Intel Key Locker Specification

343965-001US

SEPTEMBER 2020

Table of Contents

Contents

Revision History

Revision Number	Description	Date
1.0	Initial release.	September 2020

Glossary

Abbreviation / Term	**Description**
AAD	Additional Authentication Data
AES	Advanced Encryption Standard
IA	Intel® Architecture
SMM	System Management Mode
STM	SMM Transfer Monitor
SW	Software
TSX	Transactional Synchronization Extensions
VM	Virtual Machine
VMM	Virtual Machine Monitor
VMX	Virtual Machine Extensions

1 Introduction

This document describes the software programming interface for the Intel® Architecture instruction set extensions pertaining to the Key Locker feature.

Key Locker provides a mechanism to encrypt and decrypt data with an AES key without having access to the raw key value by converting AES keys into "handles". These handles can be used to perform the same encryption and decryption operations as the original AES keys, but they only work on the current system and only until they are revoked. If software revokes Key Locker handles (e.g., on a reboot), then any previous handles can no longer be used.

Once a key handle has been created, the original keys that were wrapped into those handles can be erased from memory. Most adversaries generally cannot obtain the actual AES keys, except during that brief period when software is requesting that the key handles be created.

If the OS chooses a policy that revokes the handles on each reboot, then any handles that may have been stolen should no longer be useful to the attacker after the reboot.

There is no arbitrary limit on the number of key handles that can be created. An internal wrapping key is used to create the handles, each of which is essentially an encrypted form of an underlying AES key. The internal wrapping key can be created and loaded by privileged software, or it can be randomly generated by the CPU in a way that is designed not to reveal its value to any software.

On many platforms, software can back up the current internal wrapping key and also restore it. This can enable the OS to save and restore the keys across the S3 (sleep) and S4 (hibernate) system sleep states, as well as provide a method to distribute an internal wrapping key across the entire platform without putting it in memory.

Software cryptographic libraries may be able to use the Key Locker instructions without fundamentally changing their API, providing an easy way for software to gain improved security for their AES keys without having to directly add support for the Key Locker instructions.

1.1 Basic Instructions and Usage

Key Locker consists of three types of instructions:

1) Instructions to create handles from an AES key (ENCODEKEY128 and ENCODEKEY256).
2) Instructions to use handles to perform AES encryption or decryption (AESDEC128KL, AESDEC256KL, AESDECWIDE128KL, AESDECWIDE256K, AESENC128KL, AESENC256KL, AESENCWIDE128KL, and AESENCWIDE256KL).
3) Instruction to load an internal wrapping key (LOADIWKEY).

1.1.1 Instructions to Create Handles

Key Locker adds two instructions that take AES keys and create handles. They also take input on which restrictions are requested on how the handle can be used. They use the current IWKey (see section 1.2) to create a handle. The output handle contains an encrypted version of the AES key as well as metadata. The entire handle is designed to include integrity protection, such that any modification of the handle (e.g., to edit the metadata) or usage of the handle with a different IWKey should be detected.

ENCODEKEY128 takes a 128-bit AES key as input and produces a 384-bit handle.

ENCODEKEY256 takes a 256-bit AES key as input and produces a 512-bit handle.

Many software usages will want to overwrite the AES key after the handle is generated so that later vulnerabilities should be limited to a stolen handle, not the original AES key.

More details on the algorithm used to create the handle are available in section A.4.

In addition to producing the handle, the ENCODEKEY* instructions also indicate the type of IWKey that was loaded and zero registers XMM4, XMM5 and XMM6. It is possible that future enhancements to Key Locker will produce non-zero values for XMM4-6 (e.g., to indicate further information about the IWKey specified).

1.1.1.1 Handle Restrictions

Each handle includes an AAD (Additional Authentication Data) field which is designed to be integrity protected but not encrypted. It is used to hold metadata of the handle, including its restrictions.

When a Key Locker handle is created via one of the ENCODEKEY* instructions, SW can specify the following restrictions by setting the indicated bit the handle's AAD field:

1) Ring 0 only (bit 0 of AAD): Handle can be used only in CPL 0 (supervisor mode); it cannot be used in application modes (CPL >0).
2) No-Encrypt (bit 1 of AAD): Handle cannot be used for encryption.
3) No-Decrypt (bit 2 of AAD): Handle cannot be used for decryption.

Multiple restriction bits may be set in a single handle. Handle restriction failures (including AAD reserved bits set) will result in the AES*KL instructions setting RFLAGS.ZF and not performing the requested encryption or decryption.

Ring 0 only handles may be useful for OS keys that are not intended for usage by applications. If a malicious application manages to steal such a handle, it should not be able to use it within the application itself. Note that ring 0 handles can be created at any privilege level despite only being usable for encryption/decryption at ring 0.

No-decrypt and no-encrypt handles may be useful in pairs when one side of a protocol only needs to create messages (with encryption) and the other side of the protocol only needs to read messages (with decryption). This would require using an AES mode that uses both AES encryption and decryption (e.g., AES-CBC), rather than an AES mode that only uses encryption (e.g., AES-CTR).

1.1.2 Instructions to Use Handles to Perform AES Encryption or Decryption

Key Locker instructions can take a handle and either plaintext or cipher text and encrypt/decrypt it. Details on different Key Locker instructions are shown in Table 2-1.

Table 2-1. Key Locker Instructions

Instruction	AES Key Size	Encrypt or Decrypt	Single 128-bit Block vs. Eight 128-bit Blocks
AESENC128KL	128-bit	Encrypt	Non-wide (single 128-bit block)
AESENCWIDE128KL			Wide (eight 128-bit blocks)
AESDEC128KL		Decrypt	Non-wide (single 128-bit block)
AESDECWIDE128KL			Wide (eight 128-bit blocks)
AESENC256KL	256-bit	Encrypt	Non-wide (single 128-bit block)
AESENCWIDE256KL			Wide (eight 128-bit blocks)
AESDEC256KL		Decrypt	Non-wide (single 128-bit block)
AESDECWIDE256KL			Wide (eight 128-bit blocks)

The 'wide' instructions that operate on eight 128-bit blocks have higher performance on parallel AES modes like AES-CTR than executing eight iterations of a non-wide Key Locker instruction.

After every Key Locker AES encryption/decryption operation, software should check ZF in order to ensure that the operation did not fail (e.g., due to a corrupted handle or a restriction failure). Failure to do this check might lead to using plaintext as ciphertext (or vice versa) when the instruction fails.

1.1.3 Instruction to Load the Internal Wrapping Key

The internal wrapping key (Key Locker IWKey) is used to convert between handles and the original keys. The internal wrapping key is written by the LOADIWKEY instruction. It is important that it is kept secret from attackers in order to help prevent them from manually unwrapping handles in order to obtain the original keys. For this reason there is no operation to read out the internal wrapping key, although there are "IWKeyBackup" MSRs that can be used to back up the internal wrapping key without revealing its value (see section 4).

As the internal wrapping key is considered system state, LOADIWKEY can only execute in supervisor mode (CPL 0). In order to support VM context switch and migration, a VMM can also cause a VM exit on guest execution of LOADIWKEY in order to capture the IWKey value.

It is recommended that LOADIWKEY be executed early in the OS boot in order to reduce the chance that software has been loaded that an attacker can exploit to watch the LOADIWKEY's data.

It is possible for LOADIWKEY to specify that the internal wrapping key it loads cannot be backed up through the IWKeyBackup MSRs. This is designed to ensure that the internal wrapping key cannot be revealed through any vulnerabilities found in the future in the backup or restore mechanisms; but blocks usages that require those MSRs (like maintaining handles across S3 or S4 sleep states).

LOADIWKEY can either directly load the specified argument or can request a hardware generated random internal wrapping key. When LOADIWKEY requests a hardware generated random key, the processor reads 384 bits of random data (from the same on-chip random number generator that supplied the random data read by RDSEED) and XORs it with the LOADIWKEY's arguments. Because the data is XORed, software can combine their own entropic data with that supplied by the hardware random bit generator.

There is no instruction or VM execution control for a virtual-machine monitor (VMM) to read the values of hardware generated random internal wrapping keys. This fact limits the ability of a VMM to move a virtual machine from one processor to another. Because of this limitation, a VMM may choose to not enumerate support for hardware generated random internal wrapping keys when it virtualizes the CPUID instruction.

Hardware random internal wrapping keys can be backed up and restored (unless the LOADIWKEY specified that no such backup is allowed) through MSRs. The output of the ENCODEKEY* instructions indicate properties of the internal wrapping key that was used to create the handle, including whether it is was from the on-chip hardware random number generator and whether it can be backed up through MSRs.

1.2 IWKey

The IWKey is an internal wrapping key used by the Key Locker ENCODEKEY* and AES*KL instructions. It is logical processor scoped and is written through the LOADIWKEY instruction. It is designed not to be directly readable by software.

The internal wrapping key currently consists of:

1) IntegrityKey[127:0] - A 128-bit integrity key used to check that handles have not been tampered with.
2) EncryptionKey[255:0] - A 256-bit encryption key used in wrapping/unwrapping to help protect confidentiality of the keys indicated by the handles.
3) KeySource [3:0] – The only allowed values are 0 (AES GCM SIV wrapping algorithm with SW specified keys) and 1 (AES GCM SIV wrapping algorithm with random keys enforced by hardware).
4) NoBackup flag – when set, this IWKey cannot be backed up.

IWKeyBackup (described in section 4) has the same format as IWKey.

1.3 CR4.KL

Key Locker introduces a new “KL” bit in CR4 (bit 19) in order to help prevent usage of Key Locker when it is not properly enabled by system software. This can also help prevent guests of legacy VMMs from using Key Locker.

CR4.KL existence is enumerated by CPUID.KL (CPUID.(EAX=07H, ECX=0H).ECX.KL[bit 23]).

When CR4.KL is 0, all Key Locker instructions will #UD, including the AES*KL, ENCODEKEY128, ENCODEKEY256 and LOADIWKEY instructions, and CPUID.AESKLE (CPUID.19H:EBX[0]) will be 0. The Key Locker IWKeyBackup MSRs (described later) are not affected by the value of CR4.KL.

1.4 Handle Format

Handles for 128-bit AES keys are 384 bits in size and have the following format:

- Handle[127:0] = AAD
- Handle[255:128] = Integrity Tag
- Handle[383:256] = Ciphertext

Handles for 256-bit AES keys are 512 bits in size and have the following format:

- Handle[127:0] = AAD
- Handle[255:128] = Integrity Tag
- Handle[383:256] = Ciphertext[127:0]
- Handle[511:384] = Ciphertext[255:128]

The AAD (Additional Authentication Data) format for both types of handles have the following format:

- AAD[0] = Handle is not usable if CPL > 0
- AAD[1] = Handle is not usable for encryption
- AAD[2] = Handle is not usable for decryption
- AAD[23:3] = Reserved
- AAD[27:24] = Key Type. 0 indicates AES-128 handle and 1 indicates AES-256 handle. All other key types are currently reserved.
- AAD[127:28] = Reserved

1.5 Intel® Transactional Synchronization Extensions (Intel® TSX) Operation

On some implementations, Key Locker instructions will cause Intel TSX aborts when executed inside an Intel TSX transaction.

2 CPUID Enumeration of Key Locker Support

Hardware support for Key Locker is enumerated through CPUID.KL: CPUID.(07H,0).ECX[23] = 1. This indicates that the Key Locker feature is supported by the processor and CPUID leaf 19H gives more information about Key Locker capabilities.

A separate CPUID bit, CPUID.AESKLE: CPUID.19H.EBX[0]=1 indicates that the operating system and system firmware (e.g., BIOS) have enabled Key Locker AES instructions (AES KL is enabled). Software should first determine that CPUID leaf 19H is supported (by checking that CPUID.KL is enumerated) before looking at leaf 19H bits like CPUID.AESKLE. AESKLE bit will be 0 unless CR4.KL is set. Some implementations may need system firmware enabling of Key Locker. If CR4.KL is set but AESKLE is not enumerated (reads as 0) then it may be that system firmware enabling of Key Locker is needed on that implementation and was not performed.

When deciding whether to enable Key Locker, the operating system should check CPUID.KL.

Software that wishes to use Key Locker to help protect AES keys (e.g., applications) should check that CPUID.KL and the CPUID.AESKLE bits are both set. CPUID.KL will indicate that CPUID.19H (and thus AESKLE) is valid and AESKLE will indicate that the OS (and, if needed, system firmware) have enabled Key Locker.

For determining the other features of Key Locker, use the definitions shown in Table 2-1. Invoke CPUID as (instantiating the register and bit fields appropriately): CPUID.19H:REG[bit #].

Table 2-1. Key Locker CPUID Definitions for Leaf 19H

Register	Bit Position(s)	Contents
EAX	0	KL restriction of CPL0-only supported.
EAX	1	KL restriction of no-encrypt supported.
EAX	2	KL restriction of no-decrypt supported.
EAX	31:3	Reserved.
EBX	0	AESKLE: When 1, the AES Key Locker instructions are fully enabled.
EBX	1	Reserved.
EBX	2	WIDE_KL: When 1, the AES wide Key Locker instructions are supported.
EBX	3	Reserved.
EBX	4	When 1, the platform supports the IWKeyBackup MSRs and backing up the internal wrapping key.
EBX	31:5	Reserved.
ECX	0	When 1, the NoBackup parameter to LOADIWKEY is supported.
ECX	1	When 1, KeySource encoding of 1 (randomization of the internal wrapping key) is supported.
ECX	31:2	Reserved.
EDX	31:0	Reserved.

3 Instructions

3.1 Notation

Instructions described in this chapter follow the general documentation convention established in Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A.[1] Additionally, the Key Locker instructions use notation conventions as described below.

