Intel ME: Security keys Genealogy, Obfuscation and other Magic

Dmitry Sklyarov, Maxim Goryachy

POSITIVE TECHNOLOGIES

Research Team

POSITIVE TECHNOLOGIES

Date	Presentation/Paper name	Maxim Goryachy	Mark Ermolov	Dmitry Sklyarov
2016-12-28	Tapping into the Core	+	+	
2017-03-23	Intel ME: The Way of the Static Analysis			+
2017-04-14	Intel DCI Secrets	+	+	
2017-08-28	Disabling Intel ME 11 via undocumented mode	+	+	
2017-11-09	Where there's a JTAG, there's a way: obtaining full system access via USB	+	+	
2017-12-06	How to Hack a Turned-Off Computer, or Running Unsigned Code in Intel Management Engine	+	+	
2017-12-06	Intel ME: Flash File System Explained			+
2017-12-06	Recovering Huffman tables in Intel ME 11.x			+
2017-12-27	Inside Intel Management Engine	+	+	

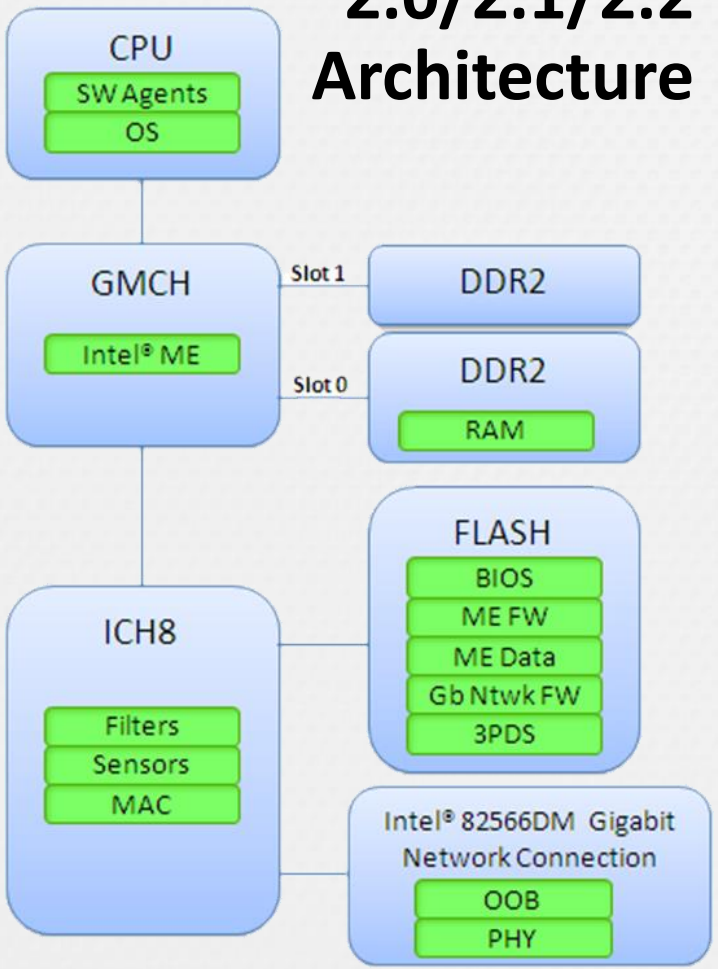
confidence

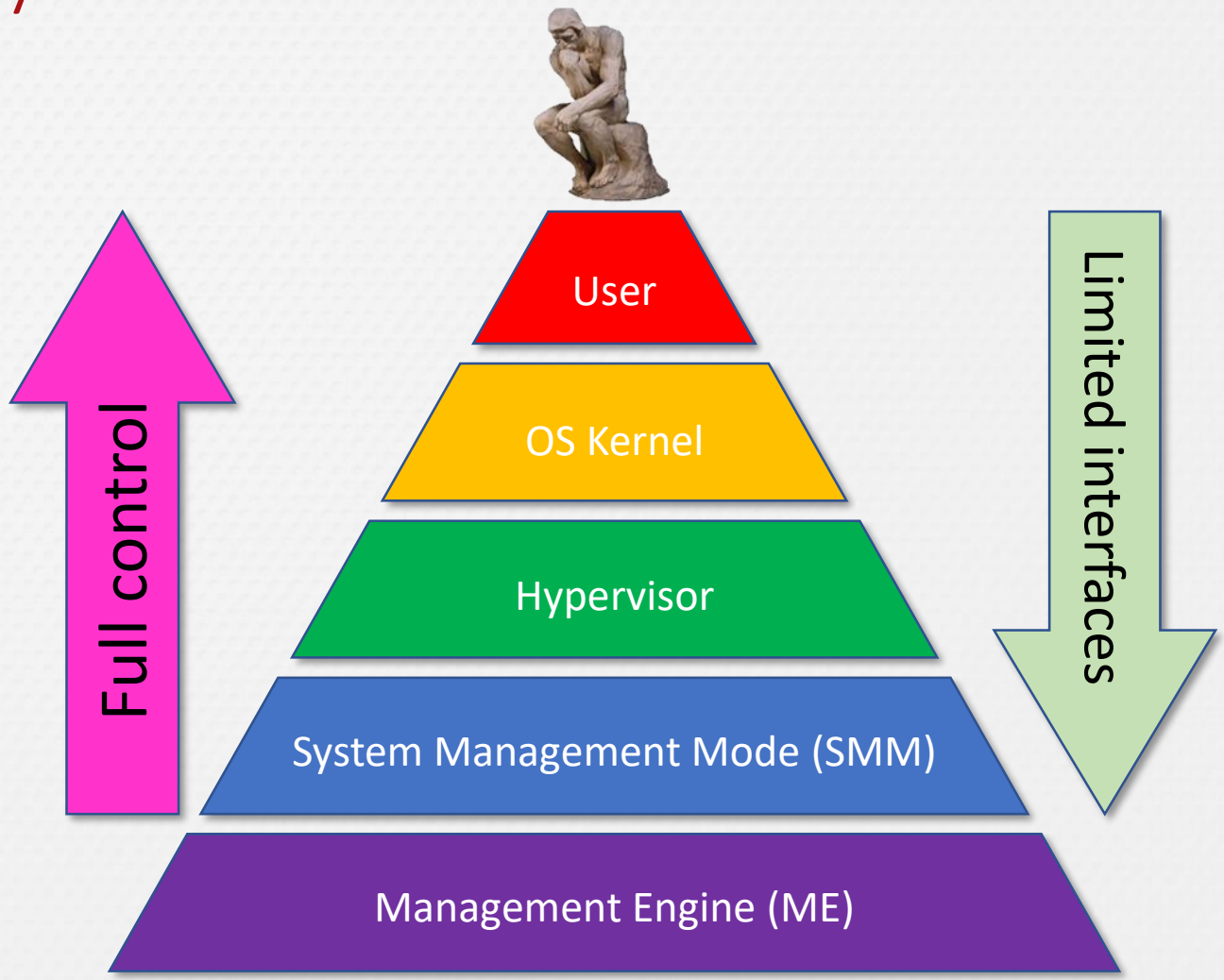
Agenda

- Security Hardware Overview
- Security Fuses
- Keys Derivation and Storage
- Fun and Magic

