or other TPM management software) if does not already exist. This can be done with the TPM_NV_DefineSpace command or corresponding higher-level TPM interface (e.g. via a TCG Software Stack or TSS).

Once the index has been created, the installation program can write the policy into the index using the TPM_NV_WriteValue command or corresponding higher-level TPM interface (e.g. via a TCG Software Stack or TSS).

Because creating and writing to the Platform Owner policy index requires the TPM owner authorization credential, care should be taken to protect the credential when it is being used and to erase or delete it from memory as soon as it is no longer needed.

Ideally, policy provisioning would occur in a secure environment or be performed by an agent that can be verified as trustworthy. An example of the former would be on an isolated network immediately after receiving the system. Another would be booting from a CD containing provisioning software. An example of the latter would be to use Intel TXT to launch a provisioning MLE or agent that was then attested to by a remote entity which could provide the owner authorization credential upon successful attestation.

4.4.2 SINIT Selection

Because the SINIT AC module is specific to a chipset, different platforms may have different SINIT ACMs. If an MLE is intended to run on multiple platforms with different chipsets, the MLE installation program will need to determine which SINIT ACM to install.

This can be done by comparing the chipset compatibility information in the SINIT ACM's Chipset ID List with the corresponding information for the platform. This would be identical to the process for verifying SINIT ACM compatibility at launch time, as described in section 2.2.3.1.

Appendix A Intel® TXT Execution Technology Authenticated Code Modules

A.1 Authenticated Code Module Format

An authenticated code module (AC module) is required to conform to a specific format. At the top level the module is composed of three sections: module header, internal working scratch space, and user code and data. The module header contains critical information necessary for the processor to properly authenticate the entire module, including the encrypted signature and RSA based public key. The processor also uses other fields of the AC module for initializing the remaining processor state after authentication.

The format of the authenticated-code module is in Table 3. This definition represents Revision 0.0 of the AC module header version (defined in the HeaderVersion field).

Table 3. Authenticated Code Module Format

Field	Offset	Size (bytes)	Description
ModuleType	0	2	Module type
ModuleSubType	2	2	Module sub-type
HeaderLen	4	4	Header length (in multiples of four bytes) (161 for version 0.0)
HeaderVersion	8	4	Module format version
ChipsetID	12	2	Module release identifier
Flags	14	2	Module-specific flags
ModuleVendor	16	4	Module vendor identifier
Date	20	4	Creation date (BCD format: year.month.day)
Size	24	4	Module size (in multiples of four bytes)
Reserved1	28	4	Reserved for future extensions
CodeControl	32	4	Authenticated code control flags
ErrorEntryPoint	36	4	Error response entry point offset (bytes)
GDTLimit	40	4	GDT limit (defines last byte of GDT)

Field	Offset	Size (bytes)	Description
GDTBasePtr	44	4	GDT base pointer offset (bytes)
SegSel	48	4	Segment selector initializer
EntryPoint	52	4	Authenticated code entry point offset (bytes)
Reserved2	56	64	Reserved for future extensions
KeySize	120	4	Module public key size less the exponent (in multiples of four bytes) (64 for version 0.0)
ScratchSize	124	4	Scratch field size (in multiples of four bytes) (2 * KeySize + 15 for version 0.0)
RSAPubKey	128	KeySize * 4	Module public key
RSAPubExp	384	4	Module public key exponent
RSASig	388	256	PKCS #1.5 RSA Signature
End of AC module header			
Scratch	644	ScratchSize * 4	Internal scratch area used during initialization (needs to be all 0s)
User Area	644 + ScratchSize* 4	N * 64	User code/data (modulo-64 byte increments)

ModuleType

Indicates the module type. The following module types are defined:

2 = Chipset authenticated code module.

Only ModuleType 2 is supported by GETSEC functions SENTER and ENTERACCS.

ModuleSubType

Indicates whether the module is capable of being executed at processor reset.

0 = ACM cannot be executed at processor reset

1 = ACM is capable of being executed at processor reset

ModuleSubType 1 is not supported for use by the GETSEC[SENTER] instruction.

HeaderLen

Length of the authenticated module header specified in 32-bit quantities. The header spans the beginning of the module to the end of the signature field. This is fixed to 161 for AC module version 0.0.

HeaderVersion

Specifies the AC module header version. Major and minor vendor field are specified, with bits 15:0 holding the minor value and bits 31:16 holding the major value. This should be initialized to zero for header version 0.0. Unsupported header versions will be rejected by the processor and result in an abort during authentication.

ChipsetID

Module-specific chipset identifier.

Flags

Module-specific flags. The following bits are currently defined:

Table 4. AC module Flags Description

Bit position	Description
13:0	Reserved (must be 0)
14	Production (0) or pre-production flag (1)
15	Production (0) or debug (1) signed

ModuleVendor

Module creator vendor ID. Use the PCI SIG* assignment for vendor IDs to define this field. The following vendor ID is currently recognized:

00008086H = Intel

Date

Creation date of the module. Encode this entry in the BCD format as follows: year.month.day with two bytes for the year, one byte for the day, and one byte for the month. For example, a value of 20040328H indicates module creation on March 28, 2004.

Size

Total size of module specified in 32-bit quantities. This includes the header, scratch area, user code and data.

Reserved1

Reserved. This should be initialized to zeros.

CodeControl

Authenticated code control word. Defines specific actions or properties for the authenticated code module.

ErrorEntryPoint

If bit 0 of the CodeControl word is 1, the processor will vector to this location if a snoop hit to a modified line was detected during the load of an authenticated code module. If bit 0 is 0, then enabled error reporting via bit 1 of a HITM during ACEA load will result in an abort of the authentication process and signaling of an Intel® Trusted Execution Technology shutdown condition.

GDTLimit

Limit of the GDT in bytes, pointed to by GDTBasePtr. This is loaded into the limit field of the GDTR upon successful authentication of the code module.

GDTBasePtr

Pointer to the GDT base. This is an offset from the authenticated code module base address.

SegSel

Segment selector for initializing CS, DS, SS, and ES of the processor after successful authentication. CS is initialized to SegSel while DS, SS, and ES are initialized to SegSel + 8.

EntryPoint

Entry point into the authenticated code module. This is an offset from the module base address. The processor begins execution from this point after successful authentication.

Reserved2

Reserved. Should contain zeros.

KeySize

Defines the width the RSA public key in dwords applied for authentication, less the size of the exponent. For version 0.0 of the AC module header, KeySize is fixed to 64 (a 2048 bit key). The information in this field is intended to support external software parsing of an AC module independent of the module version. It is the responsibility of the developer to reflect an accurate KeySize. This field is not checked for consistency by the processor.

ScratchSize

Defines the width of the scratch field size specified in 32-bit quantities. For version 0.0 of the AC module header, ScratchSize is defined by KeySize * 2 + 15. The information in this field is intended to support external software parsing of an AC module independent of the module version. It is the responsibility of software to reflect an accurate ScratchSize. This field is not checked by the processor.

RSAPubKey

Contains a public key plus a fixed 32-bit exponent to be used for decrypting the signature of the module. The size of this field is defined by the previously defined AC module field, KeySize + 1.

RSASig

The PKCS #1.5 RSA Signature of the module. The RSA Signature signs an area that includes the some of the module header and the USER AREA data field (which represents the body of the module). Parts of the module header not included are: the RSA Signature, public key, and scratch field.

Scratch

Used for temporary scratch storage by the processor during authentication. This area can be used by the user code during execution for data storage needs.

User Area

User code and data represented in modulo-64 byte increments. In addition, the boundary between data and code should be on at least modulo-1024 byte intervals. The user code and data region is allocated from the first byte after the end of the Scratch field to the end of the AC module.

The chipset AC module information table is located at the start of the User Area and contains supplementary information that is specific to chipset AC modules. The chipset ID list is described in more detail in section 2.2.3.1.

Table 5. Chipset AC Module Information Table

Field	Offset (Bytes)	Width (Bytes)	Description

UUID	0	16	UUID of the Chipset AC module information table defined as:
			ULONG UUID0; // 0x7FC03AAA
			ULONG UUID1; // 0x18DB46A7
			ULONG UUID2; // 0x8F69AC2E
			ULONG UUID3; // 0x5A7F418D
			This UUID is used to identify a file/memory image as being a chipset AC module.

ChipsetACMType	16	1	Module type (00h = BIOS; 01h = SINIT)
Version	17	1	Version of this table. Table versions are always backwards compatible. The highest version defined is currently 4.
Length	18	2	Length of this table in bytes.
ChipsetIDList	20	4	Location of the Intel TXT Chipset ID list used to identify Intel TXT chipsets supported by this AC Module. This field is an offset in bytes from the start of the AC Module. See Table 6 for details.
OsSinitDataVer	24	4	Indicates the maximum version number of the OS to SINIT data structure that this module supports. It is assumed that the module is backward compatible with previous versions.

Field	Offset (Bytes)	Width (Bytes)	Description
MinMleHeaderVer	28	4	Indicates the minimum version number of the MLE Header data structure that this module supports/requires. MLEs with more recent header versions are responsible for determining whether they can support this version of the ACM.
Capabilities	32	4	Bit vector of supported capabilities. The values match those of the Capabilities field in the MLE header. This can be used by an MLE to determine whether the ACM is compatible with it and to determine any optional capabilities it might support.
AcmVersion	36	1	Version of this AC Module. It is compared against the SINITMinVersion field in LCP to determine if the module is revoked.
Reserved	37	3	Reserved
ProcessorIDList	40	4	Location of the Intel TXT Processor ID list used to identify Intel TXT processors supported by this AC Module. This field is an offset in bytes from the start of the AC Module. See Table 8 for details.

Table 6. Chipset ID List

Field	Offset	Width (Bytes)	Description
Count	0	4	Number of entries in the array ChipsetID
ChipsetIDs[]	4	Count * sizeof(TXT_ACM _CHIPSET_ID)	An array of count entries of the structure TXT_ACM_CHIPSET_ID (see next table).

Table 7. TXT_ACM_CHIPSET_ID Format

Field	Offset	Width (Bytes)	Description
Flags	0	4	Set of flags to further describe functions of the chipset ID structure. Bit Description: [0]: RevisionIdMask – if 0, the RevisionId field must exactly match the TXT.DIDVID.RID field. If 1, the RevisionId field is a bitwise mask that can be used to test for any bits set in the TXT.DIDVID.RID field. If any bits are set, the RevisionId is a match. [31:1]: Reserved for future use. Must be 0.
VendorID	4	2	Indicates the chipset vendor this AC Module is designed to support. This field is compared against the TXT.DIDVID.VID field.