In the instruction encoding, the MODRM byte is represented several ways depending on the role it plays. The MODRM byte has 3 fields: 2-bit MODRM.MOD field, a 3-bit MODRM.REG field and a 3-bit MODRM.RM field. When all bits of the MODRM byte have fixed values for an instruction, the 2-hex nibble value of that byte is presented after the opcode in the encoding boxes on the instruction description pages. When only some fields of the MODRM byte must contain fixed values, those values are specified as follows:

If only the MODRM.MOD must be 0b11, and MODRM.REG and MODRM.RM fields are unrestricted, this is denoted as 11:rrr:bbb. The rrr correspond to the 3-bits of the MODRM.REG field and the bbb correspond to the 3-bits of the MODMR.RM field.

If the MODRM.MOD field is constrained to be a value other than 0b11, i.e., it must be one of 0b00, 0b01, or 0b10, then we use the notation !(11).

If the MODRM.REG field had a specific required value, e.g., 0b101, that would be denoted as mm:101:bbb.

[1] Note that historically the Intel® 64 and IA-32 Architectures Software Developer's Manual only specified the MODRM.REG field restrictions with the notation /0 … /7 and did not specify restrictions on the MODRM.MOD and MODRM.RM fields in the encoding boxes.

3.2 AESDEC128KL

Opcode/Instruction	Op/ En	64/32-bit Mode Support	CPUID Flag	Description
F3 0F 38 DD !(11):rrr:bbb AESDEC128KL xmm, m384	A	V/V	AESKLE	Decrypt xmm using 128-bit AES key indicated by handle at m384 and store result in xmm.

3.2.1 Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

3.2.2 Description

The AESDEC128KL instruction performs 10 rounds of AES to decrypt the first operand using the 128-bit key indicated by the handle from the second operand. It stores the result in the first operand.

3.2.3 Operation

```
AESDEC128KL
    Handle := UnalignedLoad of 384 bit (SRC);  // Load is not guaranteed to be atomic.
    Illegal Handle = (HandleReservedBitSet (Handle) ||
                      (Handle[0] AND (CPL > 0)) ||
                      Handle [2]   ||
                      HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128);
    If (Illegal Handle) {
        RFLAGS.ZF := 1;
    } ELSE {
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IWKey);
        If (Authentic == 0) {
            RFLAGS.ZF := 1;
        } ELSE {
            DEST := AES128Decrypt (DEST, UnwrappedKey);
            RFLAGS.ZF := 0;
        }
    }
    RFLAGS.OF, SF, AF, PF, CF := 0;
```

3.2.4 Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

3.2.5 Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H:ECX.KL [bit 23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE [bit 0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

3.2.6 Intrinsics

unsigned char _mm_aesdec128kl_u8(__m128i* odata, __m128i idata, const void* h);

3.3 AESDEC256KL

Opcode/Instruction	Op /En	64/32-bit Mode Support	CPUID Flag	Description
F3 0F 38 DF !(11):rrr:bbb AESDEC256KL xmm, m512	A	V/V	AESKLE	Decrypt xmm using 256-bit AES key indicated by handle at m512 and store result in xmm.

3.3.1 Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

3.3.2 Description

The AESDEC256KL instruction performs 14 rounds of AES to decrypt the first operand using the 256-bit key indicated by the handle from the second operand. It stores the result in the first operand.

3.3.3 Operation

```
AESDEC256KL
    Handle := UnalignedLoad of 512 bit (SRC); // Load is not guaranteed to be atomic.
    Illegal Handle = (HandleReservedBitSet (Handle) ||
                      (Handle[0] AND (CPL > 0)) ||
                      Handle [2]   ||
                      HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256);
    If (Illegal Handle) {
        RFLAGS.ZF := 1;
    } ELSE {
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], IWKey);
        If (Authentic == 0) {
            RFLAGS.ZF := 1;
        } ELSE {
            DEST := AES256Decrypt (DEST, UnwrappedKey) ;
            RFLAGS.ZF := 0;
        }
    }
    RFLAGS.OF, SF, AF, PF, CF := 0;
```

3.3.4 Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

3.3.5 Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H:ECX.KL [bit 23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE [bit 0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

3.3.6 Intrinsics

unsigned char _mm_aesdec256kl_u8(__m128i* odata, __m128i idata, const void* h);

3.4 AESDECWIDE128KL

Opcode/Instruction	Op /En	64/32-bit Mode Support	CPUID Flag	Description
F3 0F 38 D8 !(11):001:bbb AESDECWIDE128KL m384, <XMM0-7>	A	V/V	AESKLE WIDE_KL	Decrypt XMM0-7 using 128-bit AES key indicated by handle at m384 and store each resultant block back to its corresponding register.

3.4.1 Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2-9
A	NA	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

3.4.2 Description

The AESDECWIDE128KL instruction performs ten rounds of AES to decrypt each of the eight blocks in XMM0-7 using the 128-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding decrypted block.

3.4.3 Operation

```
AESDECWIDE128KL
    Handle := UnalignedLoad of 384 bit (SRC);  // Load is not guaranteed to be atomic.
    Illegal Handle = (HandleReservedBitSet (Handle) ||
                      (Handle[0] AND (CPL > 0)) ||
                      Handle [2]   ||
                      HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128);
    If (Illegal Handle) {
        RFLAGS.ZF := 1;
    } ELSE {
         (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IWKey);
        If Authentic == 0 {
            RFLAGS.ZF := 1;
        } ELSE {
            XMM0 := AES128Decrypt (XMM0, UnwrappedKey);
            XMM1 := AES128Decrypt (XMM1, UnwrappedKey);
            XMM2 := AES128Decrypt (XMM2, UnwrappedKey);
            XMM3 := AES128Decrypt (XMM3, UnwrappedKey);
            XMM4 := AES128Decrypt (XMM4, UnwrappedKey);
            XMM5 := AES128Decrypt (XMM5, UnwrappedKey);
            XMM6 := AES128Decrypt (XMM6, UnwrappedKey);
            XMM7 := AES128Decrypt (XMM7, UnwrappedKey);
            RFLAGS.ZF := 0;
        }
}
    RFLAGS.OF, SF, AF, PF, CF := 0;
```

3.4.4 Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

3.4.5 Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H:ECX.KL [bit 23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE [bit 0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
	If CPUID.19H:EBX.WIDE_KL [bit 2] = 0.
#NM	If CR0.TS = 1.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

3.4.6 Intrinsics

unsigned char _mm_aesdecwide128kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);

3.5 AESDECWIDE256KL

Opcode/Instruction	Op /En	64/32-bit Mode Support	CPUID Flag	Description
F3 0F 38 D8 !(11):011:bbb AESDECWIDE256KL m512, <XMM0-7>	A	V/V	AESKLE WIDE_KL	Decrypt XMM0-7 using 256-bit AES key indicated by handle at m512 and store each resultant block back to its corresponding register.

3.5.1 Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2-9
A	NA	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

3.5.2 Description

The AESDECWIDE256KL instruction performs 14 rounds of AES to decrypt each of the eight blocks in XMM0-7 using the 256-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding decrypted block.

3.5.3 Operation

```
AESDECWIDE256KL
    Handle := UnalignedLoad of 512 bit (SRC); // Load is not guaranteed to be atomic.
    Illegal Handle = (HandleReservedBitSet (Handle) ||
                      (Handle[0] AND (CPL > 0)) ||
                      Handle [2]   ||
                      HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256);
    If (Illegal Handle) {
        RFLAGS.ZF := 1;
    } ELSE {
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], IWKey);
        If (Authentic == 0) {
            RFLAGS.ZF := 1;
        } ELSE {

            XMM0 := AES256Decrypt (XMM0, UnwrappedKey);
            XMM1 := AES256Decrypt (XMM1, UnwrappedKey);
            XMM2 := AES256Decrypt (XMM2, UnwrappedKey);
            XMM3 := AES256Decrypt (XMM3, UnwrappedKey);
            XMM4 := AES256Decrypt (XMM4, UnwrappedKey);
            XMM5 := AES256Decrypt (XMM5, UnwrappedKey);
            XMM6 := AES256Decrypt (XMM6, UnwrappedKey);
            XMM7 := AES256Decrypt (XMM7, UnwrappedKey);
            RFLAGS.ZF := 0;
        }
    }

    RFLAGS.OF, SF, AF, PF, CF := 0;
```

3.5.4 Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

3.5.5 Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H:ECX.KL [bit 23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE [bit 0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
	If CPUID.19H:EBX.WIDE_KL [bit 2] = 0.
#NM	If CR0.TS = 1.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

3.5.6 Intrinsics

unsigned char _mm_aesdecwide256kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);

3.6 AESENC128KL

Opcode/Instruction	Op /En	64/32-bit Mode Support	CPUID Flag	Description
F3 0F 38 DC !(11):rrr:bbb AESENC128KL xmm, m384	A	V/V	AESKLE	Encrypt xmm using 128-bit AES key indicated by handle at m384 and store result in xmm.

3.6.1 Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

3.6.2 Description

The AESENC128KL instruction performs ten rounds of AES to encrypt the first operand using the 128-bit key indicated by the handle from the second operand. It stores the result in the first operand.

3.6.3 Operation

```
AESENC128KL
    Handle := UnalignedLoad of 384 bit (SRC); // Load is not guaranteed to be atomic.
    Illegal Handle = (
                        HandleReservedBitSet (Handle) ||
                        (Handle[0] AND (CPL > 0)) ||
                        Handle [1]   ||
                        HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128
                        );
    If (Illegal Handle) {
        RFLAGS.ZF := 1;
    } ELSE {
         (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IWKey);
        If (Authentic == 0) {
            RFLAGS.ZF := 1;
        } ELSE {
            DEST := AES128Encrypt (DEST, UnwrappedKey);
            RFLAGS.ZF := 0;
        }
    }
    RFLAGS.OF, SF, AF, PF, CF := 0;
```

3.6.4 Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

3.6.5 Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H:ECX.KL [bit 23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE [bit 0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

3.6.6 Intrinsics

unsigned char _mm_aesenc128kl_u8(__m128i* odata, __m128i idata, const void* h);

3.7 AESENC256KL

Opcode/Instruction	Op /En	64/32-bit Mode Support	CPUID Flag	Description
F3 0F 38 DE !(11):rrr:bbb AESENC256KL xmm, m512	A	V/V	AESKLE	Encrypt xmm using 256-bit AES key indicated by handle at m512 and store result in xmm.

3.7.1 Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

3.7.2 Description

The AESENC256KL instruction performs 14 rounds of AES to encrypt the first operand using the 256-bit key indicated by the handle from the second operand. It stores the result in the first operand.

3.7.3 Operation

```
AESENC256KL
    Handle := UnalignedLoad of 512 bit (SRC); // Load is not guaranteed to be atomic.
    Illegal Handle = (
                        HandleReservedBitSet (Handle) ||
                        (Handle[0] AND (CPL > 0)) ||
                        Handle [1]   ||
                        HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256
    );
    If (Illegal Handle) {
        RFLAGS.ZF := 1;
    } ELSE {
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], IWKey);
        If (Authentic == 0) {
            RFLAGS.ZF := 1;
        } ELSE {
            DEST := AES256Encrypt (DEST, UnwrappedKey) ;
            RFLAGS.ZF := 0;
        }
    }
    RFLAGS.OF, SF, AF, PF, CF := 0;
```

3.7.4 Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

3.7.5 Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H:ECX.KL [bit 23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE [bit 0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

3.7.6 Intrinsics

unsigned char _mm_aesenc256kl_u8(__m128i* odata, __m128i idata, const void* h);

3.8 AESENCWIDE128KL

Opcode/Instruction	Op /En	64/32-bit Mode Support	CPUID Flag	Description
F3 0F 38 D8 !(11):000:bbb AESENCWIDE128KL m384, <XMM0-7>	A	V/V	AESKLE WIDE_KL	Encrypt XMM0-7 using 128-bit AES key indicated by handle at m384 and store each resultant block back to its corresponding register.

3.8.1 Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2-9
A	NA	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

3.8.2 Description

The AESENCWIDE128KL instruction performs ten rounds of AES to encrypt each of the eight blocks in XMM0-7 using the 128-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding encrypted block.

3.8.3 Operation

```
AESENCWIDE128KL
    Handle := UnalignedLoad of 384 bit (SRC);  // Load is not guaranteed to be atomic.
    Illegal Handle = (
                        HandleReservedBitSet (Handle) ||
                        (Handle[0] AND (CPL > 0)) ||
                        Handle [1]   ||
                        HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128
                        );
    If (Illegal Handle) {
        RFLAGS.ZF := 1;
    } ELSE {
         (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IWKey);
        If Authentic == 0 {
            RFLAGS.ZF := 1;
        } ELSE {
            XMM0 := AES128Encrypt (XMM0, UnwrappedKey);
            XMM1 := AES128Encrypt (XMM1, UnwrappedKey);
            XMM2 := AES128Encrypt (XMM2, UnwrappedKey);
            XMM3 := AES128Encrypt (XMM3, UnwrappedKey);
            XMM4 := AES128Encrypt (XMM4, UnwrappedKey);
            XMM5 := AES128Encrypt (XMM5, UnwrappedKey);
            XMM6 := AES128Encrypt (XMM6, UnwrappedKey);
            XMM7 := AES128Encrypt (XMM7, UnwrappedKey);
            RFLAGS.ZF := 0;
        }
    }
    RFLAGS.OF, SF, AF, PF, CF := 0;
```

3.8.4 Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

3.8.5 Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H:ECX.KL [bit 23] = 0.
	If CR4.KL = 0.
	If CPUID.AESKLE = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
	If CPUID.19H:EBX.WIDE_KL [bit 2] = 0.
#NM	If CR0.TS = 1.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

3.8.6 Intrinsics

unsigned char _mm_aesencwide128kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);

3.9 AESENCWIDE256KL

Opcode/Instruction	Op /En	64/32-bit Mode Support	CPUID Flag	Description
F3 0F 38 D8 !(11):010:bbb AESENCWIDE256KL m512, <XMM0-7>	A	V/V	AESKLE WIDE_KL	Encrypt XMM0-7 using 256-bit AES key indicated by handle at m512 and store each resultant block back to its corresponding register.