ME Position in Computer System

Intel AMT Release 2.0/2.1/2.2 Architecture

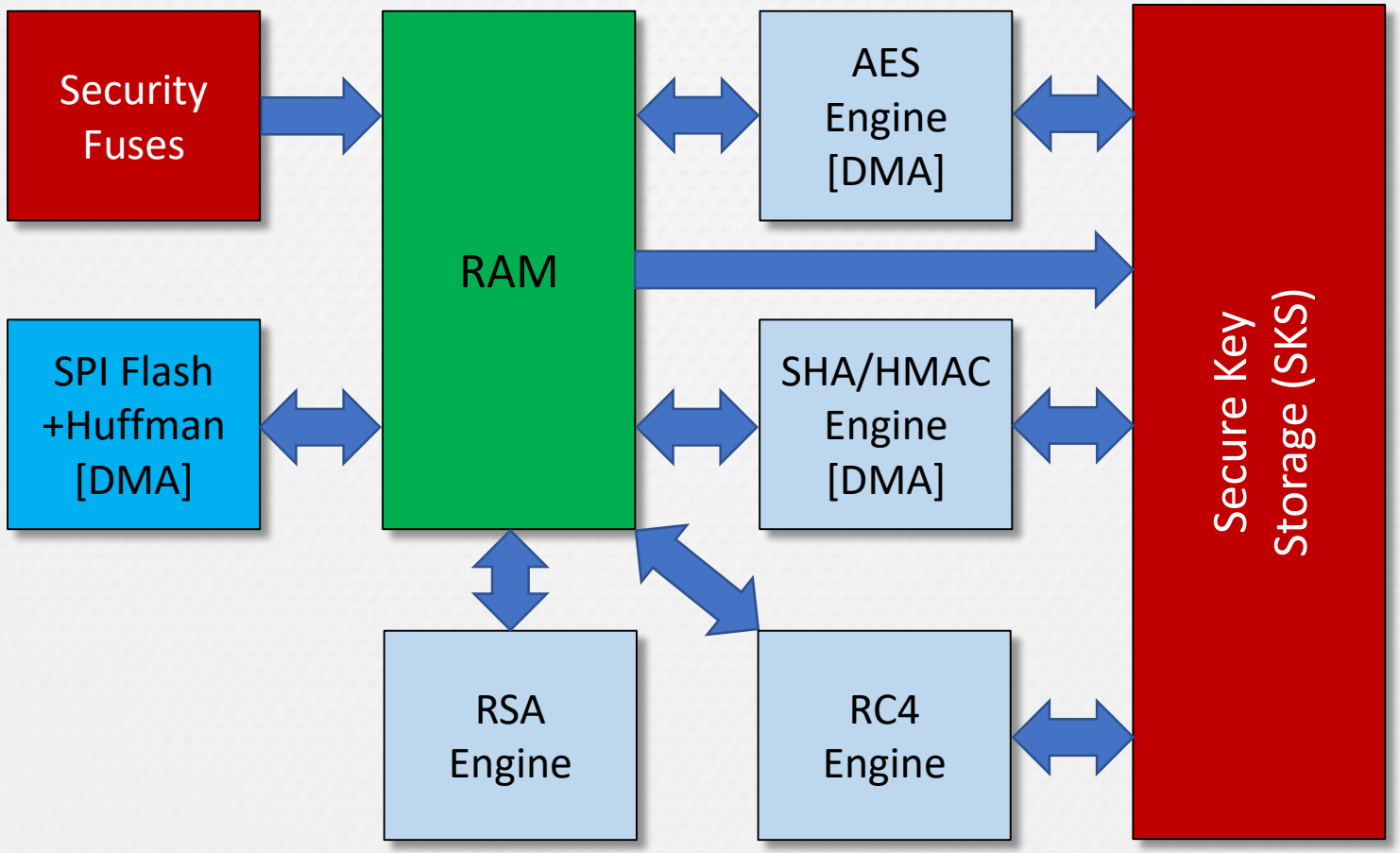




confidence

Security Hardware Overview

Intel ME: Security Hardware



SPI Flash + Controller

- Includes BIOS/UEFI, GbE, ME partition
- Holds code and configuration
- Could operate in conjunction with HMAC Engine
- Has a built-in Huffman Decompressor

	D4h	D7h	SPI Bus Requester Status (CSXE_SBRS)—Offset D4h	0h
	D8h	DBh	Huffman Decompression Compressed Page Offset (CSXE_HDCOMPOFF)—Offset D8h	0h

Security Key Storage (SKS)

- Slots 1..11 for 128-bit keys, slots 12..21 for 256-bit keys
- Key either loaded directly or obtained as a result of AES/HMAC
- Saved keys can’t be extracted into memory (only used for AES/HMAC)
- Usage policy are supported (e.g. result of AES Encrypt could be stored into memory or SKS, result of AES Decrypt – only into SKS)

AES Engine

- Keys of 128 and 256 bits are supported
- Supported block chaining modes: ECB, CBC, CTR
- AES keys could be specified directly or obtained from SKS
- Data could be transferred directly or with DMA

