tianocore

# Understanding the UEFI Secure Boot Chain

# TABLE OF CONTENTS

# UNDERSTANDING THE UEFI SECURE BOOT CHAIN

**Technical Briefing**

**06/20/2019 04:33:10**

**Revision 01.0**

## Contributed by

Jiewen Yao, Intel Corporation

Vincent J. Zimmer, Intel Corporation

## Acknowledgements

BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



## Revision History

| Revision | Revision History | Date |
|---|---|---|
| 01.0 | Initial release. | June 2019 |
| | | |

***Understanding the UEFI Secure Boot Chain***

# EXECUTIVE SUMMARY

This document introduces how to implement a secure boot chain in UEFI using the TianoCore EDK II project.

**Prerequisite**

This document assumes that the audience has basic firmware development experience with UEFI & EDK II, along with basic knowledge of UEFI boot flow and cryptography.

# OVERVIEW

System firmware, commonly referred to as Basic Input Output System (BIOS), plays an important role in platform security. Running unauthorized firmware components may introduce significant threats by creating a permanent denial of service or introducing persistent malware. In 2011, the National Institute of Standards and Technology (NIST) published BIOS Protection Guidelines (SP800-147). NIST extended the scope to all platform firmware and published Platform Firmware Resiliency Guidelines (SP800-193) in 2018. The goal of those documents is to provide a general guideline for firmware integrity.

## Integrity Models

The UEFI Secure Boot chain can be applied to the Clark-Wilson integrity policy, developed in 1987. The Clark Wilson model includes the following concepts:

1. Data Item:
    i. Constrained Data Item (CDI)
    ii. Unconstrained Data Item (UDI)
2. Procedure:
    i. Integrity Verification Procedure (IVP)
    ii. Transformation Procedures (TPs)
3. Rule:
    i. Certification Rule (CR) – integrity monitoring
        i. **C1**: (Basic: IVP Certification) All IVPs must properly ensure that all CDIs are in a valid state.
        ii. **C2**: (Basic: Validity) All TPs must be certified to be valid. For each TP and each set of CDI that it may manipulate, the security officer must specify a “relation” of the form: (TP, {CDI}).
        iii. **C3**: (Separation of Duty Certification) The list of relation in E2 must be certified to meet the separation of duty requirement.
        iv. **C4**: (Journal Certification) All TPs must be certified to write to an append-only CDI (the log) all information necessary to permit the nature of the operation to be reconstructed.

v. **C5**: (Transformation Certification) Any TP that takes a UDI as an input value must be certified to perform only valid transformations, or no transformations, for any possible value of the UDI. The transformation should take the input from a UDI to a CDI, or the UDI is rejected.

ii. Enforcement Rule (ER) – integrity preserving

i. **E1**: (Basic: Enforcement of Validity) The system must maintain the list of relation specified in C2, and must ensure that only TPs certified to run on a CDI manipulate that CDI.

ii. **E2**: (Enforcement of Separation of Duty) The system must associate a user with each TP and set of CDIs in a list of relations of the form: (User, TP, {CDI}). It must ensure that only executions described in one of the relations are performed.

iii. **E3**: (User Identity) The system must authenticate the identity of each user attempting to execute a TP.

iv. **E4**: (Initiation) Only the agent permitted to certify entities may change the list of such entities associated with other entities, specifically the one associated with a TP. An agent that can certify an entity may not have any execute rights concerning that entity.

This model is based on the relationship between authenticated principal, program, and data items. The elements of this relationship are referred to as the “Clark-Wilson Triple” (User, TP, {CDI}). The Clark-Wilson model shows the rules required to meet the security properties of integrity: (from Blake).

