Security of BIOS/UEFI System Firmware from Attacker and Defender Perspectives

6. Mitigations

Yuriy Bulygin *
Alex Bazhaniuk *
Andrew Furtak *
John Loucaides **

* Advanced Threat Research, McAfee
** Intel

License

Training materials are shared under Creative Commons “Attribution” license CC BY 4.0

Provide the following attribution:

Derived from “Security of BIOS/UEFI System Firmware from Attacker and Defender Perspective” training by Yuriy Bulygin, Alex Bazhaniuk, Andrew Furtak and John Loucaides available at https://github.com/advanced-threat-research/firmware-security-training

Section 6. Mitigations

6.1 UEFI Security Mechanisms

UEFI Security Mechanisms in SecurityPkg

1. UEFI Secure Boot (DxeVerificationLib, Chapter 27.2 of UEFI 2.4 Spec)
2. Authenticated UEFI Variables (Chapter 7 of UEFI 2.4 Spec)
3. Random Number Generator (UEFI driver implementing the EFI_RNG_PROTOCOL from the UEFI2.4 specification)
4. User Identification (DXE drivers that support multi-factor user authentication), Chapter 31 of UEFI 2.4 spec
5. TCG Measured Boot (PEI Modules & DXE drivers implementing Trusted Computing Group measured boot EFI_TCG_PROTOCOL and EFI_TREE_PROTOCOL from the TCG and Microsoft MSDN websites, respectively)

Reference implementation in UDK SecurityPkg:
https://svn.code.sf.net/p/edk2/code/trunk/edk2/SecurityPkg

Other UEFI Security Mechanisms

1. UEFI Secure “Capsule” Update

   http://comments.gmane.org/gmane.comp.bios.tianocore.devel/8402

2. Variable Lock Protocol (Read-Only Variables)

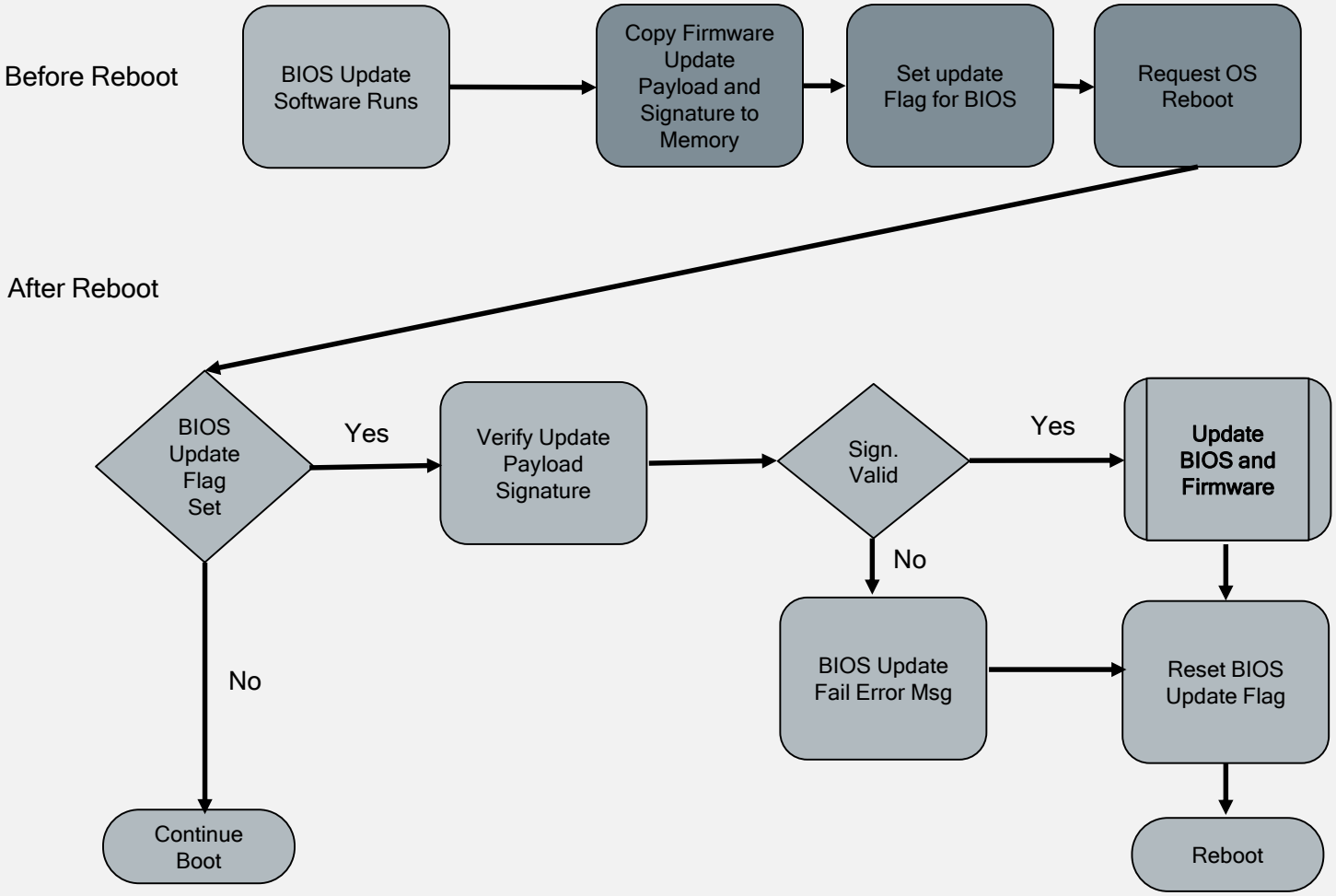
   https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Protocol/VariableLock.h

3. Lock Box (protects memory contents across S3 sleep state):

   https://github.com/tianocore/edk2-MdeModulePkg/blob/master/Include/Protocol/LockBox.h

Signed UEFI “Capsule” Update

Signed Firmware Update

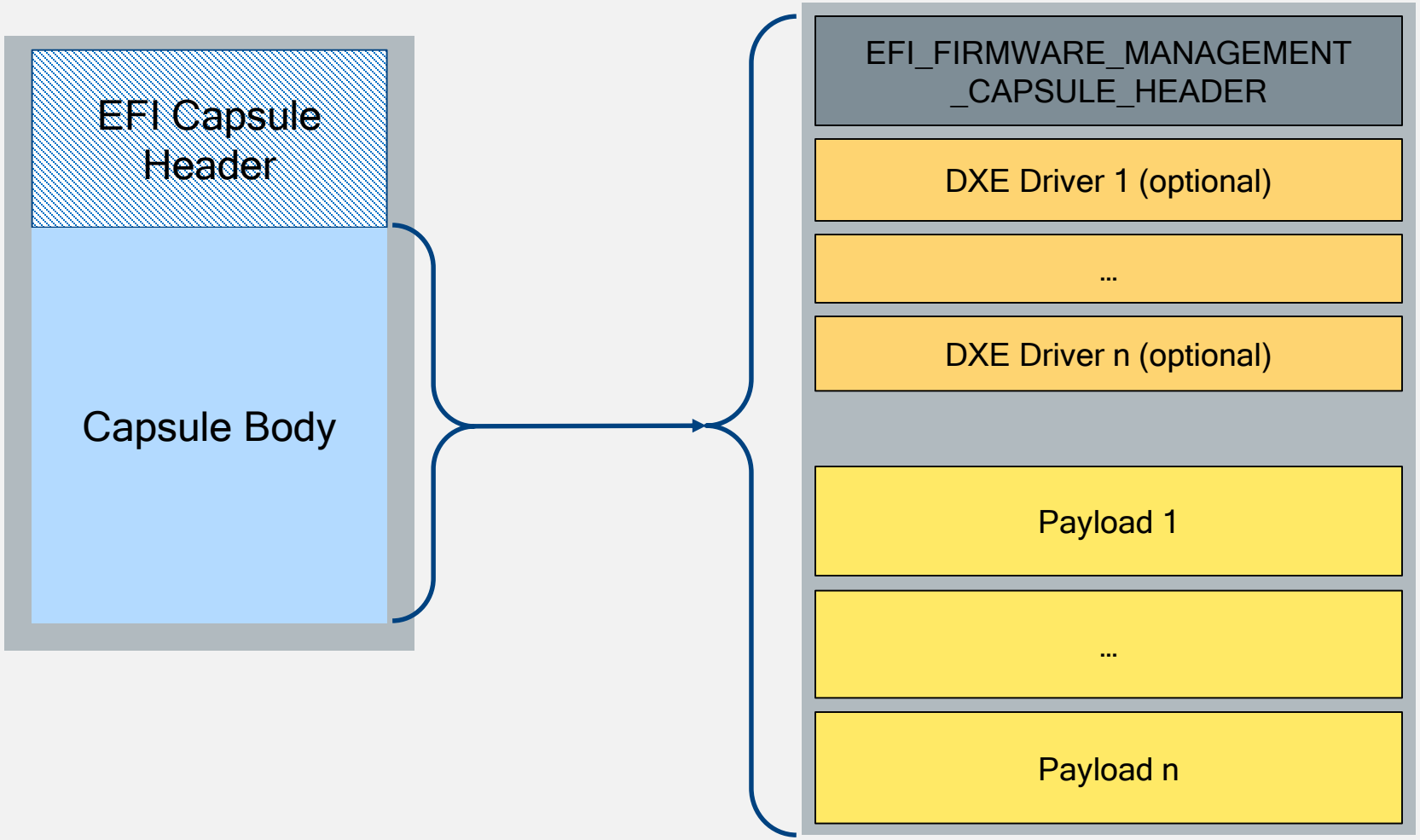


Source: Dell Signed Firmware Update (NIST 800-147)

Firmware Update Methods

1. **UEFI Signed *Capsule* Update** (update on reboot/S3)
2. Run-time SMM based update (by an SMI handler)
3. Update during BIOS Setup or from UEFI Shell
4. Remote BMC/IPMI Based Update
5. Update with physical switch