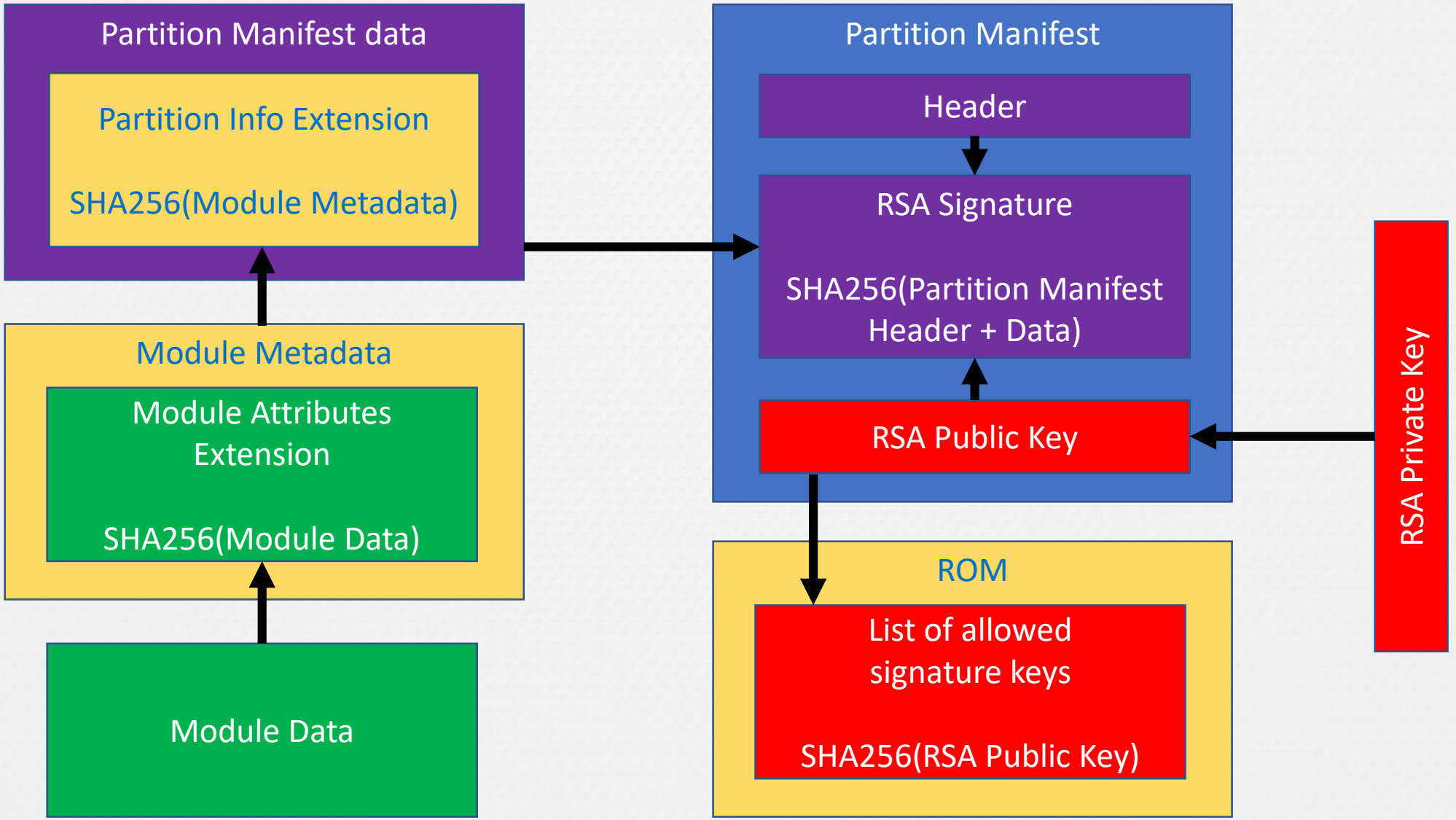
SHA/HMAC Engine

- Supported hash algorithms: SHA-1, SHA-256, SHA-384, SHA-512
- HMAC key length either 128 or 256 bits
- HMAC keys could be specified directly or obtained from SKS
- Data could be transferred directly or with DMA
- Could operate in conjunction with AES Engine (e.g. Decrypt-than-HMAC)

RSA Engine

- Capable to perform modular exponentiation

- Used for verification of Digital Signatures (e.g. ROM verifies ME Partition Manifest integrity before using any data from that partition)


Partition Manifest data
Partition Info Extension
SHA256(Module Metadata)
Module Metadata
Module Attributes Extension
SHA256(Module Data)
Module Data
Partition Manifest
Header
RSA Signature
SHA256(Partition Manifest Header + Data)
RSA Public Key
ROM
List of allowed signature keys
SHA256(RSA Public Key)
RSA Private Key

RC4 Engine

- Is it really necessary in HW developed in 2012+?
- Not used for providing security of ME itself
- Probably used by ME applications (for supporting WiFi, SSL, etc.)