Table 1-1: Clark-Wilson model

| Property | Description | Rule |
|---|---|---|
| **Integrity** | An assurance that CDIs can only be modified in constrained ways to produce valid CDIs. | C1, C2, C5, E1, E4 |
| **Access Control** | The ability to control access to resources. | C3, E2, E3 |
| **Auditing** | The ability to ascertain the changes made to CDIs and ensure that the system is in a valid state. | C1, C4 |
| **Accountability** | The ability to uniquely associate users with their actions. | E3. |

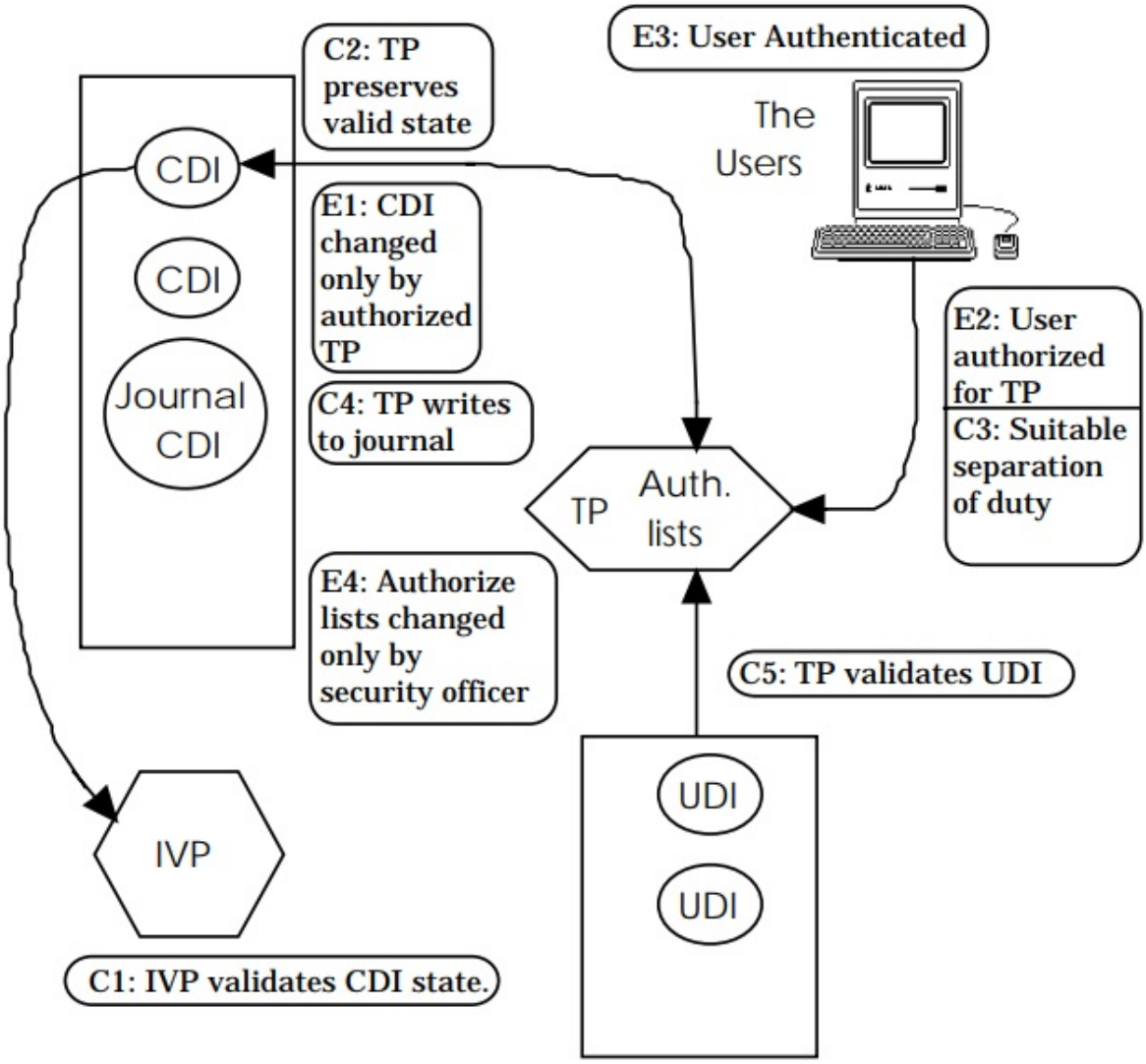



Figure 1-1: Clark-Wilson model, From Lee

Because the Clark-Wilson focuses on duty and transaction, it is more applicable to business and industry processes. Currently, some papers describe how to apply the Clark-Wilson integrity model to the existing system, such as Windows, Java or Trusted Computing Group (TCG) security.