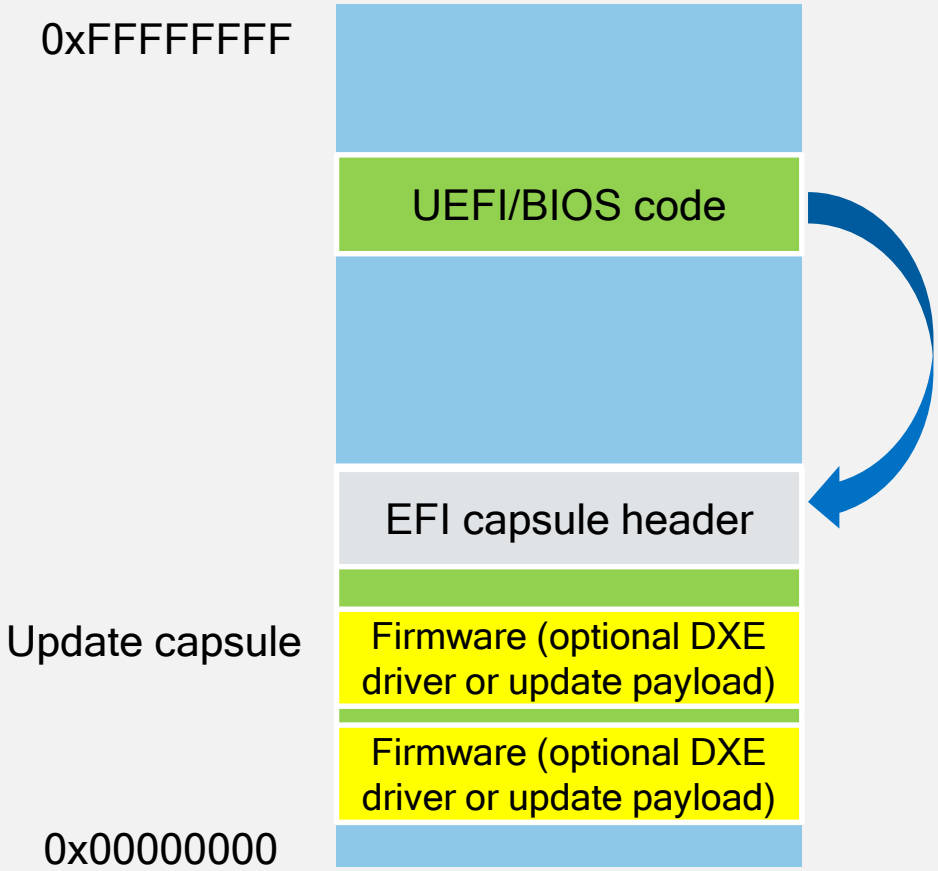
UEFI “Capsules”



Source: UEFI Spec 2.4 Facilitates Secure Update by Jeff Bobzin (Insyde Software)

UEFI Firmware Secure “Capsule” Update

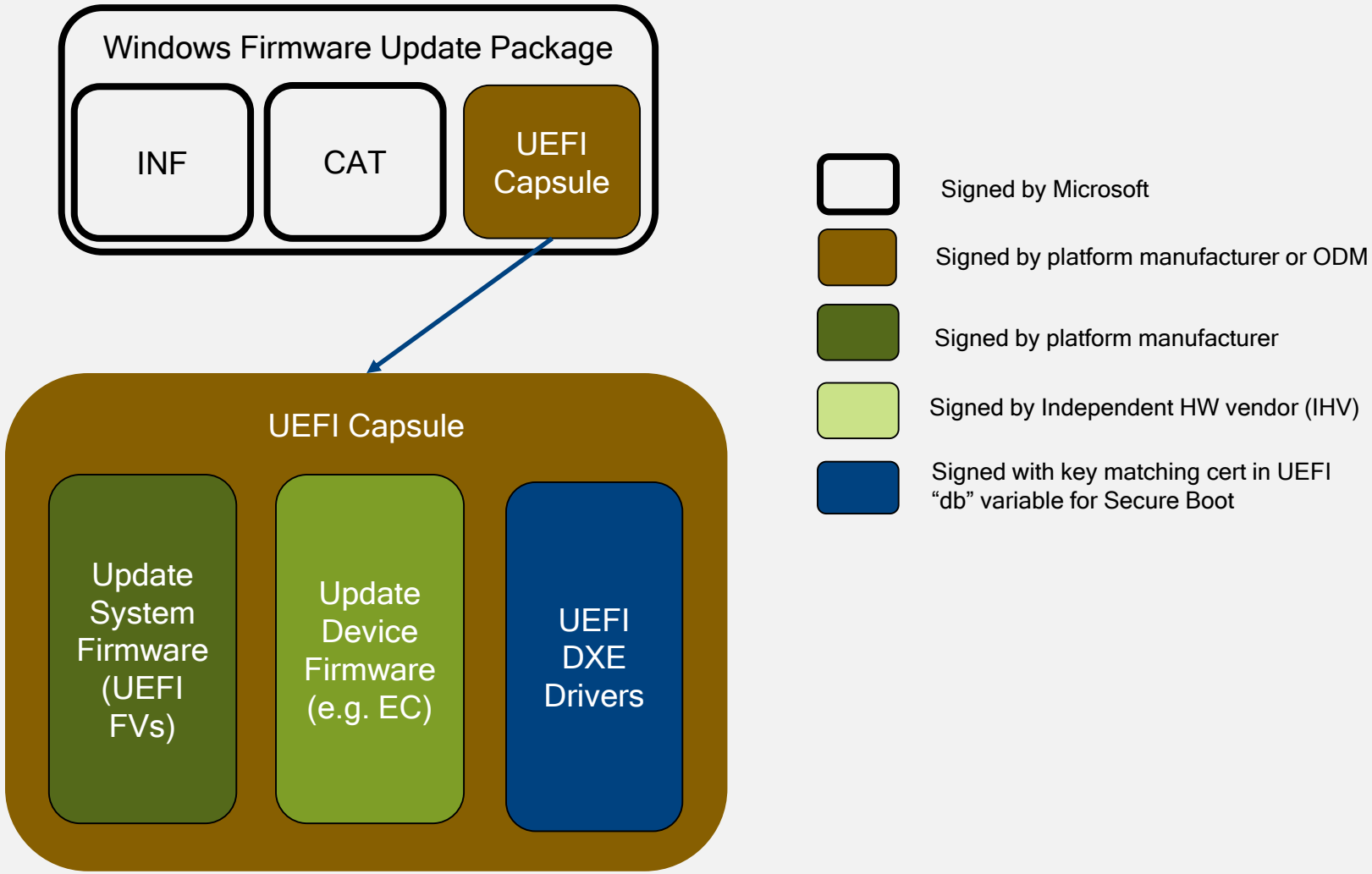
Capsule update is a runtime service used to update UEFI FW



1. Update is initiated by update application/OS run-time
2. Update application stores update “capsule” in DRAM or HDD on ESP (e.g. **`\EFI\CapsuleUpdate`**)
3. Upon reboot or S3 resume, FW finds and parses update capsule
4. After FW verifies digital signature of the capsule, FW writes new BIOS FV(s) to SPI flash memory

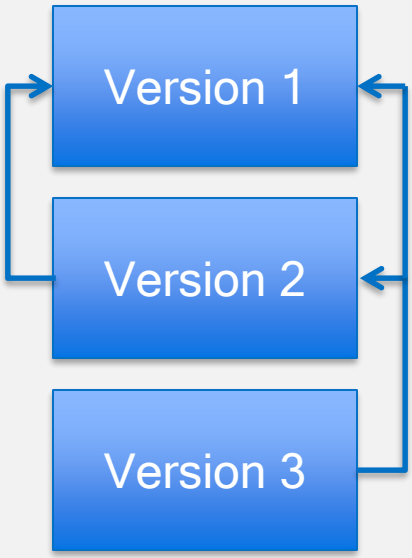
Source: UEFI Spec Version 2.4 Facilitates Secure Update UEFI Summerfest 2013

Windows Firmware Update Package

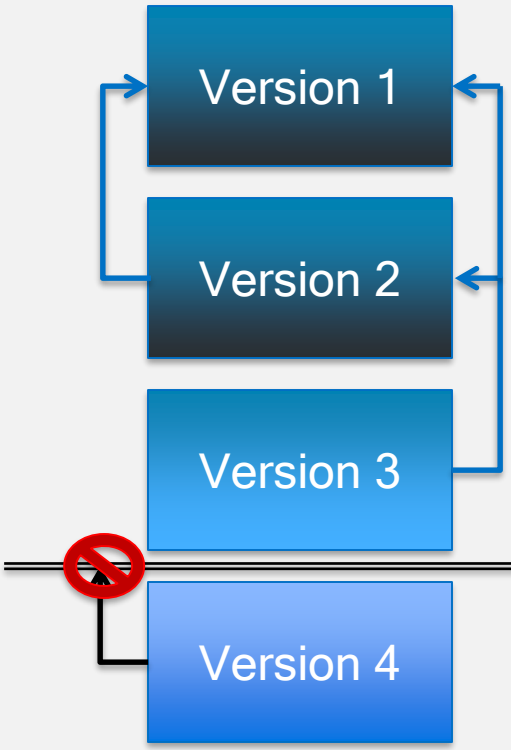


Source : [Windows UEFE Firmware Update Platform]

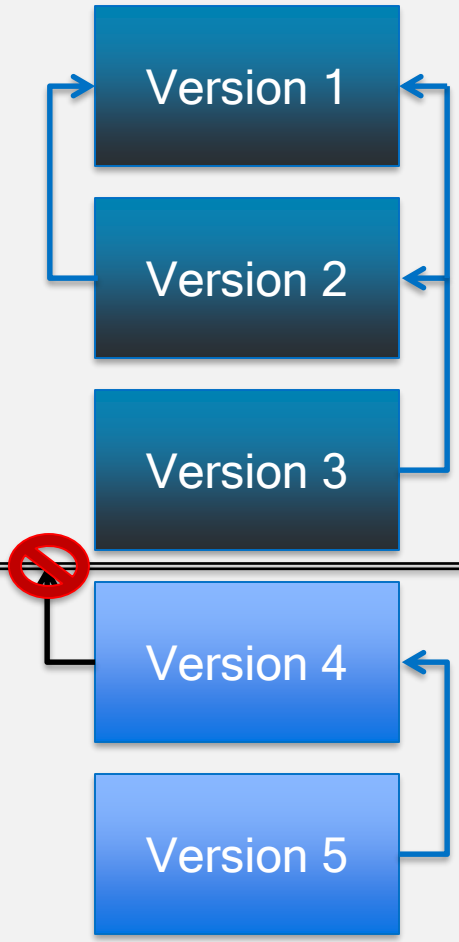
Firmware Update Rollback Protection



Each version fixes some issues with the previous. Since none are known to have security flaws, each new version allows updates to all older versions.



In V4, one of the issues fixed in V3 is realized to be a security fix. V4 will not allow updates to earlier versions, even V3 since it allows update to V2.



Version 5 can now accept only versions 5 and 4.