Security Fuses

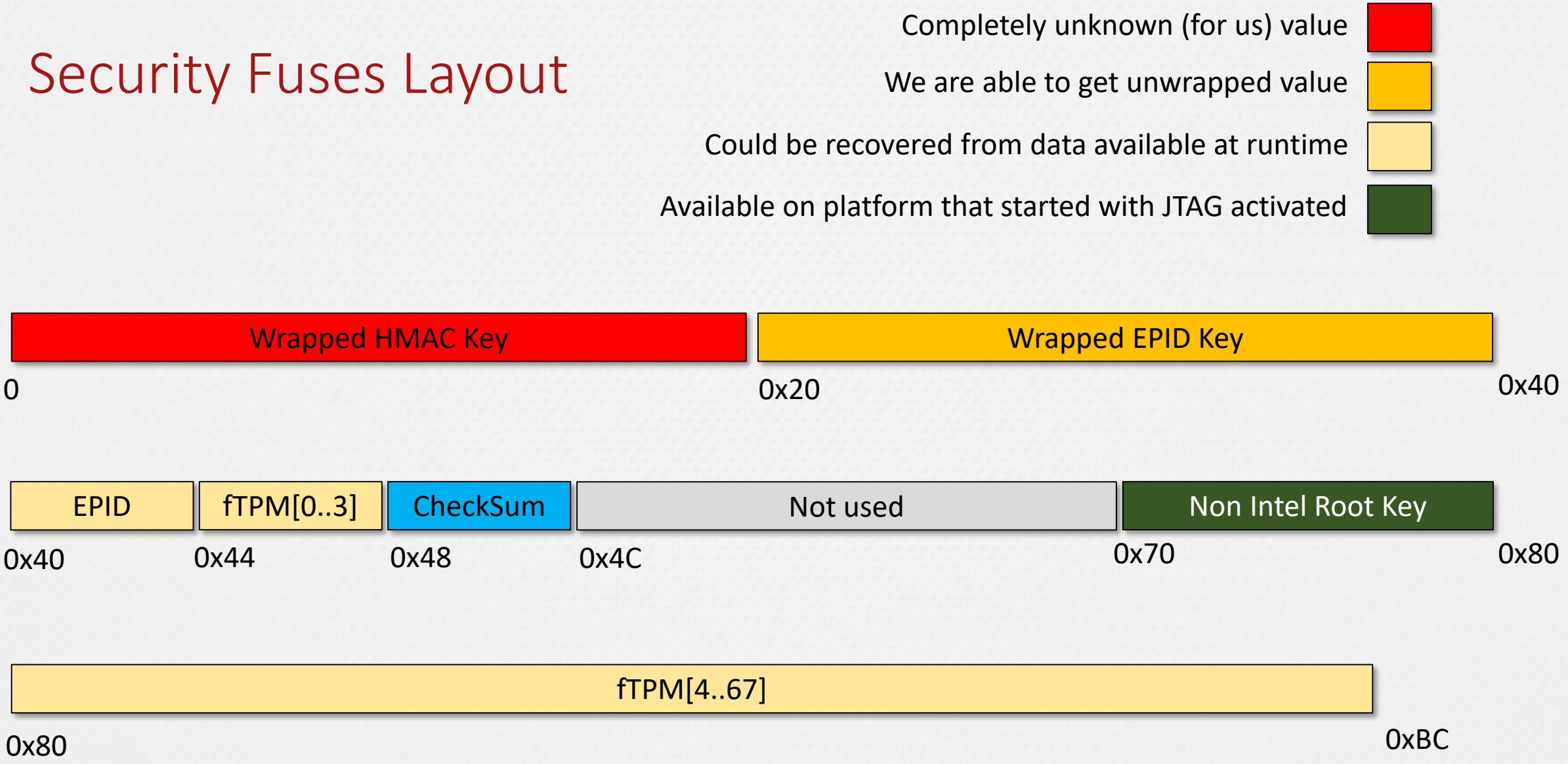
Security Fuses

- Initialized at [some] production stage
- Unique values for each PCH-chip (at least, we believe in that)
- Can not be overwritten
- Readable limited number of times (usually just once) after platform reset
- Huge part of ME security is based on confidentiality of fuses (e.g. TPM)
- Partially blocked if JTAG is enabled (even Intel's engineers can't get fuses)

Reading Data from Fuses

```
void __cdecl GetKey(T_KeysInfo *pKI) {
  unsigned int size, *pdw;
  GEN_status_reg = 1;
  while ( GEN_status_reg & 1 );
  if ( GEN_status_reg & 8 ) stage_complete(Ev_Gen_BadStatus, GEN_status_reg, 0);
  else {
    size = (GEN_status_reg >> 0x10) & 0x1FF;
    if ( size == 0x9C ) {
      pdw = (unsigned int *)&GEN;
     .... // Copy bytes from GEN to Memory
      }
    else if ( size == 0x10 ) {
      if ( isOrangeUnlock()) { // JTAG for vendor is enabled
        stage_complete(Ev_Gen_NotLoaded,(MEMORY[0xF00B1050] >> 4), 0);
      }
      else {
        *(_DWORD *)pKI->NonIntelKey = *(_DWORD *)GEN.NonIntelKey;
        *(_DWORD *)&pKI->NonIntelKey[4] = *(_DWORD *)&GEN.NonIntelKey[4];
        *(_DWORD *)&pKI->NonIntelKey[8] = *(_DWORD *)&GEN.NonIntelKey[8];
        *(_DWORD *)&pKI->NonIntelKey[0xC] = *(_DWORD *)&GEN.NonIntelKey[0xC];
        stage_complete(Ev_Gen_LoadedShort, 0, 0);
      }
```

Security Fuses Layout



Keys Derivation and Storage

FS Security Keys

There are up to 10 keys involved in FS Security

Intel Integrity	Non-Intel Integrity
Current keys (for current SVN)	
Intel Confidentiality	Non-Intel Confidentiality

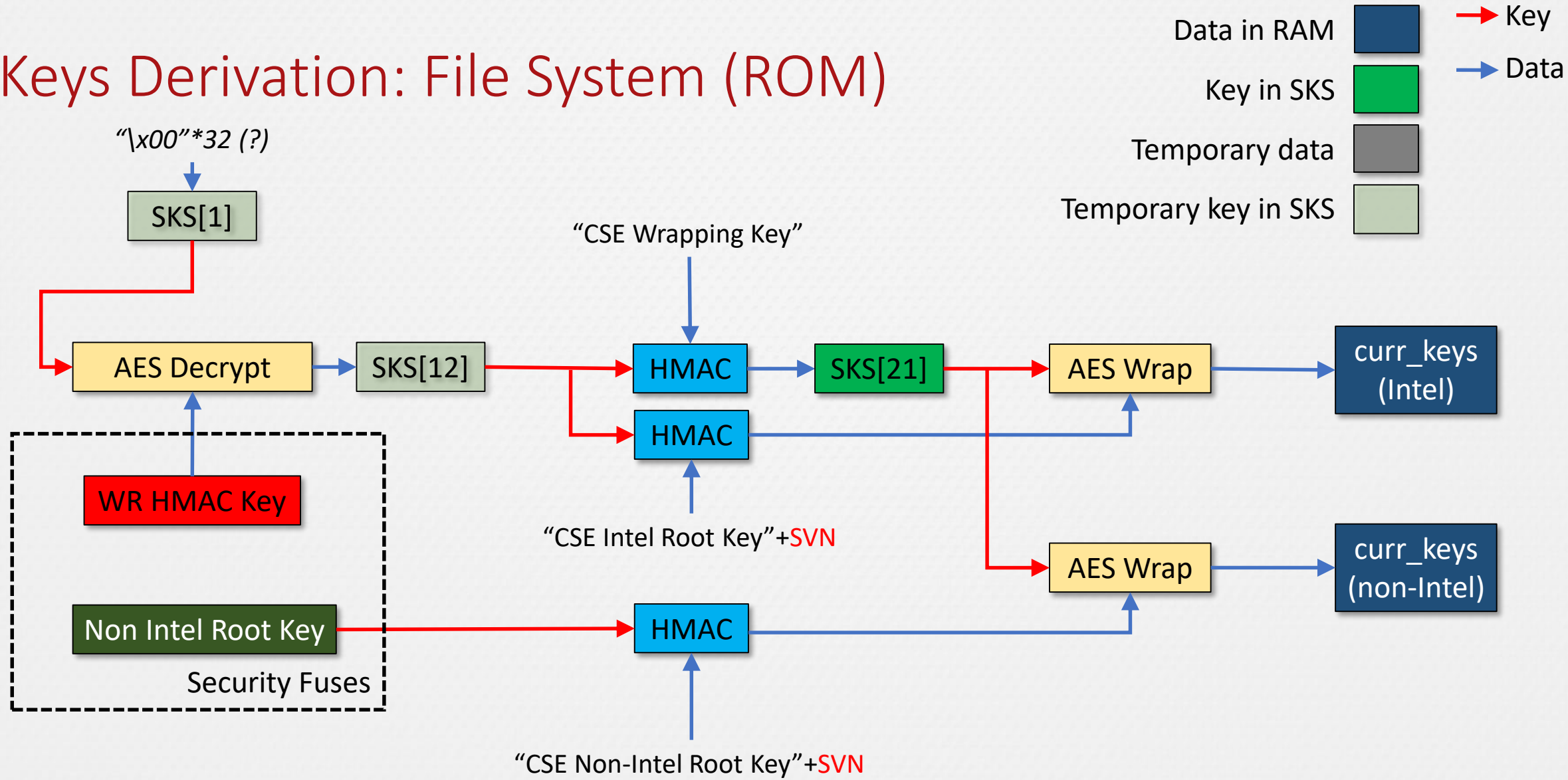
Intel Integrity	Non-Intel Integrity
Previous* keys (optional)	
Intel Confidentiality	Non-Intel Confidentiality