Field	Offset	Width (Bytes)	Description
DeviceID	6	2	Indicates the chipset vendor's device that this AC Module is designed to support. This field is compared against the TXT.DIDVID.DID field.
RevisionID	8	2	Indicates the revision of the chipset vendor's device that this AC module is designed to support. This field is used according to the RevisionIdMask bit in the Flags field.
Reserved	10	6	Reserved for future use.

Table 8. Processor ID List

Field	Offset	Width (Bytes)	Description
Count	0	4	Number of entries in the array ProcessorID
ProcessorIDs[]	4	Count * sizeof(TXT_ACM_PROCESSOR_ID)	An array of count entries of the structure TXT_ACM_PROCESSOR_ID (see next table).

Table 9. TXT_ACM_PROCESSOR_ID Format

Field	Offset	Width (Bytes)	Description
FMS	0	4	Indicates the Family/Model/Stepping of the processor this AC Module is designed to support. This field is compared against the corresponding value returned from the cpuid instruction.
FMSMask	4	4	Mask to apply to FMS
PlatformID	8	8	Indicates the Platform ID of the processor this AC Module is designed to support. This field is compared against the value in the IA32_PLATFORM_ID MSR.
PlatformMask	16	8	Mask to apply to Platform ID

A.1.1 Memory type cacheability restrictions

Prior to launching the authenticated execution environment using the GETSEC leaf functions ENTERACCS or SENTER, processor MTRRs (Memory Type Range Registers) must first be initialized to map out the authenticated RAM addresses as WB (write-back). Failure to do so may affect the ability for the processor to maintain isolation of the loaded authenticated code module. The processor will signal an Intel TXT shutdown condition with error code #BadACMMType during the loading of the authenticated code module if non-WB memory is detected.

While physical addresses within the load module must be mapped as WB, the memory type for locations outside of the module boundaries must be mapped to one of the supported memory types as returned by GETSEC[PARAMETERS] (or UC as default). This is required to support inter-operability across SMX capable processor implementations.

A.1.2 Authentication and execution of AC module

Authentication is performed after loading of the code module into the authenticated code execution area. Information from the authenticated code module header is used to support the authentication process. The RSAPubKey header field contains a public key plus a 32 bit exponent used for decrypting the signature of the authenticated code module. The signature is held in encrypted form in the RSASig header field and it represents the PKCS #1.5 RSA Signature of the module. The RSA Signature signs an area that includes the sum of the module header and the entire USER AREA data field, which represents the body of the module. Those parts of the module header not included are: the RSA Signature, the public key, and the scratch field. An inconsistent authenticated code module format, inconsistent comparison of the public key hash, or mismatch of the decrypted signature against the computed hash of the authenticated module or a corrupted signature padding value results in an abort of the authentication process and signaling of a Intel TXT shutdown condition. As part of the authentication step, the processor stores the decrypted signature of the AC module in the first 20 or 32 bytes (depending on the SINIT AC Module) of the 'Scratch' field of the AC module header.

After authentication has completed successfully, the private configuration space of the Intel TXT-capable chipset is unlocked. At this point, only the authenticated code module or system software executing in authenticated code execution mode is allowed to gain access to the restricted chipset state for the purpose of securing the platform.

The architectural state of the processor is partially initialized from contents held in the header of the authenticated code module. The processor GDTR, CS, and DS selectors are initialized from fields within the authenticated code module. Since the authenticated code module must be relocatable, all address references must be relative to the authenticated code module base address in EBX. The processor GDTR base value is initialized to the AC module header field GDT BasePtr + module base address held in EBX and the GDTR limit is set to the value in the GDT Limit field. The CS selector is initialized to the AC module header SegSel field, while the DS selector is initialized to CS + 8. The segment descriptor fields are implicitly initialized to BASE=0, LIMIT=FFFFFh, G=1, D=1, P=1, S=1, read/write access for DS, and execute/read access for CS. The processor begins the authenticated code module execution with the EIP set to the AC module header EntryPoint field + module base address (EBX). The AC module based fields used for initializing the processor state are checked for consistency and any failure results in a shutdown condition.

Appendix B SMX Interaction with Platform

B.1 Intel® Trusted Execution Technology Configuration Registers

Intel TXT configuration registers are a subset of chipset registers. These registers are mapped into two regions of memory, representing the public and private configuration spaces. Registers in the private space can only be accessed after a measured environment has been established and before the TXT.CMD.CLOSE-PRIVATE command has been issued. The private space registers are mapped to the address range starting at FED20000H. The public space registers are mapped to the address range starting at FED30000H and are available before, during and after a measured environment launch. All registers are defined as 64 bits and return 0's for the unimplemented bits. The offsets in the table are from the start of either the public or private spaces (all registers are available within both spaces, though with different permissions).

After writing to one of the command registers (e.g. TXT.CMD.SECRETS), software should read the corresponding status flag for that command (e.g. TXT.E2STS[SECRETS.STS]) to ensure that the command has completed successfully.

B.1.1 TXT.STS – Status

Description	This is the general status register. This read-only register is used by AC modules and the MLE to get the status of various Intel TXT features.
Offset	000H
Pub Attribs **Priv Attribs**	RO RO

Bits	Field Name	Field Description
0	SENTER.DONE. STS	The chipset sets this bit when it sees all of the threads have done an TXT.CYC.SENTER-ACK. When any of the threads does the TXT.CYC.SEXIT-ACK the TXT.THREADS.JOIN and TXT.THREADS.EXISTS registers will not be equal, so the chipset will clear this bit.
1	SEXIT.DONE. STS	This bit is set when all of the bits in the TXT.THREADS.JOIN register are clear. Thus, this bit will be set immediately after reset (since the bits are all 0). Once all threads have done an TXT.CYC.SEXIT-ACK, the TXT.THREAD.JOIN register will be 0, so the chipset will set this bit.

Description	This is the general status register. This read-only register is used by AC modules and the MLE to get the status of various Intel TXT features.	
5:2	Reserved	Reserved
6	MEM-CONFIG-LOCK.STS	This bit will be set to 1 when the memory configuration has been locked. Cleared by TXT.CMD.UNLOCK.MEMCONFIG or by a system reset.
7	PRIVATE-OPEN.STS	This bit will be set to 1 when TXT.CMD.OPEN-PRIVATE is performed. Cleared by TXT.CMD.CLOSE-PRIVATE or by a system reset.
14:8	Reserved	Reserved
15	TXT.LOCALITY1.OPEN.STS	This bit is set when the TXT.CMD.OPEN.LOCALITY1 command is seen by the chipset. It is cleared on reset or when TXT.CMD.CLOSE.LOCALITY1 is seen.
16	TXT.LOCALITY2.OPEN.STS	This bit is set when either the TXT.CMD.OPEN.LOCALITY2 command or the TXT.CMD.OPEN.PRIVATE is seen by the chipset. It is cleared on reset, when either TXT.CMD.CLOSE.LOCALITY2 or TXT.CMD.CLOSE.PRIVATE is seen, and by the GETSEC[SEXIT] instruction.
63:17	Reserved	Reserved

B.1.2 TXT.ESTS – Error Status

Description	This is the error status register which contains status information associated with various error conditions. The contents of this register are preserved across soft resets.
Offset	008H
Pub Attribs	RO
Priv Attribs	RO

Bits	Field Name	Field Description
0	TXT_RESET.STS	This bit is set to ‘1’ to indicate that an event occurred which may prevent the proper use of TXT (possibly including a TXT reset). To maintain TXT integrity, while this bit is set a TXT measured environment cannot be established; consequently Safer Mode Extension (SMX) instructions GETSEC[ENTERACCS] and GETSEC [SENTER] will fail. This bit is sticky and will only be cleared on a power cycle.
7:1	Reserved	Reserved

B.1.3 TXT.ERRORCODE – Error Code

Description	This register holds the Intel TXT shutdown error code. A soft reset does not clear the contents of this register; a hard reset/power cycle will clear the contents. This was formerly labeled the TXT.CRASH register.
Offset	030H
Pub Attribs **Priv Attribs**	RO RW

Bits	Field Name	Field Description
14:0	Type2	This is implementation and source specific. Provides details on the failure condition.
15	SoftwareSource	0 = Authenticated Code Module 1 = MLE
29:16	Type1	This is implementation and source specific. Provides details on the failure condition.
30	Processor/Software	0 = Error condition reported by processor (see Table 10) 1 = Error condition reported by software
31	Valid/Invalid	0 = Register content invalid. 1 = Valid error

Note 1: Upon successful execution, SINIT will put 0xC0000001 in the register.

Note 2: The format of the Type field for errors reported by SINIT is defined in an errors text file included with each SINIT AC module. This file also includes the definition of the error codes produced by that version of SINIT.

Table 10. Type Field Encodings for Processor-Initiated Intel TXT Shutdowns

Type	Error condition	Mnemonic
0	Legacy shutdown	#LegacyShutdown
1-4	Reserved	Reserved
5	Load memory type error in Authenticated Code Execution Area	#BadACMMType
6	Unrecognized AC module format	#UnsupportedACM
7	Failure to authenticate	#AuthenticateFail
8	Invalid AC module format	#BadACMFormat
9	Unexpected snoop hit detected	#UnexpectedHITM
10	Invalid event	#InvalidEvent[1]
11	Invalid MLE JOIN format	#BadJOINFormat
12	Unrecoverable machine check condition	#UnrecovMCError

Type	Error condition	Mnemonic
13	VMX abort error occurred	#VMXAbort
14	Authenticated code execution area corruption	#ACMCorrupt
15	Invalid voltage/bus ratio	#InvalidVIDBRatio
16 – 65535	Reserved	Reserved

Note: The conditions under which most of these errors are generated can be found in the pseudo code of the SMX instructions in Chapter 6, "Safer Mode Extensions Reference", of the *Intel 64 and IA-32 Software Developer Manuals, Volume 2B.*

Note 1: #InvalidEvent can be generated by the following:

- A CPU reset which is not caused by a TXT Reset
- A non-virtualized INIT event
- During RLP wakeup, bit 0 of the RLPs' IA32_SMM_MONITOR_CTL MSR does not match that of the ILP
- An SENTER/SEXIT/WAKEUP event is received post-VMXON
- A thread wakes from the wait-for-SIPI state while another thread in the same CPU is executing an AC Module

B.1.4 TXT.CMD.RESET – System Reset Command

Description	A write to this register causes a system reset. This is performed by the processor as part of an Intel TXT shutdown, after writing to the TXT.ERRORCODE register.
Offset	038H
Pub Attribs **Priv Attribs**	- WO