3.9.1 Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2-9
A	NA	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

3.9.2 Description

The AESENCWIDE256KL instruction performs 14 rounds of AES to encrypt each of the eight blocks in XMM0-7 using the 256-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding encrypted block.

3.9.3 Operation

```
AESENCWIDE256KL
    Handle := UnalignedLoad of 512 bit (SRC); // Load is not guaranteed to be atomic.
    Illegal Handle = (
                        HandleReservedBitSet (Handle) ||
                        (Handle[0] AND (CPL > 0)) ||
                        Handle [1]   ||
                        HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256
    );
    If (Illegal Handle) {
        RFLAGS.ZF := 1;
    } ELSE {
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], IWKey);
        If (Authentic == 0) {
            RFLAGS.ZF := 1;
        } ELSE {
            XMM0 := AES256Encrypt (XMM0, UnwrappedKey);
            XMM1 := AES256Encrypt (XMM1, UnwrappedKey);
            XMM2 := AES256Encrypt (XMM2, UnwrappedKey);
            XMM3 := AES256Encrypt (XMM3, UnwrappedKey);
            XMM4 := AES256Encrypt (XMM4, UnwrappedKey);
            XMM5 := AES256Encrypt (XMM5, UnwrappedKey);
            XMM6 := AES256Encrypt (XMM6, UnwrappedKey);
            XMM7 := AES256Encrypt (XMM7, UnwrappedKey);
            RFLAGS.ZF := 0;
        }
    }
```

RFLAGS.OF, SF, AF, PF, CF := 0;

3.9.4 Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

3.9.5 Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.07H:ECX.KL [bit 23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE [bit 0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
	If CPUID.19H:EBX.WIDE_KL [bit 2] = 0.
#NM	If CR0.TS = 1.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.

3.9.6 Intrinsics

unsigned char _mm_aesencwide256kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);

3.10 ENCODEKEY128

Opcode/Instruction	Op /En	64/32-bit Mode Support	CPUID Flag	Description
F3 0F 38 FA 11:rrr:bbb ENCODEKEY128 r32, r32, <XMM0-2>, <XMM4-6>	A	V/V	AESKLE	Wrap a 128-bit AES key from XMM0 into a key handle and output handle in XMM0-2.

3.10.1 Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operands 4-5	Operands 6-8
A	NA	ModRM:reg (w)	ModRM:r/m (r)	Implicit XMM0 (r, w)	Implicit XMM1-2 (w)	Implicit XMM4-6 (w)

3.10.2 Description

The ENCODEKEY128 instruction wraps a 128-bit AES key from the implicit operand XMM0 into a key handle that is then stored in the implicit destination operands XMM0-2.

The explicit source operand specifies handle restrictions, if any.

The explicit destination operand is populated with information on the source of the key and its attributes. XMM4 through XMM6 are reserved for future usages and software should not rely upon them being zeroed.

3.10.3 Operation

ENCODEKEY128

```
#GP (0) if a reserved bit[1] in SRC[31:0] is set
InputKey[127:0] := XMM0;
KeyMetadata[2:0] = SRC[2:0];
KeyMetadata[23:3] = 0; // Reserved for future usage
KeyMetadata[27:24] = 0; // KeyType is AES-128 (value of 0)
KeyMetadata[127:28] = 0; // Reserved for future usage

// KeyMetadata is the AAD input and InputKey is the Plaintext input for WrapKey128
Handle[383:0] := WrapKey128(InputKey[127:0], KeyMetadata[127:0], IWKey.Integrity Key[127:0],
   IWKey.Encryption Key[255:0]);

DEST[0] := IWKey.NoBackup;
DEST[4:1] := IWKey.KeySource[3:0];
DEST[31:5] = 0;[2]
```

---

[1] SRC[31:3] are currently reserved for future usages. SRC[2], which indicates a no-decrypt restriction, is reserved if CPUID.19H:EAX[2] is 0. SRC[1], which indicates a no-encrypt restriction, is reserved if CPUID.19H:EAX[1] is 0. SRC[0], which indicates a CPL0-only restriction, is reserved if CPUID.19H:EAX[0] is 0.

.

```
XMM0 := Handle[127:0]; // AAD
XMM1 := Handle[255:128]; // Integrity Tag
XMM2 := Handle[383:256]; // CipherText
XMM4 := 0; // Reserved for future usage
XMM5 := 0; // Reserved for future usage
XMM6 := 0; // Reserved for future usage

RFLAGS.OF, SF, ZF, AF, PF, CF := 0;
```

3.10.4 Flags Affected

All arithmetic flags (OF, SF, ZF, AF, PF, CF) are cleared to 0. Although they are cleared for the currently defined operations, future extensions may report information in the flags.

3.10.5 Exceptions

#GP	If reserved bit is set in source register value.
#UD	If the LOCK prefix is used.
	If CPUID.07H:ECX.KL [bit 23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE [bit 0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.

3.10.6 Intrinsics

unsigned int _mm_encodekey128_u32(unsigned int htype, __m128i key, void* h);

3.11 ENCODEKEY256

Opcode/Instruction	Op /En	64/32-bit Mode Support	CPUID Flag	Description
F3 0F 38 FB 11:rrr:bbb ENCODEKEY256 r32, r32 <XMM0-6>	A	V/V	AESKLE	Wrap a 256-bit AES key from XMM1:XMM0 into a key handle and store it in XMM0-3.

3.11.1 Instruction Operand Encoding

Op/ En	Tuple	Operand 1	Operand 2	Operands 3-4	Operands 5-9
A	NA	ModRM:reg (w)	ModRM:r/m (r)	Implicit XMM0-1 (r, w)	Implicit XMM2-6 (w)

3.11.2 Description

The ENCODEKEY256 instruction wraps a 256-bit AES key from the implicit operand XMM1:XMM0 into a key handle that is then stored in the implicit destination operands XMM0-3.

The explicit source operand is a general-purpose register and specifies what handle restrictions should be built into the handle.

The explicit destination operand is populated with information on the source of the key and its attributes. XMM4 through XMM6 are reserved for future usages and software should not rely upon them being zeroed.

3.11.3 Operation

```
ENCODEKEY256
    #GP (0) if a reserved bit[1] in SRC[31:0] is set
    InputKey[255:0] := XMM1:XMM0;
    KeyMetadata[2:0] = SRC[2:0];
    KeyMetadata[23:3] = 0; // Reserved for future usage
    KeyMetadata[27:24] = 1; // KeyType is AES-256 (value of 1)
    KeyMetadata[127:28] = 0; // Reserved for future usage

    // KeyMetadata is the AAD input and InputKey is the Plaintext input for WrapKey128
    Handle[511:0] := WrapKey256(InputKey[255:0], KeyMetadata[127:0], IWKey.Integrity Key[127:0],
      IWKey.Encryption Key[255:0]);

    DEST[0] := IWKey.NoBackup;
    DEST[4:1] := IWKey.KeySource[3:0];
    DEST[31:5] = 0;
```

[1] SRC[31:3] are currently reserved for future usages. SRC[2], which indicates a no-decrypt restriction, is reserved if CPUID.19H:EAX[2] is 0. SRC[1], which indicates a no-encrypt restriction, is reserved if CPUID.19H:EAX[1] is 0. SRC[0], which indicates a CPL0-only restriction, is reserved if CPUID.19H:EAX[0] is 0.

```
XMM0 := Handle[127:0]; // AAD
XMM1 := Handle[255:128]; //  Tag
XMM2 := Handle[383:256]; // CipherText[127:0]
XMM3 := Handle[511:384]; // CipherText[255:128]

XMM4 := 0; // Reserved for future usage
XMM5 := 0; // Reserved for future usage
XMM6 := 0; Integrity// Reserved for future usage

RFLAGS.OF, SF, ZF, AF, PF, CF := 0;
```

3.11.4 Flags Affected

All arithmetic flags (OF, SF, ZF, AF, PF, CF) are cleared to 0. Although they are cleared for the currently defined operations, future extensions may report information in the flags.

3.11.5 Exceptions

#GP	If reserved bit is set in source register value.
#UD	If the LOCK prefix is used.
	If CPUID.07H:ECX.KL [bit 23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE [bit 0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.

3.11.6 Intrinsics

unsigned int _mm_encodekey256_u32(unsigned int htype, __m128i key_lo, __m128i key_hi, void* h);

3.12 LOADIWKEY

Opcode/Instruction	Op /En	64/32-bit Mode Support	CPUID Flag	Description
F3 0F 38 DC 11:rrr:bbb LOADIWKEY xmm1, xmm2, <EAX>, <XMM0>	A	V/V	KL	Load internal wrapping key from xmm1, xmm2, and XMM0.

3.12.1 Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r)	ModRM:r/m (r)	Implicit EAX (r)	Implicit XMM0 (r)

3.12.2 Description

The LOADIWKEY instruction writes the Key Locker internal wrapping key, which is called IWKey. This IWKey is used by the ENCODEKEY* instructions to wrap keys into handles. Conversely, the AESENC/DEC* instructions use IWKey to unwrap those keys from the handles and help verify the handle integrity. For security reasons, no instruction is designed to allow software to directly read the IWKey value.

IWKey includes two cryptographic keys as well as metadata. The two cryptographic keys are loaded from register sources so that LOADIWKEY can be executed without the keys ever being in memory.

The key input operands are:

- The 256-bit encryption key is loaded from the two explicit operands.
- The 128-bit integrity key is loaded from the implicit operand XMM0.

The implicit operand EAX specifies the KeySource and whether backing up the key is permitted:

- EAX[0] – When set, the wrapping key being initialized is not permitted to be backed up to platform-scoped storage.
- EAX[4:1] – This specifies the KeySource, which is the type of key. Currently only two encodings are supported. A KeySource of 0 indicates that the key input operands described above should be directly stored as the internal wrapping keys. LOADIWKEY with a KeySource of 1 will have random numbers from the on-chip random number generator XORed with the source registers (including XMM0) so that the software that executes the LOADIWKEY does not know the actual IWKey encryption and integrity keys. Software can choose to put additional random data into the source registers so that other sources of random data are combined with the hardware random number generator supplied value. Software should always check ZF after executing LOADIWKEY with KeySource of 1 as this operation may fail due to it being unable to get sufficient full-entropy data from the on-chip random number generator. Both KeySource of 0 and 1 specify that IWKey be used with the AES-GCM-SIV algorithm. CPUID.19H.ECX[1] enumerates support for KeySource of 1. All other KeySource encodings are reserved.
- EAX[31:5] – Reserved.

3.12.3 Operation

```
LOADIWKEY
    IF CPL > 0 { //  LOADKWKEY only allowed at ring 0 (supervisor mode)
        #GP (0);
    }
    IF “LOADIWKEY exiting” VM execution control set {
        VMexit;
    }
    IF EAX[4:1] > 1  {  // Reserved KeySource encoding used
        #GP (0);
    }
    IF EAX[31:5] != 0 {  // Reserved bit in EAX is set
        #GP (0);
    }
    IF EAX[0] AND (CPUID.19H.ECX[0] == 0) { // NoBackup is not supported on this part
         #GP (0);
    }
    IF (EAX[4:1] == 1) AND (CPUID.19H.ECX[1] == 0) { // KeySource of 1 is not supported on this part
        #GP (0);
    }
    IF (EAX[4:1] == 0) { // KeySource of 0.
            IWKey.Encryption Key[127:0] := SRC2[127:0]:
            IWKey.Encryption Key[255:128] := SRC1[127:0];
            IWKey.IntegrityKey[127:0] := XMM0[127:0];
            IWKey.NoBackup = EAX [0];
            IWKey.KeySource = EAX [4:1];
            RFLAGS.ZF := 0;
    } ELSE { // KeySource of 1. See RDSEED definition for details of randomness
            IF HW_NRND_GEN.ready == 1 { // Full-entropy random data from RDSEED was received
                IWKey.Encryption Key[127:0] := SRC2[127:0] XOR HW_NRND_GEN.data[127:0];
                IWKey.Encryption Key[255:128] := SRC1[127:0] XOR HW_NRND_GEN.data[255:128];
                IWKey.Encryption Key[255:0] := SRC2[127:0]:SRC1[127:0] XOR HW_NRND_GEN.data[255:0];
                IWKey.IntegrityKey[127:0] := XMM0[127:0] XOR HW_NRND_GEN.data[383:256];
                IWKey.NoBackup = EAX [0];
                IWKey.KeySource = EAX [4:1];
                RFLAGS.ZF := 0;
            } ELSE {  // Random data was not returned from RDSEED. IWKey was not loaded
                RFLAGS.ZF := 1;
            }
    }

    RFLAGS.OF, SF, AF, PF, CF := 0;
```

3.12.4 Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to full-entropy random data not being received from RDSEED. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

3.12.5 Exceptions

#GP	If CPL > 0.
	If EAX[4:1] > 1.
	If EAX[31:5] != 0.
	If (EAX[0] == 1) AND (CPUID.19H.ECX[0] == 0).
	If (EAX[4:1] == 1) AND (CPUID.19H.ECX[1] == 0).
#UD	If the LOCK prefix is used.
	If CPUID.07H:ECX.KL [bit 23] = 0.
	If CR4.KL = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.

3.12.6 Intrinsics

void _mm_loadiwkey(unsigned int ctl, __m128i intkey, __m128i enkey_lo, __m128i enkey_hi);

4 Backup MSRs

4.1 Backing Up and Restoring the Internal Wrapping Key

When IWKeyBackup support is enumerated, the logical processor scoped IWKey can be copied to or from a platform scoped state called IWKeyBackup. Copying IWKey to IWKeyBackup is called 'backing up IWKey' and copying IWKeyBackup to IWKey is called 'restoring IWKey'.