*Previous keys are calculated if current SVN > 1 and PSVN partition contains valid data. These keys are used for migrating files created before the SVN was updated.

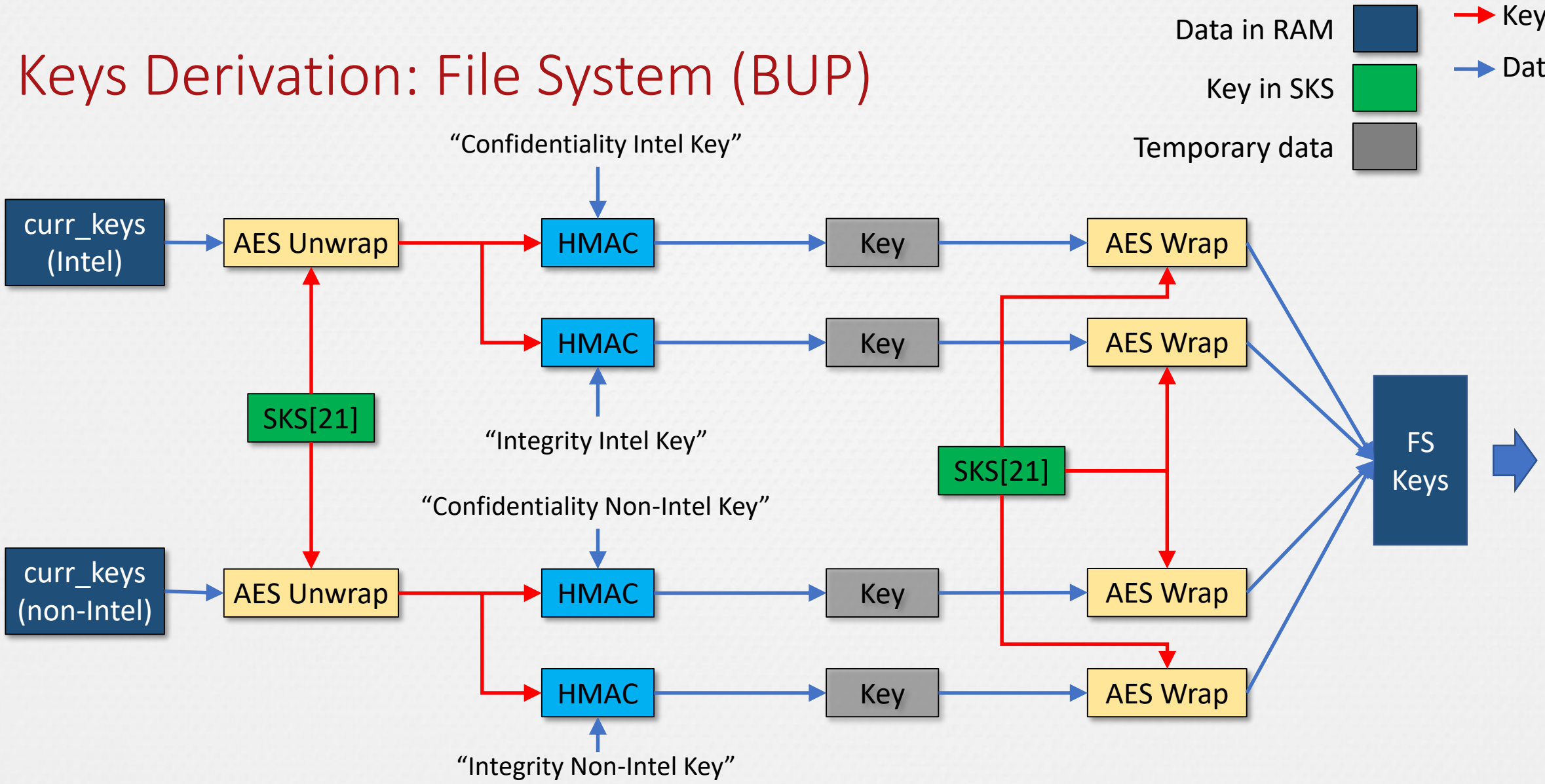
RPMC HMAC #0	RPMC HMAC #1