Protecting UEFI Variables

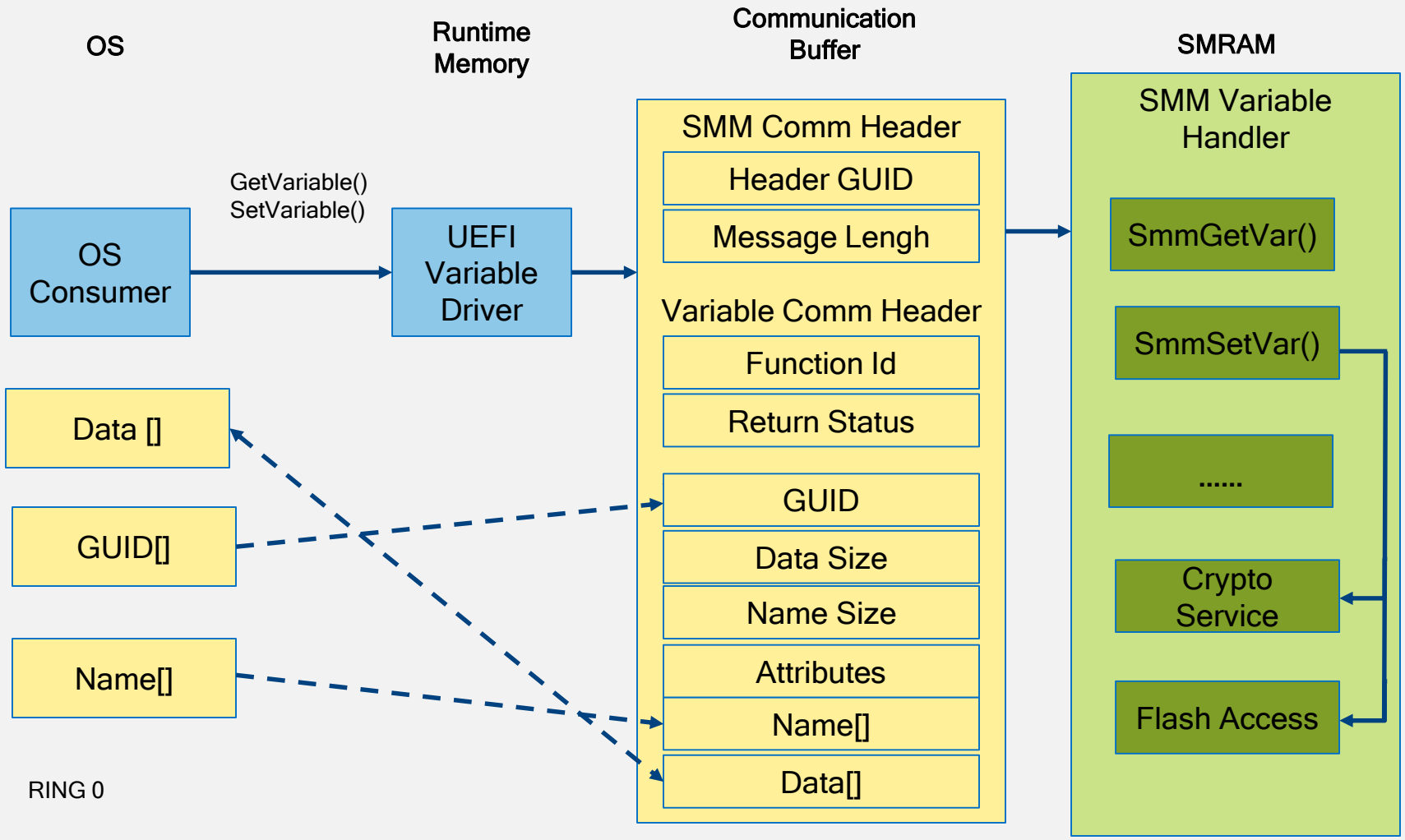
Protecting UEFI Variables

1. Separate critical settings from other and stored in different variables
2. If a variable is only used by the FW then it shouldn't be Runtime
3. Make variables, not updateable by the OS, "read-only" (*Variable Lock)*
4. Use authenticated variables or Pcd for security settings
5. Don't store debug/validation config. (e.g. HW locks) in variables
6. Whenever possible, allow only predefined values written to the variables
7. Validate contents of the variables (e.g. check pointers)
8. Have default config to allow system to boot if variable is corrupted
9. No sensitive data like BIOS passwords in variables in clear
10. Update of some variables may require physically present user
11. Other integrity checks on the contents of critical variables

Why Authenticating Variables?

1. Secure Boot stores Platform Key (PK), Key Exchange Keys (KEK), white-list and black-list (db, dbx) in UEFI Variables
2. These need to be updateable yet protected from unauthorized software changing them
3. UEFI Authenticated Variable allows update/removal of authenticated variables only if new variable is signed and corresponding public cert already present in NVRAM (e.g. in KEK)
4. Counter or Time Based Authenticated Variable to prevent from replay attacks

UEFI Variable Update



Source: A Tour Beyond Implementing UEFI Auth Variables in SMM with EDKII (Jiewen Yao, Vincent Zimmer)

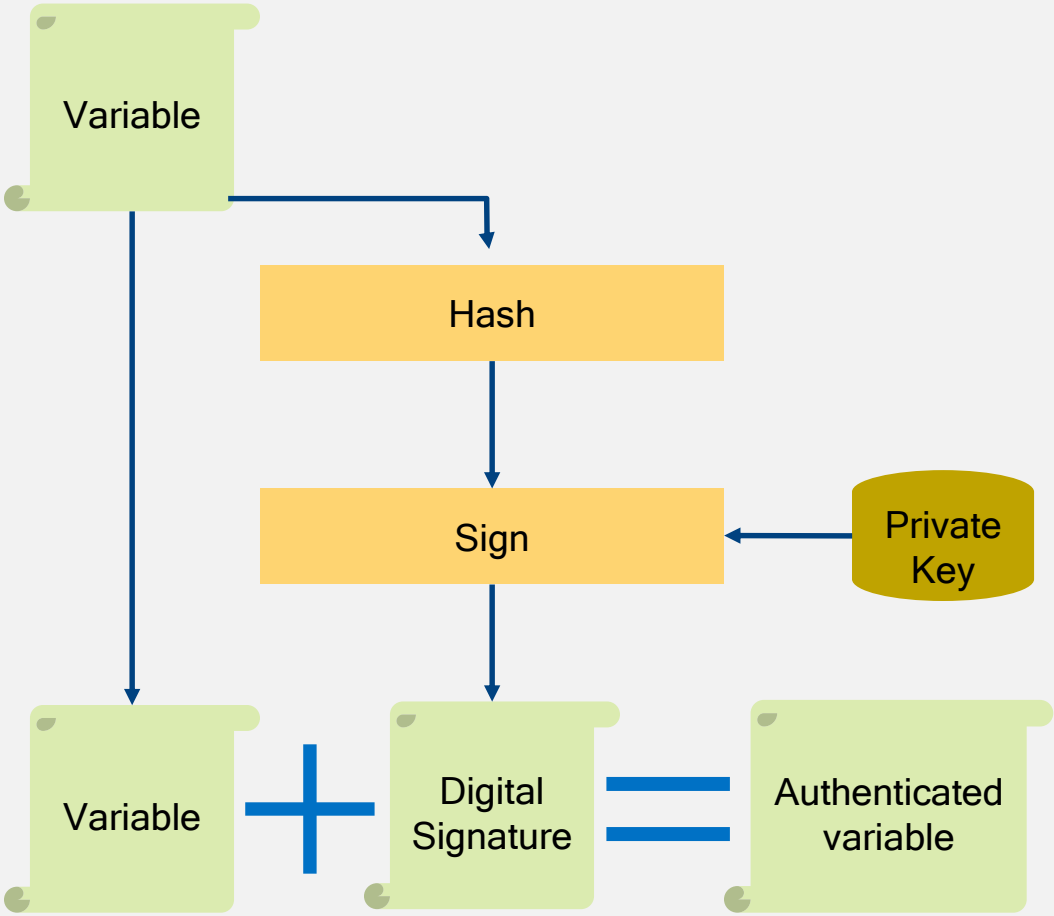
UEFI Variable Authentication

Counter-based authenticated variable

- Monotonic counter against replay
- SHA256 and RSA-2048

Time-based authenticated variable

- EFI_TIME as rollback protection
- SHA256 and X.509 certificate chains
  - Intermediate certificate support (non-root certificate as trusted certificate)



Implementing a Secure Boot Path with UEFI by Jeff Bobzin (Insyde Software), UEFI Plugfest 2011

Variables Protection Attributes

Boot Service (BS)

- Accessible to DXE drivers / Boot Loaders at boot time
- No longer accessible at run-time (after `ExitBootServices`)

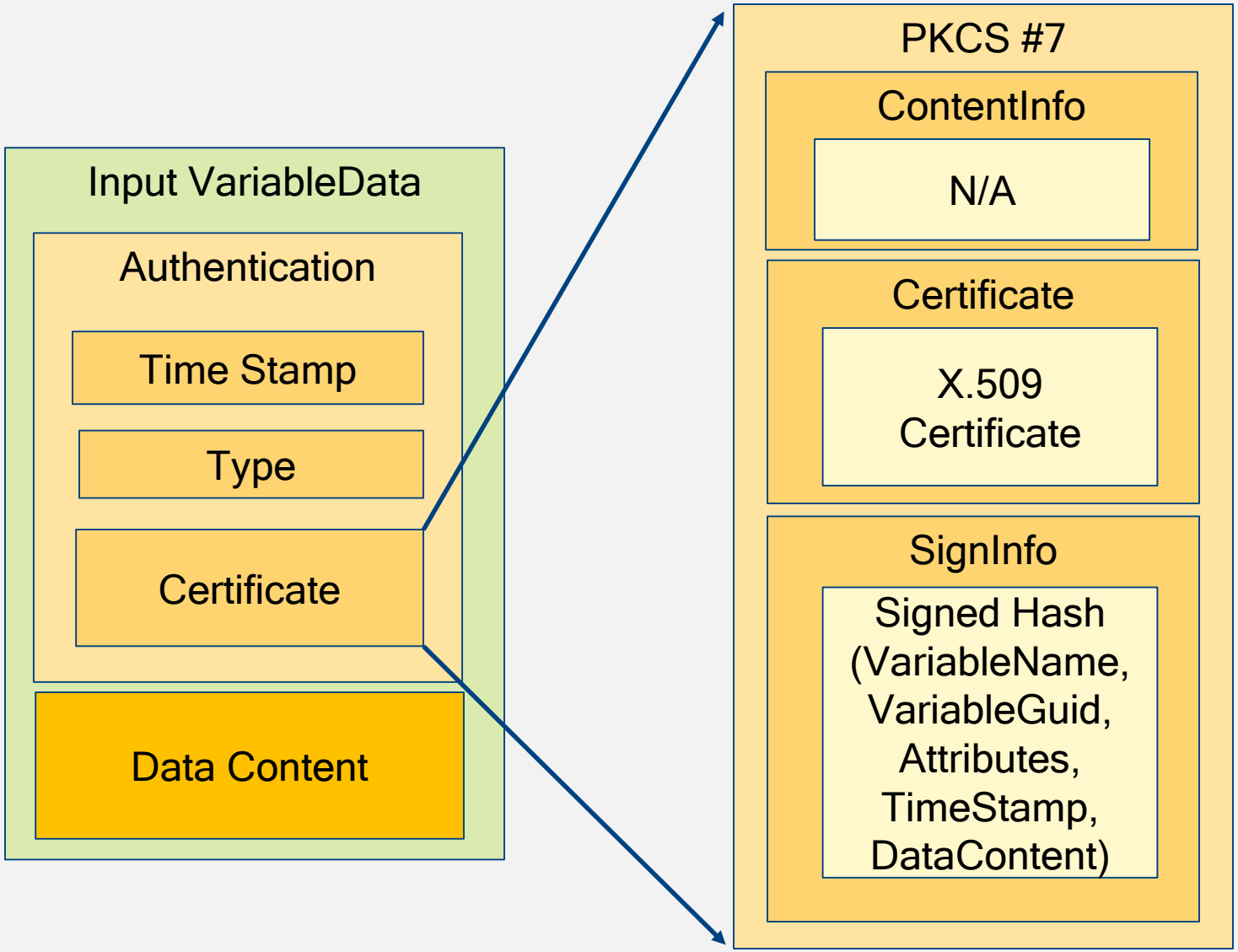
Authenticated Write Access

- Digitally signed with *MonotonicCount* incrementing each successive variable update to protect from replay attacks
- List of signatures supported by the firmware is stored in `SignatureSupport` variable