IWKeyBackup and the path between it and IWKey are designed to be protected against software and simple hardware attacks (e.g., encrypted and integrity protected on physical buses between sockets). This means that IWKeyBackup can be used to distribute an IWKey within the logical processors in a platform in a more protected manner. One logical processor can write IWKey with a secret value and then back up that IWKey to IWKeyBackup. The other logical processors can then copy IWKeyBackup to their own IWKey.

IWKeyBackup is also maintained across S3 (sleep) and S4 (hibernate) sleep states on platforms supporting those states. This allows the OS to use IWKeyBackup to back up a platform's IWKey across S3 and S4 sleep states to maintain handles using that IWKey across those sleep states.

IWKeyBackup may also be maintained across S5 (soft off) and G3 (mechanical off) state on some systems, but this is not architecturally guaranteed and thus software should not depend on that. Because it may be maintained, software that needs to ensure that all handles are truly revoked (e.g., before powering off the system) should overwrite IWKeyBackup so that it is not available later to make those handles work.

It is also possible for a VMM to use IWKeyBackup to maintain the host's IWKey so that it can be restored after executing a guest (which ran with the guest's IWKey loaded). Because of the high latency of copying to and from IWKeyBackup on current processors, this would not be performant to execute frequently (e.g., before each VM entry or after each VM exit) and these MSRs should not be included in VM entry or VM exit MSR load areas.

Backing up or restoring IWKey involves several MSRs, all of which are enumerated by CPUID.19H:EBX[4]:

- IA32_COPY_LOCAL_TO_PLATFORM (write-only, address D91H)
- IA32_COPY_PLATFORM_TO_LOCAL (write-only, address D92H)
- IA32_COPY_STATUS (read-only, address 990H)
- IA32_IWKEYBACKUP_STATUS (read-only, address 991H)

IA32_COPY_LOCAL_TO_PLATFORM is a write-only MSR that can be used to issue commands to copy data from the current logical processor to a platform scoped state. It can be used to copy IWKey for this logical processor to the IWKeyBackup register for the platform.

IA32_COPY_PLATFORM_TO_LOCAL is a write-only MSR that can be used to issue commands to copy data from platform scoped state to the current logical processor. It can be used to copy IWKeyBackup for the platform to the IWKey for this logical processor.

IA32_COPY_STATUS is a read-only logical processor scoped MSR that indicates whether the most recent write to IA32_COPY_PLATFORM_TO_LOCAL or IA32_COPY_LOCAL_TO_PLATFORM executed from this logical processor succeeded or failed.

IA32_IWKEYBACKUP_STATUS is a read-only MSR that indicates attributes of IWKeyBackup.

On some systems, system firmware enabling of Key Locker may be needed for CPUID.19H:EBX[4] to enumerate as 1.

4.2 IA32_COPY_LOCAL_TO_PLATFORM MSR

IA32_COPY_LOCAL_TO_PLATFORM MSR is designed to support copying IWKey content to the IWKeyBackup register when a WRMSR sets IA32_COPY_LOCAL_TO_PLATFORM[0]. It is possible for this write to fail, for example because IWKey.NoBackup is set, so software should always check after a write that it succeeded.

Software can determine whether a copy of IWKey to IWKeyBackup succeeded through checking IA32_COPY_STATUS[0] immediately after the WRMSR to set IA32_COPY_LOCAL_TO_PLATFORM[0]. A value of 1 in IA32_COPY_STATUS[0] indicates that the write to IWKeyBackup succeeded.

IWKeyBackup also includes an integrity measurement. It may be possible to corrupt IWKeyBackup due to two simultaneous writes from different logical processors that are writing different values. It is also possible to corrupt an IWKeyBackup read (restore) due to that read occurring simultaneously with a write of a different value. Because of the IWKeyBackup integrity measurement, the CPU is designed to detect that the IWKeyBackup read would be of a corrupted value and in that situation it is designed not to mark the read valid (meaning IA32_COPY_STATUS[0] will be 0) or modify IWKey.

After IWKeyBackup is written, the platform will ensure it is written to a storage which is persistent across sleep (S3) and hibernate (S4) sleep states. If this persistent storage is not on the same die as IWKeyBackup, then it is designed to be encrypted, integrity protected and (when available) replay protected. An older write to IWKeyBackup (IA32_COPY_LOCAL_TO_PLATFORM[0] set) that has not yet completed updating the persistent storage (and thus has not yet set IA32_IWKEYBACKUP_STATUS[Backup/Restore Valid] may cause younger writes to fail.

Although younger writes to IWKeyBackup may be blocked by an older write that has not yet updated the persistent storage, reads are not blocked. A read of IWKeyBackup (using IA32_COPY_PLATFORM_TO_LOCAL[0]) may be initiated (e.g., on another logical processor) as soon as a write is verified to have completed successfully (IA32_COPY_STATUS[0] was set).

Register address		Architectural MSR Name / Bit Fields	MSR/Bit Description	Comment
Hex	Decimal			
D91	3473	IA32_COPY_LOCAL_TO_PLATFORM	Copy local state to platform state (W)	IF ((CPUID.19H:EBX[4] == 1) && (CPUID.(07H,0).ECX[23] == 1))
		0	IWKeyBackup - Copy IWKey to IWKeyBackup	IF ((CPUID.19H:EBX[4] == 1) && (CPUID.(07H,0).ECX[23] == 1))
		63:1	Reserved	

4.3 IA32_COPY_PLATFORM_TO_LOCAL MSR

IA32_COPY_PLATFORM_TO_LOCAL MSR is designed to support copying IWKeyBackup content (a platform register) to the IWKey register of the current logical processor when a WRMSR sets IA32_COPY_PLATFORM_TO_LOCAL[0]. It is possible for this write to fail, for example if it is reading IWKeyBackup simultaneously with a write to IWKeyBackup, so software should always check after the write that it succeeded. Software can determine whether a copy of IWKeyBackup to IWKey succeeded through checking IA32_COPY_STATUS[0] after the WRMSR to set IA32_COPY_PLATFORM_TO_LOCAL[0]; a value of 1 in IA32_COPY_STATUS[0] indicates that the write to IWKey succeeded.

Register address		Architectural MSR Name / Bit Fields	MSR/Bit Description	Comment
Hex	Decimal			
D92	3474	IA32_COPY_PLATFORM_TO_LOCAL	Copy platform state to local state (W)	IF ((CPUID.19H:EBX[4] == 1) && (CPUID.(EAX=07H, ECX=0H).ECX[23] == 1))
		0	IWKeyBackup - Copy IWKeyBackup to IWKey	IF ((CPUID.19H:EBX[4] == 1) && (CPUID.(EAX=07H, ECX=0H).ECX[23] == 1))
		63:1	Reserved	

4.4 IA32_COPY_STATUS MSR

Each bit in the read-only IA32_COPY_STATUS MSR at address 990H indicates whether the most recent command executed through the corresponding bit in IA32_PLATFORM_TO_LOCAL MSR or IA32_LOCAL_TO_PLATFORM MSR was successful (a value of 1) or unsuccessful (a value of 0). The reset value is 0.

This MSR is logical processor scoped and can be read immediately after an MSR write to cause a backup or restore operation. It is cleared by cold/warm reset and unaffected by INIT.

Register address		Architectural MSR Name / Bit Fields	MSR/Bit Description	Comment
Hex	Decimal			
990	2448	IA32_COPY_STATUS	Status of most recent platform to local or local to platform copies	IF ((CPUID.19H:EBX[4] == 1) && (CPUID.(07H,0).ECX[23] == 1))
		0	IWKEY_COPY_SUCCESSFUL: Status of most recent copy to or from IWKeyBackup	IF ((CPUID.19H:EBX[4] == 1) && (CPUID.(07H,0).ECX[23] == 1))
		63:1	Reserved	

4.5 IA32_IWKEYBACKUP_STATUS MSR

The IA32_IWKEYBACKUP_STATUS MSR (read-only, platform scoped) provides information about the status of the Key Locker IWKeyBackup register. It is cleared on a successful copy of IWKey to IWKeyBackup by any logical processor (one that sets IA32_COPY_STATUS[IWKEY_COPY_SUCCESSFUL]) and becomes evaluated again at a later point. It is also cleared by cold/warm reset and unaffected by INIT.

Register address		Architectural MSR Name / Bit Fields	MSR/Bit Description	Comment
Hex	Decimal			
991	2449	IA32_IWKEYBACKUP_STATUS	Information about IWKeyBackup register	IF ((CPUID.19H:EBX[4] == 1) && (CPUID.(07H,0).ECX[23] == 1))
		0	**Backup/restore valid.** Cleared when a write to IWKeyBackup is initiated, and then set when the latest write of IWKeyBackup has been written to storage that persists across	IF ((CPUID.19H:EBX[4] == 1) && (CPUID.(07H,0).ECX[23] == 1))

Register address		Architectural MSR Name / Bit Fields	MSR/Bit Description	Comment
Hex	Decimal			
			S3/S4 sleep state. If S3/S4 is entered between when an IWKeyBackup write occurs and when this bit is set, then IWKeyBackup may not be recovered after S3/S4 exit. During S3/S4 sleep state exit (system wakeup), this bit is cleared. It is set again when IWKeyBackup is restored from persistent storage and thus available to be copied to IWKey using IA32_COPY_PLATFORM_TO_LOCAL MSR. Another write to IWKeyBackup (via IA32_COPY_LOCAL_TO_PLATFORM MSR) may fail if a previous write has not yet set this bit.	
		1	Reserved	
		2	**Backup key storage read/write error.** Updated prior to backup/restore valid being set. Set when an error is encountered while backing up or restoring a key to persistent storage	IF ((CPUID.19H:EBX[4] == 1) && (CPUID.(07H,0).ECX[23] == 1))
		3	**IWKeyBackup consumed.** Set after the previous backup operation has been consumed by the platform. This does not indicate that the system is ready for a second IWKeyBackup write as the previous IWKeyBackup write may still need to set Backup/restore valid.	IF ((CPUID.19H:EBX[4] == 1) && (CPUID.(07H,0).ECX[23] == 1))
		63:4	Reserved	

5 OS Enabling

5.1 OS Boot

On processors where CPUID.KL is enumerated, the OS boot process should set CR4.KL and then execute LOADIWKEY with a random number as input. When support for HW randomized IWKey and IWKeyBackup is enumerated (CPUID.19H.ECX[1] and CPUID.19H.EBX[4] are both set), software may wish LOADIWKEY to specify a KeySource of 1 in order to have hardware random numbers be combined with whatever is specified by software (e.g., random numbers from a software defined pool). When specifying a KeySource of 1, software should check after LOADIWKEY that RFLAGS.ZF is 0 and retry the LOADIWKEY if RFLAGS.ZF is 1 (which indicates that a lack of availability of full-entropy data caused LOADIWKEY to not complete). If repeated LOADIWKEYs do not succeed (RFLAGS.ZF set each time), then full-entropy data from the on-chip random number generator was unavailable and software may wish to instead use a KeySource of 0.

The OS should not allow applications to execute with CR4.KL set unless LOADIWKEY successfully loaded as applications may presume that IWKey is properly initialized when CR4.KL is set.

An OS should ensure that the IWKey is the same on each logical processor so that an application's handles work regardless of which logical processor it is executing on. This can be done by using a software specified IWKey (KeySource of 0) and passing that through memory to the other logical processors so that each can execute LOADIWKEY with the same value. Alternatively, it can be done by executing LOADIWKEY on one logical processor, then copying the IWKey from that logical processor to IWKeyBackup by using IA32_COPY_LOCAL_TO_PLATFORM MSR, and then, on every other logical processor, copying to IWKey from IWKeyBackup using IA32_COPY_PLATFORM_TO_LOCAL MSR. Software can immediately do a copy to IWKey from IWKeyBackup after a successful backup (write) to IWKeyBackup completed (IA32_COPY_STATUS[0] was set on the writing logical processor), without needing to check IA32_IWKEYBACKUP_STATUS MSR. A second write to IWKeyBackup may need to wait until the previous write has completed and set IA32_IWKEY_BACKUP_STATUS[Backup/Restore Valid]. Most usages will not need to do a second write to IWKeyBackup soon after boot, but it may occur when a system quickly does a kernel soft reset or revokes the handles by overwriting all IWKeys as well as IWKeyBackup; particularly in artificial test environments. IWKeyBackup supports both software specified IWKey (KeySource of 0) as well as keys combined with data from the on-chip hardware random number generator(KeySource of 1) and is enumerated through CPUID.19H.EBX[4].

By loading IWKey early in the boot process (before most software components are loaded) and not recording the IWKey value, vulnerabilities in later loaded software components will not be able to single step the system to watch the IWKey value being loaded (as the IWKey loading has already occurred).

On some platforms, system firmware enabling may be needed for Key Locker to be used. On such platforms, if the system firmware has not properly enabled Key Locker then CPUID.AESKLE may stay 0 even though CR4.KL is set. It is also possible that some systems will not enumerate support for IWKeyBackup if system firmware did not properly enable Key Locker.

On systems that support S3 and S4 sleep states, the OS may want to copy the LoadIWKey value to IWKeyBackup by setting IA32_COPY_LOCAL_TO_PLATFORM[0] at boot. This gives the platform more time to write IWKeyBackup into persistent storage (and set IA32_IWKEYBACKUP_STATUS[0]) before there is a request to enter S3 or S4 sleep states (in order that entry to those sleep states is not delayed. Platforms that support S3 and S4 sleep states but do not support IWKeyBackup will need to find some other means to maintain the IWKey across the S3/S4 sleep states. If no other means is available, they may choose to not enable Key Locker.