Bits	Field Name	Field Description
7:0		

B.1.5 TXT.CMD.CLOSE-PRIVATE – Close Private Space Command

Description	A write to this register causes the Intel TXT-capable chipset private configuration space to be locked. Locality 2 will also be closed. Once locked, conventional memory read/write operations can no longer be used to access these registers. The private configuration space can only be opened for the MLE by successfully executing GETSEC[SENTER].
Offset	048H
Pub Attribs **Priv Attribs**	- WO (a serializing operation, such as a read of the register, is required

	after the write to ensure that any future chipset operations see the write)

Bits	Field Name	Field Description
7:0		

B.1.6 TXT.VER.FSBIF – Frontside Bus Interface

Description	This register identifies whether the chipset is debug or release fused. On certain chipsets, a 4 byte read to this address will return either 0xFFFF_FFFF or 0x0000_0000. In these cases, the MLE should read an alternate offset (TXT.VER.EMIF, 200H) to capture this information.
Offset	100H
Pub Attribs	RO
Priv Attribs	RO

Bits	Field Name	Field Description
30:0	Reserved	Reserved
31	DEBUG.FUSE	0 = Chipset is debug fused 1 = Chipset is production fused

B.1.7 TXT.DIDVID – TXT Device ID

Description	This register contains the vendor, device, and revision IDs for the memory controller or chipset.
Offset	110H
Pub Attribs	RO
Priv Attribs	RO

Bits	Field Name	Field Description
15:0	VID	Vendor ID: 8086 for Intel® components
31:16	DID	Device ID: specific to the chipset/platform
47:32	RID	Revision ID: specific to the chipset/platform
63:48	ID-EXT	Extended ID: specific to the chipset/platform

B.1.8 TXT.VER.QPIIF – Intel® QuickPath Interconnect Interface

Description	This register identifies whether the memory controller or chipset is debug or release fused. On certain chipsets, a 4 byte read to TXT.VER.FSBIF will return 0xFFFF_FFFF or 0x0000_0000. In these cases, the MLE should read this register to determine if the chipset is debug or release fused.
Offset	200H
Pub Attribs **Priv Attribs**	RO RO

Bits	Field Name	Field Description
30:0	Reserved	Reserved
31	DEBUG.FUSE	0 = Chipset is debug fused 1 = Chipset is production fused

B.1.9 TXT.CMD.UNLOCK-MEM-CONFIG – Unlock Memory Config Command

Description	When this command is invoked, the chipset unlocks all memory configuration registers.
Offset	218H
Pub Attribs **Priv Attribs**	- WO (a serializing operation, such as a read of the register, is required after the write to ensure that any future chipset operations see the write)

Bits	Field Name	Field Description
7:0		

B.1.10 TXT.SINIT.BASE – SINIT Base Address

Description	This register contains the physical base address of the memory region set aside by the BIOS for loading an SINIT AC module. If BIOS has provided an SINIT AC module, it will be located at this address. System software that provides an SINIT AC module must store it to this location.
Offset	270H
Pub Attribs **Priv Attribs**	RW RW

Bits	Field Name	Field Description
31:0		

B.1.11 TXT.SINIT.SIZE – SINIT Size

Description	This register contains the size (in bytes) of the memory region set aside by the BIOS for loading an SINIT AC module. This register is initialized by the BIOS.
Offset	278H
Pub Attribs	RW
Priv Attribs	RW

Bits	Field Name	Field Description
31:0		

B.1.12 TXT.MLE.JOIN – MLE Join Base Address

Description	Holds a physical address pointer to the base of the join data structure used to initialize RLPs in response to GETSEC[WAKEUP].
Offset	290H
Pub Attribs	RW
Priv Attribs	RW

Bits	Field Name	Field Description
31:0		

B.1.13 TXT.HEAP.BASE – TXT Heap Base Address

Description	This register contains the physical base address of the Intel TXT Heap memory region. The BIOS initializes this register.
Offset	300H
Pub Attribs	RW
Priv Attribs	RW

Bits	Field Name	Field Description
31:0		

B.1.14 TXT.HEAP.SIZE – TXT Heap Size

Description	This register contains the size (in bytes) of the Intel TXT Heap memory region. The BIOS initializes this register.
Offset	308H
Pub Attribs **Priv Attribs**	RW RW

Bits	Field Name	Field Description
31:0		

B.1.15 TXT.DPR – DMA Protected Range

Description	This register defines the DMA Protected Range of memory in which the TXT heap and SINIT region are located.
Offset	330H
Pub Attribs **Priv Attribs**	RW RW

Bits	Field Name	Field Description
0	Lock	Bits 19:0 are locked down in this register when this bit is set.
3:1	Reserved	Reserved
11:4	Size	This is the size of memory, in MB, that will be protected from DMA accesses. A value of 0x00 in this field means no additional memory is protected. The DPR range works independently of any other DMA protections, such as VT-d, and is done post any VT-d translation or TXT checks.
19:12	Reserved	Reserved
31:20	Top	Top address + 1 of DPR. This is the base of TSEG.

B.1.16 TXT.CMD.OPEN.LOCALITY1 – Open Locality 1 Command

Description	Writing to this register “opens” the TPM locality 1 address range, enabling decoding by the chipset and thus access to the TPM. This locality is not automatically opened after GETSEC[SENTER] and must be opened explicitly. This locality will not be closed by the TXT.CMD.CLOSE-PRIVATE command.
Offset	380H

Pub Attribs **Priv Attribs**	- WO

Bits	Field Name	Field Description
7:0		

B.1.17 TXT.CMD.CLOSE.LOCALITY1 – Close Locality 1 Command

Description	Writing to this register “closes” the TPM locality 1 address range, disabling decoding by the chipset and thus access to the TPM. This locality will not be closed by the TXT.CMD.CLOSE-PRIVATE command.
Offset	388H
Pub Attribs **Priv Attribs**	- WO

Bits	Field Name	Field Description
7:0		

B.1.18 TXT.CMD.OPEN.LOCALITY2 – Open Locality 2 Command

Description	Writing to this register “opens” the TPM locality 2 address range, enabling decoding by the chipset and thus access to the TPM. This locality is automatically opened after GETSEC[SENTER]. This locality will be closed by the TXT.CMD.CLOSE-PRIVATE command.
Offset	390H
Pub Attribs **Priv Attribs**	- WO

Bits	Field Name	Field Description
7:0		

B.1.19 TXT.CMD.CLOSE.LOCALITY2 – Close Locality 2 Command

Description	Writing to this register “closes” the TPM locality 2 address range, disabling decoding by the chipset and thus access to the TPM.

	This locality will be closed by the TXT.CMD.CLOSE-PRIVATE command or by the GETSEC[SEXIT] instruction.
Offset	398H
Pub Attribs	-
Priv Attribs	WO

Bits	Field Name	Field Description
7:0		

B.1.20 TXT.PUBLIC.KEY – AC Module Public Key Hash

Description	This register contains the hash of the public key used for the verification of AC Modules. The size, hash algorithm, and value are specific to the memory controller or chipset.
Offset	400H
Pub Attribs	RO
Priv Attribs	RO

Bits	Field Name	Field Description
255:0		

B.1.21 TXT.CMD.SECRETS – Set Secrets Command

Description	Writing to this register indicates to the chipset that there are secrets in memory. The chipset tracks this fact with a sticky bit. If the platform reboots with this sticky bit set the BIOS AC module (or BIOS on multiprocessor TXT systems) will scrub memory. The chipset also uses this bit to detect invalid sleep state transitions. If software tries to transition to S3, S4, or S5 while secrets are in memory then the chipset will reset the system. The MLE issues the TXT.CMD.SECRETS command prior to placing secrets in memory for the first time.
Offset	8E0H
Pub Attribs	-
Priv Attribs	WO

Bits	Field Name	Field Description
7:0		

B.1.22 TXT.CMD.NO-SECRETS – Clear Secrets Command

Description	Writing to this register indicates there are no secrets in memory. The MLE will write to this register after removing all secrets from memory as part of the TXT teardown process.
Offset	8E8H
Pub Attribs **Priv Attribs**	- WO

Bits	Field Name	Field Description
7:0		

B.1.23 TXT.E2STS – Extended Error Status

Description	This register is used to read the status associated with various errors that might be detected. The contents of this register are preserved across soft resets.
Offset	8F0H
Pub Attribs **Priv Attribs**	RO RW

Bits	Field Name	Field Description
0	Reserved	Reserved
1	SECRETS.STS	0 = Chipset acknowledges that no secrets are in memory 1 = Chipset believes that secrets are in memory and will provide reset protection
63:2	Reserved	Reserved

B.2 TPM Platform Configuration Registers

The TPM contains Platform Configuration Registers (PCRs). The purpose of a PCR is to contain measurements. From a TPM standpoint, the TPM does not care what entity uses a PCR to store a measurement.

The TPM provides two types of PCRs: static and dynamic. Static PCRs only reset on system reset; dynamic PCRs reset upon request. Static PCRs are written by the static root of trust for measurement (SRTM). In the PC, the SRTM begins with the BIOS boot block. The dynamic PCRs are written by the dynamic root of trust for measurement (DRTM). In the PC, the DRTM is the process initiated by GETSEC[SENTER].

A PC TPM requires a minimum of 24 PCRs. The first 16 are designated the static Root of Trust and the next eight are designated the dynamic Root of Trust. Intel TXT uses PCRs 17 and 18 within the dynamic Root of Trust to measure the MLE.

All PCRs, static or dynamic, have the same size and same updating mechanism. The size is 160 bits. This size allows the PCRs to contain a SHA-1 hash digest value. Storing a measurement value in the PCRs involves a TPM_Extend operation, which is itself a hash operation.

B.3 Intel® Trusted Execution Technology Device Space

There are several memory ranges within Intel TXT address space provided to access Intel TXT related devices. The first range is 0xFED4_xxxx which is divided up into 16 pages. Each page in the FED4 range has specific access attributes. A page in this region may be accessed by Intel TXT cycles only, by Intel TXT cycles and via private space, or by Intel TXT cycles, private and public space.

Table 11. TPM Locality Address Mapping

Address Range	TPM Locality
FED4 0xxxH	Locality 0 (fully public)
FED4 1xxxH	Locality 1 (trusted OS)
FED4 2xxxH	Locality 2 (MLE access only)
FED4 3xxxH	Locality 3 (AC modules access only)
FED4 4xxxH	Locality 4 (Hardware or microcode access only)
All others	Reserved

The first five pages of the 0xFED4_xxxx region are used for TPM access. Each page represents a different locality to the TPM. Locality is an attribute used by the TPM to define how it treats certain transactions. Locality is defined by the address range used for commands sent to the TPM. All Intel TXT chipsets must support all localities. Locality 0 is considered public and accesses it is accepted by the chipset under all circumstances. Accesses to locality 0 are sent to the ICH even if Intel TXT is disabled, there has been no SENTER, or private space is closed. Locality 4 is never open, but may only be accessed with Intel TXT cycles. There are Intel TXT commands that will open localities 1 through 3. Localities 2-3 require that both LocalityX.OPEN and TXT.CMD.OPEN-PRIVATE be done before allowing accesses in that range to be accepted. At reset, localities 1 through 3 are closed.

