# INTEL® ARCHITECTURE MEMORY ENCRYPTION TECHNOLOGIES SPECIFICATION

Ref: #336907-001US

Rev: 1.1

December 2017

# Disclaimers

# Terminology

1. TME – Total Memory Encryption - this is a base line capability for memory encryption with a single ephemeral key.
2. MKTME – Add support to use multiple keys for page granular memory encryption with additional support for software provisioned keys.

# 1 Introduction

*This document is a work in progress and is subject to change based on customer feedback and internal analysis. This document does not imply any product commitment from Intel to anything in terms of features and/or behaviors.*

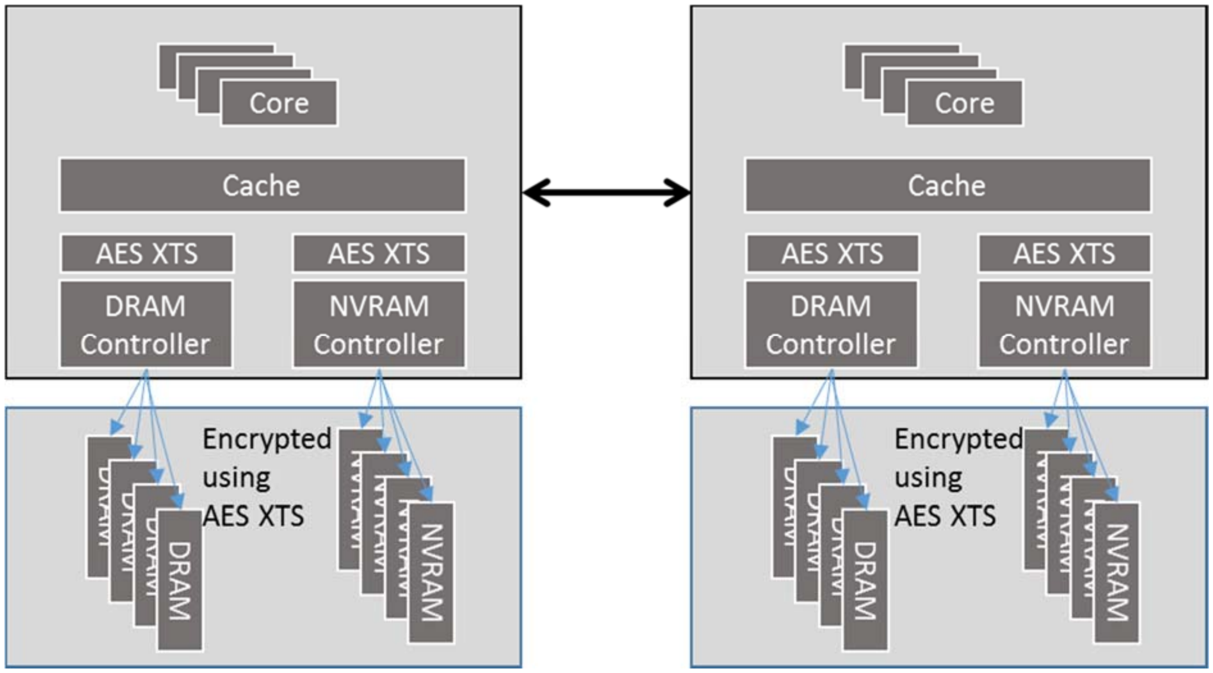
This document describes the memory encryption support targeting future Intel processors. Note that Intel platforms supports many different types of memory and not all SOC would support this capability for all types of memory. Initial implementation is likely to focus on traditional DRAM and NVRAM.

Total Memory Encryption (TME) – as name would imply is a capability to encrypt entirety of physical memory of a system. This capability is typically enabled in very early stages of boot process with small change to BIOS and once configured and locked will encrypt all the data on external memory buses of an SOC using NIST standard AES-XTS algorithm with 128-bit keys. The encryption key used for TME uses hardware random number generator implemented in Intel SOC and the keys are not accessible by software or using external interfaces to Intel SOC. TME capability is intended to provide protections of AES-XTS to external memory buses and DIMMs. The architecture is flexible and will support additional memory protections schemes in future. This capability when enabled is intended to support (unmodified) existing system and application software. Overall performance impact of this capability is likely to be relatively small and is highly dependent on workload.

Multi-Key Total Memory Encryption (MKTME) builds on TME and adds support for multiple encryption keys. The SOC implementation will support a fixed number of encryption keys, and software can configure SOC to use a subset of available keys. Software manages the use of keys and can use each of the available key for encrypting any page of the memory. Thus, MKTME allows page granular encryption of memory. By default MKTME uses TME encryption key unless explicitly specified by software. In addition to supporting CPU generated ephemeral key (not accessible by software or using external interfaces to SOC), MKTME also supports software provided keys. Software provided keys are particularly useful when used with non-volatile memory or when combined with attestation mechanisms and/or used with key provisioning services. In virtualization scenario, we anticipate VMM or hypervisor to manage use of keys to transparently support legacy operating systems without any changes (thus, MKTME can also be viewed as TME virtualization in such deployment scenario). An OS may be enabled to take additional advantage of MKTME capability both in native or virtualized environment. When properly enabled, MKTME is available to each guest OS in virtualized environment, and guest OS can take advantage of MKTME in same was as native OS.

# 2 Introduction to Total Memory Encryption (TME)

The diagram below shows basic idea behind total memory encryption in a two socket configuration. Actual implementation may vary.