5.2 OS Shutdown

Shutting down the system (either S5 soft off or G3 mechanical off) involves closing all applications and thus is a good time to revoke handles. Although the next boot of the OS will normally load a new IWKey and thus revoke those handles, some security usage models may not want to trust the next boot of the OS to perform that revocation (e.g., in case the OS is swapped with a malicious one).

In order to reduce the risk that an attacker could re-use handles across the shutdown even in such situations, the OS can overwrite IWKey and IWKeyBackup before powering off the system. Specifically, it can use LOADIWKEY on each logical processor to overwrite IWKey (e.g., with zeroes or a new random value) and can copy an overwritten IWKey from one of those logical processors to IWKeyBackup using IA32_COPY_LOCAL_TO_PLATFORM[0]. Note that this write to IWKeyBackup may fail if a previous write to IWKeyBackup (e.g., the boot write) has not already completed and set IA32_IWKEY_BACKUP_STATUS[Backup/Restore Valid]. Before resetting the processor, software can help ensure that this new overwriting of IWKeyBackup has overwritten its persistent storage copy by waiting for IA32_IWKEYBACKUP_STATUS[Backup/Restore Valid] to be set again.

5.3 Entering S3 or S4 System Sleep States (Sleep or Hibernate)

Entering S3 (sleep) or S4 (hibernate) states will power off all processors and thus will lead to losing the IWKey (as it is cleared by reset when the system is powered back on). In order to maintain IWKey (so that application handles created before entering S3 or S4 continue to work after waking up), software should make sure that IWKeyBackup is written (e.g., at boot) and IA32_IWKEYBACKUP_STATUS[Backup/Restore Valid] is set so that it is maintained across S3/S4. If there was an error when writing to persistent storage, then IA32_IWKEYBACKUP_STATUS[Backup key storage read/write error] will be set.

5.4 Exiting S3 or S4 System Sleep States (Waking from Sleep or Hibernate)

On waking up from S3 or S4 sleep states, the OS will want to recover the previous IWKey so that application handles created before entering S3/S4 state continue to work after waking up from S3/S4.

If the previous IWKeyBackup completed its write to persistent storage before entering S3/S4 sleep states, then the restoration process will automatically start on exiting S3/S4 sleep states.

The OS should check that IWKeyBackup is ready to be copied by waiting until IA32_IWKEYBACKUP_STATUS[Backup/Restore Valid] is set. When IWKeyBackup is ready, WRMSR to set IA32_COPY_PLATFORM_TO_LOCAL[0] can be executed on each logical processor in order to restore IWKey on those logical processors. If the OS is unable to properly restore IWKey after an S3/S4 (e.g. IA32_IWKEYBACKUP_STATUS[Backup/restore valid] is not set after a retry or the IWKey restore from IWKeyBackup fails), then it may want to log an error and either clear CR4.KL or shutdown the system.

6 Application Enabling

Applications can either directly use the Key Locker instructions or can use a software library that allows selecting Key Locker for maintenance of its AES keys.

Software can use CPUID.AESKLE to determine that the system and OS support Key Locker. This CPUID bit is only set when the OS has set CR4.KL. Any OS which has done that should also have written IWKey to a random value[1].

Once the application has obtained the AES key that it wants to use (e.g., the result of key negotiation with another entity or what is unsealed from a TPM), it should use ENCODEKEY128 (for 128-bit AES key) or ENCODEKEY256 (for 256-bit AES key) to create the handle. Handle restrictions (e.g., no-encrypt or no-decrypt) can be specified through the SRC register and are detailed in section 1.1.1.1.

Along with the handle, ENCODEKEY128 and ENCODEKEY256 also produce information about the IWKey used to create the handle. For example, bit 0 of the destination register indicates whether IWKey is forbidden from being written to IWKeyBackup. A value of 0 in bits 4:1 of the destination register indicates that the IWKey used to help protect this handle was specified by system software and a value of 1 indicates that the IWKey is random and thus is designed not to be known by any software (including system software).

Intel SGX enclaves that do not have system software within their trust boundary may be designed to refuse to use an IWKey that does not use the on-chip hardware random number generator in order to help ensure that Key Locker handles provide defense in depth against system software attackers that also obtain the handle; but this may limit Key Locker usage. Note that Key Locker is designed so that it does not significantly reduce security for enclave software to use it when a system software adversary knows the IWKey value, although such usage also does not help improve the enclave security.

When the application wishes to perform encryption or decryption instructions, it should pass the handle along with the corresponding plaintext or ciphertext to the AES*KL instructions. Software doing parallel AES operations (like AES-CTR mode) may wish to use the wide instructions in order to encrypt/decrypt multiple blocks in parallel. Software doing serial AES operations (like AES-CBC encryption) should use the non-wide instructions. As an example, software doing an AES CBC encryption using a 256-bit AES key should use AESENC256KL.

If a handle or IWKey becomes corrupted (e.g., through malicious system software that changes the IWKey) or a restriction is violated, the AES*KL fails and sets RFLAGS.ZF. In this situation, the destination is unmodified and thus holds plaintext (if an encryption operation is being performed) or ciphertext (if a decryption operation is being performed). Software should thus be careful to check that ZF is 0 after each execution of an AES*KL instruction.

The application should protect the handle as it would normally protect a key; even though it is designed to not be usable remotely or after handle revocation, it could still be used by an adversary on that system until handles are revoked (by the OS or VMM overwriting IWKey and any backup of it).

---

[1] Note that if IWKey is not initialized (and thus all of its fields are 0), then an ENCODEKEY128 with input of 0 will create a handle with an integrity tag of 0x8720849214a248ad_898940a278c095dc and ciphertext of 0xd3e9d22b334fb3c2_3382228c8474c308. The AES GCM SIV algorithm's integrity check requires a non-zero IWKey to properly detect changes in the handle.

7 Virtualization Support

7.1 Virtualization Strategies

A VMM may need the ability to be able to context switch and migrate guests as well as have the ability to save a guest to disk and resume it later (e.g., after the host has rebooted and thus revoked its handles).

A VMM can do this by causing a VM exit when a guest loads IWKey and recording the IWKey that the guest expects. The VMM can then load this IWKey (using LOADIWKEY) before running that guest. This may mean doing a LOADIWKEY on each context switch to a guest that is using Key Locker (has CR4.KL set).

Note that a VMM may wish to enumerate no support for hardware generated random IWKeys to the guest (i.e., enumerate CPUID.19H:ECX[1] as 0) as such IWKeys cannot be easily context switched. A guest ENCODEKEY* will return the type of IWKey used (IWKey.KeySource) and thus is designed to notice if a VMM virtualized a hardware generated random IWKey with a software specified IWKey.

Although a system using virtualization will need to hold the IWKey of guests so that they can be loaded on guest context switch, an attacker of a guest not only needs to recover the handle but also needs to observe the IWKey values maintained by the VMM in order to unwrap that handle and recover the AES key. This is more difficult for an attacker than simply stealing the handle.

Because the IWKey is designed not to be directly readable from the processor, a VMM that starts after IWKey is loaded should be unable to determine that IWKey value. If the VMM is also using Key Locker (separate from guest usage), then each loading of a guest IWKey will overwrite the VMM's IWKey. The VMM that needs to use Key Locker may either need to save away its own IWKey in memory/registers (which would impact security as an adversary that can observe arbitrary VMM memory may be able to steal both the handles and IWKey as well as require the VMM to be running before the first IWKey load) or need to use IWKeyBackup to restore its own IWKey before using VMM handles or not allow guest usage of Key Locker (so that no guest IWKey needs to be loaded). Initial implementations may take a significant amount of time to perform a copy of IWKeyBackup to IWKey (via an MSR write to IA32_COPY_PLATFORM_LOCAL[0]) so it may cause a significant performance impact to reload IWKey after each VM exit. It is thus recommended that VMMs only restore IWKey right before needing to use a handle, or even avoid using Key Locker for VMM usages if Key Locker is enumerated to guests (so that either the VMM uses Key Locker or the guests use Key Locker but not both).

The VMM can use the MSR bitmap to catch guest usage of IA32_COPY_LOCAL_TO_PLATFORM, IA32_COPY_PLATFORM_TO_LOCAL, IA32_COPY_STATUS, and IA32_IWKEYBACKUP_STATUS MSRs. They can be virtualized through software recording the guest expectation of IWKeyBackup as well as IWKey. This avoids the guest needing to actually read or write the actual platform IWKeyBackup; the guest interaction would be with the virtual IWKeyBackup.

VMMs that are unaware of Key Locker should not allow guests to set CR4.KL (as the VMM should already be preventing guests from setting CR4 bits of which the VMM is not aware). This will cause Key Locker instructions to generate a #UD exception on guests of such VMMs.

7.2 Tertiary Processor-Based VM-Execution Controls

A new 64-bit vector of controls is defined to govern the handling of synchronous events, mainly those caused by the execution of specific instructions. This vector is called the tertiary processor-based VM-execution controls. Software can use the VMREAD and VMWRITE instructions to access the tertiary processor-based VM-execution controls using the encoding pair 2034H/2035H.

Bit 17 of the primary processor-based VM-execution controls is defined as "activate tertiary controls." It determines whether the tertiary processor-based VM-execution controls are used. If that bit is 0, VM entry and VMX non-root operation function as if all the tertiary processor-based VM-execution controls were 0. Processors that support only the 0-setting of bit 17 of the primary processor-based VM-execution controls do not support the tertiary processor-based VM-execution controls. Thus, a processor supports the tertiary processor-based VM-execution controls if and only if IA32_VMX_PROCBASED_CTLS[49] = 1.[1]

Processors that support the tertiary processor-based VM-execution controls also support the IA32_VMX_PROCBASED_CTLS3 MSR (index 492H). This MSR enumerates the allowed 1-settings of these controls. Specifically, VM entry allows bit X of the tertiary processor-based VM-execution controls to be 1 if and only if bit X of the MSR is set to 1. If bit X of the MSR is cleared to 0, VM entry fails if control X and the "activate tertiary controls" primary processor-based VM-execution control are both 1.

7.3 LOADIWKEY VM Exiting

For a VMM to capture the IWKey values of guests, processors that support Key Locker also support a new "LOADIWKEY exiting" VM-execution control in bit 0 of the tertiary processor-based VM-execution controls. A processor supports the 1-setting of this control if it sets bit 49 of the IA32_VMX_PROCBASED_CTLS MSR and bit 0 of the IA32_VMX_PROCBASED_CTLS3 MSR. See Section 7.2 for details.

If the "activate tertiary controls" VM-execution control and the "LOADIWKEY exiting" VM-execution control are both 1, an execution of LOADIWKEY in VMX non-root operation causes a VM exit. Such a VM exit uses a basic exit reason of 45H (69 decimal) but no exit qualification. The length of the instruction is stored in the VM-exit instruction-length field and information about the instruction operands is stored in the VM-exit instruction-information field (details are provided in Table 7-1).

Table 7-1. Format of the VM-Exit Instruction-Information Field as Used for LOADIWKEY

Bit Position(s)	Contents
2:0	Undefined
6:3	Reg1: 0=XMM0 1=XMM1 … 7=XMM7 8-15 represent XMM8-XMM15, respectively (used only on processors that support Intel® 64 architecture)
9:7	Undefined
10	Set to 1 (reg format)
14:11	Undefined
17:15	Undefined (Segreg)
27:18	Undefined (IndexReg and BaseReg)

[1] If IA32_VMX_BASIC[55] = 1, the IA32_VMX_TRUE_PROCBASED_CTLS MSR exists and is identical to IA32_VMX_PROCBASED_CTLS in bit positions 63:32. Thus, if IA32_VMX_BASIC[55] = 1, a processor supports the tertiary processor-based VM-execution controls if and only if IA32_VMX_TRUE_PROCBASED_CTLS [49] = 1.

31:28	Reg2: 0=XMM0 1=XMM1 … 7=XMM7 8-15 represent XMM8-XMM15, respectively (used only on processors that support Intel 64 architecture)

Appendix A

A.2 Key Locker Performance

The long-term goal for Key Locker is to have performance that is comparable with direct usage of AES-NI instructions. Serial algorithm modes like AES CBC encryption should use instructions that encrypt or decrypt a single block at a time (e.g., AESENC128KL), while parallel algorithm modes like AES CTR or AES GCM should use instructions that encrypt or decrypt multiple blocks at a time (e.g., AESENCWIDE128KL).

In order to improve performance, the processor may store recently used handles and their associated keys or round keys for the currently loaded IWKey in a 'handle cache'. For security reasons, this handle cache is designed not to be accessible except through using the Key Locker instructions; the keys or round keys are designed to be not directly readable by software. The size of the handle cache may differ between implementations.

Some initial implementations may have lower performance than direct usage of AES-NI, particularly serial algorithms.

A.3 Functions Used in Instructions

A.3.1 HandleReservedBitSet (Handle)

This will evaluate to true if a reserved bit is set in the handle. The reserved bits are bits 127:28 and 23:3 of the handle.

A.3.2 HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128

This will evaluate to true if the Handle key type field (bits 27:24) is not 0.

A.3.3 HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256

This will evaluate to true if the Handle Key Type field (bits 27:24) is not 1.

A.3.4 UnwrapKeyAndAuthenticate384 (Handle[383:0], IWKey)

Intel Key Locker uses the AES-GCM-SIV algorithm to unwrap and verify the integrity handles using IWKey.EncryptionKey as the encryption key and IWKey.IntegrityKey as the integrity key[1]. The results will be whether the integrity check passes and the decrypted result. The unwrapped result should not be used if the integrity check fails.

The AES-GCM-SIV algorithm is described in detail in section A.4. This algorithm uses AES GCM SIV with input of 128 bits of ciphertext (handle[383:256]), 128 bits of additional authenticated data (handle[127:0]), and 128 bits of an integrity tag (handle[255:128]) and it results in a 128-bit unwrapped result as well as an indication of whether the integrity check passes.