Time Based Authenticated Write Access

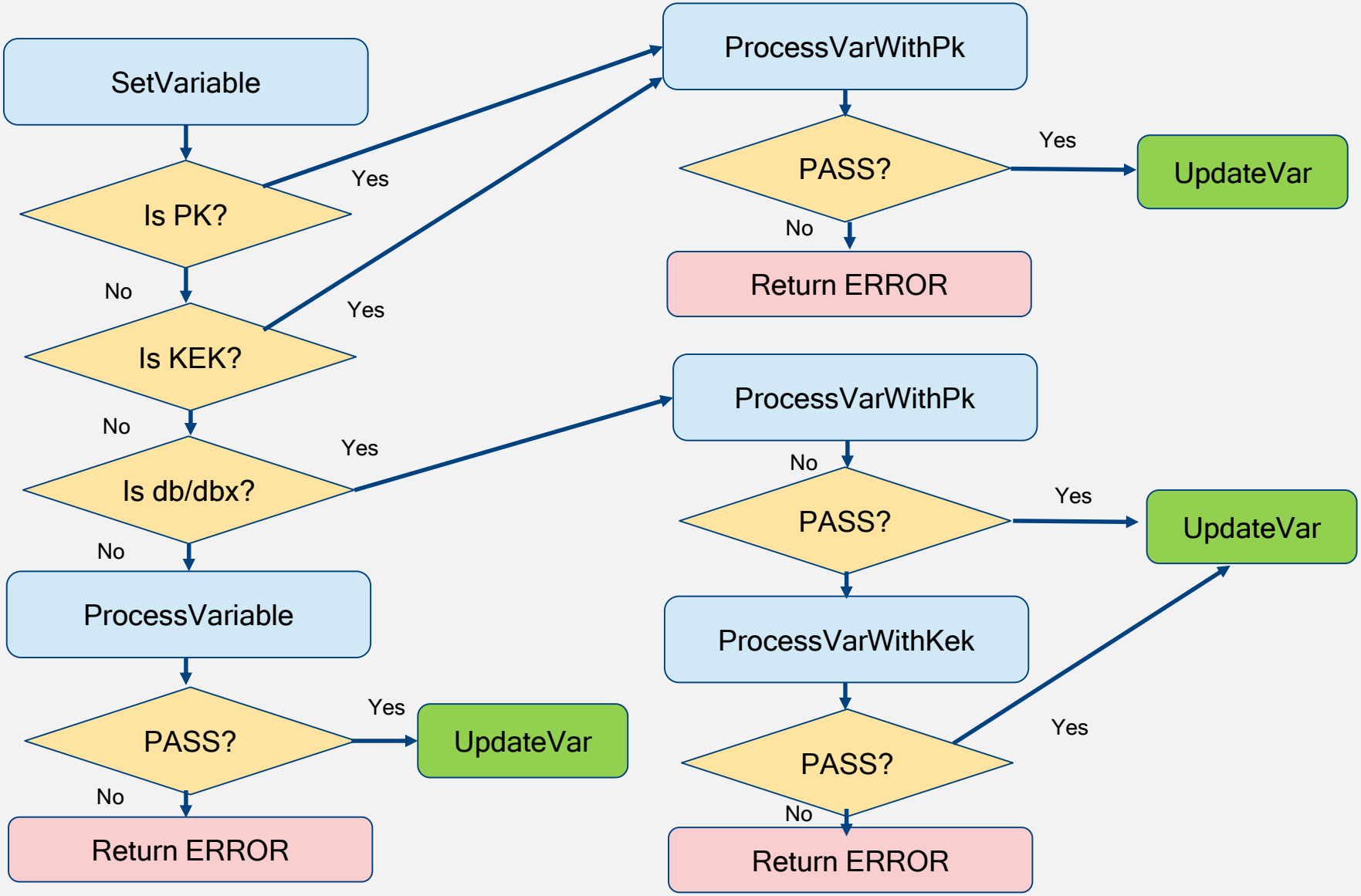
- Signed with `TimeStamp` (time at signing) to protect from replay attacks
- `TimeStamp` should be greater than `TimeStamp` in existing variable
- Used by Secure Boot: PK verifies PK/KEK update, KEK verifies db/dbx update
- `certdb` variable stores certificates to verify non PK/KEK/db(x) variables

Time Based Authenticated Variables



Source: A Tour Beyond Implementing UEFI Auth Variables in SMM with EDKII (Jiewen Yao, Vincent Zimmer)

Authenticated Variable Update Flow



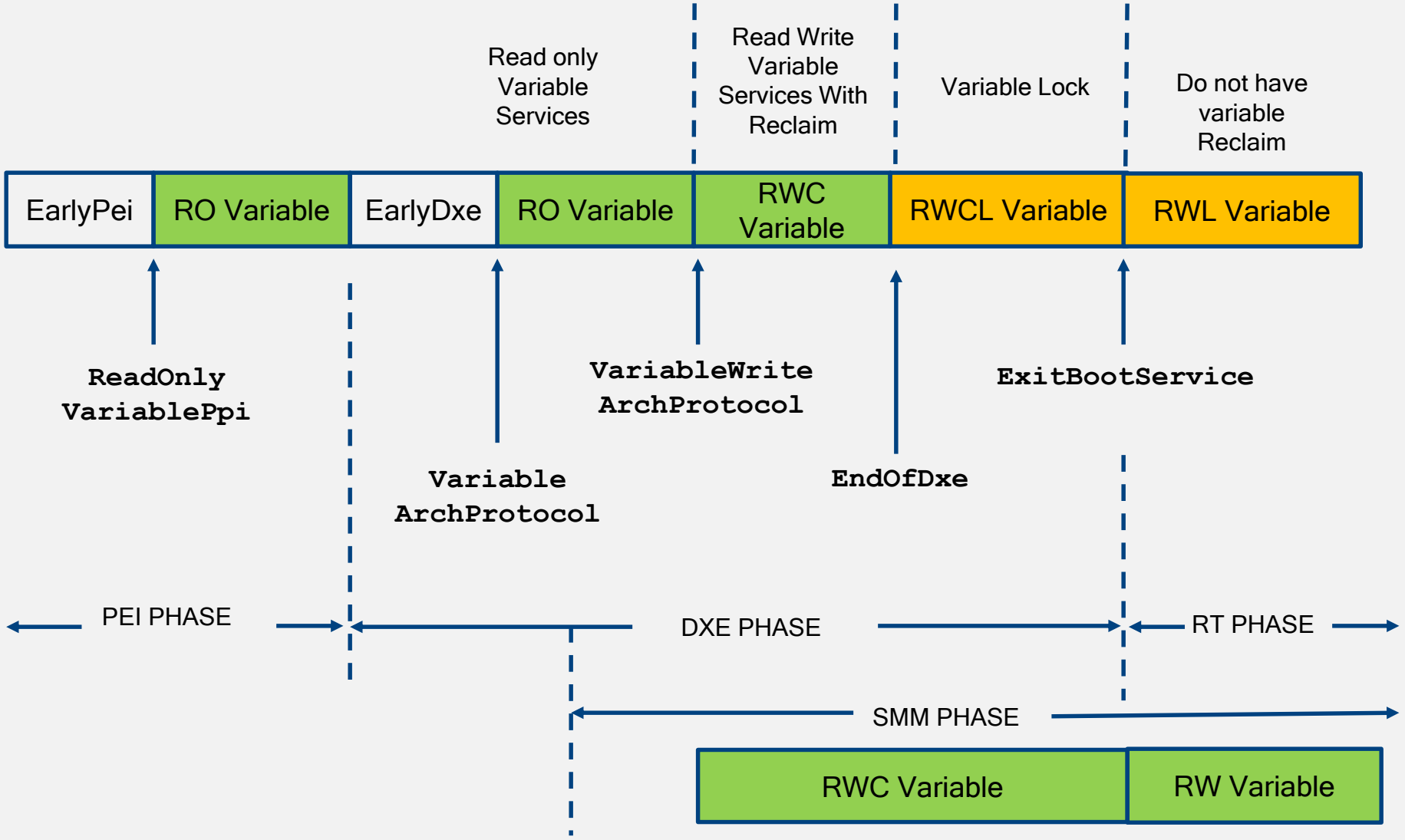
Source: A Tour Beyond Implementing UEFI Auth Variables in SMM with EDKII (Jiewen Yao, Vincent Zimmer)

UEFI Variable Lock Protocol (Read-Only Variables)

UEFI Read-Only Variables

- EDKII implements **`VARIABLE_LOCK_PROTOCOL`** which provides a mechanism to make some variables "**`Read-Only`**" during Run-time OS
- DXE drivers make UEFI variables **`Read-Only`** using **`RequestToLock()`** API before **`EndOfDxe`** event
- After **`EndOfDxe`** event (e.g. during OS runtime), all registered variables cannot be updated or removed (enforced by **`SetVariable`** API)
- Lock is transient, firmware has to request locking variables every boot. Before **`EndOfDxe`** variables are not locked

Variable Lock Flow



Source: A Tour Beyond Implementing UEFI Auth Variables in SMM with EDKII (Jiewen Yao, Vincent Zimmer)