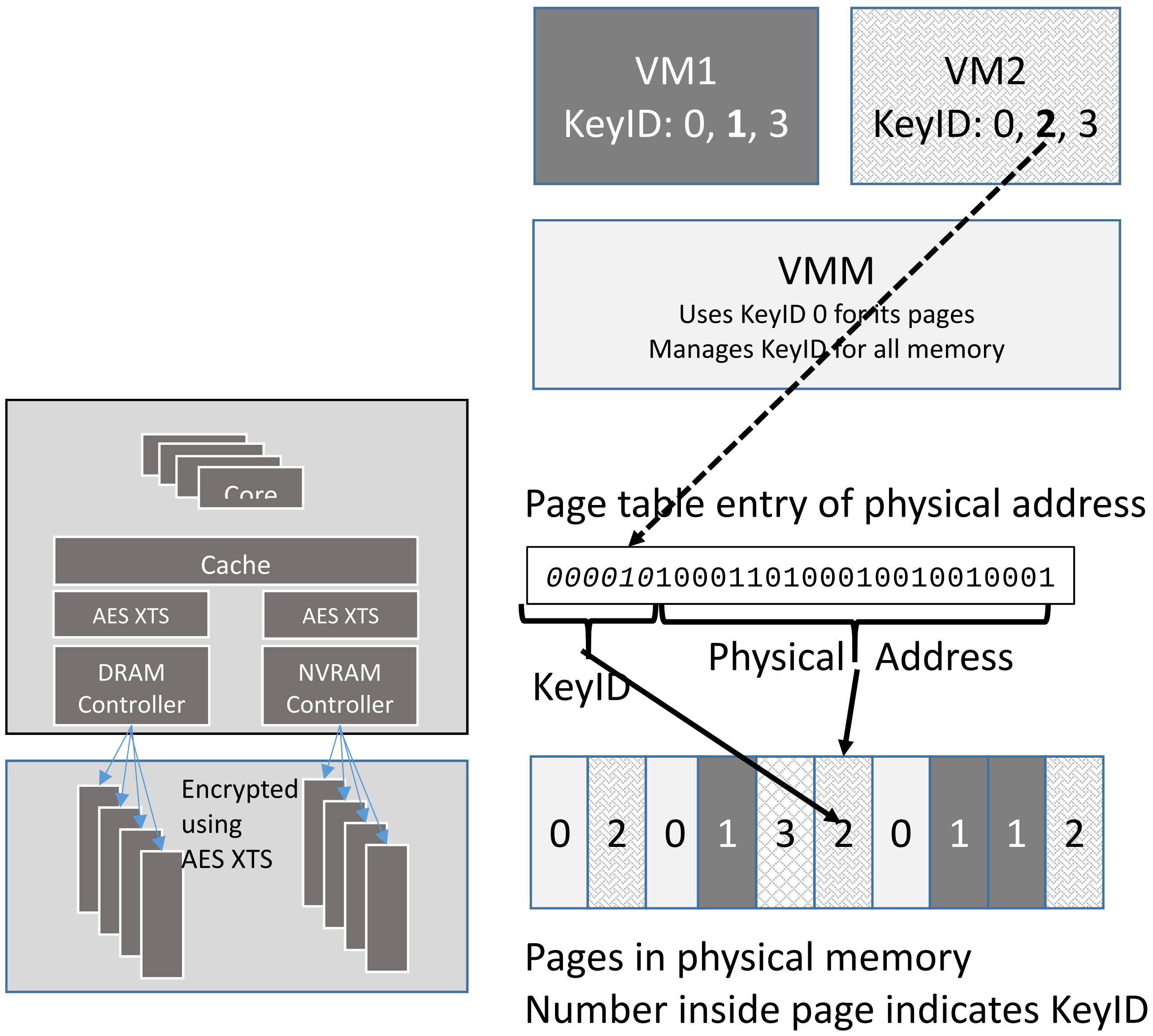
AES XTS encryption engine is in the direct data path to external memory buses and therefore, all the memory data entering and/or leaving SOC on memory buses is encrypted using AES XTS. The data inside the SOC (in caches, etc.) remains plain text and therefore, supports all the existing software and I/O models.

In a typical deployment, the encryption key is generated by the CPU and therefore, is not visible to the software. When the system is configured with NVRAM, if the NVRAM is to be treated as DRAM, the it can also use CPU generated keys. However, if NVRAM were to be treated as non-volatile memory, there is an option to be able to have same key generated/reused across platform power cycles/reboots.

# 3 Introduction to Multi-Key Total Memory Encryption (MKTME)

## 3.1 High-level Architecture

The highlevel architecture of MKTME is shown in the figure below.



The figure above shows basic architecture of MKTME which shares basic hardware architecture with TME with exception that AES XTS now supports multiple keys. To the right the figure shows use of MKTME in a virtualized environment though architecture support use of MKTME in a native OS deployment scenario as well. In this example, we show one hypervisor/VMM and two VMs. By default hypervisor uses KeyID 0 (same as TME) though it can use a different keyID for its own memory as well. VM1 uses KeyID1 for its own private pages and VM2 is using KeyID 2 for its own private pages. In addition, VM1 can always use KeyID 0 (TME KeyID) for any page and is also opting to use KeyID 3 for shared memory between itself and VM2. The KeyID is included in Page Table Entry as upper bits of physical address field. As in this example, KeyID 2 is shown. The remainder of the bits in physical address field are used to actually address bits in the memory. The figure shows one possible page assignment along with KeyID for illustration purpose though in this case hypervisor has full freedom to be able to use any KeyID with any pages for itself or any of its guest VM. Note that the idea of oversubscribing physical address bits in the page table extends to other pages tables as well including IA page tables and IOMMU page tables. The KeyID remains part of physical address bits everywhere in SOC, with exception of tweak for AES XTS and on external memory buses. KeyID is not used outside of SOC or in tweak for AES XTS.

# 4 TME & MKTME: Enumeration and Control Registers

This document is applicable to only to CPUs that enumerate TME and/or, MKTME capabilities.

## 4.1 Enumeration

TME and MKTME capability is exposed to the BIOS/Software via MSR described in this section. The Max number of keys available/supported in the processor for MKTME are enumerated. BIOS will need to activate this capability via an MSR (described later) and it must select the number of keys to be supported/used for MKTME during early boot process. Upon activation, all memory (except in TME Exclusion range) attached to CPU/SoC is encrypted using AES-XTS 128 bit ephemeral key (platform key) that is generated by the CPU on every boot.

### 4.1.1 TME

CPUID.TME (CPUID.7.0.ECX.13) enumerates the existence of the 4 (architectural) MSRs and their MSR addresses:

IA32_TME_CAPABILITY – 981H

IA32_TME_ACTIVATE – 982H

IA32_TME_EXCLUDE_MASK – 983H

IA32_TME_EXCLUDE_BASE – 984H

### 4.1.2 Multi-key TME

The CPUID.TME bit indicates the presence of the TME_CAPABILITY MSR, and that MSR will further enumerate the TME characteristics as well as the MK-TME availability and characteristics. MKTME is enabled/configured by BIOS using the IA32_TME_ACTIVATE MSRs. MKTME requires TME and therefore, cannot be enabled without enabling TME.