## Introduction to the Secure Boot Chain

According to NIST SP800-147 and SP800-193, the system needs to maintain integrity and availability during the firmware boot process. In firmware, secure boot (aka verified boot) uses a set of policy objects to verify the next entity before execution. For example, to match C5, the system uses the TP (verification procedure) to verify the UDI (untrusted firmware component), transforms the UDI into a CDI (trusted firmware component), and executes it.

In contrast, a trusted boot (aka measured boot) process does not verify the next entity. It only records the digest of the next boot entity to a trusted location, such as a Platform Configuration Register (PCR) in the Trusted Platform Module (TPM). This allows a trusted

boot chain to be verified later in the boot process. Many security models use secure boot and trusted boot capabilities in combination for maximum effectiveness.

Table 1-2: Clark-Wilson model in Firmware

| Property | Description | Rule | Firmware Secure Boot |
|---|---|---|---|
| **Integrity** | An assurance that CDIs can only be modified in constrained ways to produce valid CDIs. | C1, C2, C5, E1, E4 | Yes. Firmware needs to verify the next component |
| **Access Control** | The ability to control access to resources. | C3, E2, E3 | No. There is no user concept in secure boot. |
| **Auditing** | The ability to ascertain the changes made to CDIs and ensure that the system is in a valid state. | C1, C4 | Yes, if TCG trusted boot is enabled. TCG event log may record such information. |
| **Accountability** | The ability to uniquely associate users with their actions. | E3. | No. There is no user concept in secure boot. |

## Patterns in the Secure Boot Chain

Definition:

1. Firmware[N] - the N level firmware binary. Any firmware layer is updatable.

   Firmware[0] means the component verified by Hardware.

   Firmware[N] means the component verified by Firmware[N-1].

   It may include both code (Firmware[N].Code) and data (Firmware[N].Data).

2. Firmware[N].Code - the code of the N level firmware binary.

   It may include the verifier (Firmware[N].Code.Verifier.)

3. Firmware[N].Data - the data of the N level firmware binary.

   It may include the verification policy. (Firmware[N].Data.Policy.)

4. Firmware[N].Code.Verifier - the verification function of the N level firmware binary.

5. Firmware[N].Data.Policy - the policy data inside of the N level firmware binary. This data is used by the verification function. Both verification function and policy data have below subcategory:

i. Boot - the firmware boot

ii. FirmwareUpdate - the firmware update (it may or might not include policy data)

iii. PolicyUpdate - the policy update. It may or might not exist.

iv. Recovery - the firmware recovery

v. Communication - the firmware runtime communication

6. Hardware – the hardware, including Register Transfer Level (RTL) and register. The hardware is not updatable. The hardware must be fused when it is shipped to the end user.

There are two types of verification:

1. The verifier for boot (verified boot). The read-only code and read-only data are in this category. This category includes both initial installation and upgrade. For example, UEFI Secure Boot is for code installation, or signed capsule update is for code/data upgrade. In most cases, the verification is based upon a crypto-algorithm, such as Secure Hash Algorithm (SHA) or Rivest-Shamir-Adleman Algorithm (RSA). The policy data can be the hash value of the firmware or the public key hash of the firmware. Above 5.a, 5.b, 5.c, 5.d belongs to this type.
2. The verifier for communication (verified communication). The read/write data are in this category. This category is for cross-boundary data passing such as SMM communication, including the UEFI non-volatile variable. In most cases, the verification is based upon the boundary check, valid range check, etc. Above 5.e belongs to this type.