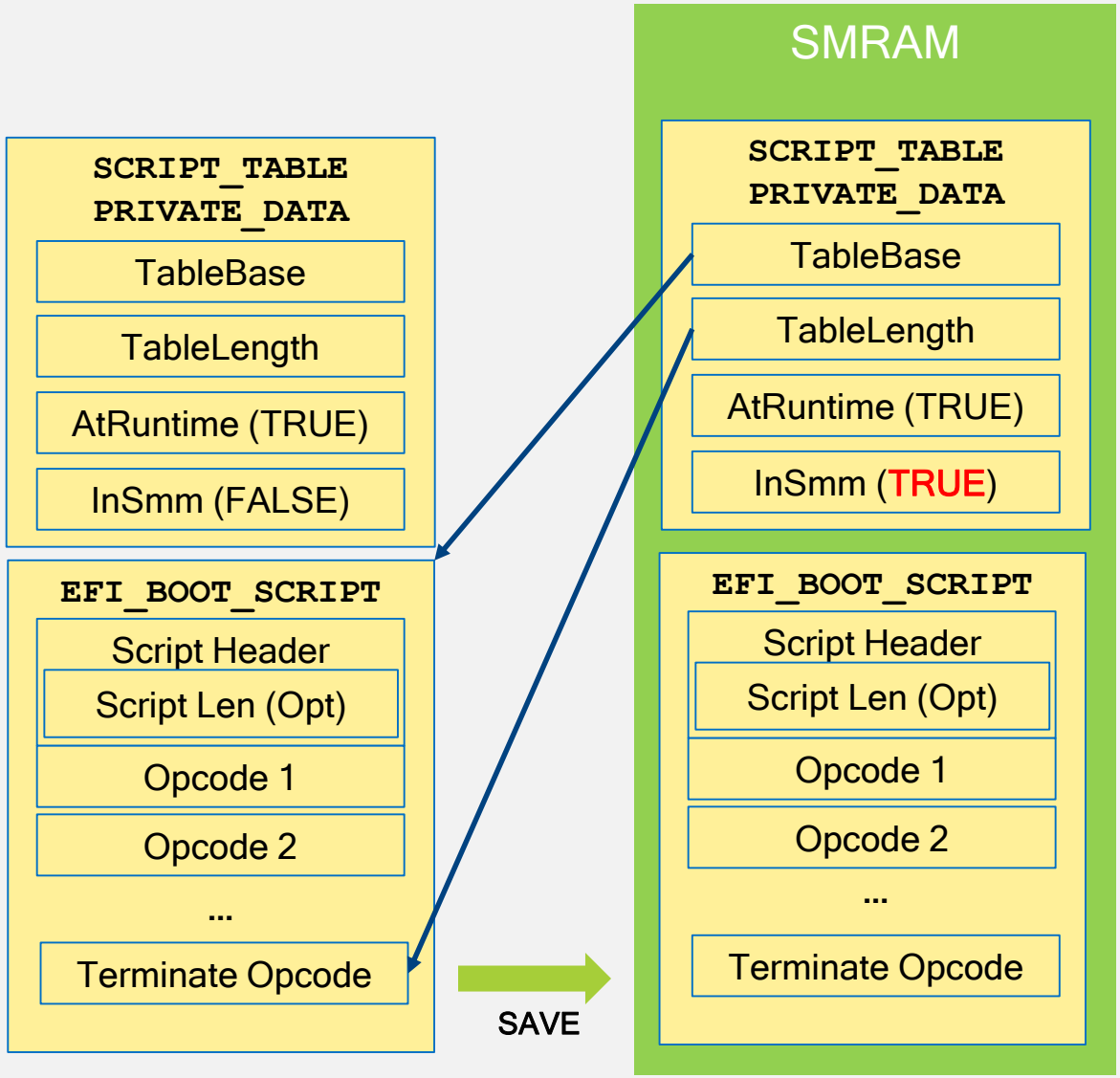
Protecting S3 Resume Boot Script (LockBox)

EDK2 LockBox Overview

- LockBox is a protected storage inaccessible to OS or DMA even across S3 sleep state
- LockBox can be backed by anything (e.g. ReadOnly NV UEFI variable or encrypted media)
- Current implementation uses SMRAM as LockBox
- Firmware copies data it needs to protect from the OS and DMA to SMRAM via **SaveLockBox()** API
- SMM LockBox is used in EDKII reference implementation to protect the S3 Resume Boot Script across S3 transition

UEFI LockBox details: A Tour Beyond BIOS Implementing S3 Resume with EDKII

Saving S3 Boot Script to LockBox



Saving S3 Boot Script to LockBox

SaveBootScriptDataToLockBox():

…

```
//
// mS3BootScriptTablePtr->TableLength does not include
EFI_BOOT_SCRIPT_TERMINATE, because we need add entry at runtime.
// Save all info here, just in case that no one will add boot
script entry in SMM.
//
Status = SaveLockBox (
           &mBootScriptDataGuid,
           (VOID *)mS3BootScriptTablePtr->TableBase,
           mS3BootScriptTablePtr->TableLength +
           sizeof(EFI_BOOT_SCRIPT_TERMINATE)
           );
ASSERT_EFI_ERROR (Status);
Status = SetLockBoxAttributes (&mBootScriptDataGuid,
LOCK_BOX_ATTRIBUTE_RESTORE_IN_PLACE);
```

https://svn.code.sf.net/p/edk2/code/trunk/edk2/MdeModulePkg/Library/PiDxeS3BootScriptLib/BootScriptSave.c

6.2 Hardware Based Firmware Protections

Protection From SMM Cache Attacks

System Management Range Registers (SMRR)

- 2 MSRs: **`SMRR_PHYSBASE, SMRR_PHYSMASK`**
- Physical address PA hits SMRR range when:

  `PA & SMRR_PHYSMASK == SMRR_PHYSBASE & SMRR_PHYSMASK`
- Force region of memory defined by SMRR as un-cacheable (**`UC`**) for non-SMM software access (e.g. from OS) to prevent cache fills in the range
- Non-SMM memory writes are dropped, non-SMM memory reads return all F's
- If CPU is in SMM, SMRR define memory type (typically **`WB`**)