### 4.1.3 Memory Encryption Capability MSR

IA32_TME_CAPABILITY MSR – 981H

| Register Address | Architectural MSR Name and bit fields (Former MSR Name) | MSR/Bit Description | Comment |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 981H | IA32_TME_CAPABILITY MSR | Memory Encryption Capability MSR | One MSR for TME and MKTME |
| | 0 | Suports for AES-XTS 128 bit encryption algorithm | NIST standard |
| | 15:1 | [Reserved] | For additional encryption algorithms |
| | 23:16 | [Reserved] | |
| | 31:24 | [Reserved] | For future TME usage |
| | 35:32 | MK_TME_MAX_KEYID_BITS<br><br>Number of bits which can be allocated for usage as key identifiers for multi-key memory encryption.<br><br>Zero if MKTME is not supported | 4 bits allows for a max value of 15, which could address 32K keys |
| | 50:36 | MK_TME_MAX_KEYS<br><br>Indicates the maximum number of keys which are available for usage.<br><br>This value may not be a power of 2.<br><br>This maximum value of this field will be (2^MK_TME_MAX_KEYID_BITS)-1<br><br>Zero if MKTME is not supported | KeyID 0 is reserved for TME and this number does not include the TME key.<br><br>Max value is 32K-1 keys |
| | 63:51 | [Reserved] | |

### 4.1.4 Note on CPUID reporting of MAX_PA_WIDTH

CPUID enumeration of MAX_PA_WIDTH (leaf 80000008.EAX) is unaffected by MKTME activation and will continue to report the maximum number of physical address bits available for software to use, irrespective of the number of KeyID bits.

## 4.2 Memory Encryption Configuration and Status Registers

### 4.2.1 ACTIVATION MSR

IA32_TME_ACTIVATE MSR – 982H

This MSR is used to lock the following MSRs. Any write to the following MSRs will be ignored after they are locked. The lock is reset when CPU is reset

IA32_TME_ACTIVATE

IA32_TME_EXCLUDE_MASK

IA32_TME_EXCLUDE_BASE

Note that IA32_TME_EXCLUDE_MASK and IA32_TME_EXCLUDE_BASE expected to be configured before IA32_TME_ACTIVATE

To enable MKTME, TME Enable RWL bit in IA32_TME_ACTIVATE MSR must be set and bits 35:32 must have a non-zero value (which will specify the number of KeyID bits configured for MKTME).

| Register Address | Architectural MSR Name and bit fields (Former MSR Name) | MSR/Bit Description | Comment |
|---|---|---|---|
| 982H | IA32_TME_ACTIVATE MSR | | |
| | 0 | Lock RO – Will be set upon successful WRMSR (Or first SMI); written value ignored. | |
| | 1 | TME Enable RWL – Enable Total Memory encryption using CPU generated ephemeral key based on hardware random number generator | This bit also enables & locks MKTME, MKTME cannot be enabled without enabling TME |
| | 2 | Key select<br><br>0 – Create a new TME key (expected cold/warm boot) | |

| | | | |
|---|---|---|---|
| | | 1- Restore the TME key from storage (Expected when resume from standby) | |
| | 3 | Save TME key for standby – Save key into storage to be used when resume from standby | May not be supported in all CPUs |
| | 7:4 | TME policy/encryption algorithm<br><br>Only algorithms enumerated in IA32_TME_CAPABILITY are allowed<br><br>For example: 0000 – AES-XTS-128<br><br>Other values are invalid | TME Encryption algorithm to be used |
| | 31:8 | Reserved | |
| | 35:32 | Reserved if MKTME is not enumerated | |
| | | MK_TME_KEYID_BITS<br><br>The number of key identifier bits to allocate to MKTME usage. Similar to enumeration, this is an encoded value.<br><br>Writing a value greater than MK_TME_MAX_KEYID_BITS will result in #GP<br><br>Writing a non-zero value to this field will #GP if bit 1 of EAX (TME Enable) is not also set to '1, as TME must be enabled to use MKTME. | Example: To support 255 keys, this field would be set to a value of 8. |
| | 47:36 | [Reserved] | |
| | 63:48 | Reserved if MKTME is not enumerated | |
| | | MK_TME_CRYPTO_ALGS<br><br>Bit 48: AES-XTS 128<br>Bit 63:49: Reserved (#GP)<br><br>Bitmask for BIOS to set which encryption algorithms are allowed for MKTME, would be later enforced by the key loading ISA ('1 = allowed) | |

### 4.2.2 IA32_TME_ACTIVATE WRMSR Response and Error Handling

| Conditions | Response |
|---|---|
| WRMSR when not enumerated | #GP |
| WRMSR while lock status=1 | #GP |
| WRMSR with 63:8 (reserved) ≠ 0 | #GP |
| WRMSR with Unsupported policy value (IA32_TME_CAPABILITY[IA32_TME_ACTIVATE[7:4]]=0) | #GP |
| WRMSR with enabled=0 | TME disabled, MSR locked subsequent RDMSR returns x..x01b |
| WRMSR with enabled=1 and key select=0 (new key); RNG success | TME enabled and MSR locked subsequent RDMSR returns x..x011b |
| WRMSR with enabled=1 and key select=0; RNG fail | Not enabled subsequent RDMSR returns x..x000b |
| WRMSR with enabled=1 and key select=1; Non zero key restored from CPU | TME enabled and MSR locked subsequent RDMSR returns x..x111b |
| WRMSR with enabled=1 and key select=1; Fail - Zero key restored from CPU | Not enabled subsequent RDMSR returns x..x100b |
| WRMSR with any other legal values | Subsequent RDMSR returns written values + lock status=1 |
| If MK_TME_KEYID_BITS > MK_TME_MAX_KEYID_BITS | #GP |
| If MK_TME_KEYID_BITS > 0 && (TME) Enable == 0 (TME must be enabled at the same point as MK-TME) | #GP |
| If MK_TME_KEYID_BITS > 0 and TME is not successfully activated (lock is not set) | Write not committed |
| If MK_TME_CRYPTO_ALGS reserved bits are set | #GP |

### 4.2.3 Core Address Masking MSR

MK_TME_CORE_ACTIVATE – Addr TBD (BIOS-only)

After successful activation using the IA32_TME_ACTIVATE MSR, this register should be written on each physical core with a value of 0 in EDX:EAX, failure to do so may result in unpredictable behavior. . Accesses to this MSR will #GP if MKTME is not supported.