## Patterns for Verified Boot

Table 1-3: Patterns for Verified Boot

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Firmware[N].Code.Verifier.Boot (Firmware[N].Data.Policy.Boot, Firmware[N+1]) | Firmware Owner | Same as Firmware[N] |
| **CDI** | Firmware[N] | Firmware Owner | Originally on Flash, loaded into RAM by Firmware[N-1] |
| **UDI** | Firmware[N+1] | Firmware Owner | Originally on Flash, loaded into RAM by Firmware[N] |

NOTE: If N == 0, Firmware[-1] means the hardware.

## Patterns for Verified Policy Update

Table 1-4: Patterns for Verified Policy Update

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Firmware[N].Code.Verifier.PolicyUpdate (Firmware[N].Data.Policy.PolicyUpdate, Firmware[N].Data.Policy:New) | Firmware Owner | |
| **CDI** | Firmware[N].Code.Verifier.PolicyUpdate + Firmware[N].Data.Policy.PolicyUpdate | Firmware Owner | In an isolated execution environment. As such the rest of Firmware[N] cannot tamper with it. |
| **UDI** | Firmware[N].Data.Policy:New | Policy Data Owner | Memory, loaded into an isolated environment, by Firmware[N]. Code.Verifier. PolicyUpdate |

- ## Patterns for Verified Firmware Update

Table 1-5: Patterns for Verified Firmware Update

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Firmware[N].Code.Verifier.FirmwareUpdate (Firmware[N].Data.Policy.FirmwareUpdate, Firmware[N]:New) | Firmware Owner | |
| **CDI** | Firmware[N] | Firmware Owner | Flash unlockable environment, loaded by Firmware[N-1] |
| **UDI** | Firmware[N]:New | Firmware Owner | Flash unlockable environment, loaded by original Firmware[N] |

- ## Patterns for Verified Recovery

Table 1-6: Patterns for Verified Recovery

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Firmware[N].Code.Verifier.Recovery (Firmware[N].Data.Policy.Recovery, Firmware[N+1]:Recovery) | Firmware Owner | |
| **CDI** | Firmware[N] | Firmware Owner | Originally on flash, loaded into RAM by Firmware[N-1] |
| **UDI** | Firmware[N+1]:Recovery | Firmware Owner | Originally on recovery storage (Flash, USB, Hard drive), loaded into RAM by Firmware[N] |

- **Patterns for Verified Runtime Communication**

Table 1-7: Patterns for Verified Runtime Communication

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Firmware[N].Code.Verifier.RuntimeCommunication (Firmware[N].Data.Policy.RuntimeCommunication, Data:New) | Firmware Owner | |
| **CDI** | Firmware[N].Code.Verifier.RuntimeCommunication + Firmware[N].Data.Policy.RuntimeCommunication | Firmware Owner | In an isolated execution environment. As such the rest of Firmware[N] cannot tamper it. |
| **UDI** | Data:New | Any | Memory, loaded into an isolated environment, by Firmware[N]. Code.Verifier. PolicyUpdate. This can be any Data, as long as the format is known by the producer and consumer. |

## Comparing Clark-Wilson and UEFI Secure Boot

The following table illustrates how the UEFI Secure Boot Chain maps to Clark-Wilson certification and enforcement rules.