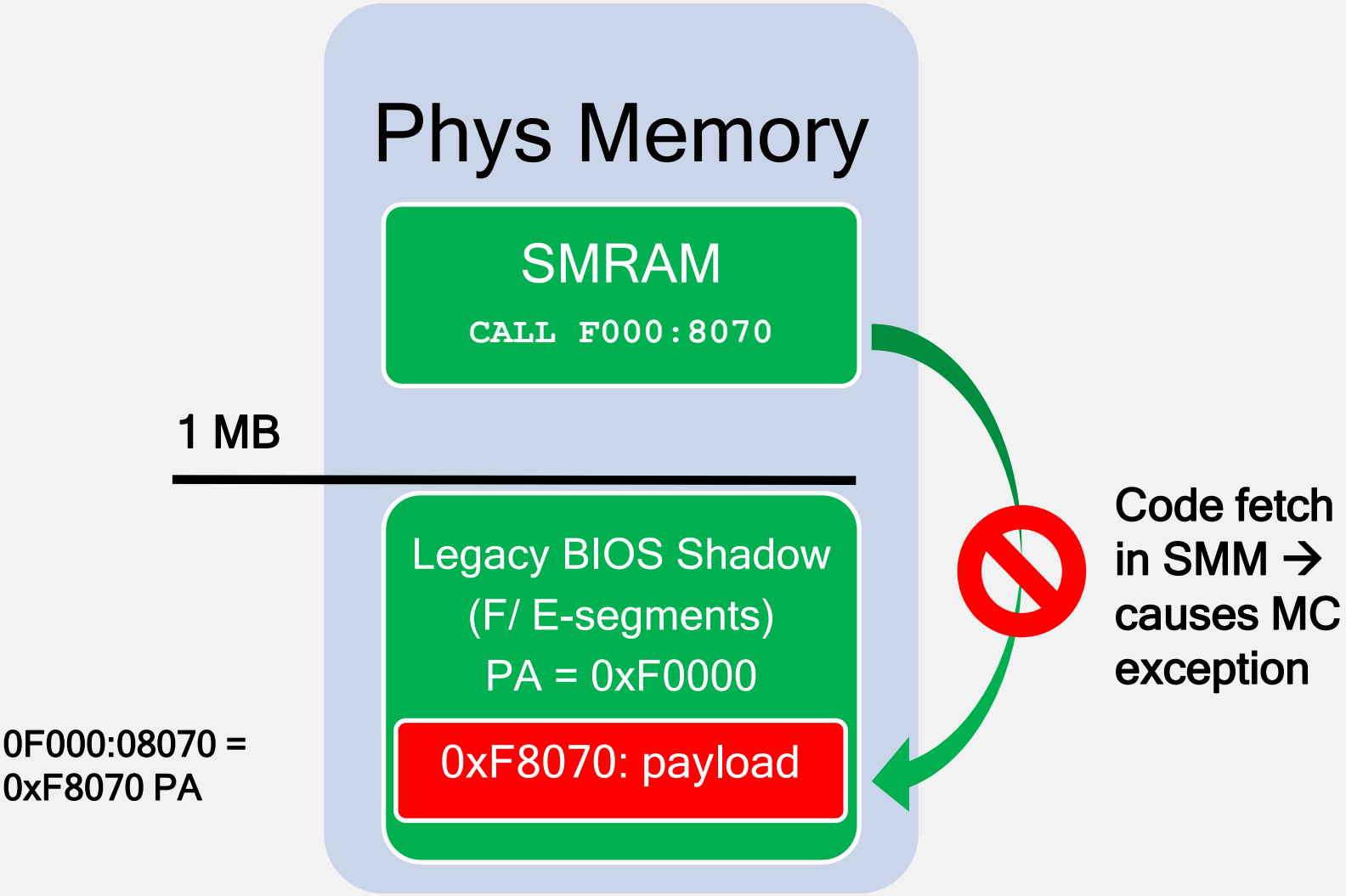
SMM Code Access Check

Mitigating SMM Call-Outs

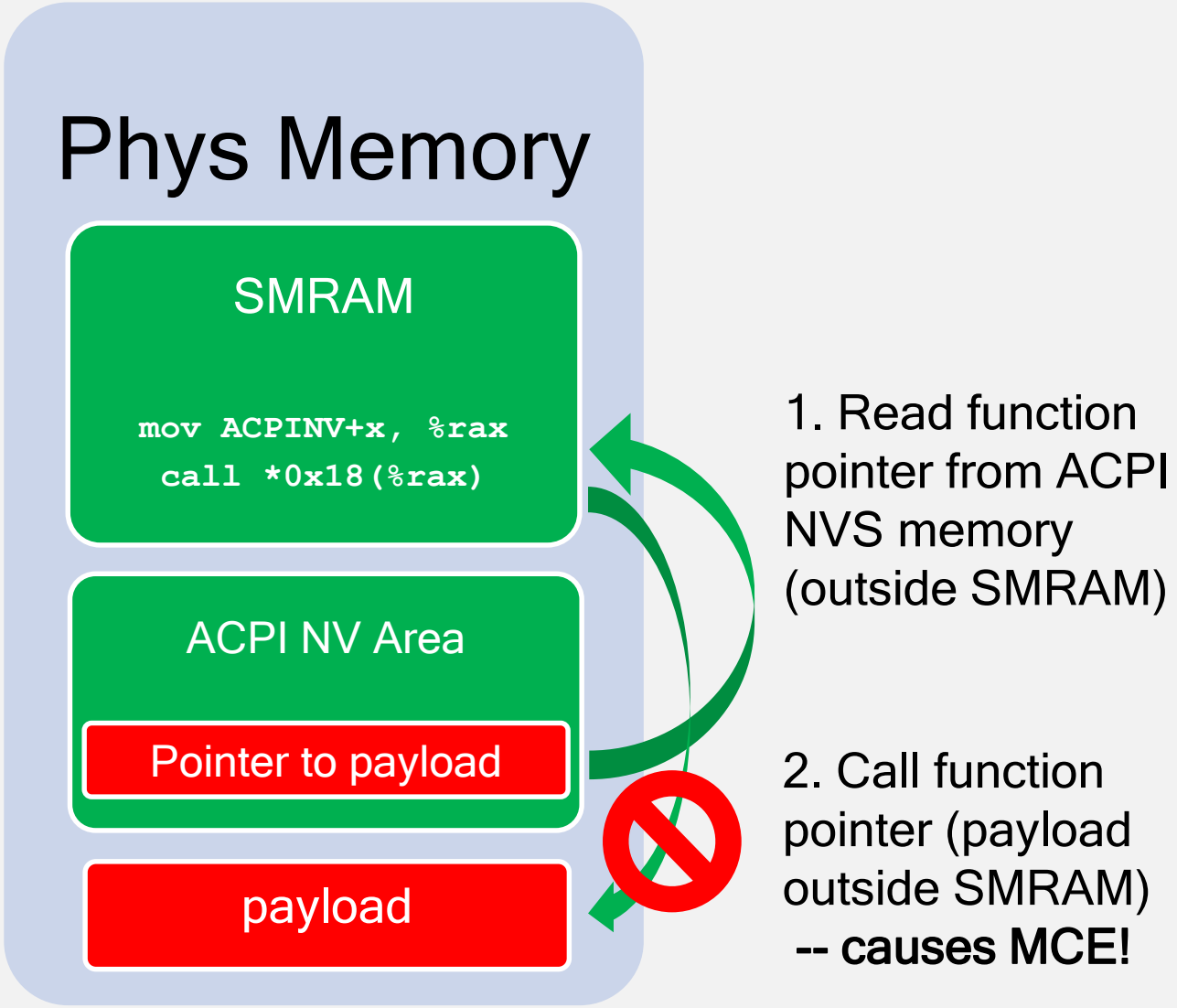
1. Don’t call any function outside of protected SMRAM
   - Violates “No read down” rule of classical Biba integrity model

2. Enable SMM Code Access Check CPU protection
   - Available starting in Haswell based CPUs
   - Available if **`MSR_SMM_MCA_CAP[58] == 1`**
   - When enabled, attempts to execute code not within the ranges defined by the SMRR while inside SMM result in a Machine Check Exception

Blocking Code Fetch Outside of SMRAM



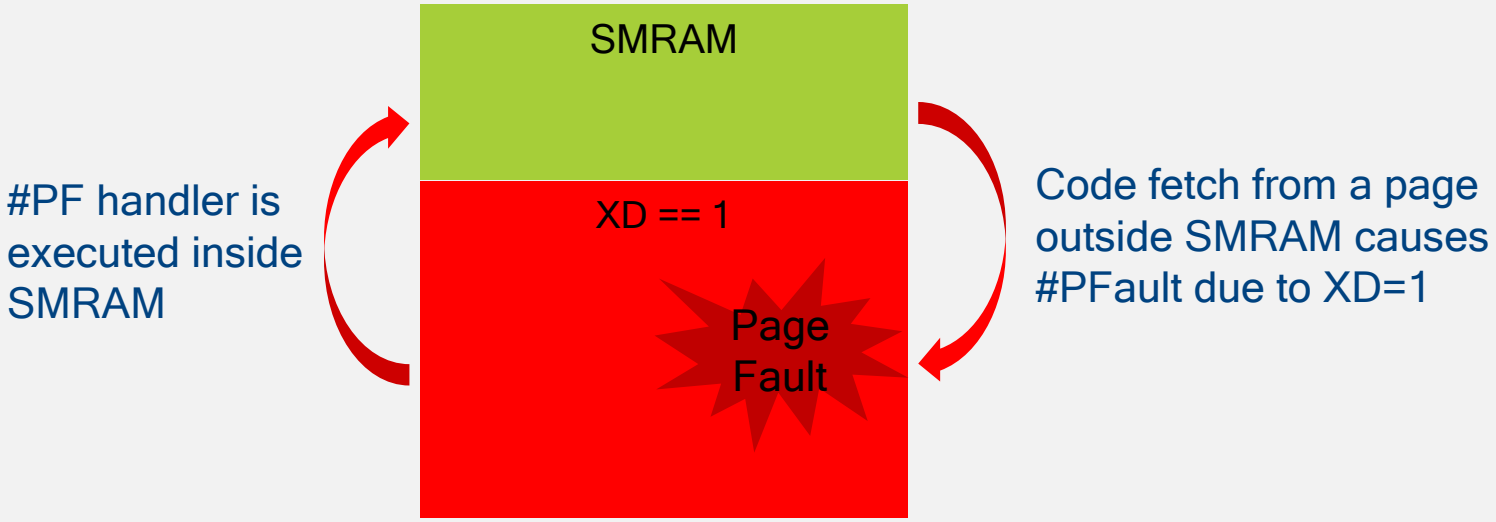
Function Pointers Outside of SMRAM (DXE SMI)



Paging based SMM code access check

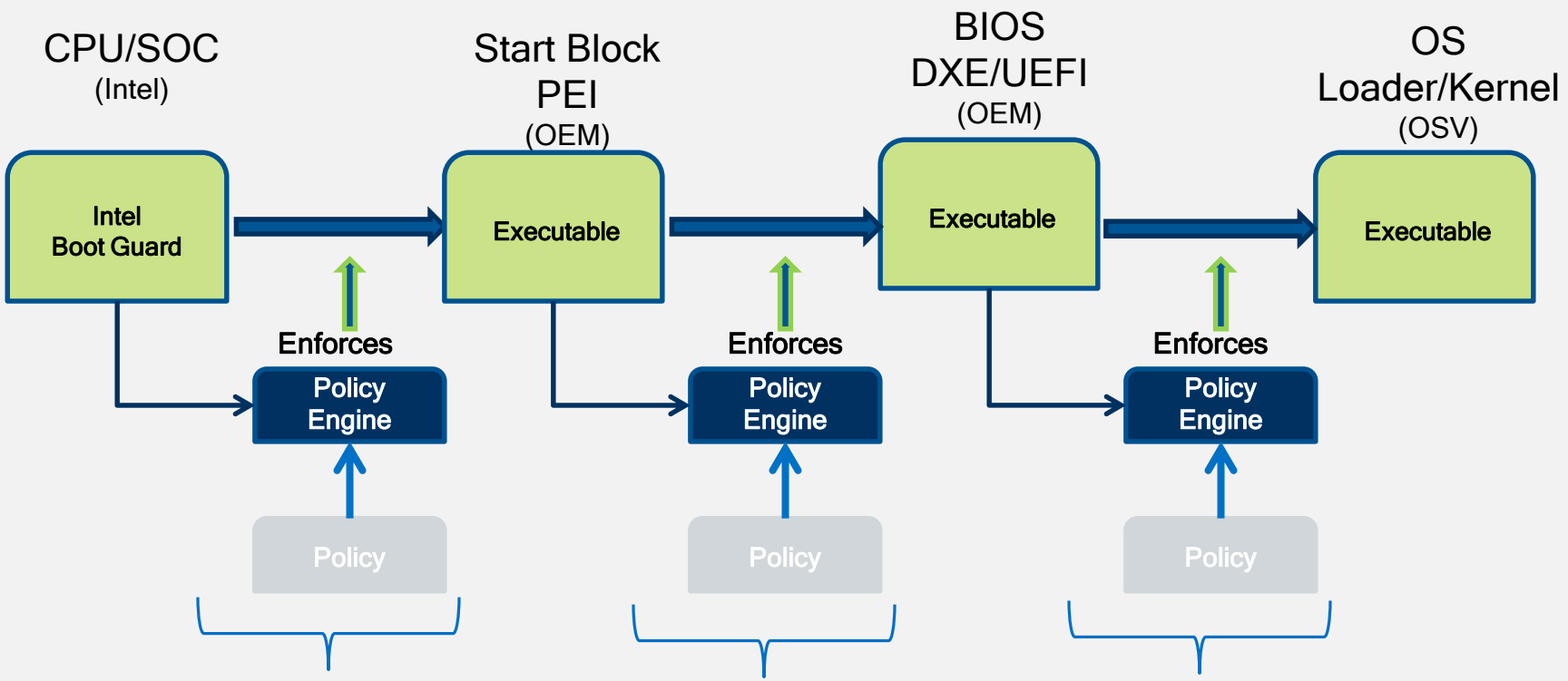
NX based soft SMM Code Access Check patches by Phoenix

- SMM paging/NX are enabled when CPU enters SMM
- PTEs outside of SMRAM have XD=1
- #PF is signaled when SMI handler attempts to fetch from any page outside of SMRAM



Hardware Secure Boot

Example: Intel® Boot Guard



http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/4th-gen-core-family-mobile-brief.pdf