BIOS is expected to write to this MSR on each core after doing MKTME activation. The first SMI on each core will also cause this value to be synchronized with the package MSR value.

| Register Address | MSR Name and bit fields | MSR/Bit Description | Comment |
|---|---|---|---|
| TBD | MK_TME_CORE_ACTIVATE MSR | This MSR will #GP if MKTME is not supported | |
| | 31:0 | RESERVED | |
| | 35:32 | MK_TME_KEYID_BITS (read-only)<br><br>The number of key identifier bits to allocate to MKTME usage. Similar to enumeration, this is an encoded value.<br><br>This is a read-only field. #GP on non-zero write. | Will be shadowed from the package MSR value on write |
| | 63:36 | RESERVED | |

### 4.2.4 Exclusion Range MSRs

TME and MKTME (for KeyID=0 Only) supports one exclusion range to be used for special cases. (Note: For all KeyIDs other than 0, TME Exclusion Range does not apply to MK-TME) The range of physical addresses specified in this MSR does not apply memory encryption described in this document. This range is primarily intended to be used for memory not available to OS and typically configured by BIOS. However, TME/MKTME (for KeyID=0) architecture does not place any restrictions on use of the exclusion range. The software is able to determine this range by reading MSR. The definition of this range follows definition of many range registers implemented in Intel processors.

| Register Address | MSR Name and bit fields | MSR/Bit Description | Comment |
|---|---|---|---|
| 983H | IA32_TME_EXCLUDE_MASK MSR | | |

| | | | |
|---|---|---|---|
| | 10:0 | RESERVED | |
| | 11 | Enable - When set to ‘1’, then TME_EXCLUDE_BASE and TME_EXCLUDE_MASK are used to define an exclusion region for TME/MKTME (for KeyID=0). | |
| | MAXPHYSADDR-1:12 | TMEEMASK - This field indicates the bits that must match TMEEBASE in order to qualify as a TME/MKTME (for KeyID=0) exclusion memory range access. | |
| | 63:MAXPHYSADDR | RESERVED – Must be Zero | |

| Register Address | MSR Name and bit fields | MSR/Bit Description | Comment |
|---|---|---|---|
| 984H | IA32_TME_EXCLUDE_BASE MSR | | |
| | 11:0 | RESERVED | |
| | MAXPHYSADDR-1:12 | TMEEBASE - Base physical address to be excluded for TME/MKTME (for KeyID=0) encryption. | |
| | 63:MAXPHYSADDR | RESERVED – Must be Zero | |

Note: Writing ‘1’ into bits above the max supported physical size will result in #GP

IA32_TME_EXCLUDE_MASK must define a contiguous region

- WRMSR will #GP if TMEEMASK field does not specify contiguous region

These MSRs are locked by TME_ACTIVATE MSR. If lock=1 then WRMSR to TME_EXCLUDE_MASK/TME_RXCLUDE_BASE will result in #GP.

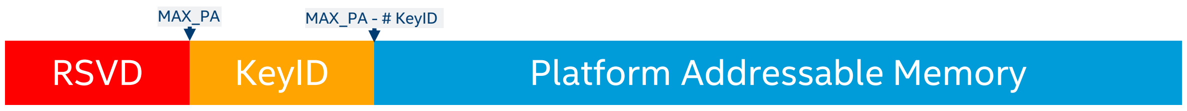
# 5 Runtime Behavior of MKTME

After MKTME is activated by the BIOS, there are a number of changes to the runtime behavior of the processor which will be described in this section.

## 5.1 Changes to Specification of Physical Address

The most significant change for MKTME is the repurposing of physical address bits to communicate the KeyID to the encryption engine(s) in the memory controller(s). This change necessitates a number of other hardware and software changes in order to maintain proper behavior.

When MKTME is activated the upper bits of platform physical address (starting with the highest order bit available as enumerated by the CPUID MAX_PA info) are repurposed for usage as a KeyID as shown below.



### 5.1.1 IA Paging

For the case that IA paging is being used without EPT, the upper bits starting with MAX_PA for each level of the IA page table are repurposed for usage as KeyID bits. Similarly, the upper bits of physical address in CR3 will be treated in the same manner.

Note that when EPT is active, IA paging does not generate/use platform physical addresses, instead it produces/uses guest physical addresses. Guest physical addresses are not modified by MKTME, and will continue to index into EPT page table walks as they did prior to MKTME.

### 5.1.2 EPT Paging

When EPT is enabled during VMX non-root operation, the upper bits for each level of EPT page walk are repurposed for usage as KeyID bits. Similarly, the upper bits of physical address in EPTP will be treated in the same manner. Note that guest OS may also use KeyID in IA page address, and full guest PA (including KeyID) is used by EPT.

### 5.1.3 Other physical addresses

Other physically addressed structures such as VMCS pointers, physically addressed bitmaps, etc. will receive similar treatment, with the upper bits of the address starting with MAX PA being repurposed as KeyID bits. Note that any reserved bit checking remains unchanged, which means that checking of these addresses will only be based upon the CPUID MAX_PA value.

# 6 MKTME Key Programming

## 6.1 Overview

Figure 1 shows a high-level overview of the MKTME engine meant to introduce the terminology that we will be using for the rest of the discussion and does not imply implementation.