Replay-Protected Monotonic Counter (RPMC) is optional feature of SPI Flash chip

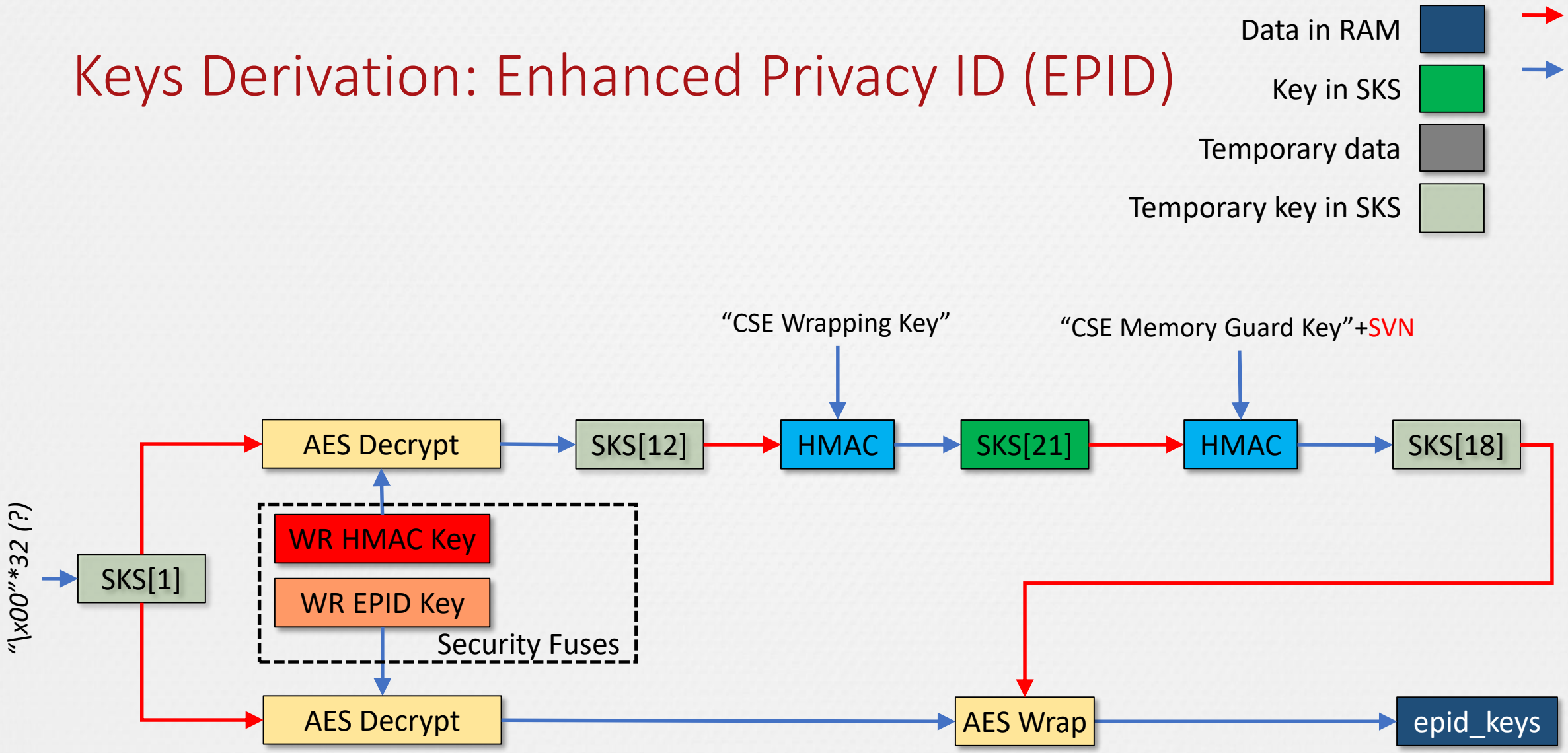
Keys Derivation: File System (ROM)



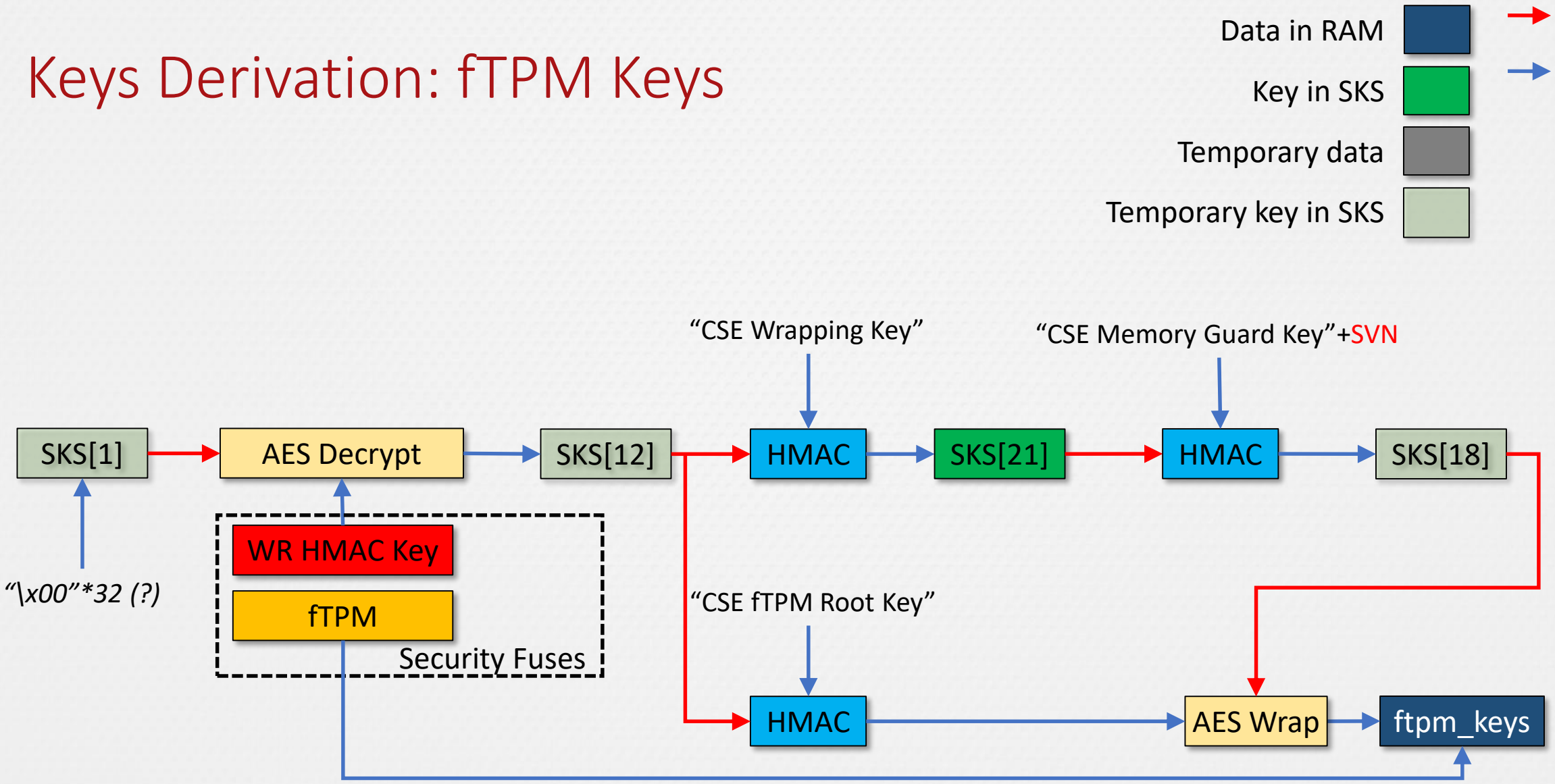
Keys Derivation: File System (BUP)