Table 1-8: Comparison between Clark-Wilson and Secure Boot Chain

| Rule | Clark-Wilson | Secure Boot Chain |
|---|---|---|
| **C1** | The system will have an IVP for validating the integrity of any CDI. | Not applied today. No one validates the CDI. The integrity may be verified by using a signature check.<br>If TCG trusted boot is enabled, PCR validation can also be done. |
| **C2** | The application of a TP to any CDI must maintain the integrity of that CDI | Not applied. No User in UEFI.<br>UEFI does not provide isolation. Ideally, the TP should not change CDI not managed by TP. But the reality is hard to enforce.<br>SMM might be OK. ? |
| **C3** | A CDI can only be changed by a certified TP. Separation of duties / least privilege. | Not applied. No User in UEFI.<br>Similar to C2. Only SMM has isolation.<br>Data in SMM can only be changed in SMM.<br>But SMM only used for UEFI Secure Boot authenticated variable trust anchors, and Intel® BIOS Guard update. |
| **C4** | TP actions are logged. | TPM Event Log |
| **C5** | TP actions on UDIs result in valid CDIs. | YES. Input Verification – secure boot chain |
| **E1** | Only certified TPs may act on CDIs. | The verification TP is inside of verified firmware. |
| **E2** | Subjects may access CDIs only through TPs for which they are authorized. | Not applied. No User in UEFI.<br>All code in same privilege, except SMM. |
| **E3** | Subjects attempting to execute a TP must first be authenticated. | SMM |
| **E4** | Only administrators can specify TP authorizations. | NO. CPU – hardware owner. |

# SECURE BOOT CHAIN IN UEFI

This section describes the overview of the UEFI Secure Boot chain including the following:

- UEFI Secure Boot
- Intel® Boot Guard
- OEM boot block OBB Verification
- Boot Chain – Putting it all together
    - Signed Capsule Update
    - Intel® BIOS Guard
    - Signed Recovery
    - S3 Resume
    - SMM Runtime Communication

# UEFI Secure Boot

UEFI Secure Boot is a feature defined in the UEFI Specification. It guarantees that only valid 3rd party firmware code can run in the Original Equipment Manufacturer (OEM) firmware environment. UEFI Secure Boot assumes the system firmware is a trusted entity. Any 3rd party firmware code is not trusted, including the bootloader installed by the Operating System Vendor (OSV) and peripherals provided by an Independent Hardware Vendor (IHV). The end user may choose to enroll and revoke entries in the UEFI Secure Boot image security database as part of managing verification policy.

UEFI Secure Boot includes two parts - verification of the boot image and verification of updates to the image security database. Figure 2-1 shows the UEFI Secure Boot verification flow. Table 2-1 shows the key/image security database used in UEFI Secure Boot.