---

[1] Note that the integrity key is called the message-authentication key and the encryption key is called the message-encryption key in the AES-GCM-SIV RFC8452 specification at https://tools.ietf.org/html/rfc8452

A.3.5 UnwrapKeyAndAuthenticate512 (Handle[511:0], IWKey)

Intel Key Locker uses the AES-GCM-SIV algorithm to unwrap and verify the integrity handles using IWKey.EncryptionKey as the encryption key and IWKey.IntegrityKey as the integrity key[1]. The results will be whether the integrity check passes and the decrypted result. The unwrapped result should not be used if the integrity check fails.

The AES-GCM-SIV algorithm is described in detail in section A.4. This algorithm uses AES GCM SIV with input of 256 bits of ciphertext (handle[511:256]), 128 bits of additional authenticated data (handle[127:0]), and 128 bits of an integrity tag (handle[255:128]) and it results in a 256-bit unwrapped result as well as an indication of whether the integrity check passes.

A.3.6 WrapKey128(Plaintext[127:0], AAD[127:0], Integrity Key[127:0], Encryption Key[255:0])

Intel Key Locker uses the AES-GCM-SIV algorithm to unwrap handles and verify their integrity.

The AES-GCM-SIV algorithm is described in detail in section A.4. Note that the C-code in that section contains a char *handle input that holds the output of the function.

This version is for 128-bit plaintext.

A.3.7 WrapKey256(Plaintext[255:0], AAD[127:0], Integrity Key[127:0], Encryption Key[255:0])

Intel Key Locker uses the AES-GCM-SIV algorithm to unwrap handles and verify their integrity.

The AES-GCM-SIV algorithm is described in detail in section A.4. Note that the C-code in that section contains a char *handle input that holds the output of the function.

This version is for 256-bit plaintext.

A.4 AES ECB Algorithm

This section specifies the algorithm for AES ECB including AES128Encrypt, AES128Decrypt, AES256Encrypt, AES256Decrypt functions which are used in the Key Locker pseudocode. AES is also described in the NIST FIPS 197 document.

A.4.1 AES-128 Key Expansion

```
#include <wmmintrin.h>
inline __m128i AES_128_ASSIST (__m128i temp1, __m128i temp2)
 {
      __m128i temp3;
      temp2 = _mm_shuffle_epi32 (temp2 ,0xff);
      temp3 = _mm_slli_si128 (temp1, 0x4);
      temp1 = _mm_xor_si128 (temp1, temp3);
      temp3 = _mm_slli_si128 (temp3, 0x4);
      temp1 = _mm_xor_si128 (temp1, temp3);
      temp3 = _mm_slli_si128 (temp3, 0x4);
      temp1 = _mm_xor_si128 (temp1, temp3);
```

[1] Note that the integrity key is called the message-authentication key and the encryption key is called the message-encryption key in the AES-GCM-SIV RFC8452 specification at https://tools.ietf.org/html/rfc8452

```
    temp1 = _mm_xor_si128 (temp1, temp2);
    return temp1;
}

void AES_128_Key_Expansion (const unsigned char *userkey,  unsigned char *key)
 {
    __m128i temp1, temp2;
    __m128i *Key_Schedule = (__m128i*)key;
    temp1 = _mm_loadu_si128((__m128i*)userkey);
    Key_Schedule[0] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1 ,0x1);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[1] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x2);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[2] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x4);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[3] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x8);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[4] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x10);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[5] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x20);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[6] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x40);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[7] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x80);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[8] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x1b);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[9] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x36);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[10] = temp1;
 }
```

A.4.2 AES-256 Key Expansion

```
#include <wmmintrin.h>
inline void KEY_256_ASSIST_1(__m128i* temp1, __m128i * temp2)
{
    __m128i temp4;
    *temp2 = _mm_shuffle_epi32(*temp2, 0xff);
    temp4 = _mm_slli_si128 (*temp1, 0x4);
    *temp1 = _mm_xor_si128 (*temp1, temp4);
    temp4 = _mm_slli_si128 (temp4, 0x4);
    *temp1 = _mm_xor_si128 (*temp1, temp4);
    temp4 = _mm_slli_si128 (temp4, 0x4);
```

```
        *temp1 = _mm_xor_si128 (*temp1, temp4);
        *temp1 = _mm_xor_si128 (*temp1, *temp2);
}
inline void KEY_256_ASSIST_2(__m128i* temp1, __m128i * temp3)
{
        __m128i temp2,temp4;
        temp4 = _mm_aeskeygenassist_si128 (*temp1, 0x0);
        temp2 = _mm_shuffle_epi32(temp4, 0xaa);
        temp4 = _mm_slli_si128 (*temp3, 0x4);
        *temp3 = _mm_xor_si128 (*temp3, temp4);
        temp4 = _mm_slli_si128 (temp4, 0x4);
        *temp3 = _mm_xor_si128 (*temp3, temp4);
        temp4 = _mm_slli_si128 (temp4, 0x4);
        *temp3 = _mm_xor_si128 (*temp3, temp4);
        *temp3 = _mm_xor_si128 (*temp3, temp2);
}
void AES_256_Key_Expansion (const unsigned char *userkey, unsigned char *key)
{
        __m128i temp1, temp2, temp3;
        __m128i *Key_Schedule = (__m128i*)key;
        temp1 = _mm_loadu_si128((__m128i*)userkey);
        temp3 = _mm_loadu_si128((__m128i*)(userkey+16));
        Key_Schedule[0] = temp1;
        Key_Schedule[1] = temp3;
        temp2 = _mm_aeskeygenassist_si128 (temp3,0x01);
        KEY_256_ASSIST_1(&temp1, &temp2);
        Key_Schedule[2]=temp1;
        KEY_256_ASSIST_2(&temp1, &temp3);
        Key_Schedule[3]=temp3;
        temp2 = _mm_aeskeygenassist_si128 (temp3,0x02);
        KEY_256_ASSIST_1(&temp1, &temp2);
        Key_Schedule[4]=temp1;
        KEY_256_ASSIST_2(&temp1, &temp3);
        Key_Schedule[5]=temp3;
        temp2 = _mm_aeskeygenassist_si128 (temp3,0x04);
        KEY_256_ASSIST_1(&temp1, &temp2);
        Key_Schedule[6]=temp1;
        KEY_256_ASSIST_2(&temp1, &temp3);
        Key_Schedule[7]=temp3;
        temp2 = _mm_aeskeygenassist_si128 (temp3,0x08);
        KEY_256_ASSIST_1(&temp1, &temp2);
        Key_Schedule[8]=temp1;
        KEY_256_ASSIST_2(&temp1, &temp3);
        Key_Schedule[9]=temp3;
        temp2 = _mm_aeskeygenassist_si128 (temp3,0x10);
        KEY_256_ASSIST_1(&temp1, &temp2);
        Key_Schedule[10]=temp1;
        KEY_256_ASSIST_2(&temp1, &temp3);
        Key_Schedule[11]=temp3;
        temp2 = _mm_aeskeygenassist_si128 (temp3,0x20);
        KEY_256_ASSIST_1(&temp1, &temp2);
        Key_Schedule[12]=temp1;
        KEY_256_ASSIST_2(&temp1, &temp3);
        Key_Schedule[13]=temp3;
```

```
    temp2 = _mm_aeskeygenassist_si128 (temp3,0x40);
    KEY_256_ASSIST_1(&temp1, &temp2);
    Key_Schedule[14]=temp1;
}
```

A.4.3 AES128Encrypt

```
#include <wmmintrin.h>
void AES128Encrypt(unsigned char *dest, //pointer to the PLAINTEXT/CIPHERTEXT
 const char *userkey) //pointer to AES128 encryption key
{
    __m128i tmp;
    __m128i *key[10];
    int i;
    tmp = _mm_loadu_si128 ((__m128i*)dest));
    AES_128_Key_Expansion (userkey,key);
    tmp = _mm_xor_si128 (tmp,((__m128i*)key)[0]);
    for(i=1; i <10; i++) {
        tmp = _mm_aesenc_si128 (tmp,((__m128i*)key)[i]);
    }
    tmp = _mm_aesenclast_si128 (tmp,((__m128i*)key)[i]);
    _mm_storeu_si128 ((__m128i*)dest),tmp);
}
```

A.4.4 AES128Decrypt

```
#include <wmmintrin.h>
void AES128Decrypt(unsigned char *dest, //pointer to the CIPHERTEXT/PLAINTEXT
const char *userkey) //pointer to the expanded key schedule
{
    __m128i tmp;
    __m128i key[10], key_decrypt[10];
    int i, round;
    AES_128_Key_Expansion (userkey, key);
    key_decrypt [0] = key [10];
     for (round = 1; round <10; round++) {
        key_decrypt [round] = _mm_aesimc_si128(key[10 - round]);
    }
     key_decrypt [10] = key[0];
    tmp = _mm_loadu_si128 ((__m128i*)dest);
    tmp = _mm_xor_si128 (tmp,((__m128i*)key_decrypt)[0]);
    for(i=1; i <10; i++) {
        tmp = _mm_aesdec_si128 (tmp,((__m128i*)key_decrypt)[i]);
    }
    tmp = _mm_aesdeclast_si128 (tmp,((__m128i*)key)[i]);
    _mm_storeu_si128 ((__m128i*)dest)[i], tmp);
}
```

A.4.5 AES256Encrypt

```
#include <wmmintrin.h>
void AES256Encrypt(unsigned char *dest, //pointer to the PLAINTEXT/CIPHERTEXT
```

```
const char *userkey) //pointer to AES128 encryption key
{
      __m128i tmp;
      __m128i *key[14];
      int i;
      tmp = _mm_loadu_si128 ((__m128i*)dest));
      AES_256_Key_Expansion (userkey,key);
      tmp = _mm_xor_si128 (tmp,((__m128i*)key)[0]);
      for(i=1; i <14; i++) {
            tmp = _mm_aesenc_si128 (tmp,((__m128i*)key)[i]);
      }
      tmp = _mm_aesenclast_si128 (tmp,((__m128i*)key)[i]);
      _mm_storeu_si128 ((__m128i*)dest),tmp);
}
```

A.4.6 AES256Decrypt

```
#include <wmmintrin.h>
void AES256Decrypt(unsigned char *dest, //pointer to the CIPHERTEXT/PLAINTEXT
const char *userkey) //pointer to the expanded key schedule
{
      __m128i tmp;
      __m128i key[14], key_decrypt[14];
      int I, round;
      AES_256_Key_Expansion (userkey, key);
      key_decrypt [0] = key [14];
      for (round = 1; round <14; round++){
            key_decrypt [round] = _mm_aesimc_si128(key[14 - round]);
      }
       key_decrypt [14] = key[0];
      tmp = _mm_loadu_si128((__m128i*)dest);
      tmp = _mm_xor_si128 (tmp,((__m128i*)key_decrypt)[0]);
      for(i=1; i <14; i++) {
            tmp = _mm_aesdec_si128 (tmp,((__m128i*)key_decrypt)[i]);
      }
      tmp = _mm_aesdeclast_si128 (tmp,((__m128i*)key_decrypt)[i]);
      _mm_storeu_si128((__m128i*)dest)[i], tmp);
}
```

A.5 AES-GCM-SIV Algorithm

A.5.1 Background on AES-GCM-SIV and Usage by Key Locker

Key Locker uses the AES-GCM-SIV[1] algorithm in order to wrap the keys supplied to the ENCODEKEY* instructions. It is designed to be high performance and to support both confidentiality as well as integrity and to support additional authentication data. The Key Locker usage does not use a nonce and allows directly specifying a 128-bit integrity (aka message-authentication) key and a 256-bit confidentiality (aka message-encryption) key via the inputs to the LOADIWKEY instruction instead of deriving those keys from a single key and a nonce.

[1] Described in more detail at https://tools.ietf.org/html/rfc8452.