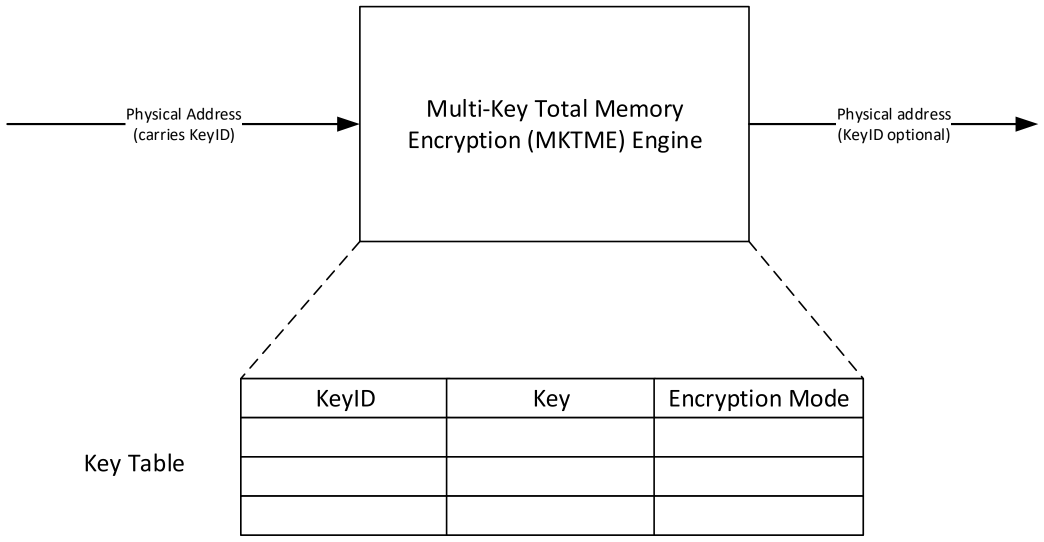

Physical Address (carries KeyID)
Multi-Key Total Memory Encryption (MKTME) Engine
Physical address (KeyID optional)
Key Table
KeyID
Key
Encryption Mode

**Figure 1: MKTME Engine Overview**

The MKTME engine maintains an internal key table not accessible by software to store the information (key and encryption mode) associated with each KeyID. Each KeyID may be associated with three encryption modes: Encryption using key specified, do not encrypt at all (memory will be plain text), or encrypt using TME Key. Future implementation may support additional encryption modes. PCONFIG is a new instruction that is used to program KeyID attributes for MKTME. While initial implementation may only use PCONFIG for MKTME, it may be extended in future to support additial usages. Therefore, PCONFIG is enumerated separately from MKTME.

## 6.2 PCONFIG Instruction

| Opcode | Instruction | Description |
|---|---|---|
| 0F 01 C5 | PCONFIG | This instruction is used to execute functions for configuring platform features<br><br>EAX: Leaf function to be invoked<br><br>RBX/RCX/RDX: Leaf-specific purpose |

### 6.2.1 PCONFIG Description

PCONFIG instruction is invoked by software for configuring platform features. PCONFIG supports multiple leafs and a leaf function is invoked by setting the appropriate leaf value in EAX. RBX, RCX, and RDX have a leaf-specific purpose. An attempt to execute an undefined leaf results in a #GP(0).

Table 1 shows the leaf encodings for PCONFIG.

**Table 1: PCONFIG Leafs**

| Leaf | Encoding | Description |
|---|---|---|
| MKTME_KEY_PROGRAM | 0x00000000 | This leaf is used to program the key and encryption mode associated with a KeyID. |
| RESERVED | 0x00000001-0xFFFFFFFF | Reserved for future use (#GP(0) if used) |

#### 6.2.1.1 MKTME_KEY_PROGRAM Leaf

MKTME_KEY_PROGRAM leaf of PCONFIG is used by software to manage the key associated with a KeyID. The leaf function is invoked by setting the leaf value of ‘0 in EAX and the address of MKTME_KEY_PROGRAM_STRUCT in RBX. Successful execution of the leaf clears RAX (set to zero) and ZF, CF, PF, AF, OF, and SF are cleared. In case of failure, the failure reason is indicated in RAX with ZF set to 1 and CF, PF, AF, OF, and SF are cleared. The MKTME_KEY_PROGRAM leaf works using the MKTME_KEY_PROGRAM_STRUCT in memory shown in Table 2.

**Table 2: MKTME_KEY_PROGRAM_STRUCT Format**

| Field | Offset (bytes) | Size (bytes) | Comments |
|---|---|---|---|
| KEYID | 0 | 2 | Key Identifier |

| | | | |
|---|---|---|---|
| KEYID_CTRL | 2 | 4 | KeyID control:<br>- Bits [7:0]: COMMAND<br>- Bits [23:8]: ENC_ALG<br>- Bits [31:24]: RSVD, MBZ |
| RSVD | 6 | 58 | RSVD, MBZ |
| KEY_FIELD_1 | 64 | 64 | Software supplied KeyID data key or entropy for KeyID data key |
| KEY_FIELD_2 | 128 | 64 | Software supplied KeyID tweak key or entropy for KeyID tweak key |

The following sub-sections provide a description of each of the fields in MKTME_KEY_PROGRAM_STRUCT

**KEYID**
Key Identifier being programmed to the MKTME engine

**KEYID_CTRL**
The KEYID_CTRL field carries two sub-fields used by software to control the behavior of a KeyID: Command and KeyID encryption algorithm.

The command used controls the encryption mode for a KeyID. Table 3 provides a summary of the commands supported.

**Table 3: Supported Key Programming Commands**

| Command | Encoding | Description |
|---|---|---|
| KEYID_SET_KEY_DIRECT | 0 | Software uses this mode to directly program a key for use with KeyID |
| KEYID_SET_KEY_RANDOM | 1 | CPU generates and assigns an ephemeral key for use with a KeyID. Each time the instruction is executed, the CPU generates a new key using hardware random number generator and the keys are discarded on reset |
| KEYID_CLEAR_KEY | 2 | Clear the (software programmed) key associated with the KeyID. On execution of this command, the KeyID gets TME behavior (encrypt with platform TME key) |
| KEYID_NO_ENCRYPT | 3 | Do not encrypt memory when this KeyID is in use. |

The encryption algorithm field (ENC_ALG) allows software to select one of the activated encryption algorithms for the KeyID. As discussed previously, the BIOS can activate a set of algorithms to allow for use when programming keys using the IA32_TME_ACTIVATE MSR (does not apply to KeyID 0 which uses TME policy). The ISA checks to ensure that the algorithm selected by software is one of the algorithms that has been activated by the BIOS.

**KEY_FIELD_1**
This field carries software supplied data key to be used for the KeyID if direct key programming option is used (KEYID_SET_KEY_DIRECT). When random key programming option is used (KEYID_SET_KEY_RANDOM), this field carries software supplied entropy to be mixed in the CPU generated random data key. It is software's responsibility to ensure that the key supplied for direct programming option or the entropy supplied for random programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys. When AES XTS-128 is used, the upper 48B are treated as reserved and must be zeroed out by software before executing the instruction.