No status read check of the TPM is performed by the processor GETSEC[SENTER] instruction ahead of the TPM.HASH write sequence. If the TPM is not in acquiesced state at this time, then the PCRs 17-20 reset and hash registration to PCR 17 may not succeed. To insure reliable system software functionality for TPM support, it is recommended that the GETSEC[SENTER] instruction only be executed once the TPM has acquiesced and ownership has been established in the context of the SENTER initiating process.

Appendix C Intel® TXT Heap Memory

Intel TXT Heap memory is a region of physically contiguous memory which is set aside by BIOS for the use of Intel TXT hardware and software. The system software that launches the measured environment passes data to both the SINIT AC module and the MLE using Intel TXT Heap memory. The system software is responsible for filling in the table contents prior to executing the SENTER instruction. An incorrect format or incorrect content of this table or tables described by this table will result in failure to launch the protected environment.

Table 12. Intel® Trusted Execution Technology Heap

Offset	Length (bytes)	Name	Description
0	8	BiosDataSize	Size in bytes of the Intel TXT specific data passed from the BIOS to system software for the purposes of launching the MLE. This size includes the number of bytes for this field, so this field cannot be less than a value of 8. Note 1.
8	BiosDataSize - 8	BiosData	BIOS specific data. The format of this data is described below in Table 13.
BiosDataSize	8	OsMleDataSize	Size in bytes of the data passed from the launching system software to the MLE. This size includes the number of bytes for this field, so this field cannot be less than a value of 8. Note 1.
BiosDataSize + 8	OsMleDataSize – 8	OsMleData	System software -specific data. Format of data in this field is considered specific to the system software vendor.
BiosDataSize + OsMleDataSize	8	OsSinitDataSize	Size in bytes of the data passed from the launching system software to the SINIT AC module. This size includes the number of bytes for this field, so this field cannot be less than a value of 8. Note 1.
BiosDataSize + OsMLEDataSize + 8	OsSinitDataSize - 8	OsSinitData	System software data passed to the SINIT AC module. The format of this data is described below in Table 14.

Offset	Length (bytes)	Name	Description
BiosDataSize + OsMleDataSize + OsSinitDataSize	8	SinitMleDataSize	Size in bytes of the data passed from the launched SINIT AC module to the MLE. This size includes the number of bytes for this field, so this field cannot be less than a value of 8. Note 1.
BiosDataSize + OsMleDataSize + OsSinitDataSize + 8	SinitMleDataSize - 8	SinitMleData	SINIT data passed to the MLE. The format of this data is described below in Table 15.

NOTES:

1. For proper data alignment on 64bit processor architectures this field must be a multiple of 8 bytes. OsMleDataSize + OsSinitDataSize + SinitMleDataSize must be less than or equal to TXT.HEAP.SIZE.

C.1 BIOS Data Format

The format of the data passed from the BIOS to the system software for the purposes of launching the measured environment is shown in Table 13.

Table 13. BIOS Data Table

Offset	Length (bytes)	Name	Description
0	4	Version	Version number of the BiosData table. The current value is 4 (versions < 2 are not supported). This value is incremented for any change to the definition of this table. Future versions will always be backwards compatible with previous versions (new fields will be added at the end).
4	4	BiosSinitSize	This field indicates the size of the SINIT AC module provided by system BIOS. A value of 0 indicates the BIOS is not providing an SINIT AC module for system software use. A non-0 value indicates that the AC module will be at the location specified by the TXT.SINIT.BASE register and be of the specified size.
8	8	LcpPdBase	Physical base address of the Platform Default Launch Control Policy, LCP_POLICY_DATA structure. Ignored if Platform Default Policy does not require additional data or does not exist.
16	8	LcpPdSize	Size of the Launch Control Policy Platform Default Policy Data. Ignored if Platform Default Policy does not require additional data or does not exist.

Offset	Length (bytes)	Name	Description
24	4	NumLogProcs	This is the total number of logical processors in the system. The minimum value in this register must be at least 1.
Versions >= 3			
28	8	Flags	BIOS-provided information for SINIT AC module consumption. Bit definition will be dependent on the chipset. Version 3 only.
Versions >= 4			
36	BiosDataSize - 36	ExtDataElements []	Array/list of extended data element structures. See section C.5 for element definitions.

C.2 OS to MLE Data Format

Each system software vendor may have a different format for this data, and any MLE being launched by system software must understand the format of that software's handoff data.

C.3 OS to SINIT Data Format

Table 14 defines the format of the data passed from the launching system software to the SINIT AC module in the OsSinitData field.

Table 14. OS to SINIT Data Table

Offset	Length (bytes)	Name	Description
0	4	Version	Version number of the OsSinitData table. Current values are 4 or 5 (versions < 4 are not supported). This value is incremented for any change to the definition of this table. Future versions will always be backwards compatible with previous versions (new fields will be added at the end).
4	4	Reserved	Reserved for future use
8	8	MLE PageTableBase	Physical address of MLE page table (the MLE page directory pointer table address)
16	8	MLE Size	Size in bytes of the MLE image
24	8	MLE HeaderBase	Linear address of MLE header (linear address within the MLE page tables)

Offset	Length (bytes)	Name	Description
32	8	PMR Low Base	Physical base address of the PMR Low region (must be 2MB aligned). Can be set to zero if not desired to be enabled by SINIT. The MVMM must be loaded in one of the DPR, PRM low, or the PMR high regions.
40	8	PMR Low Size	Size of the PMR Low Region (must be 2MB granular). Set to zero if not desired to be enabled by SINIT.
48	8	PMR High Base	Physical base address of the PMR High region (must be 2MB aligned). Can be set to zero if not desired to be enabled by SINIT.
56	8	PMR High Size	Size of the PMR HIGH Region (must be 2MB granular). Set to zero if not desired to be enabled by SINIT.
64	8	LCP PO Base	Physical base address of the Platform Owner's Launch Control Policy, LCP_POLICY_DATA structure.
72	8	LCP PO Size	Size of the Launch Control Policy Platform Owner's Policy Data.
80	4	Capabilities	Bit vector of capabilities that SINIT is requested to use. This must be a subset of the ones SINIT supports.
Versions >= 5			
84	8	EFI RSDT Pointer	Physical address of RSDT table when an EFI boot was performed. This will be ignored if SINIT finds the standard ACPI RSDT table.

C.4 SINIT to MLE Data Format

Table 15 defines the format of the SINIT data presented to the MLE.

Table 15. SINIT to MLE Data Table

Offset	Length (bytes)	Name	Description
0	4	Version	Version number of the SinitMleData table. Current values are 6 through 8 (versions < 5 are not supported). This value is incremented for any change to the definition of this table. Future versions will always be backwards compatible with previous versions (new fields will be added at the end).
4	20	BiosAcmID	ID of the BIOS AC module in the system

Offset	Length (bytes)	Name	Description
24	4	EdxSenterFlags	Value of EDX SENTER control flags
28	8	MsegValid	MSEG MSR (Valid bit only)
36	20	SinitHash	SHA-1 hash of the SINIT AC module
56	20	MleHash	SHA-1 hash of the MLE
76	20	StmHash	SHA-1 hash of STM. This is only valid if MsegValid = 1, else will contain zero
96	20	LcpPolicyHash	SHA-1 Hash of the LCP policy that was enforced; if no hash is needed based on the LCP policy control field this will contain zero
116	4	PolicyControl	Taken from the LCP policy used
120	4	RlpWakeupAddr	MONITOR physical address used for waking up RLPs (write 32bit non-0 value)
124	4	Reserved	Reserved for future use
128	4	NumberOfSinitMdrs	Number of SINIT Memory Descriptor Records
132	4	SinitMdrTableOffset	Pointer to the start of an array of SINIT Memory Descriptor Records as defined below. Each record describes a memory region as defined by the SINIT AC module (see Table 16). This field is an offset in bytes from the start of the SinitMleDataSize field.
136	4	SinitVtdDmarTableSize	Length of the Intel® Virtualization Technology (Intel® VT) for Directed I/O (Intel® VT-d) DMAR table pointed to by the SinitVtdDmarTable field
140	4	SinitVtdDmarTableOffset	Pointer to the start of the SINIT provided DMAR table dump for the MLE. This field is an offset in bytes from the start of the SinitMleDataSize field.
Versions >= 8			
144	4	ProcessorSCRTMStatus	Bit 0 - 1 if PCR 0 measurement for this boot was rooted in processor hardware. This is possible only if all logical processors implement S-CRTM and the platform is designed to take advantage of that capability. Bit 0 - 0 if PCR0 measurement for this boot was rooted in BIOS. Bits 31:1 – Reserved for future use The ProcessorSCRTMStatus field is reflected in PCR 17 measurement. See section PCR 17.

Table 16. SINIT Memory Descriptor Record

Offset	Length (bytes)	Name	Description
0	8	Address	Physical address of the memory range described in this record.
8	8	Length	Length of the memory range.
16	1	Type	Memory range type. Valid values: 0 Usable, good memory 1 SMRAM– Overlayed – deprecated 2 SMRAM– Non-Overlayed – deprecated 3 PCIe*- PCIe Extended Config Region 4–255 Reserved
17	7	Reserved	Reserved for future use

The array of Memory Descriptor Records (MDRs) is not necessarily ordered and some MDRs may be of 0 length, in which case they should be ignored.

Memory of type 0 is usable for the MLE and any code or data that it may load. SINIT will verify that the MLE and its page table are located in memory of this type.

Memory types 1 and 2 are deprecated in future versions of SINIT, as SMRAM regions are not of use to the MLE.

Memory of type 3 is the PCI Express extended configuration region. The MLE may use this to verify that the PCIE configuration specified in the ACPI tables is using the appropriate address space.

C.5 Extended Data Elements

Extended data elements are self-describing data structures that will be used for all future extensions to TXT heap tables. The *ExtDataElements[]* field in each of the heap tables is an array/list of individual elements, terminated by a HEAP_END_ELEMENT:

ExtDataElements[] ::= <HEAP_EXT_DATA_ELEMENT>* | <HEAP_END_ELEMENT>

Each element consists of the following data structure:

```
typedef struct {
    UINT32     Type;             // one of HEAP_EXTDATA_TYPE_*
    UINT32     Size;
    UINT8      Data[Size – 8];
} HEAP_EXT_DATA_ELEMENT;
```

The structures in the following sub-sections correspond to the contents of the *Data* field for the specific type of element.