Keys Derivation: Enhanced Privacy ID (EPID)



Keys Derivation: fTPM Keys



Keys Derivation and Storage Summary

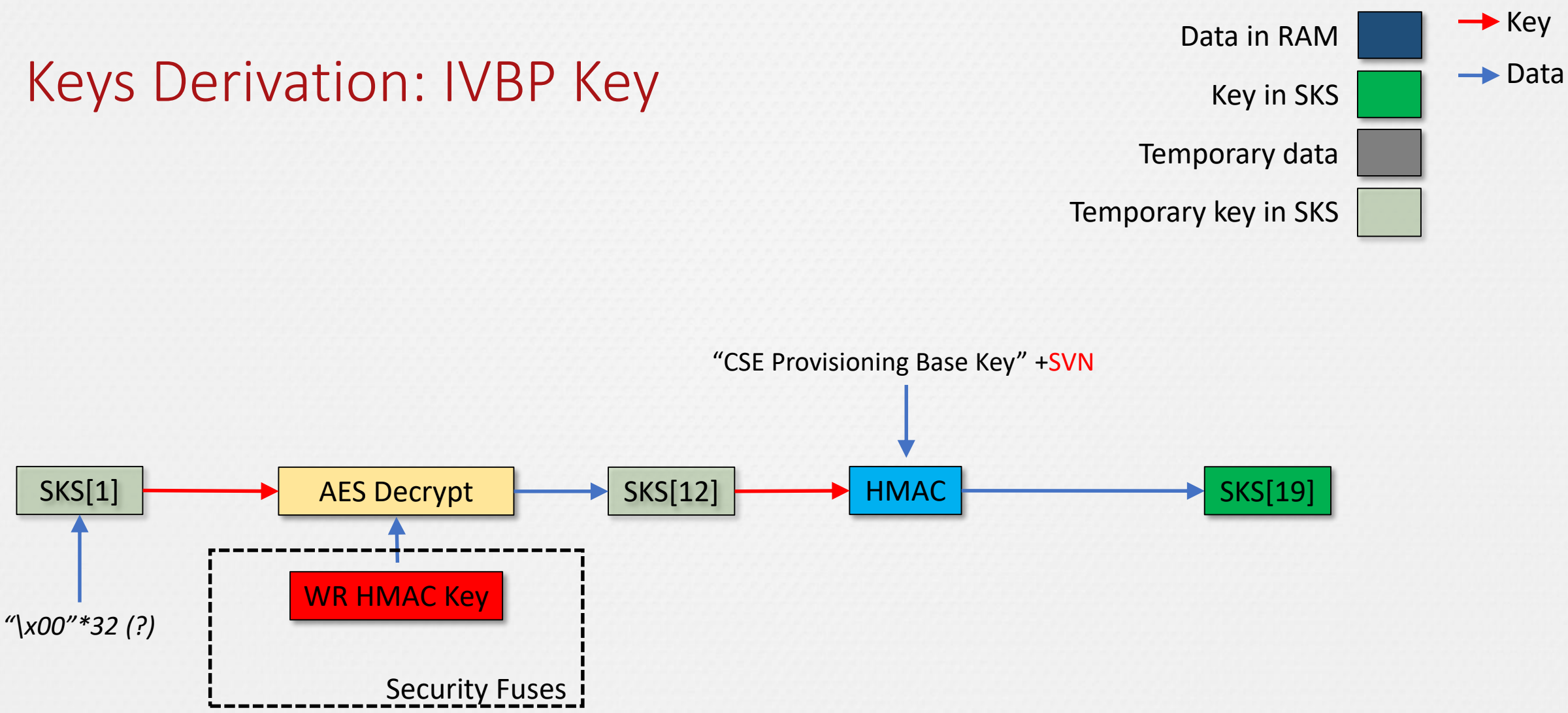
- Security keys never holds in memory in plaintext
- Usually wrapped by SKS[21] or SKS[18]
- Almost all keys depends on Wrapped HMAC Key (which unavailable)
- Having Code Execution in BUP we could recalculate some keys (but not Wrapped/Unwrapped HMAC Key == SKS[12])
- Even Intel's engineers (with JTAG) are unable to get HMAC key

Fun and Magic

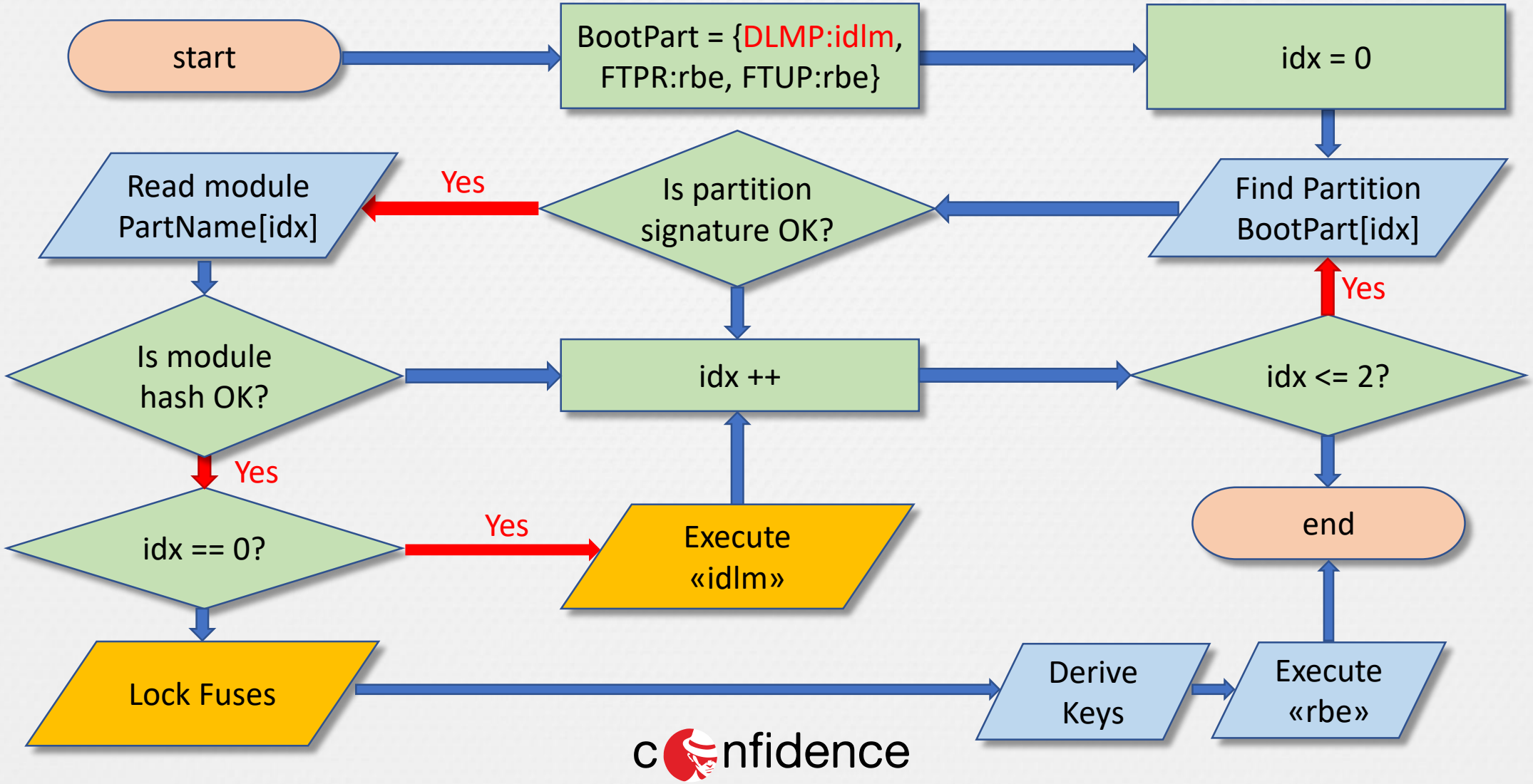
IVBP (IVB Partition)