**KEY_FIELD_2**
This field carries software supplied tweak key to be used for the KeyID if direct key programming option is used (KEYID_SET_KEY_DIRECT). When random key programming option is used (KEYID_SET_KEY_RANDOM), this field carries software supplied entropy to be mixed in the CPU generated random tweak key. It is software's responsibility to ensure that the key supplied for direct programming option or the entropy supplied for random programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys. When AES XTS-128 is used, the upper 48B are treated as reserved and must be zeroed out by software before executing the instruction.

All KeyIDs use TME key on MKTME activation. Software can at any point decide to change key for a KeyID using PCONFIG instruction. Change of keys for a KeyID does NOT change state of TLB, caches or memory pipeline. It is software's responsibility to take appropriate actions to ensure correct behavior. Example of software flows are provided in section 7.

Table 4 shows the return values associating with the MKTME_KEY_PROGRAM leaf of PCONFIG. On instruction execution, RAX is populated with the return value.

**Table 4: Programming Status for MKTME_KEY_PROGRAM**

| Return Value | Encoding | Description |
|---|---|---|
| PROG_SUCCESS | 0 | KeyID was successfully programmed |
| INVALID_PROG_CMD | 1 | Invalid KeyID programming command |
| ENTROPY_ERROR | 2 | Insufficient entropy |
| INVALID_KEYID | 3 | KeyID not valid |
| INVALID_ENC_ALG | 4 | Invalid encryption algorithm chosen (not supported) |
| DEVICE_BUSY | 5 | Failure to access key table (Section 6.2.4) |

### 6.2.2 PCONFIG Virtualization

Software in VMX root mode can control the execution of PCONFIG in VMX non-root mode using the following execution controls introduced for PCONFIG:

- PCONFIG_ENABLE: This control is a single bit control and enables the PCONFIG instruction in VMX non-root mode. If 0, the execution of PCONFIG in VMX non-root mode causes #UD else, execution of PCONFIG works according to PCONFIG_EXITING
- PCONFIG_EXITING: This is a 64b control and allows VMX root mode to cause a VM-exit for various leaf functions of PCONFIG. This control does not have any effect if the PCONFIG_ENABLE control is clear.

### 6.2.3 PCONFIG Enumeration

PCONFIG is enumerated in extended features (CPUID.7.0.EDX[18]). When 0, PCONFIG will #UD. A new CPUID leaf, PCONFIG_LEAF (leaf encoding, 0x1B), returns PCONFIG information. More specifically, sub-leaf n (n>=0) returns information about targets supported on the platform. The software developer manual will define the sub-leaf types and information returned. Software is expected to scan all sub-leafs to get the information about all targets supported on the platform. It should also be noted that the sub-leafs of the same target need not be consecutive.

**CPUID.PCONFIG_LEAF.n (n>=0)**
Returns information about supported targets on the platform. The information returned is shown below:

- EAX: Sub-leaf type
  - Bits 11:0: 0: Invalid sub-leaf, 1: Target Identifier
- If EAX[11:0] == 0
  - EAX:EBX:ECX:EDX = 0
  - Sub-leafs m>n return all 0s
- If EAX[11:0] == 1
  - EAX[31:12] = 0
  - EBX: Target_ID_1
  - ECX: Target_ID_2
  - EDX: Target_ID_3

Software is expected to scan all sub-leafs till an invalid sub-leaf is returned. All sub-leafs after the first invalid sub-leaf are invalid as well.

### 6.2.4 PCONFIG Concurrency

In a scenario, where MKTME_KEY_PROGRAM leaf of PCONFIG is executed concurrently on multiple logical processors, only one logical processor will succeed in updating the key table. PCONFIG execution will return with an error code (DEVICE_BUSY) on other logical processors and software must retry. In cases where the instruction execution fails with a DEVICE_BUSY error code, the key table is not updated, thereby ensuring that either the key table is updated in its entirety with the information for a KeyID or it is not updated at all. In order to accomplish this, MKTME_KEY_PROGRAM leaf of PCONFIG maintains a writer lock for updating the key table. This lock is referred to as the Key table lock and denoted in the instruction flows as KEY_TABLE_LOCK. The lock can either be unlocked, when no logical processor is holding the lock (also the initial state of the lock) or in an exclusive state where a logical processor is trying to update the key table. There can be only one logical processor holding the lock in exclusive state. The lock, being exclusive, can only be acquired when the lock is in unlocked state.

PCONFIG uses the following syntax to acquire KEY_TABLE_LOCK in exclusive mode and release the lock.

KEY_TABLE_LOCK.ACQUIRE(WRITE)

KEY_TABLE_LOCK.RELEASE()