OEM PI Verification Using PI Signed Firmware Volumes

Vol 3, section 3.2.1.1 of PI 1.3 Specification

OEM UEFI 2.4 Secure Boot

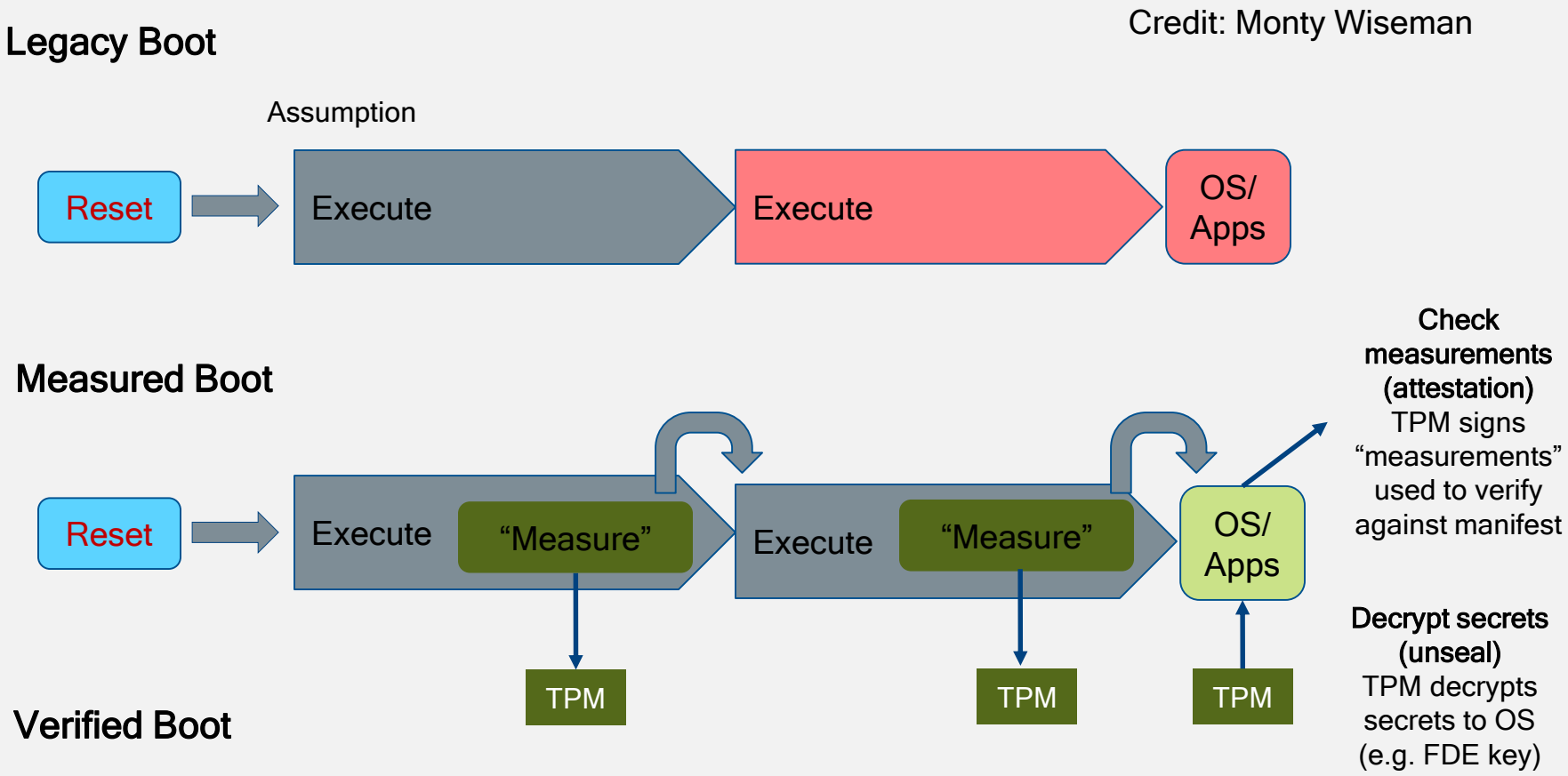
Chapter 27.2 of The UEFI 2.4 Specification

Example: Intel® Boot Guard

1. CPU loads BootGuard specific Authenticated Code Module (ACM) to validate the initial firmware boot block (IBB)
2. IBB validation includes signature verification and/or measurement into the TPM PCR
3. Verified boot uses 2 manifests: OEM public key manifest and IBB manifest structures
4. OEM programs 256 bits of SHA-256 of RSA public key used by ACM to verify the IBB signature into one-time field programmable fuses at the manufacturing
5. OEM programs policies into the fuses
   - Verified and/or Measured Boot
   - ACM or IBB verification failure response

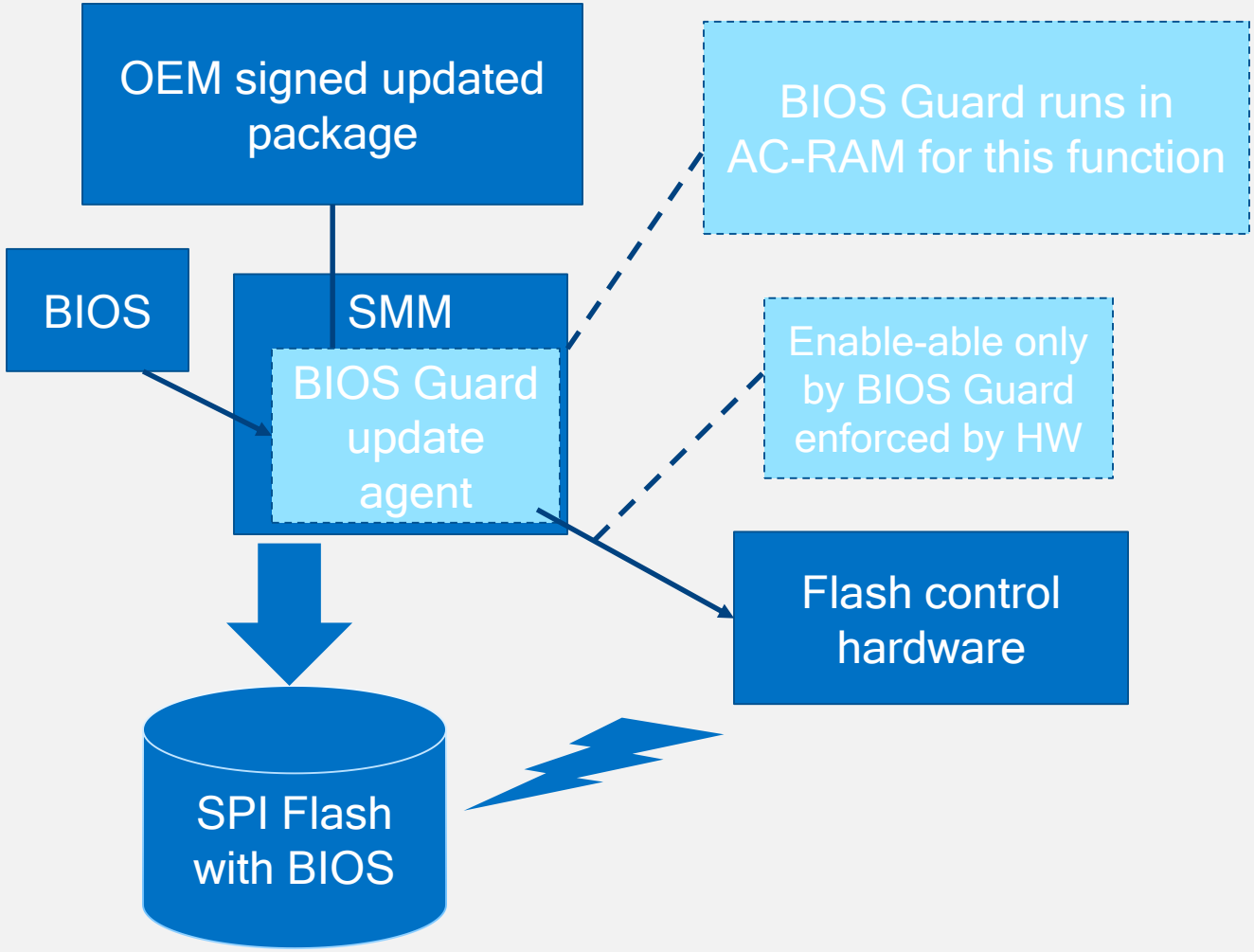
Verified vs Measured Boot

Credit: Monty Wiseman



Hardware Based System Firmware Update

Example: Intel® BIOS Guard

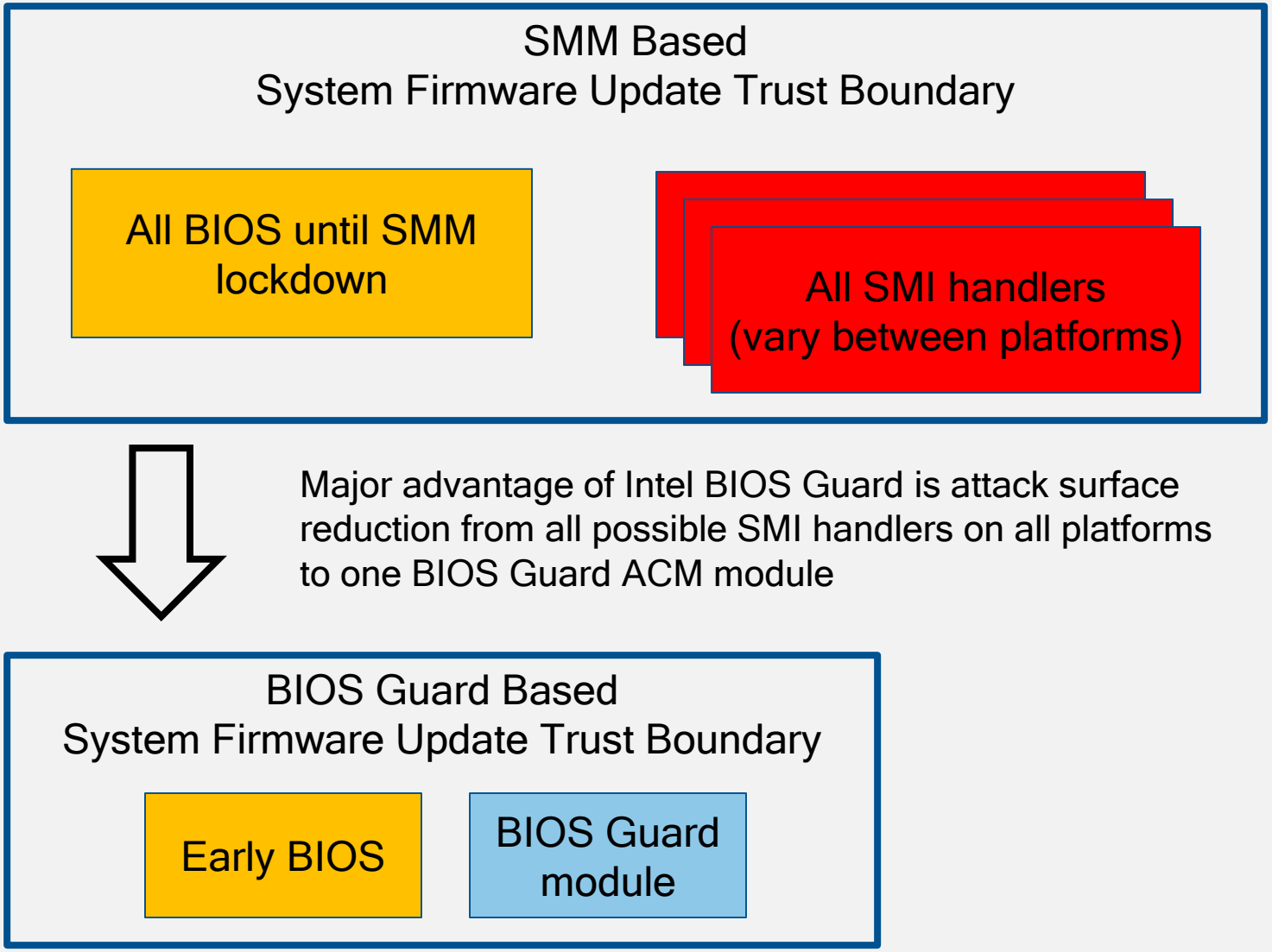


Source: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/security-technologies-4th-gen-core-retail-paper.pdf