- System partition for «warm» starting (like hibernation restoration)
- Encrypted, integrity protected with HMAC
- Unique for each platform (PCH-chip) and each boot
- If you know IVBP key, you have arbitrary execution on this platform

Keys Derivation: IVBP Key



ROM: Boot stages (it's about IDLM)



IDLM module (DLM Partition)

- The only place (except ROM) where data read from Fuses is available
- IDLM must be signed with RSA-2048
  - SHA-256 for 8 Public RSA Keys are hardcoded in ROM
  - 4 of 8 keys can be used for IDLM signing (on tested platforms)
- Owner of permitted RSA signing key can extract HMAC key with IDLM!
- IDLM partition seen “in the wild” on ME 11.8 (svn 3) releases

Conclusion

- Intel’s engineers implemented complex and well-thought-out security model for handling keys
- Even Code Execution (in any place except ROM) do not give an attacker the ability to fully compromise security model
- But IDLM feature looks like backdoor ;)
- And who knows – are Fuses Data really unique and unpredictable?

c  nfidence

Q&A

dsklyarov@ptsecurity.com
mgoryachy@ptsecurity.com

POSITIVE TECHNOLOGIES

Bonus Slides

ROM bypass vs ROM

ROM

```
parity = check_parity(&g_KeysInfo, 0x12);// 0x48 bytes
if ( odd == (g_KeysInfo.check & 1) ) {
  pBuf.d.pb = (unsigned __int8 *)&g_KeysInfo;
  pBuf.cb = 72;
  SHA256(&pBuf, SHAF_Final|SHAF_Init);
  SHA256_GetResult(hash, 0);
  key.u.pb = hash;
  key.location = KEY_IN_MEM;
  key.cb = 0x20;
  out.location = KEY_IN_MEM;
  out.u.pdw = adw_mask;
  out.cb = 32;
  data.d.pb = "CSE_PART_ID";
  data.cb = 0x20;
  ROM_HMAC_derive_key(&key, &out, 0, &data);
  b1 = (LOBYTE(g_KeysInfo.check) >> 1) & 1;
  if ( b1 != is_bit(adw_mask, 2u)
    || (b2 = (BYTE(g_KeysInfo.check) >> 2) & 1, b2 != is_bit(adw_mask, 28u))
    || (b3 = (BYTE(g_KeysInfo.check) >> 3) & 1, b3 != is_bit(adw_mask, 47u))
    || (b4 = (BYTE(g_KeysInfo.check) >> 4) & 1, b4 != is_bit(adw_mask, 72u))
  )
```

ROM bypass

```
parity = check_parity(&g_KeysInfo, 8);
if ( parity == (g_KeysInfo.check & 1) ) {
  pBuf.pb = (unsigned __int8 *)&g_KeysInfo;
  pBuf.cb = offsetof(T_KeysInfo, unused);
  SHA256(&pBuf, SHAF_Final|SHAF_Init);
  pBuf.pb = hash;
  pBuf.cb = 32;
  SHA256_GetResult(hash, 0);
  b1 = (LOBYTE(g_KeysInfo.check) >> 1) & 1;
  if ( b1 != is_bit((unsigned int *)hash, 32u)
    || (b2 = (BYTE(g_KeysInfo.check) >> 2) & 1, b2 != is_bit(hash, 69u))
    || (b3 = (BYTE(g_KeysInfo.check) >> 3) & 1, b3 != is_bit(hash, 109u))
    || (b4 = (BYTE(g_KeysInfo.check) >> 4) & 1, b4 != is_bit(hash, 239u))
  )
```

Some Initial Value for AES Engine (in ROM)

```
st260 = AES_r_st260;
st264 = AES_r_st264;
AES_r_secret[0] = 0xE103F8A3;
AES_r_secret[1] = 0xA4B79CD6;
AES_r_secret[2] = 0x56216728;
AES_r_secret[3] = 0x30E039B6;
if ( g_gen_cfg[0] & 4 )
{
  st260 = AES_r_st260 & ~0x12u;
  st264 = AES_r_st264 | 1;
}
AES_r_st260 = st260 & ~1u;
AES_r_st264 = st264;
```

Some hardcoded key for AES Engine (in Crypto)

```
if ( byte_6584F & 1 )
  {
    v6 = getDW_sel(devAES, 0x260u);
    putDW_sel(15, 0x260u, v6 | 4);
  }
  else
  {
    putDW_sel(devAES, key0, 0x09CF4F3C);
    putDW_sel(devAES, key4, 0xABF71588);
    putDW_sel(devAES, key8, 0x28AED2A6);
    putDW_sel(devAES, keyC, 0x2B7E1516);
  }
  putDW_sel(devAES, 0, (*(_BYTE *)a3 == 0) << 6);
```