While not required, it is recommended that *Size* be a 4-byte multiple.

C.5.1 HEAP_END_ELEMENT

```
#define HEAP_EXTDATA_TYPE_END                  0

typedef struct {
   UINT32        Type;           // = 0
   UINT32        Size;           // = 8
} HEAP_END_ELEMENT;
```

The HEAP_END_ELEMENT represents the terminating element of a given ExtDataElements[] list. It contains no *Data[]* field.

C.5.2 HEAP_BIOS_SPEC_VER_ELEMENT

```
#define HEAP_EXTDATA_TYPE_BIOS_SPEC_VER        1

typedef struct {
   UINT16        SpecVerMajor;
   UINT16        SpecVerMinor;
   UINT16        SpecVerRevision;
} HEAP_BIOS_SPEC_VER_ELEMENT;
```

The HEAP_BIOS_SPEC_VER_ELEMENT contains fields that indicate the version of the TXT BIOS specification to which this platforms BIOS corresponds. This element type is mainly useful for diagnostic tools.

C.5.3 HEAP_ACM_ELEMENT

```
#define HEAP_EXTDATA_TYPE_ACM                  2

typedef struct {
   UINT32        NumAcms;
   UINT64        AcmAddrs[NumAcms];  // phys addr of ACM
} HEAP_ACM_ELEMENT;
```

The HEAP_ACM_ELEMENT allows BIOS to indicate the ACMs that it contains and their locations in memory.

BIOSes that support this element type should report all ACMs that they carry, both BIOS ACMs and SINIT ACMs. Since the TXT architecture requires that BIOS provide at least one BIOS ACM, *NumAcms* must always be greater than 0.

AcmAddrs[] is an array of physical addresses of each of the ACMs.

C.5.4 HEAP_CUSTOM_ELEMENT

```
#define HEAP_EXTDATA_TYPE_CUSTOM               3

typedef struct {
   UINT32 data1;
   UINT16 data2;
   UINT16 data3;
   UINT16 data4;
   UINT8 data5[6];
} UUID;
```

```
typedef struct {
   UUID          Uuid;
   UINT8         Data[];
} HEAP_CUSTOM_ELEMENT;
```

The HEAP_CUSTOM_ELEMENT allows for platform suppliers to communicate supplier-specific data through a standard location and mechanism. Software wishing to use this data must understand its format.

Uuid is a UUID value that uniquely identifies the format of the *Data* field. It is important to generate the UUID value using a process that will provide a statistically unique value.

The *Data* field's contents are defined by the platform supplier. The size of this data must be included within the size of the HEAP_EXTDATA_ELEMENT::Size field.

Appendix DLCP v1

The version of Launch Control Policy described in this appendix applies to TXT-capable platforms produced before 2009.

D.1 Overview

The Launch Control Policy architecture consists of the following components:

- LCP Policy Engine – part of the SINIT ACM and enforces the policies stored on the platform
- LCP Policies – stored in the TPM, they specify the policies SINIT ACM will enforce.
- LCP PolicyData objects – referenced by the Policy structures in the TPM; each contains a list of valid Measured Launched Environments or a list of possible platform configurations.

Figure 4. Launch Control Policy Components shows how these components relate to each other. The figure also shows that there are two possible policies available on the platform: the Platform Default policy, as established by the Platform Supplier, and the Platform Owner's policy.

When the platform boots, its state is measured and recorded by the Static Root of Trust for Measurement (SRTM) and the other components which make up the static chain of trust; these events occur from when the platform is powered on until the Intel TXT measured launch (or until some component breaks the static chain). At this point GETSEC[SENTER] is invoked, and control is passed through the authenticated code execution area to the SINIT authenticated code module. The LCP engine in SINIT reads the LCP Policy Indices in the TPM NV, decides which policy to use, and checks the Platform Configuration and the Measured Launched Environment as require by the chosen policy. The measured environment is then launched.

Figure 4. Launch Control Policy Components

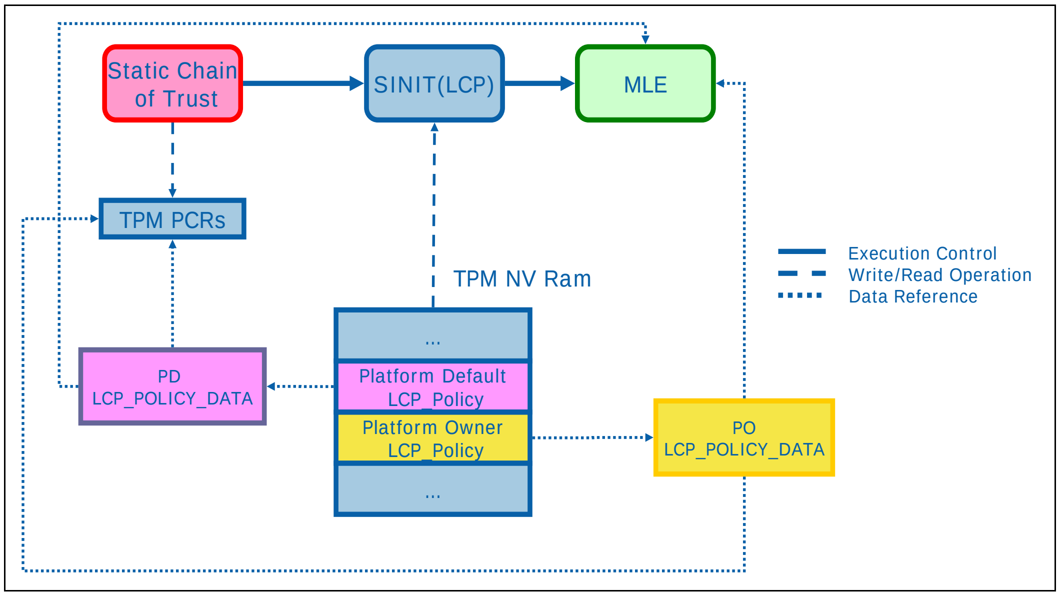
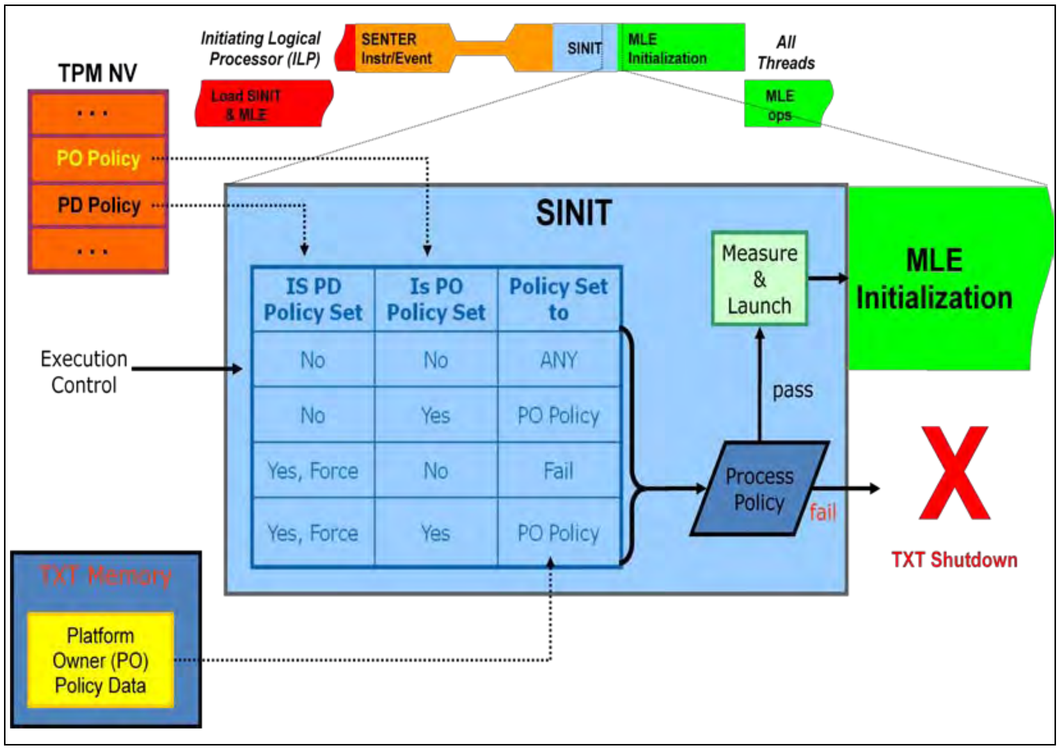


Figure 5. SINIT LCP Internal Flow shows the policy decision flow within the LCP engine within the SINIT authenticated code module. If no policy on the platform is configured then SINIT behaves as the policy allows ANY Measured Launched Environment and ANY platform configuration. When both the Platform Default and the Platform Owner Policies have been established on the platform the policies are combined (see section D.2.1), the Platform Owner policy taking precedence when there is a conflict between the two policies.

Figure 5. SINIT LCP Internal Flow



IS PD Policy Set	Is PO Policy Set	Policy Set to
No	No	ANY
No	Yes	PO Policy
Yes, Force	No	Fail
Yes, Force	Yes	PO Policy

D.1.1 LCP Components

The description of the policy that the SINIT AC module implements, consists of two policies: one policy is set by the Platform Supplier, known as the Platform Default (PS), and the other belongs to the Platform Owner (PO). Both policies are stored in the non-volatile store of the Trusted Platform Module (TPM NV). By storing the policy in the TPM NV, access controls can be applied to it; it also enables the policy to persist across platform power cycles.

D.1.1.1 LCP Policy

The *LCP_POLICY* structure (for a full listing see section D.6.1) is used for both the Platform Default and the Platform Owner policies. The size of the structure currently needs to be kept to a minimum in order to preserve the scarce resources of the TPM NV storage, which is why additional structures for both Default and Owner policies (*LCP_POLICY_DATA*) that can be persisted elsewhere are provided to handle additional information.

Figure 6. LCP_POLICY Structure

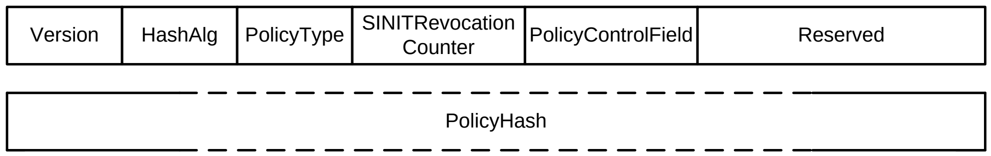


Figure 6. LCP_POLICY Structure diagrammatically illustrates the LCP_POLICY structure (fields not to scale): *HashAlg* identifies the hashing algorithm used through this policy. *PolicyType* indicates whether an additional LCP_POLICY_DATA structure is required and if this data is to be treated as unsigned.