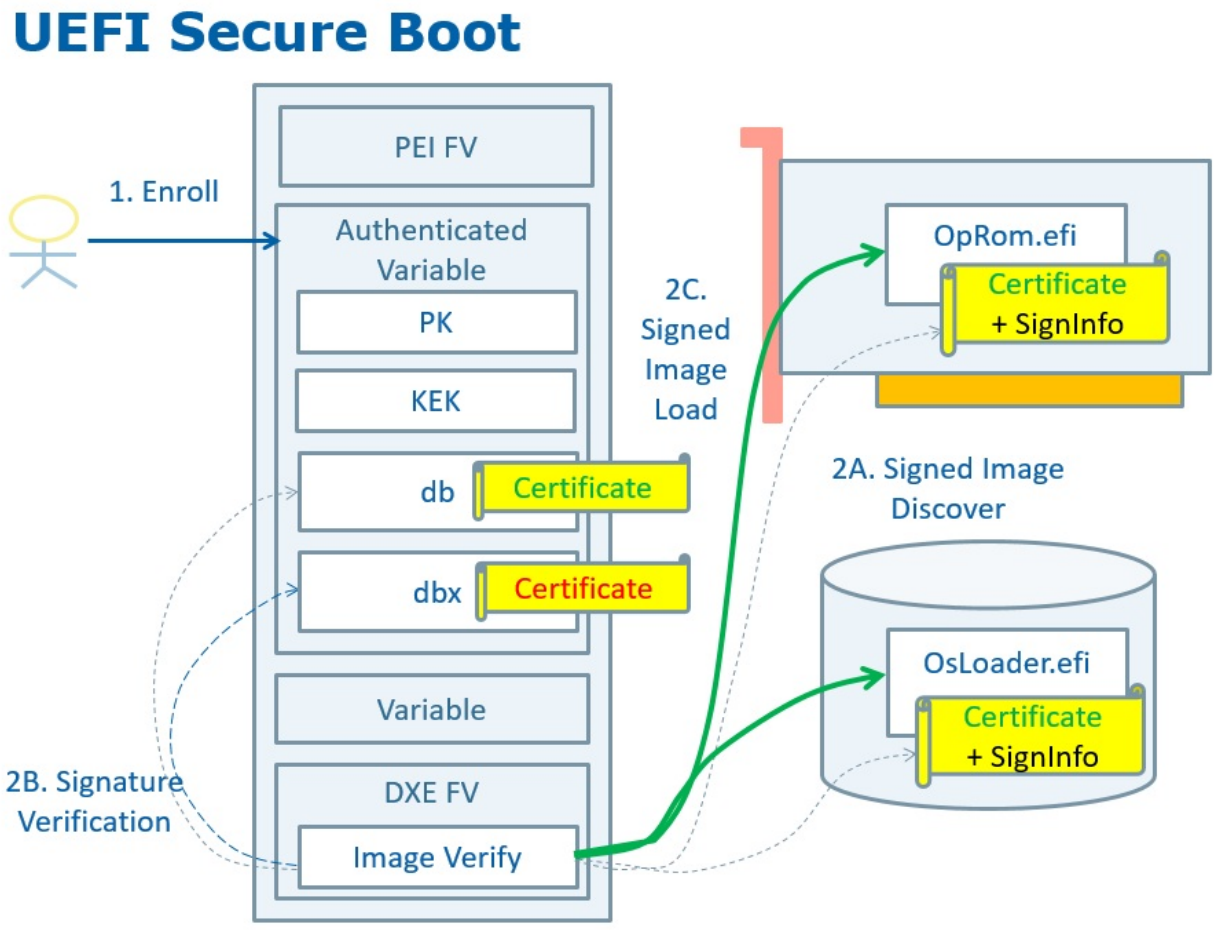



**Figure 2-1: UEFI Secure Boot**

Table 2-1: Key Usage in UEFI Secure Boot

| Key | Verifies | Update is verified by | NOTES |
|---|---|---|---|
| PK | New PK<br>New KEK<br>New db/dbx/dbt/dbr<br>New OsRecoveryOrder<br>New OsRecovery#### | PK | Platform Key |
| KEK | New db/dbx/dbt/dbr<br>New OsRecoveryOrder<br>New OsRecovery#### | PK | Key Exchange Key |
| db | UEFI Image | PK/KEK | Authorized Image Database |
| dbx | UEFI Image | PK/KEK | Forbidden Image Database |
| dbt | UEFI Image + dbx | PK/KEK | Timestamp Database |
| dbr | New OsRecoveryOrder<br>New OsRecovery#### | PK/KEK | Recovery Database |

## UEFI Secure Boot Image Verification

Table 2-2: UEFI Secure Boot Image Verification

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | UEFI Secure Boot Image Verification | OEM | Originally on flash, loaded into DRAM |
| **CDI** | Manufacture Firmware Code | OEM | Originally on flash, loaded into DRAM |
| | UEFI Secure Boot Image Security Database (Policy) | End user (or OEM default) | Originally on flash, authenticated variable region, loaded into DRAM |
| **UDI** | 3rd party Firmware Code, (OS boot loader) | OSV | Originally on external storage (e.g. Hard drive, USB), loaded into DRAM |
| | 3rd party Firmware Code, (PCI Option ROM) | IHV | Originally on PCI card, loaded into DRAM |
| | 3rd party Firmware Code, (UEFI Shell Tool) | Any | External Storage (e.g. hard drive, USB), loaded into DRAM |

Table 2-2 shows the component involved in the UEFI Secure Boot Image Verification.

## Signing

In UEFI Secure Boot, the UDI is any 3rd part firmware code, including the OS boot loader, PCI option ROMs, or a UEFI shell tool. The component provider needs to sign these components with a private key and publish the public key.

## Public Key Storage

The OEM or end user may enroll the public key as a CDI (UEFI Secure Boot Image Security Database). The database is in a UEFI Authenticated Variable region. The database can also be

updated during runtime. It can be read by anyone but only be written after data authentication. See Table 2 below.