### 6.2.5 PCONFIG Operation

| Variable Name | Type | Size (Bytes) | Description |
|---|---|---|---|
| TMP_KEY_PROGRAM_STRUCT | MKTME_KEY_PROGRAM_STRUCT | 192 | Structure holding the key programming structure |
| TMP_RND_DATA_KEY | UINT128 | 16 | Random data key generated for random key programming option |
| TMP_RND_TWEAK_KEY | UINT128 | 16 | Random tweak key generated for random key programming option |

```
(* #UD if PCONFIG is not enumerated or CPL>0 *)
if (CPUID.7.0:EDX[18] == 0 OR CPL > 0) #UD;

if (in VMX non-root mode)
{
   if (VMCS.PCONFIG_ENABLE == 1)
   {
      if ((EAX > 62 AND VMCS.PCONFIG_EXITING[63] ==1) OR
         (EAX < 63 AND VMCS.PCONFIG_EXITING[EAX] == 1))
      {
         Set VMCS.EXIT_REASON = PCONFIG; //No Exit qualification
         Deliver VMEXIT;
      }
   }
   else
   {
      #UD
   }
}

(* #GP(0) for an unsupported leaf *)
```

```
if(EAX != 0) #GP(0)

(* KEY_PROGRAM leaf flow *)
if (EAX == 0)
{
   (* #GP(0) if TME_ACTIVATE MSR is not locked or does not enable TME or
   multiple keys are not enabled *)
   if (IA32_TME_ACTIVATE.LOCK != 1 OR IA32_TME_ACTIVATE.ENABLE != 1 OR
   IA32_TME_ACTIVATE.MK_TME_KEYID_BITS  == 0) #GP(0)

   (* Check MKTME_KEY_PROGRAM_STRUCT is 256B aligned *)
   if(DS:RBX is not 256B aligned) #GP(0);

   (* Check that MKTME_KEY_PROGRAM_STRUCT is read accessible *)
   <<DS: RBX should be read accessible>>

   (* Copy MKTME_KEY_PROGRAM_STRUCT to a temporary variable *)
   TMP_KEY_PROGRAM_STRUCT = DS:RBX.*;

   (* RSVD field check *)
   if(TMP_KEY_PROGRAM_STRUCT.RSVD != 0) #GP(0);

   if(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.RSVD !=0) #GP(0);

   if(TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1.BYTES[63:16] != 0) #GP(0);

   if(TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2.BYTES[63:16] != 0) #GP(0);

   (* Check for a valid command *)
   if(TMP_KEY_PROGRAM_STRUCT. KEYID_CTRL.COMMAND is not a valid command)
   {
      RFLAGS.ZF = 1;
      RAX = INVALID_PROG_CMD;
      goto EXIT;
   }
   (* Check that the KEYID being operated upon is a valid KEYID *)
   if(TMP_KEY_PROGRAM_STRUCT.KEYID >
                     2^IA32_TME_ACTIVATE.MK_TME_KEYID_BITS – 1
      OR TMP_KEY_PROGRAM_STRUCT.KEYID >
                     IA32_TME_CAPABILITY.MK_TME_MAX_KEYS
      OR TMP_KEY_PROGRAM_STRUCT.KEYID == 0)
   {
      RFLAGS.ZF = 1;
      RAX = INVALID_KEYID;
      goto EXIT;
   }

   (* Check that only one algorithm is requested for the KeyID and it is
      One of the activated algorithms *)
   if(NUM_BITS(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG) != 1 ||
      (TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG &
         IA32_TME_ACTIVATE. MK_TME_CRYPTO_ALGS == 0))
```

```
{
   RFLAGS.ZF = 1;
   RAX = INVALID_ENC_ALG;
   goto EXIT;
}
(* Try to acquire exclusive lock *)
if (NOT KEY_TABLE_LOCK.ACQUIRE(WRITE))
{
   //PCONFIG failure
   RFLAGS.ZF = 1;
   RAX = DEVICE_BUSY;
   goto EXIT;
}

(* Lock is acquired and key table will be updated as per the command
   Before this point no changes to the key table are made *)

switch(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.COMMAND)
{
case KEYID_SET_KEY_DIRECT:
   <<Write
      DATA_KEY=TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1,
      TWEAK_KEY=TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2,
      ENCRYPTION_MODE=ENCRYPT_WITH_KEYID_KEY,
      to MKTME Key table at index TMP_KEY_PROGRAM_STRUCT.KEYID
   >>
   break;

case KEYID_SET_KEY_RANDOM:
   TMP_RND_DATA_KEY = <<Generate a random key using RDSEED>>
   if (NOT ENOUGH ENTROPY)
   {
      RFLAGS.ZF = 1;
      RAX = ENTROPY_ERROR;
      goto EXIT;
   }
   TMP_RND_TWEAK_KEY = <<Generate a random key using RDSEED>>
   if (NOT ENOUGH ENTROPY)
   {
      RFLAGS.ZF = 1;
      RAX = ENTROPY_ERROR;
      goto EXIT;
   }
   (* Mix user supplied entropy to the data key and tweak key *)
   TMP_RND_DATA_KEY = TMP_RND_KEY XOR
         TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1.BYTES[15:0];
   TMP_RND_TWEAK_KEY = TMP_RND_TWEAK_KEY XOR
         TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2.BYTES[15:0];

   <<Write
      DATA_KEY=TMP_RND_DATA_KEY,
      TWEAK_KEY=TMP_RND_TWEAK_KEY,
            ENCRYPTION_MODE=ENCRYPT_WITH_KEYID_KEY,
            to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
         >>
         break;

      case KEYID_CLEAR_KEY:
            <<Write
            DATA_KEY='0,
            TWEAK_KEY='0,
            ENCRYPTION_MODE = ENCRYPT_WITH_TME_KEY,
            to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
            >>

         break;
      case KD_NO_ENCRYPT:
         <<Write
            ENCRYPTION_MODE=NO_ENCRYPTION,
            to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
         >>
         break;
      }

      RAX = 0;
      RFLAGS.ZF = 0;

      //Release Lock
      KEY_TABLE_LOCK(RELEASE);

      EXIT:
      RFLAGS.CF=0;
      RFLAGS.PF=0;
      RFLAGS.AF=0;
      RFLAGS.OF=0;
      RFLAGS.SF=0;
   }

   end_of_flow
```

### 6.2.6 Flags Affected

| | |
|---|---|
| ZF | Cleared if instruction completes successfully.<br>Set if error occurred. RAX is set to the error code |
| CF, PF, AF, OF, SF | Cleared |

### 6.2.7 Use of prefixes

| | |
|---|---|
| LOCK | Causes #UD |
| REP* | Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ) |
| Operand size | Causes #UD |
| VEX | Causes #UD |
| Segment overrides | Ignored |
| Address size | Ignored |
| REX | Ignored |

### 6.2.8 Protected Mode Exceptions

| | |
|---|---|
| #UD | If any of the LOCK/REP/OSIZE/VEX prefix is used.<br>If current privilege level is not 0.<br>If CPUID.7.0:EDX[18] = 0<br>If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0 |
| #GP(0) | If input value in EAX encodes an unsupported leaf<br>If IA32_TME_ACTIVATE MSR is not locked<br>If TME and MKTME capability is not enabled in IA32_TME_ACTIVATE MSR<br>If memory operand is not 256B aligned.<br>If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set<br>If a memory operand effective address is outside the DS segment limit. |

#PF(fault code) If a page fault occurs in accessing memory operands.