- If the *PolicyType* field is *POLTYPE_ANY* then the value in the *PolicyHash* field is ignored and the environment to be launched is simply measured before execution control is passed to it. No corresponding *LCP_POLICY_DATA* is expected.
- If the *PolicyType* field is *POLTYPE_HASHONLY* then the value in the *PolicyHash* field is the only value against which the Measured Launched Environment is to be compared. No additional *LCP_POLICY_DATA* is required. No corresponding *LCP_POLICY_DATA* is expected. This policy type is not currently supported for the default policy.
- If the *PolicyType* field is *POLTYPE_UNSIGNED* then the value of *PolicyHash* is the result of computing a hash over the LCP_POLICY_DATA. The corresponding *LCP_POLICY_DATA* structure will contain data of type *LCP_UNSIGNED_POLICY_DATA*. This policy type is not currently supported for the default policy.
- If the *PolicyType* field is *POLTYPE_FORCEOWNERPOLICY* the value of *PolicyHash* is ignored and the LCP engine will process the Platform Owner's LCP Policy and fail if it is not present. No corresponding *LCP_POLICY_DATA* is expected. This value can only be used in the Platform Default LCP policy.

SINITRevocationCounter specifies the minimum version of SINIT that can be used. This value corresponds to the AcmVersion field in the AC module Information Table (see Table 5).

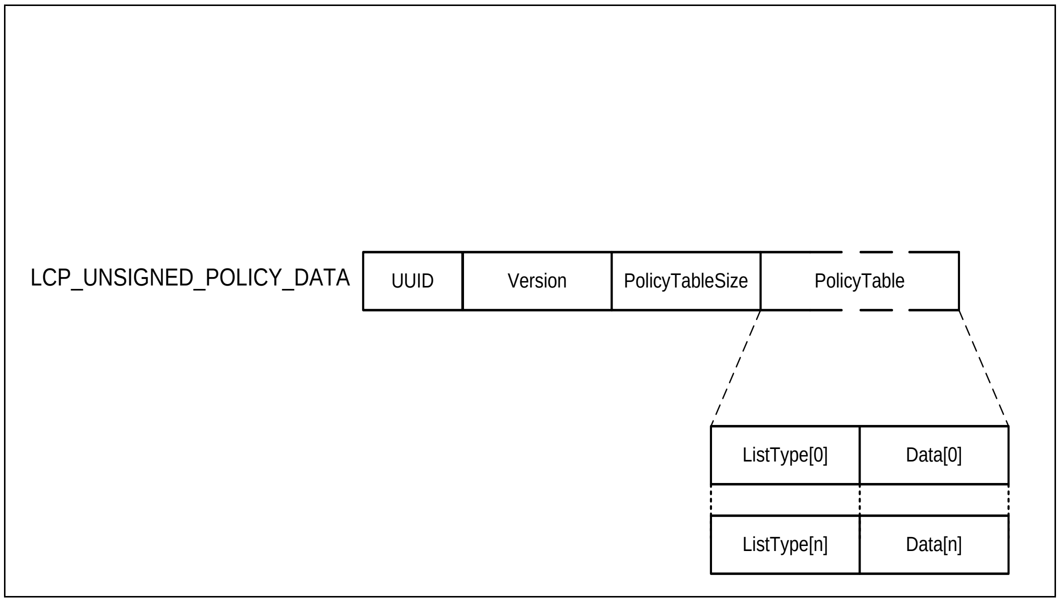
The *PolicyControlField* provides a number of control bits which are defined as:

- Bits31:2 Reserved and should set to zero
- Bit 1 Identifies whether the platform will allow AC Modules which have been marked as pre-production to be used to launch the MLE.
  Note: Setting the bit to 1 will require a pre-production SINIT AC module and will result in PCRs 17 and 18 being capped with a random value.
- Bit 0 Identifies whether a hash of an unsigned policy is extended into PCR 17

D.1.1.2 LCP Policy Data

The purpose of the *LCP_POLICY_DATA* structure is to provide the additional data needed to enforce the policy but in a separate entity that doesn't have to consume TPM NV space. The *LCP_POLICY_DATA* structure contains an unsigned policy data (indicated by *LCP_POLICY::PolicyType*); the structures for which are shown in Figure 7. *_POLICY_DATA Structures and a full description of *LCP_POLICY_DATA* can be found in section D.6.2.

Figure 7. *_POLICY_DATA Structures



The *LCP_UNSIGNED_POLICY_DATA::PolicyTable* structure allows the object to contain a number of lists which identify either:

- the types of measured environment that can be launched, or
- the states the platform may be in – as recorded in the TPM by the Static chain of Trust.

D.1.2 Policy List Types

D.1.2.1 MLE List

When processing a measured environment list the LCP engine computes the hash (using the *HashAlg* identified in *LCP_POLICY*) over the environment to be launched and compares it to the hash values in the MLE list. If a match can be found, then the LCP engine proceeds.

The following rules apply to the Measured Launched Environment List:

- There can only be one MLE list within the *LCP_POLICY_DATA* structure
- Each measured environment must be enumerated using the same cryptographic hashing algorithm specified by *HashAlg* in *LCP_POLICY*. This algorithm must be supported by SINIT.
- All measurements are of the environment to be launched, as measured by SINIT, not the value of the PCR after it's been extended.

D.1.2.2 Platform Configuration List

When processing the platform configuration list the LCP engine reads the appropriate PCR's as defined by the first TPM*_PCR_INFO_SHORT* value in the list and concatenates them and cryptographically hashes them together. The result is compared to the hash value in the *TPM_PCR_INFO_SHORT*. If there is no match this process is repeated for each and every member of the list. As soon as a match is found, the LCP engine proceeds.

The follow rules apply to the Platform configuration List:

- There can only be one platform configuration list within the *LCP_POLICY_DATA* structure
- Each platform configuration in the list must be enumerated as a *TPM_PCR_INFO_SHORT* structure.

Additionally, it is recommended, although not necessary, that all *TPM_PCR_INFO_SHORT* structures in the platform configuration list test the same set of PCR values.

D.1.3 Supported Cryptographic Algorithms

The following algorithms are defined for the current version of the Launch Control Policy:

Hashing – SHA-1

It is the responsibility of the policy author to ensure that their policy uses an algorithm supported by the version of the SINIT AC module being used. If the policy specifies an unsupported algorithm, the policy will fail and the environment will not be permitted to launch.

D.2 Policy Engine Logic

The logic above means that all the records (i.e., an *LCP_POLICY_LIST* structure) in the *PolicyTable* must evaluate to true. However, a record may represent a list which contains multiple possible hash values only one of which must match in order for the record to evaluate true..

In a very simple case where the *LCP_POLICY_LIST* is a simple MLE list, the logic would be as follows:

```
MLEList:          MLE1     or MLE2 or MLE3
```

The measured environment can be launched if, and only if, the hash of the measured environment matched any one of the listed MLE1, MLE2 or MLE3.

While a more complex set of policies could contain both an MLE list and a platform configuration list. In this case the logic would be as follows:

```
MLEList:          MLE1 or MLE2 or MLE3
                     AND
PltConfList:      PCONF1 or PCONF2 or PCONF3
```

Here, the measured environment can be launched if, and only if, the hash of the measured environment matches one the values MLE1, MLE2 or MLE3 and the values of the TPM PCRs at the time SINIT is executed matches one of the values PCONF1, PCONF2, or PCONF3.

If either the MLE or the PCONF fails to match any of its respective hash values, an error code is written to the TXT.ERRORCODE register and a TXT Shutdown is triggered.

D.2.1 Combining Policies

When both the Platform Supplier and the Platform Owner have established policies on the platform, the two policies are combined to give a resultant policy which the LCP Policy Engine will enforce.

D.3 Measuring the Enforced Policy

The LCP engine will extend to PCR 17 a hash value which represents the policy against which the environment was launched. This hash value is determined by the rules in the following sections and in some cases the settings in the relevant *LCP_POLICY::PolicyControlField*, these relate to the possible combinations:

- *No Policy* – The platform has no configured policy.
- *Allow Any Policy* – The selected policy allows any MLE to launch.
- *LCP Policy Hash Only* – The selected policy only uses a single hash stored in the LCP_POLICY
- *Unsigned LCP_POLICY_DATA* – The selected policy uses an unsigned LCP_POLICY_DATA structure
- *Force Owner Policy* – The Platform Default policy requires that a Platform Owner's Policy is present on the platform.

As a matter of integrity the *LCP_POLICY::PolicyControlField* field will always be extended into PCR 17.

D.3.1 No Policy Data

When no LCP Policy is executed, 20 bytes of 0s will be extended to PCR 17 to prevent any Measured Launched Environment from later extending a value of any other policy into the PCR 17. As PCR 17 will be open to Locality 2 extends, the Measured Launched Environment would gain access to sealed secrets which were sealed against some known policy by another Measured Launched Environment.

D.3.2 LCP Policy Allow Any

When the LCP_POLICY indicates that the Policy Type is ALLOW ANY, 20 bytes of 0s will extended to PCR 17 to prevent any Measured Launched Environment from later extending a value of any other policy into the PCR 17.

D.3.3 LCP Policy Hash Only

When the LCP_POLICY indicates that the *HashValue* is to be used as the only measurement to be compared against, and that comparison is successful then this value is extended to PCR 17.

D.3.4 Unsigned LCP_POLICY_DATA

When the *LCP_POLICY* indicates that the *LCP_POLICYDATA* object associated with the policy is unsigned, the *LCP_POLICY_DATA* may contain unsigned MLE & Platform Configuration lists. The value extended into PCR 17 will not simply be the hash of the list. The LCP engine will generate a hash value for each list present within *LCP_POLICY_DATA* and hash them together and extend this value. The hash for a list is calculated by hashing the entire list. When Bit 0 of the LCP_POLICY::PolicyControlField is set, then the hash value for unsigned LCP_POLICY_DATA is extended into PCR 17; when it is clear then it is not.

D.3.5 Force Platform Owner Policy

When the Platform Default policy indicates that the Platform Owner policy must be present then the value of the Platform Owner policy will be extended in PCR 17.

D.4 Revocation

D.4.1 SINIT Revocation

LCP Structures also enable a limited self-revocation mechanism for SINIT. This is supported in the LCP_POLICY structure in the *SINITRevocationCounter* field. This field is a UINT8 and corresponds to a similar field within the SINIT image. When the value of the *LCP_POLICY::SINITRevocationCounter* field is less than or equal to the value of AcmVersion in the SINIT image then SINIT may proceed, otherwise SINIT shall cause a TXT Shutdown condition with the value LCP_SINIT_REVOKED in the TXT.ERRORCODE register.