SMM BIOS Update Trust Boundary

- For runtime BIOS Update (e.g. on server platforms), all complex SMI handlers code is in the trust boundary of the firmware update
- Different systems have different SMI handlers which makes it difficult to ensure consistent security level of SMI code across all system and security level of firmware update
- BIOS Guard reduces SMI handler attack surface, using one signed BIOS Guard authenticated code module (ACM)
- Platforms enabling BIOS Guard only need to use one module for a given processor generation

Trust Boundary with BIOS Guard



BIOS Guard Based Firmware Update

- BIOS Guard can update contents of the BIOS region in system SPI flash and EC firmware on EC flash memory
- BIOS Guard module is authenticated code module (ACM) executing in internal processor AC RAM
- When BIOS Guard is enabled, only BIOS Guard module is able to write to system SPI flash memory
- BIOS Guard verifies the signature of a firmware update package signed by a platform manufacturer prior to writing to system SPI flash memory

6.3 Trusted (Measured) Boot with Trusted Platform Module

Secure or Measured Boot?

How do you mitigate risk of a lost/stolen laptop?

- Example: Software based Full Disk Encryption like Microsoft BitLocker

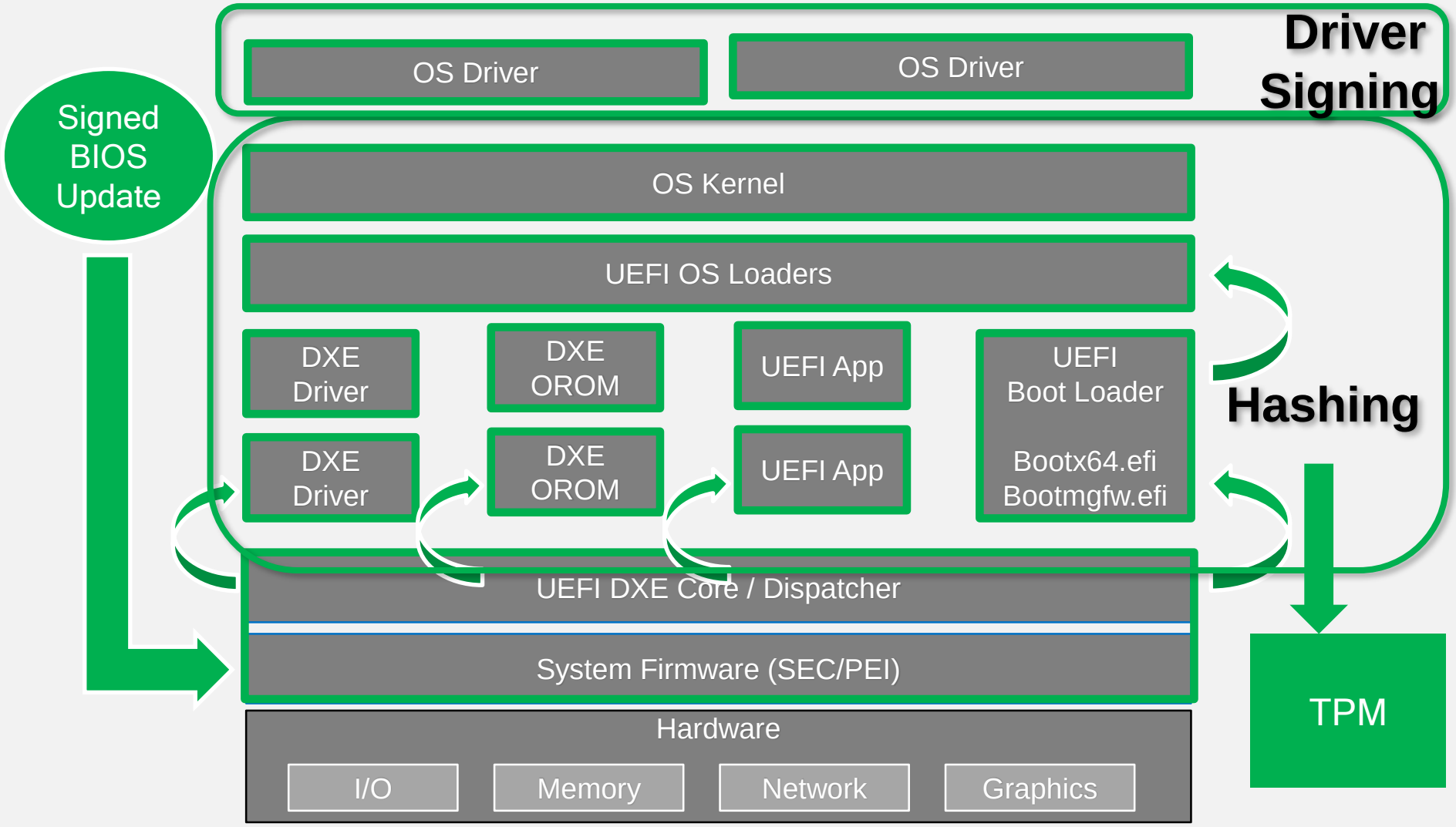
How do you attack full disk encryption?

- Example: Evil Maid attack infecting the firmware or boot loaders to log/steal FDE PIN

Protecting Against the Evil Maid Attack

- Evil Maid Attack
- Get a system with Trusted Platform Module (TPM)
- You trust TPM (including its firmware) & system firmware (BIOS)
  - Problem: Angry Evil Maid Attack
- During measured boot process, initial firmware creates hashes of the next firmware stages and boot loaders what is known as *measuring*
- Initial firmware is *Static Root of Trust for Measurement (STRM)*
- FDE enabled boot loader then *seals* volume encryption key(s) to these measurements in the PCRs
- Thus each boot, the keys can only be decrypted (*unsealed*) and the volume decrypted when all firmware and boot loaders have the same measurements (hashes)

Measured Boot



Hash into TPM PCRs instead of signature check

Measurements into TPM PCRs

⊗ Initial startup FW at CPU reset vector

PCR[0] ← CRTM, UEFI Firmware, PEI/DXE [BIOS], UEFI Boot and Runtime Services, Embedded EFI OROMs, SMI Handlers, Static ACPI Tables

PCR[1] ← SMBIOS, ACPI Tables, Platform Configuration Data

PCR[2] ← EFI Drivers from Expansion Cards [Option ROMs]

PCR[3] ← [Option ROM Data and Configuration]

PCR[4] ← UEFI OS Loader, UEFI Applications [MBR]

PCR[5] ← EFI Variables, GUID Partition Table [MBR Partition Table]

PCR[6] ← State Transitions and Wake Events

PCR[7] ← UEFI Secure Boot keys (PK/KEK) and variables (db, dbx..)

PCR[8] ← TPM Aware OS specific hashes [NTFS Boot Sector]

PCR[9] ← TPM Aware OS specific hashes [NTFS Boot Block]

PCR[10] ← [Boot Manager]

PCR[11] ← BitLocker Access Control

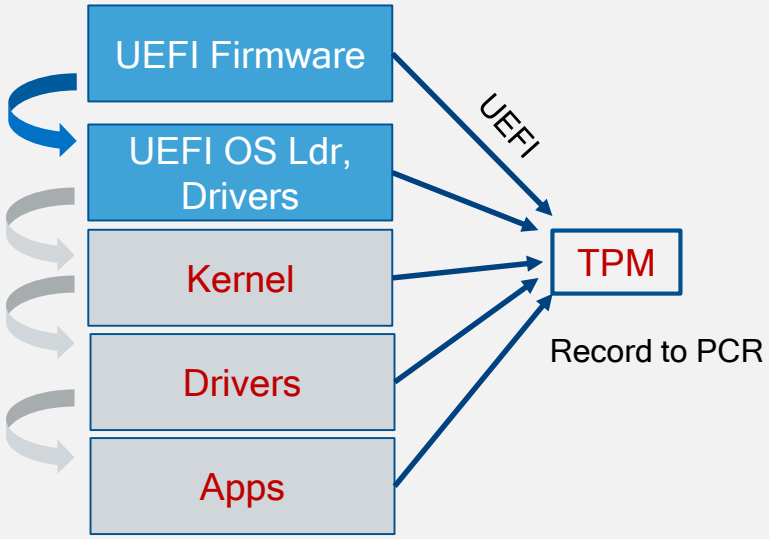
UEFI Secure Boot vs TCG Trusted Boot

UEFI Secure Boot

- UEFI authenticates OS loader (pub key and policy)
- Checks signature of before loading
- UEFI Secure boot will stop platform boot if signature is not valid
- UEFI requires remediation mechanisms if boot fails

UEFI TCG Measured Boot

- UEFI PI measures OS loader & UEFI drivers into TPM PCRs
- TCG Trusted boot never fails to boot firmware/OS
- Incumbent upon other SW to make security decision using attestation and/or unsealing



Source: Secure Boot Ecosystem Challenges by Vincent Zimmer

What does it all mean?

Secure Boot

- System firmware looks for authorized signature before execution

Measured Boot

- System firmware hashes images into PCRs before execution

Both

- Trust system firmware to do checks (hash or sig check)
- Only do startup checks, not runtime

Example with Measured Boot

1. Let's look at some PCRs

```
cat /sys/class/misc/tpm0/device/pcrs
```

2. Let's seal something using the `keyctl` command:
   - the "trusted" keys are encrypted and sealed in the TPM
   - options include sealing to PCR values

```
keyctl add trusted name "new keylen [options]" ring
```

Training materials are available on Github

https://github.com/advanced-threat-research/firmware-security-training

Yuriy Bulygin	@c7zero
Alex Bazhaniuk	@ABazhaniuk
Andrew Furtak	@a_furtak
John Loucaides	@JohnLoucaides