### 6.2.9 Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If any of the LOCK/REP/OSIZE/VEX prefix is used.<br>If current privilege level is not 0.<br>If CPUID.7.0:EDX[PCONFIG_BIT] = 0<br>If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0 |
| #GP(0) | If input value in EAX encodes an unsupported leaf<br>If IA32_TME_ACTIVATE MSR is not locked<br>If TME and MKTME capability is not enabled in IA32_TME_ACTIVATE MSR<br>If memory operand is not 256B aligned.<br>If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set |

### 6.2.10 Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | PCONFIG instruction is not recognized in virtual-8086 mode |

### 6.2.11 Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 6.2.12 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If any of the LOCK/REP/OSIZE/VEX prefix is used.<br>If current privilege level is not 0.<br>If CPUID.7.0:EDX[18] = 0<br>If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0 |
| #GP(0) | If input value in EAX encodes an unsupported leaf<br>If IA32_TME_ACTIVATE MSR is not locked<br>If TME and MKTME capability is not enabled in IA32_TME_ACTIVATE MSR<br>If memory operand is not 256B aligned.<br>If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set<br>If a memory operand is non-canonical form |
| #PF(fault code) | If a page fault occurs in accessing memory operands. |

# 7 Software life cycle: Managing Pages with KeyID

## 7.1 Overview

As mentioned earlier in the document, the KeyID is integral part of physical address; meaning it is not only present in page tables but is also present in the TLB, Caches, etc. Therefore, software needs be aware of this and must take appropriate steps to maintain correctness of operations and security.

Note that while this section focuses on virtualization scenarios, the TME and MKTME architecture is applicable to both native OS and virtualized environments, and for DRAM and NVRAM types of memory.

## 7.2 Restrictions and Cache Management

The hardware/CPU does not enforce coherency between mappings of the same physical page with different KeyIDs or encryption keys. System software is responsible for carefully managing the caches in regard to usage of key identifiers (KeyIDs) and maintaining cache coherency when the KeyID or a Key associated with a physical page is changed by the software. Specifically, the CPU will treat two physical addresses that are identical except for the KeyID bits as two different physical addresses though these two addresses reference the same location in memory. Software must take necessary steps to ensure that this does not result in unpredictable or incorrect behavior, or violate security properties desired. MK-TME retains the existing behavior of the caches and TLB for the entire physical address including the KeyID portion of the physical address and expects software to properly flush the caches and/or perform TLB shootdowns.

The sections below are intended to give examples of algorithms that shouldt be used by software to ensure correctness and security. Please check the final version of this specification for any updated algorithms or requirements in this area.

## 7.3 General software guidance for dealing with Aliased Address Mappings

The following are some general guidelines for OS/VMM software vendors to consider when using MKTME with more than the default single KeyID.

1. Software should avoid mapping the same physical address with multiple KeyIDs.
2. If Software must map same physical address with multiple KeyIDs, it should mark those pages as read-only, except for one KeyID.

3. If Software must map same physical address with multiple KeyIDs as read-write, then software must ensure that all writes are done with single KeyID (This includes locked and non-locked writes that do not modify the data).

## 7.4 AddPage: Associating a KeyID to a page

The following algorithm should be used by OS/VMM when assigning a new KeyID to a physical page.

1. Program a new key for the KeyID, if not already programmed (using PCONFIG instruction)
2. Map the physical page to VMM's address space (with the new KeyID) by updating its paging structure entries (IA-PT), if not already mapped
3. Ensure that the step 1 has successfully completed
4. Zero page contents via the new mapping (with new KeyID) to avoid data leakage between KeyID domains
5. Make the page available to a new VM with the new KeyID set in the EPT page-table entry

This will ensure against data leakage between KeyID domains, such as VMs with KeyIDs, when the KeyID is changed for a physical page (but the data is in clear in the CPU caches). The assumption is that before using this algorithm to assign a new KeyID, software/ VMM makes sure that the page was evicted correctly from the previous KeyID (using the algorithm defined in the next section).

## 7.5 EvictPage: Disassociating a KeyID from a Page

The following algorithm should be used by OS/VMM when changing the KeyID of a physical page so that the current KeyID is no longer used with the page.

1. Steps to be completed before changing the KeyID
    1. Make the physical page not accessible to the VM (by updating the EPT page-table entry)
    2. Invalidate all page mappings/aliases (the INVEPT instruction and IOMMU (VT-d) invalidation if page was mapped as device accessible) from the TLB (across the logical processors, with the old KeyID)
    3. Map the page to VMM address space (with the old KeyID) by updating its paging structure entries (IA-PT) if not already mapped
    4. OS/VMM Flushes dirty cache lines (for page using old KeyID) to prevent aliasing overwrite/data corruption
        1. a) Options: CLFLUSH, CLWB+fence, CLFLUSHOPT+fence or WBINVD

2. b) Software can optionally avoid doing these flushing if it tracks page modification using EPT page-modification logging or accessed and dirty flags for EPT (optimization)

2. The page is now ready to be used with a new KeyID (example, using steps in previous section)

This will ensure that no cache lines aliased by physical address exist in the CPU caches, when the KeyID of the physical page is changed.

Note - Guidance for usage of WBINVD instruction: WBINVD instruction should be run on each socket if those invalidates invalidate all coherent caches on the sockets.

## 7.6 Paging by OS/VMM example

Below is an example of a software sequence where OS/VMM is reallocating a page from VM2 to VM3. VM2 memory uses KeyID2, and VM3 memory uses KeyID3.

1. Evict a page with KeyID2 from VM2 using EvictPage algorithm described in section 7.5

2. OS/VMM reads the evicted page with KeyID2, encrypts the page contents with Full Disk Encryption key (optional) and writes the page to disk/stores on a swap file (or in OS/VMM memory, using the VMM KeyID=0)

3. Add the evicted page to VM3 KeyID3 using the AddPage algorithm in section 7.4.

## 7.7 OS/VMM access to guest memory

OS/VMM can access guest memory (in clear) for emulation purposes (MMIO) by setting the guest KeyID bits in its paging structure entries (IA-PT).

## 7.8 I/O interactions

OS/VMM can use the TME key (KeyID=0) to setup shared memory between Guest VM and the VMM as needed for I/O purposes. For directed I/O (e.g., SR-IOV), OS/VMM should program the KeyID as part of physical addresses in IOMMU (VT-d) page tables corresponding to the KeyID as part of physical addresses in EPT (for the Guest VM). This will allow DMAs to be able to access memory in clear without requiring changes to I/O devices and/or I/O drivers in the guest VM or OS/VMM.