A.5.2 AES GCM SIV Algorithm

This section explains the usage of AES-GCM-SIV by the Key Locker instructions. It does this through the below C code, which gives the same results as the CPU implementation of the AES-GCM-SIV functions in the Key Locker instruction pseudocode. This C-code can be examined to understand what the Key Locker instruction pseudocode does.

```
#include <stdint.h>
#include <string.h>
#include <wmmintrin.h>
#include <emmintrin.h>
# include <smmintrin.h>
#include "measurements.h"

#if !defined (ALIGN16)
#if defined (__GNUC__)
#  define ALIGN16  __attribute__  ( (aligned (16)))
# else

#  define ALIGN16 __declspec (align (16))
# endif
#endif

void Horner_Step(uint8_t* A, uint8_t* B, uint8_t* H, uint8_t* res)
{
   register __m128i _A, _B, _H, TMP1, TMP2, TMP3, TMP4, POLY;
   POLY = _mm_setr_epi32(0x1,0,0,0xc2000000);
   _H = _mm_loadu_si128((__m128i*)H);
   _A = _mm_loadu_si128((__m128i*)A);
   _B = _mm_loadu_si128((__m128i*)B);
   _A = _mm_xor_si128(_A, _B);
   TMP1 = _mm_clmulepi64_si128(_A, _H, 0x00);
   TMP4 = _mm_clmulepi64_si128(_A, _H, 0x11);
   TMP2 = _mm_clmulepi64_si128(_A, _H, 0x10);
   TMP3 = _mm_clmulepi64_si128(_A, _H, 0x01);
   TMP2 = _mm_xor_si128(TMP2, TMP3);
   TMP3 = _mm_slli_si128(TMP2, 8);
   TMP2 = _mm_srli_si128(TMP2, 8);
   TMP1 = _mm_xor_si128(TMP3, TMP1);
   TMP4 = _mm_xor_si128(TMP4, TMP2);
   TMP2 = _mm_clmulepi64_si128(TMP1, POLY, 0x10);
   TMP3 = _mm_shuffle_epi32(TMP1, 78);
   TMP1 = _mm_xor_si128(TMP3, TMP2);
   TMP2 = _mm_clmulepi64_si128(TMP1, POLY, 0x10);
   TMP3 = _mm_shuffle_epi32(TMP1, 78);
   TMP1 = _mm_xor_si128(TMP3, TMP2);
   _A = _mm_xor_si128(TMP4, TMP1);
   _mm_storeu_si128((__m128i*)res, _A);
}

void AES256_KS1_ENC1_On_The_Fly(const unsigned char* PT, unsigned char* CT,
            unsigned char* KS, unsigned char* key){
   register __m128i xmm1, xmm2, xmm3, xmm4, con3, xmm14, b1, mask, con1, _P1, _AAD, POLY, _LENBLK;
   int i=0;
```

```
    __m128i* Key_Schedule =  (__m128i*)KS;
    mask = _mm_setr_epi32(0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d);
    con1 = _mm_setr_epi32(1,1,1,1);
    con3 = _mm_setr_epi8(-1,-1,-1,-1,-1,-1,-1,-1,4,5,6,7,4,5,6,7);
    xmm4 = _mm_setzero_si128();
    xmm14 = _mm_setzero_si128();
    xmm1 = _mm_loadu_si128((__m128i*)key);
    xmm3 = _mm_loadu_si128(&(((__m128i*)key)[1]));
    _mm_storeu_si128(&Key_Schedule[0], xmm1);
    b1 = _mm_loadu_si128((__m128i*)PT);
    b1 = _mm_xor_si128(b1, xmm1);
    b1 = _mm_aesenc_si128(b1, xmm3);
    _mm_storeu_si128(&Key_Schedule[1], xmm3);
    for (i=0; i<6; i++)
    {
       xmm2 = _mm_shuffle_epi8(xmm3, mask);
       xmm2 = _mm_aesenclast_si128(xmm2, con1);
       con1 = _mm_slli_epi32(con1, 1);
       xmm4 = _mm_slli_epi64 (xmm1, 32);
       xmm1 = _mm_xor_si128(xmm1, xmm4);
       xmm4 = _mm_shuffle_epi8(xmm1, con3);
       xmm1 = _mm_xor_si128(xmm1, xmm4);
       xmm1 = _mm_xor_si128(xmm1, xmm2);
       _mm_storeu_si128(&Key_Schedule[(i+1)*2], xmm1);
       b1 = _mm_aesenc_si128(b1, xmm1);

       xmm2 = _mm_shuffle_epi32(xmm1, 0xff);
       xmm2 = _mm_aesenclast_si128(xmm2, xmm14);
       xmm4 = _mm_slli_epi64(xmm3, 32);
       xmm3 = _mm_xor_si128(xmm4, xmm3);
       xmm4 = _mm_shuffle_epi8(xmm3, con3);
       xmm3 = _mm_xor_si128(xmm4, xmm3);
       xmm3 = _mm_xor_si128(xmm2, xmm3);
       _mm_storeu_si128(&Key_Schedule[(i+1)*2+1], xmm3);
       b1 = _mm_aesenc_si128(b1, xmm3);
    }
    xmm2 = _mm_shuffle_epi8(xmm3, mask);
    xmm2 = _mm_aesenclast_si128(xmm2, con1);
    xmm4 = _mm_slli_epi64 (xmm1, 32);
    xmm1 = _mm_xor_si128(xmm1, xmm4);
    xmm4 = _mm_shuffle_epi8(xmm1, con3);
    xmm1 = _mm_xor_si128(xmm1, xmm4);
    xmm1 = _mm_xor_si128(xmm1, xmm2);
    _mm_storeu_si128(&Key_Schedule[14], xmm1);
    b1 = _mm_aesenclast_si128(b1, xmm1);
    _mm_storeu_si128((__m128i*)CT, b1);
}

void AES_256_Encrypt(uint8_t* PT, uint8_t* CT, uint8_t* KS)
{
    __m128i block1;
    uint8_t TMP_KS[16*15]={0};
    block1 = _mm_loadu_si128((__m128i*)PT);
    memcpy(TMP_KS,KS,(sizeof(uint8_t)*16*15));
```

```
    block1 = _mm_xor_si128(block1, *((__m128i*)(TMP_KS+16*0)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*1)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*2)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*3)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*4)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*5)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*6)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*7)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*8)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*9)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*10)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*11)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*12)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*13)));
    block1 = _mm_aesenclast_si128(block1, *((__m128i*)(TMP_KS+16*14)));
    _mm_storeu_si128((__m128i*)CT, block1);
}
void AES_256_Encrypt_x2(uint8_t* PT, uint8_t* PT1, uint8_t* CT, uint8_t* CT1, uint8_t* KS)
{
    __m128i block1, block2;
    uint8_t TMP_KS[16*15]={0};
    block1 = _mm_loadu_si128((__m128i*)PT);
    block2 = _mm_loadu_si128((__m128i*)PT1);
    memcpy(TMP_KS,KS,(sizeof(uint8_t)*16*15));
    block1 = _mm_xor_si128(block1, *((__m128i*)(TMP_KS+16*0)));
    block2 = _mm_xor_si128(block2, *((__m128i*)(TMP_KS+16*0)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*1)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*1)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*2)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*2)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*3)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*3)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*4)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*4)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*5)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*5)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*6)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*6)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*7)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*7)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*8)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*8)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*9)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*9)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*10)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*10)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*11)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*11)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*12)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*12)));
    block1 = _mm_aesenc_si128(block1, *((__m128i*)(TMP_KS+16*13)));
    block2 = _mm_aesenc_si128(block2, *((__m128i*)(TMP_KS+16*13)));
    block1 = _mm_aesenclast_si128(block1, *((__m128i*)(TMP_KS+16*14)));
    block2 = _mm_aesenclast_si128(block2, *((__m128i*)(TMP_KS+16*14)));
```

```
    _mm_storeu_si128((__m128i*)CT, block1);
    _mm_storeu_si128((__m128i*)CT1, block2);
}
```

A.5.2.1 SIV_KEY_WRAP_16B

```
void SIV_KEY_WRAP_16B(uint8_t* K1,uint8_t* K2, uint8_t* AAD, uint8_t* PT, uint8_t* CT, uint8_t* T)
{
    __m128i zero = _mm_setzero_si128();
    uint8_t TMP[64] = {0};
    uint8_t KS[16*15] = {0};
    __m128i LENBLK;
    __m128i AND_MASK, TOP_ONE;
    uint8_t TMP_PT[64]={0};
    TOP_ONE = _mm_setr_epi32(0,0,0,0x80000000);
    LENBLK = _mm_setr_epi32(16*8, 0, 16*8, 0);
    AND_MASK = _mm_setr_epi32(0xffffffff,0xffffffff,0xffffffff,0x7fffffff);
    Horner_Step(AAD, (uint8_t*)&zero, K1, TMP);
    Horner_Step(PT, TMP, K1, TMP);
    Horner_Step(TMP, (uint8_t*)&LENBLK, K1, TMP);

    // TMP = POLYVAL;
    *(__m128i*)TMP = _mm_and_si128(*(__m128i*)TMP, AND_MASK);

    AES256_KS1_ENC1_On_The_Fly(TMP, TMP, KS, K2);

    _mm_storeu_si128((__m128i*)T, *(__m128i*)TMP);
    *(__m128i*)TMP = _mm_or_si128(*(__m128i*)TMP, TOP_ONE);
    AES_256_Encrypt(TMP, TMP, KS);
      memcpy(TMP_PT,PT,(sizeof(uint8_t)*32));
    *(__m128i*)TMP = _mm_xor_si128(*(__m128i*)TMP, *((__m128i*)TMP_PT));
    _mm_storeu_si128((__m128i*)CT, *(__m128i*)TMP);
}
```

A.5.2.2 SIV_KEY_WRAP_32B

```
void SIV_KEY_WRAP_32B(uint8_t* K1,uint8_t* K2, uint8_t* AAD, uint8_t* PT, uint8_t* CT, uint8_t* T)
{
    __m128i zero = _mm_setzero_si128();
    uint8_t TMP[64] = {0};
    uint8_t TMP1[64] = {0};
    uint8_t KS[16*15] = {0};
    __m128i ONE, AND_MASK, LENBLK, TOP_ONE;
    uint8_t TMP_PT[64]={0};
    ONE = _mm_setr_epi32(1,0,0,0);
    TOP_ONE = _mm_setr_epi32(0,0,0,0x80000000);
    LENBLK = _mm_setr_epi32(16*8, 0, 32*8, 0);
    AND_MASK = _mm_setr_epi32(0xffffffff,0xffffffff,0xffffffff,0x7fffffff);
    Horner_Step(AAD, (uint8_t*)&zero, K1, TMP);
    Horner_Step(PT, TMP, K1, TMP);
    Horner_Step(PT+16, TMP, K1, TMP);
    Horner_Step(TMP, (uint8_t*)&LENBLK, K1, TMP);
    // TMP = POLYVAL;
    *(__m128i*)TMP = _mm_and_si128(*(__m128i*)TMP, AND_MASK);
    AES256_KS1_ENC1_On_The_Fly(TMP, TMP, KS, K2);
```

```
    _mm_storeu_si128((__m128i*)T, *(__m128i*)TMP);
    *(__m128i*)TMP = _mm_or_si128(*(__m128i*)TMP, TOP_ONE);
    *(__m128i*)TMP1 = _mm_add_epi32(*(__m128i*)TMP, ONE);
    AES_256_Encrypt_x2(TMP, TMP1, TMP, TMP1, KS);
    memcpy(TMP_PT,PT,(sizeof(uint8_t)*32));
    *(__m128i*)TMP = _mm_xor_si128(*(__m128i*)TMP, *((__m128i*)TMP_PT));
    *(__m128i*)TMP1 = _mm_xor_si128(*(__m128i*)TMP1, *((__m128i*)(TMP_PT+16)));
    _mm_storeu_si128((__m128i*)CT, *(__m128i*)TMP);
    _mm_storeu_si128((__m128i*)(CT+16), *(__m128i*)TMP1);
}
```

A.5.2.3 SIV_KEY_UNWRAP_32B

```
int SIV_KEY_UNWRAP_32B(uint8_t* K1,uint8_t* K2, uint8_t* AAD, uint8_t* CT, uint8_t* T, uint8_t* PT)
{
    __m128i zero = _mm_setzero_si128();
    uint8_t TMP[64] = {0};
       uint8_t TMP_T[64] = {0};
    uint8_t TMP1[64] = {0};
    uint8_t TMP2[64] = {0};
    uint8_t KS[16*15] = {0};
    uint8_t TMP_CT[64]={0};
    __m128i ONE, AND_MASK, LENBLK, TOP_ONE;
    ONE = _mm_setr_epi32(1,0,0,0);
    TOP_ONE = _mm_setr_epi32(0,0,0,0x80000000);
    LENBLK = _mm_setr_epi32(16*8, 0, 32*8, 0);
    AND_MASK = _mm_setr_epi32(0xffffffff,0xffffffff,0xffffffff,0x7fffffff);
       memcpy(TMP_T,T,(sizeof(uint8_t)*16));
    *(__m128i*)TMP = _mm_or_si128(*(__m128i*)TMP_T, TOP_ONE);
    *(__m128i*)TMP1 = _mm_add_epi32(*(__m128i*)TMP, ONE);
    AES256_KS1_ENC1_On_The_Fly(TMP, TMP, KS, K2);
    AES_256_Encrypt(TMP1, TMP1, KS);
    memcpy(TMP_CT,CT,(sizeof(uint8_t)*32));
    *(__m128i*)TMP = _mm_xor_si128(*(__m128i*)TMP, *((__m128i*)TMP_CT));
    *(__m128i*)TMP1 = _mm_xor_si128(*(__m128i*)TMP1, *((__m128i*)(TMP_CT+16)));
    Horner_Step(AAD, (uint8_t*)&zero, K1, TMP2);
    Horner_Step(TMP, TMP2, K1, TMP2);
    Horner_Step(TMP1, TMP2, K1, TMP2);
    Horner_Step(TMP2, (uint8_t*)&LENBLK, K1, TMP2);
    // TMP = POLYVAL;
    *(__m128i*)TMP2 = _mm_and_si128(*(__m128i*)TMP2, AND_MASK);
    AES_256_Encrypt(TMP2, TMP2, KS);
    if (memcmp(TMP2, TMP_T, 16)==0)
    {
       _mm_storeu_si128((__m128i*)(PT), *(__m128i*)TMP);
       _mm_storeu_si128((__m128i*)(PT+16), *(__m128i*)TMP1);
       return 1;
    }
    return 0;
}
```

A.5.2.4 SIV_KEY_UNWRAP_16B

```
int SIV_KEY_UNWRAP_16B(uint8_t* K1,uint8_t* K2, uint8_t* AAD, uint8_t* CT, uint8_t* T, uint8_t* PT)
{
```

```
    __m128i zero = _mm_setzero_si128();
    uint8_t TMP[64] = {0};
    uint8_t TMP2[64] = {0};
    uint8_t KS[16*15] = {0};
    __m128i AND_MASK, LENBLK, TOP_ONE;
    uint8_t TMP_CT[64];
    uint8_t TMP_T[64];
    memcpy(TMP_T,T,sizeof(uint8_t)*16);

    TOP_ONE = _mm_setr_epi32(0,0,0,0x80000000);
    LENBLK = _mm_setr_epi32(16*8, 0, 16*8, 0);
    AND_MASK = _mm_setr_epi32(0xffffffff,0xffffffff,0xffffffff,0x7fffffff);
    *(__m128i*)TMP = _mm_or_si128(*(__m128i*)TMP_T, TOP_ONE);
    AES256_KS1_ENC1_On_The_Fly(TMP, TMP, KS, K2);
    memcpy(TMP_CT,CT,sizeof(uint8_t)*32);
    *(__m128i*)TMP = _mm_xor_si128(*(__m128i*)TMP, *((__m128i*)TMP_CT));
    Horner_Step(AAD, (uint8_t*)&zero, K1, TMP2);
    Horner_Step(TMP, TMP2, K1, TMP2);
    Horner_Step(TMP2, (uint8_t*)&LENBLK, K1, TMP2);
    // TMP = POLYVAL;
    *(__m128i*)TMP2 = _mm_and_si128(*(__m128i*)TMP2, AND_MASK);
    AES_256_Encrypt(TMP2, TMP2, KS);
    if (memcmp(TMP2, T, 16)==0)
    {
        _mm_storeu_si128((__m128i*)(PT), *(__m128i*)TMP);
        return 1;
    }
    return 0;
}
```