D.5 Platform Owner Index

The Platform Owner policy index is intended to represent the policy defined by the owner of the platform, and as such should be provisioned with access control permissions that enforce that control over the policy remains with the platform owner.

The following attribute settings are required:

Index Value:	0x40000001
Size:	34 bytes
Read Locality:	3 (can be others as well)
Read Auth:	None
PCR Read:	None

The simplest access control setting that maintains platform owner control is to set the Write Auth to Owner. However, this also means that updating the policy requires the owner authorization. As the owner authorization can be used for almost all TPM management operations, it may be desirable to limit its use.

Another possibility would be to use a separate authorization just for this index. That would eliminate having to provide the owner authorization just to update the policy.

If trusted software, running in the context of the MLE or with access to TPM locality 2, needs to be able to update the policy, then it would be possible to have no Write Auth but set the Write Locality to 2. This would permit whatever software was able to successfully launch to update the policy.

Note that it is not advisable to use PCR Write controls, since it would mean that the specified PCR could not change over time (e.g. if the software measured into it was upgraded). This is because the index attributes cannot be changes once the index is created.

D.6 Data Structures

D.6.1 LCP_POLICY

Both the Platform Owner and Platform Default SINIT policy structures are of the same type, i.e. *LCP_POLICY*. These objects are stored in the TPM. The required fields for *LCP_POLICY* are as follows:

```
#define POLHALG_SHA1                0

#define POLTYPE_HASHONLY            0
#define POLTYPE_UNSIGNED            1
#define POLTYPE_ANY                 3
#define POLTYPE_FORCEOWNERPOLICY    4

typedef struct {
    UINT8                   Sha1[20];
} LCP_HASH;

typedef struct {
    UINT8                   Version;
    UINT8                   HashAlg;                 // one of POLHALG_*
    UINT8                   PolicyType;              // one of POLTYPE_*
    UINT8                   SINITRevocationCounter;
    UINT32                  PolicyControlField;
    UINT16                  Reserved[3];
    LCP_HASH                PolicyHash;
 } LCP_POLICY;

LCP_POLICY       PlatformOwnerLCPPolicy;
LCP_POLICY       PlatformDefaultLCPPolicy;
```

HashAlg not only determines the algorithm and size of the hash used for *PolicyHash* but all other values used with the LCP structures. If the algorithm type is not supported by SINIT ACM then LCP shall cause an Intel TXT shutdown condition with the value LCP_UNKNOWN_POLICY_TYPE in the TXT.ERRORCODE register.

If the type specified in the *PolicyType* field is not supported by SINIT then LCP shall cause an Intel TXT shutdown condition with the value LCP_UNKNOWN_POLICY_TYPE in the TXT.ERRORCODE register.

The *SINITRevocationCounter* field provides a mechanism for determining whether the loaded version of SINIT ACM should be used. This value must be less than or equal to a corresponding value, AcmVersion, in the SINIT Image to determine whether SINIT can be used.

The *PolicyControlField* provides a way for the policy authority to control some of the logic in the LCP Policy Engine. The field is defined as:

- *Bits31:3 Reserved and should set to zero*
- *Bit 2 Identifies whether the OsSinitData.Capabilities field will be extended into PCR 17 (1).*
- *Bit 1 Identifies whether the platform will allow Authenticated Code Modules which have been marked as pre-production can be used to launch the MLE.*
  Note: Setting the bit to 1 will require a pre-production SINIT ACM and will result in PCRs 17 and 18 being capped with a random value.
- *Bit 0 Identifies whether a hash of an unsigned policy is extended into PCR 17*

The *PolicyControlField* must always be reported into PCR 17.

The version of the LCP_POLICY structure defined here is 1.

D.6.2 LCP_POLICY_DATA

For each policy on the platform there must exist a block of data which the SINIT policy engine will process. Where this data resides on the platform is platform dependent, but it must provisioned into the Intel TXT heap data structures (see Appendices C.1 and C.3) by the pre-GETSEC[SENTER] environment.

```
typedef struct {
    UUID                         Uuid;
    LCP_UNSIGNED_POLICY_DATA  UnsignedData;
} LCP_POLICY_DATA
```

Where:

```
typedef struct {
    UINT32    data1;
    UINT16    data2;
    UINT16    data3;
    UINT16    data4;
    UINT8     data5[6];
} UUID;
```

The UUID value for LCP will be:

```
AB0D1925-EEE7-48eb-A9FC-0BAC5A262D0E or
{ 0xab0d1925, 0xeee7, 0x48eb, 0xa9fc { 0xb, 0xac, 0x5a,
0x26, 0x2d, 0xe }
```

D.6.3 LCP_UNSIGNED_POLICY_DATA

Unsigned policy data is a table of elements which are lists of policy values (all of the same type), LCP_POLICY_LIST. The version of the LCP_UNSIGNED_POLICY_DATA defined here is 0H.

```
typedef struct {
    UINT8            Version;
    UINT8            PolicyTableSize;
    LCP_POLICY_LIST  PolicyTable[PolicyTableSize];
} LCP_UNSIGNED_POLICY_DATA
```

D.6.3.1 LCP_POLICY_LIST

The least significant 15 bits identifies the type of list.

```
#define POLDESC_MLE_UNSIGNED     0x0001

#define POLDESC_PConf_UNSIGNED   0x0002

typedef struct {
    UINT16              ListType;          // One of POLDESC_*
    LCP_UNSIGNED_LIST   UnsignedList;
} LCP_POLICY_RECORD
```

D.6.3.2 LCP_UNSIGNED_LIST

Unsigned lists are either a list of the environments which may be launched or a list of the allowable platform configurations.

```
typedef struct {
       UINT8    Version;
       UINT8    ListSize;
       choice {
           LCP_HASH             HashList[ListSize];
           TPM_PCR_INFO_SHORT  PCRInfoList[ListSize]];
       };
} LCP_UNSIGNED_LIST;
```

HashAlg identifies the hashing algorithm and hence the size of each hash; *ListSize* identifies either the number of hashes, or the number of TPM_PCR_INFO_SHORT structures (see the TPM Main Specification Version 1.2 for its full definition), in the list. *Version* is 0H for the structure defined here.

The various TPM_* structures have been copied below to facilitate understanding of the list structure.

```
typedef struct tdTPM_PCR_INFO_SHORT{
    TPM_PCR_SELECTION          pcrSelection;
    TPM_LOCALITY_SELECTION     localityAtRelease;
    TPM_COMPOSITE_HASH         digestAtRelease;
} TPM_PCR_INFO_SHORT;
```

```
typedef struct tdTPM_PCR_SELECTION {
    UINT16                              sizeOfSelect;
    [size_is(sizeOfSelect)] BYTE        pcrSelect[];
} TPM_PCR_SELECTION;

#define TPM_LOCALITY_SELECTION BYTE    // each bit is the
                                       // corresponding locality

typedef struct tdTPM_PCR_COMPOSITE {
    TPM_PCR_SELECTION                      select;
    UINT32                                 valueSize;
    [size_is(valueSize)] TPM_PCRVALUE      pcrValue[];
} TPM_PCR_COMPOSITE;

typedef struct tdTPM_DIGEST{
    BYTE digest[digestSize];
} TPM_DIGEST;
typedef TPM_DIGEST  TPM_COMPOSITE_HASH;   // hash of
                                          // TPM_PCR_COMPOSITE
                                          // object
typedef TPM_DIGEST  TPM_PCRVALUE;
```

D.6.4 Structure Endianness

Endianness deals with the sequencing order of stored bytes. There are two common sequencing orders; Little Endian (format used by Intel) and Big Endian (in this case, TPM NVRAM). In order to successfully compare the values of these two formats, we need to convert TPM NVRAM data to Little Endian when comparing within Intel code as well as converting data into Big Endian format when writing to TPM NVRAM space.

Example 36 bit value: 0x12345678

Little Endian (from least significant byte to most significant byte):

0x78, 0x56, 0x34, 0x12

Big Endian (from least significant byte to most significant byte):

0x12, 0x34, 0x56, 0x78

Appendix E LCP v2 Data Structures

E.1 LCP_POLICY

Both the Platform Owner and Platform Supplier policy structures are of the type *LCP_POLICY*. These objects are stored in the TPM NV. The required fields for *LCP_POLICY* are as follows:

```
#define LCP_POLHALG_SHA1          0

#define LCP_POLTYPE_LIST          0
#define LCP_POLTYPE_ANY           1

typedef struct {
    UINT8               Sha1[20];
} LCP_HASH;

#define LCP_MAX_LISTS             8

typedef struct {
    UINT16              Version;
    UINT8               HashAlg;              // one of LCP_POLHALG_*
    UINT8               PolicyType;           // one of LCP_POLTYPE_*
    UINT8               SINITMinVersion;
    UINT8               Reserved1;
    UINT16              DataRevocationCounters[LCP_MAX_LISTS];
    UINT32              PolicyControl;
    UINT32              Reserved2[2];
    LCP_HASH            PolicyHash;
 } LCP_POLICY;
```

E.2 LCP_POLICY_DATA

For each policy of type LCP_POLTYPE_LIST, there must exist a block of data which the SINIT policy engine will process. Where this data resides on the platform is platform dependent, but it must provisioned into the Intel TXT heap data structures (see Appendices C.1 and C.3) before executing the GETSEC[SENTER] instruction. While not required, it is recommended that software place the LCP_POLICY_DATA on a 4-byte aligned boundary to reduce access alignment penalties.

```
typedef struct {
    char             FileSignature[32];
    UINT8            Reserved[3];
    UINT8            NumLists;
    LCP_POLICY_LIST  PolicyLists[NumLists];
} LCP_POLICY_DATA
```

FileSignature is the string “Intel(R) TXT LCP_POLICY_DATA\0\0\0\0”, where ‘\0’ is a single byte whose value is 0x00. This field is intended for use by software that needs to determine if a given file is an LCP_POLICY_DATA file.

The *Reserved* field must be set to all 0s.

The *NumLists* field must be less than or equal to LCP_MAX_LISTS.

Each list in *PolicyLists* may be either signed or unsigned.

E.3 LCP_POLICY_LIST

```
#define LCP_POLSALG_NONE 0
#define LCP_POLSALG_RSA_PKCS_15         1

typedef struct {
    UINT16       RevocationCounter;
    UINT16       PubkeySize;
    UINT8        PubkeyValue[PubkeySize];
    UINT8        SigBlock[PubkeySize];
} LCP_SIGNATURE;
```

The *RevocationCounter* field is a monotonically increasing value that can be used, in conjunction with the corresponding index of the *DataRevocationCounters* field in LCP_POLICY, to provide a method of revoking (or preventing rollback of) signed policies.