A.5.2.5 WrapKey128

```
char *  WrapKey128(unsigned char *aeskey, unsigned char *aeskeymetadata, unsigned char *integritykey,
unsigned char *encryptionkey, unsigned char *handle)
{
    unsigned char ciphertext[16] = {0};
    unsigned char authtag[16] = {0};
    int i;
    SIV_KEY_WRAP_16B (integritykey, encryptionkey, aeskeymetadata, aeskey, ciphertext, authtag);
    for (i=0; i<16; i++) {
        handle[i] = aeskeymetadata[i];
        handle[i+16] = authtag[i];
        handle[i+32] = ciphertext[i];
    }
    return handle;
}
```

A.5.2.6 WrapKey256

```
char * WrapKey256(unsigned char *aeskey, unsigned char *aeskeymetadata, unsigned char *integritykey, unsigned
char *encryptionkey, unsigned char *handle)
{
    unsigned char ciphertext[32] = {0};
    unsigned char authtag[16] = {0};
    int i;
```

```
   SIV_KEY_WRAP_32B(integritykey, encryptionkey, aeskeymetadata, aeskey, ciphertext, authtag);
   for (i=0; i<16; i++) {
      handle[i] = aeskeymetadata[i];
      handle[i+16] = authtag[i];
      handle[i+32] = ciphertext[i];
      handle[i+48] = ciphertext[i+16];
   }
   return handle;
}
```

A.5.2.7 UnwrapKeyAndAuthenticate384

int **UnwrapKeyAndAuthenticate384**(const unsigned char *handle, const unsigned char *iwkey, unsigned char *unwrappedkey)

```
{
   unsigned char integritykey[16] = {0};
   unsigned char encryptionkey[32] = {0};
   unsigned char aad[16] = {0};
   unsigned char authtag[16] = {0};
   int i;
   for (i=0; i<32; i++) {
        encryptionkey[i] = iwkey[i+16];
   }
   for (i=0;i<16;i++) {
        integritykey[i] = iwkey[i];
        aad[i] = handle[i];
        authtag[i] = handle[i+16];
  }

  unsigned char ciphertext[16];
  for(i=0;i<16;i++)
    ciphertext[i] = handle[i+32];
  return SIV_KEY_UNWRAP_16B(integritykey, encryptionkey, aad, ciphertext, authtag, unwrappedkey);
}
```

A.5.2.8 UnwrapKeyAndAuthenticate512

int **UnwrapKeyAndAuthenticate512**(const unsigned char *handle, const unsigned char *iwkey, unsigned char *unwrappedkey)

```
{
   unsigned char integritykey[16] = {0};
   unsigned char encryptionkey[32] = {0};
   unsigned char aad[16] = {0};
   unsigned char authtag[16] = {0};
   int i;
    for (i=0; i<32; i++) {
        encryptionkey[i] = iwkey[i+16];
   }
   for (i=0;i<16;i++) {
        integritykey[i] = iwkey[i];
        aad[i] = handle[i];
        authtag[i] = handle[i+16];
   }
   unsigned char ciphertext[32];
   for(i=0; i<32; i++)
```

```
        ciphertext[i] = handle[i+32];
    return SIV_KEY_UNWRAP_32B(integritykey, encryptionkey, aad, ciphertext, authtag, unwrappedkey);
}
```

A.6 Key Locker Security Properties

Key Locker is designed to help to prevent an attacker who breaks into a system after a key is wrapped from being able to use that wrapped key after revocation, or on another system, or in violation of the specified handle restrictions (e.g., at ring 3 for a handle restricted to ring 0).

When used outside of a Trusted Execution Environment (TEE), an attacker who breaks into a system before the key is wrapped may be able to observe the key before it is turned into a handle (e.g., through single stepping and watching the ENCODEKEY* instruction inputs), or observe the IWKey that is used to help protect the Key Locker handles (e.g., through single stepping and watching the LOADIWKEY inputs when KeySource of 0 is used).

Key Locker may be particularly effective in adding protection to a key when the key is turned into a handle early in the boot process, as less software being loaded also means less victims for an attacker to target in order to escalate privileges or find an information disclosure. However, Key Locker should also reduce risk when used later due to the significant reduction of when the key can be disclosed (e.g., before the handle is generated instead of during the entire lifetime of the key).

ENCODEKEY* instructions take the key value from registers instead of memory in order to support software that wants to avoid storing the key to memory until it has the increased protection resulting from being wrapped into a Key Locker handle.

Key Locker handles contain integrity measurements to help detect attempts to change the handle (e.g., changing the restrictions). These measurements also are designed to detect using the wrong key size or using a different IWKey than the handle was created with. Software should use care to check the ZF after each operation in order to detect these cases.

Key Locker does not prevent denial of service by more privileged system software overwriting the IWKey or IWKeyBackup. There are many existing methods for more privileged software to deny execution to less privileged software.

Key Locker was not designed to prevent usage of a wrapped key in ways allowed by the handle restrictions unless the IWKey was revoked. Revoking a Key Locker handle involves overwriting the IWKey value used to create the handle. This may include overwriting the IWKey value on other logical processors (if they contain it), as well as overwriting the IWKey value in IWKeyBackup (if the IWKey was backed up there) and waiting for the new IWKeyBackup to overwrite the persistent storage (which causes IA32_IWKEYBACKUP_STATUS[Backup/Restore valid] to be set).

Key Locker provides some level of protection against simple hardware attacks including probes of external buses, but was not designed to fully protect against all attackers that are able to physically probe values within the CPU die or modify data values on external buses.

If an attacker discovers both a handle and the IWKey used to generate it, they can unwrap the handle in order to obtain the original key, which can then be directly used by the attacker for encryption and decryption. Key Locker security relies upon keeping the IWKey secret from potential attackers.

A.6.1 Key Locker Usage with TEE

Trusted Execution Environments (TEE) modes can provide a more trusted place to execute operations. One example of such a TEE is an Intel SGX enclave. Another example is a VMX guest that runs trusted services that are isolated from other less trusted code that runs outside that VMX guest.

A Key Locker handle can be generated by ENCODEKEY* inside a TEE in order to further guard the key. This provides improved protection against an attacker who is already present on the system before handle generation but has not penetrated the TEE. Such an attacker cannot single step and observe

the key value before it is converted into a handle when that conversion process is protected by the TEE. In order to better guard TEEs that do not control IWKey (e.g., enclaves) against an attacker knowing the IWKey (as handles only increase protection against attackers who do not know the IWKey), the TEE can check the destination register after the ENCODEKEY* and refuse to run unless the IWKey has a KeySource of 1. Note that such a TEE policy may prevent that software from working with virtualization, as IWKey with a KeySource of 1 cannot be fully virtualized at this time.

A more privileged entity that changes IWKey underneath a TEE will result in the previously created handles not working for encryption or decryption and setting RFLAGS.ZF to indicate that. Usages of Key Locker should always check ZF in order to help detect this. Although this could be used as a denial of service on the TEE, there already exist many techniques that more privileged entities (e.g., OSes or VMMs) can use to deny service to less privileged entities that they manage.

Some usage models may be served by having a TEE generate the key and wrap it to a handle but allowing the handle itself to be used outside of the TEE. To support usages like this, applications at ring 3 can create handles restricted to system software which runs at ring 0 usage. These usages require that the TEE and the SW using the handle both use the same IWKey. If there is no such usage, then separate IWKeys could be used for the TEE and the rest of the software (e.g., when the TEE is a separate VMX guest). This would help prevent the handles of the TEE from being used by software outside of the TEE if they are stolen (e.g., through an information disclosure attack), provided that the attacker is not also able to steal the IWKey value used for the TEE.

A TEE that trusts the IWKey when it is first launched, but not when it is later called (due to concerns of a later privilege escalation attack that overwrites the IWKey with a known non-random value) may be able to generate a handle of a known value when it is first called. It can later use that handle to help confirm that the IWKey that was used early in the boot is still the current IWKey. This may be less effective if the attacker is able to single step the TEE as it may be able to restore the original IWKey only when the TEE is checking the IWKey and use the attacker-known IWKey the rest of the time.

A.6.2 Ability of Other Software to Use Handles

Key Locker handles are linked to the IWKey that was used to create them. This means that an attacker who steals the handle but not the original key should not be able to encrypt or decrypt with that original key (e.g., using the handle) unless they can use the stolen handle with its corresponding IWKey.

If the attacker is not able to discover the IWKey value (e.g., because it was loaded into the processor before the attacker was able to perform information disclosure or privilege escalation), then the attacker should not be able to set up that IWKey value on another system or after it is revoked and thus should not be able to use the stolen handles on another system or after IWKey revocation. If IWKey revocation is performed on reboot, as recommended, this can help prevent usage of previously obtained handles after a reboot.

If a VMM is using different IWKeys for different guests (which is expected to be the normal case), then a handle stolen from one guest generally will not be usable in a different attacker's guest unless the attacker also manages to obtain the IWKey from the VMM. A VMM that wishes paravirtualized guests to share handles would need to use the same IWKey for both.

It is also possible to create handles that include restrictions (e.g., a ring 0 only handle that can only encrypt and not decrypt).

Ring 0 only handles may be useful for OS keys that are not intended for usage by applications. If a malicious application manages to steal such a handle, it should not be able to use it within the application itself. Note that ring 0 handles can be created at any privilege level despite only being usable for encryption/decryption at CPL 0.

No-decrypt and no-encrypt handles may be useful in pairs where one side of a protocol only needs to create messages (with encryption) and the other side of the protocol only needs to read messages (with decryption). This would require using an AES mode that uses both AES encryption and decryption (e.g., AES-CBC) rather than an AES mode that only uses encryption (e.g., AES-CTR).

An attacker that obtains the victim's handle and is using the same IWKey as the victim can use that handle in ways allowed by the handle's restrictions. The expected usage model is to have the same IWKey for all applications, which may allow handles stolen by a different application to be used until they are revoked (e.g., by reboot). Currently, restrictions do not limit handle usage to a specific application.

A.6.3 CCA/CPA – Limitations of Encryption/Decryption as a Service

Although usage of Key Locker can help protect the value of the key within the handle, chosen ciphertext attacks (CCA) and chosen plaintext attacks (CPA) may reveal data patterns. For example, AES ECB mode will always encrypt the same plaintext value into the same ciphertext value. Thus, an attacker who knows that a given block of ciphertext was created from a plaintext of 0 knows that other ciphertext blocks with the same value also map to plaintext of 0 and may be able to find patterns in the ciphertext.

Users of AES should be careful that the algorithm mode they select provides the necessary resistance against CCA and CPA attacks. This is true regardless of whether the key value is guarded by Key Locker or by another mechanism (e.g., using a hardware security module (HSM)).

A.6.4 Resistance to Side Channels

To improve performance, Key Locker instructions may use a cache of recently used handles. Although software should not be able to directly observe the values in the cache, usage of handles inside the cache will be faster than using handles that are not in the cache. This means that a timing-based side channel attack may be able to observe if another entity on the same processor is using the exact same handle.

Additionally, some implementations may do a partial lookup of cache entries based on a subset of the integrity tag of the handle, which can lead to different latency for handles if they match only that subset of a handle store in the Key Locker handle cache. Only bits within the bottom 64 bits of the integrity tag will be used this way. The AAD, ciphertext, and upper 64 bits of the tag will not be used for partial lookups. Because some subset of the bottom 64 bits of the tag may be used for a partial lookup, a malicious user of Key Locker instructions may be able to use a timing-based side channel attack to reveal that subset of handle bits of other Key Locker users. That is why only bits within the bottom 64 bits of the integrity tag are used for partial lookups. An attacker who is able to infer 64 bits of the integrity tag will have no information about the rest of the integrity tag or the ciphertext portion of the handle. This provides a 320-bit container for a 256-bit AES key and a 192-bit container for a 128-bit AES key. It will be more difficult for the attacker to guess the unknown bits of the handle than it would be for them to directly guess the entire AES key itself.

Although an attacker should not be able to use the key cache to infer enough bits of the other handle to guess their value, an attacker may theoretically be able to use the key cache to infer the pattern of handle usage of other Key Locker users on the processor.

A.7 System Firmware Enabling

A.7.1 SMM and STM

Current implementations of Key Locker will not allow usage of AES Key Locker instructions in SMM. This is enumerated to SMM software through CPUID.AESKLE being 0 when in SMM, including both default treatment and dual-monitor treatment of SMIs. Future implementations that support SMM may allow CPUID.AESKLE to be 1 when in SMM to indicate that AES*KL and ENCODEKEY* instructions can be used in this mode.

A.7.2 Feature Config

On parts that support MSR_FEATURE_CONFIG (MSR address 13CH), AES Key Locker instructions are not available if the MSR_FEATURE_CONFIG[1:0] has a value of 11b. MSR_FEATURE_CONFIG[1:0] value of 11b will also cause CPUID.AESKLE to be 0, thus causing all AES Key Locker instructions to generate a #UD exception.