Supported public key sizes are 1024, 2048 and 3072 bits. It is recommended that a public key size of at least 2048 bits be used. Larger sizes may take longer to verify.

The exponent is fixed and must be 65537.

As specified for all policy data, both the PubkeyValue and SigBlock must be in little-endian byte order. This may require tools that generate policies to reverse the byte order of keys and signatures produced by tools that use the ASN.1/big-endian format.

```
typedef struct {
    UINT16                    Version;
    UINT8                     Reserved;
    UINT8                     SigAlgorithm;       // one of SIGALG_*
    UINT32                    PolicyElementsSize;
    LCP_POLICY_ELEMENT        PolicyElements[];
    optionally LCP_SIGNATURE  Signature;
} LCP_POLICY_LIST;
```

The *Reserved* field must be set to 0.

PolicyElementsSize specifies the size (in bytes) of all of the LCP_POLICY_ELEMENTS structures in the object. It may be 0. A LCP_POLICY_LIST with no elements can be used as a “placeholder” signed list that can be updated at runtime with the actual signed data but without having to re-provision the LCP_POLICY in TPM NV.

If *SigAlgorithm* is SIGALG_RSA_PKCS_15 then the *Signature* field must be present (else it must not). For a signed list, the RSA signature will be calculated over the entire LCP_POLICY_LIST structure, including the *Signature* member, except for the *SigBlock* field.

The version of the LCP_POLICY_LIST structure defined here is 1.0 (100H).

E.4 LCP_POLICY_ELEMENT

```
typedef struct {
    UINT32     Size;
    UINT32     Type;
    UINT32     PolEltControl;
    UINT8      Data[Size – 12];
} LCP_POLICY_ELEMENT;
```

The structures in the following sub-sections correspond to the contents of the *Data* field for the specific type of element.

While not required, it is recommended that *Size* be a 4-byte multiple.

E.4.1 LCP_MLE_ELEMENT

```
#define LCP_POLELT_TYPE_MLE           0

typedef struct {
    UINT8                SINITMinVersion;
    UINT8                HashAlg;              // one of LCP_POLHALG_*
    UINT16               NumHashes;
    LCP_HASH             Hashes[NumHashes];
} LCP_MLE_ELEMENT;
```

The LCP_MLE_ELEMENT represents a list of the acceptable MLEs, as measured by their hashes. An MLE will match the policy if its hash (as calculated when traversing its pages in the MLE page table; not the value of PCR 18 after it has been extended) matches any hash within the list.

SINIT will use the largest of the *SINITMinVersion* fields (the one in LCP_POLICY and the one in the LCP_MLE_ELEMENT which contains the matching MLE hash) to determine the minimum allowable version of SINIT.

HashAlg specifies the hash algorithm to use when measuring the MLE and also of the values in *Hashes[]*.

If *NumHashes* is 0 then this element will evaluate to false for all MLEs.

E.4.2 LCP_PCONF_ELEMENT

```
#define LCP_POLELT_TYPE_PCONF         1

typedef struct {
    UINT16                     NumPCRInfos;
    TPM_PCR_INFO_SHORT         PCRInfos[NumPCRInfos];
} LCP_PCONF_ELEMENT;
```

The LCP_PCONF_ELEMENT represents a list of acceptable PCR values on the platform at the time of launch. The platform will satisfy the policy if the PCR values at the time of launch match any of the *PCRInfos* within the list. When processing the platform configuration list the LCP engine reads the appropriate PCR's as defined by the first TPM*_PCR_INFO_SHORT* value in the list and concatenates them and cryptographically hashes them together. The result is compared to the hash value in the *TPM_PCR_INFO_SHORT*. If there is no match this process is repeated for each and every member of the list. As soon as a match is found, the LCP engine proceeds.

Additionally, it is recommended, although not necessary, that all *TPM_PCR_INFO_SHORT* structures in the platform configuration list test the same set of PCR values.

If *NumPCRInfos* is 0 then this element will evaluate to false for all platform configurations.

The various TPM_* structures have been copied below to facilitate understanding of the list structure.

```
typedef struct tdTPM_PCR_INFO_SHORT{
    TPM_PCR_SELECTION              pcrSelection;
    TPM_LOCALITY_SELECTION         localityAtRelease;
    TPM_COMPOSITE_HASH             digestAtRelease;
} TPM_PCR_INFO_SHORT;

typedef struct tdTPM_PCR_SELECTION {
    UINT16                             sizeOfSelect;
    [size_is(sizeOfSelect)] BYTE       pcrSelect[];
} TPM_PCR_SELECTION;

#define TPM_LOCALITY_SELECTION BYTE    // each bit is the
                                       // corresponding locality

typedef struct tdTPM_DIGEST{
    BYTE digest[digestSize];
} TPM_DIGEST;
typedef TPM_DIGEST  TPM_COMPOSITE_HASH;   // hash of
                                          // TPM_PCR_COMPOSITE
                                          // object
```

E.4.3 LCP_SBIOS_ELEMENT

```
#define LCP_POLELT_TYPE_SBIOS         2

typedef struct {
    UINT8                HashAlg;         // one of LCP_POLHALG_*
    UINT8                Reserved1[3];
    LCP_HASH             FallbackHash;
    UINT16               Reserved2;
    UINT16               NumHashes;
    LCP_HASH             Hashes[NumHashes];
} LCP_SBIOS_ELEMENT;
```

The LCP_SBIOS_ELEMENT represents a list of the acceptable startup BIOSes (that portion of BIOS measured by the BIOS ACM), as measured by their hashes.

HashAlg specifies the hash algorithm to use when measuring the startup BIOS and also of the values in *Hashes[]* and *FallbackHash*.

The *FallbackHash* field represents a version of the startup BIOS that is always acceptable, regardless of the contents of the *Hashes[]* field or lack thereof. The expected use is for OEMs that do not wish to sign their BIOS (i.e. not sign the LCP_POLICY_LIST for the Supplier Policy) and thus will use auto-promotion for their startup BIOS policy. In that case, they would specify *NumHashes* as 0 and set a valid *FallbackHash* to correspond to their fallback or recovery BIOS. This hash would be valid regardless of the current value of the auto-promotion hash. If not used (i.e. the OEM is providing the hash in the *Hashes[]* field) then it can be set to anything.

If *NumHashes* is not 0 then the current startup BIOS must be specified by one of the hashes in *Hashes[]* or by the *FallbackHash* otherwise the BIOS ACM will fail to boot that processor. If *NumHashes* is 0 then the BIOS ACM will use the auto-promotion method for startup BIOS policy.

The *Reserved1* and *Reserved2* fields must be set to 0.

E.4.4 LCP_CUSTOM_ELEMENT

```
#define LCP_POLELT_TYPE_CUSTOM        3

typedef struct {
    UINT32    data1;
    UINT16    data2;
    UINT16    data3;
    UINT16    data4;
    UINT8     data5[6];
} UUID;

typedef struct {
    UUID      Uuid;
    UINT8     Data[];
} LCP_CUSTOM_ELEMENT;
```

The LCP_CUSTOM_ELEMENT allows for users, ISVs, IT, etc. to define policy-related data which can then be carried as part of a policy and interpreted by user/ISV/IT software. Because the data is contained within a policy, its integrity will be verified by SINIT as part of policy processing.

Uuid is a UUID value that uniquely identifies the format of the *Data* field. This field will be used by all custom software that may have its own policy data. It is thus important to generate the UUID value using a process that will provide a statistically unique value.

The *Data* field's contents are defined by the entity which "owns" the UUID of the element. The size of this data must be included within the size of the LCP_POLICY_ELEMENT::Size field.

E.5 Structure Endianness

Endianness deals with the sequencing order of stored bytes. There are two common sequencing orders: Little Endian (format used by Intel) and Big Endian. All structures and data are in Little Endian format, even LCP_POLICY.

Appendix F Processor and System State after GETSEC[SENTER] and RLP Wakeup

The following table describes the processor state of the ILP after returning to the MLE from GETSEC[SENTER] and the RLPs after waking from SENTER. This will be the state seen by the MLE.

Table 17. Processor State after SENTER and RLP Wakeup

Register state	ILP after GETSEC[SENTER]	RLP after wakeup
CR0	PG←0, AM←0,WP←0; others unchanged	PG←0, CD←0, NW←0, AM←0, WP←0; PE←1, NE←1
CR4	00004000H	00004000H
EFLAGS	000000XXH (XX = Undefined)	000000XXH (XX = Undefined)
EIP	[MLEHeader.EntryPoint]	[TXT.MLE.JOIN + 12]
ESP	Undefined	Undefined
EBP	Undefined	Undefined
ECX	Ptr to MLE page table[1]	Undefined
EBX	[MLEHeader.EntryPoint]	Undefined
EAX, EDI, ESI	Undefined	Undefined

CS	Sel=[SINIT.SegSel], base=0, limit=FFFFFH,
	G=1, D=1, AR=9BH
		Sel=[TXT.MLE.JOIN + 8], base=0, limit=FFFFFH,
		G=1, D=1, AR=9BH

DS, ES, SS	Undefined	Sel=[TXT.MLE.JOIN + 8] + 8, base=0, limit=FFFFFH, G=1, D=1, AR=93H
GDTR	Base=SINIT.Base (EBX) + [SINIT.GDTBasePtr], Limit=[SINIT.GDTLimit]	Base=[TXT.MLE.JOIN + 4], Limit=[TXT.MLE.JOIN]
DR7	00000400H	00000400H
IA32_DEBUGCTL MSR	0H	0H
IA32_EFER MSR	0	0
IA32_MISC_ENABLE MSR	IA32_MISC_ENABLE & FFF37CEAH[2][3]	IA32_MISC_ENABLE & FEE324A8H[3]

[1] If bit 2 of the Capabilities field in SINIT's Chipset AC Module Information Table is set then ECX will contain the pointer to the MLE's page table. If clear, the contents of ECX are undefined.
[2] Bit 3 (thermal monitor enable) will be set to 1 if it was previously clear.
[3] Bit 18 (MONITOR/MWAIT enable) will be set to 1 if it was previously clear, when bit 1 of OsSinitData.Capabilities (use of MONITOR for RLP wakeup) is set.

Performance counters and counter control registers	0H	0H
IA32_APIC_BASE MSR	35:12 cleared to 0FEE00H	35:12 cleared to 0FEE00H, bit 8 (BSP) cleared to 0

The TPM will not have any locality active following SENTER.