## Verification

During boot, the TP (Image Verification Procedure) verifies the UDI (3rd party firmware code), according to the CDI (UEFI Secure Boot Image Security Database) as policy. If the verification passes, the UDI is transformed into a CDI and the 3rd party firmware code is executed. If the verification fails, the 3rd party firmware code is discarded.

Figure 2-2 shows a verification flow using db/dbx.

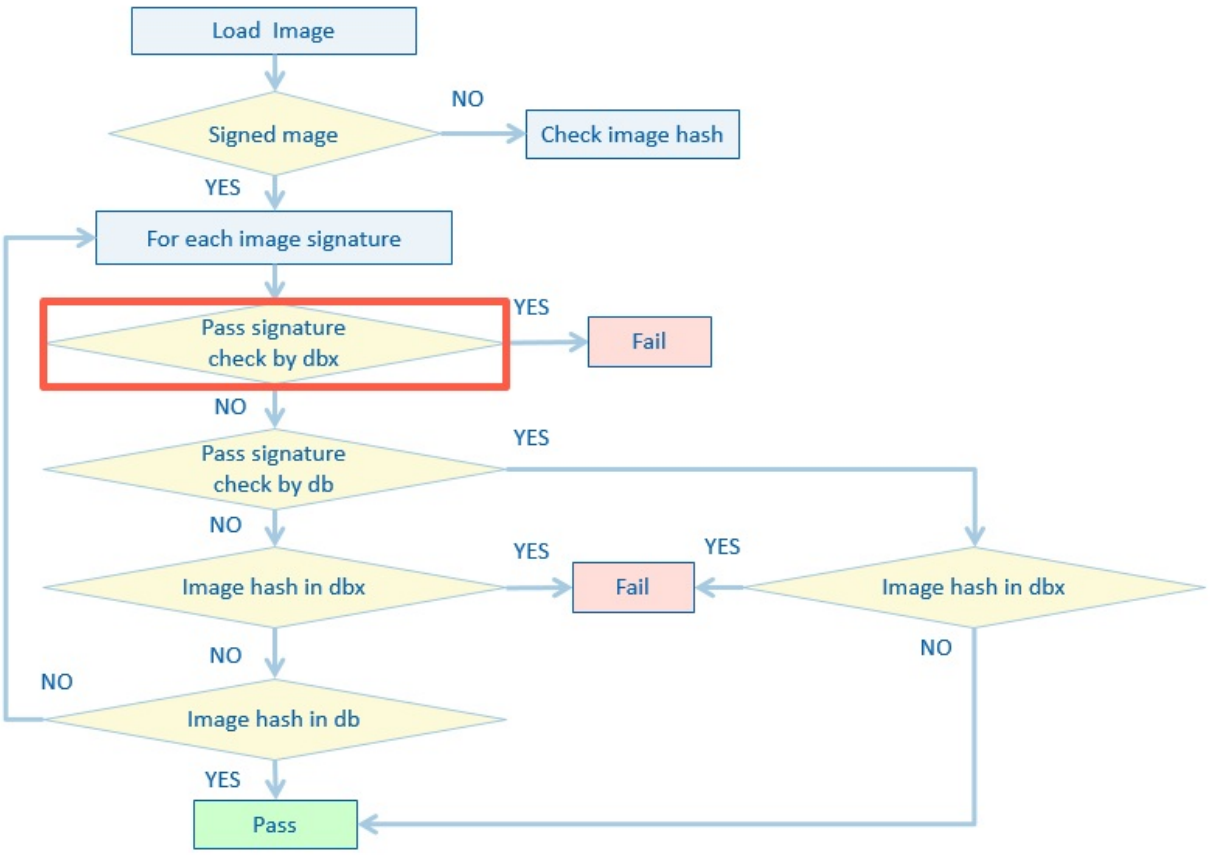


Figure 2-2: Image Verification flow

Figure 2-3 shows a verification flow introducing dbt. An additional check is required based dbx signature.

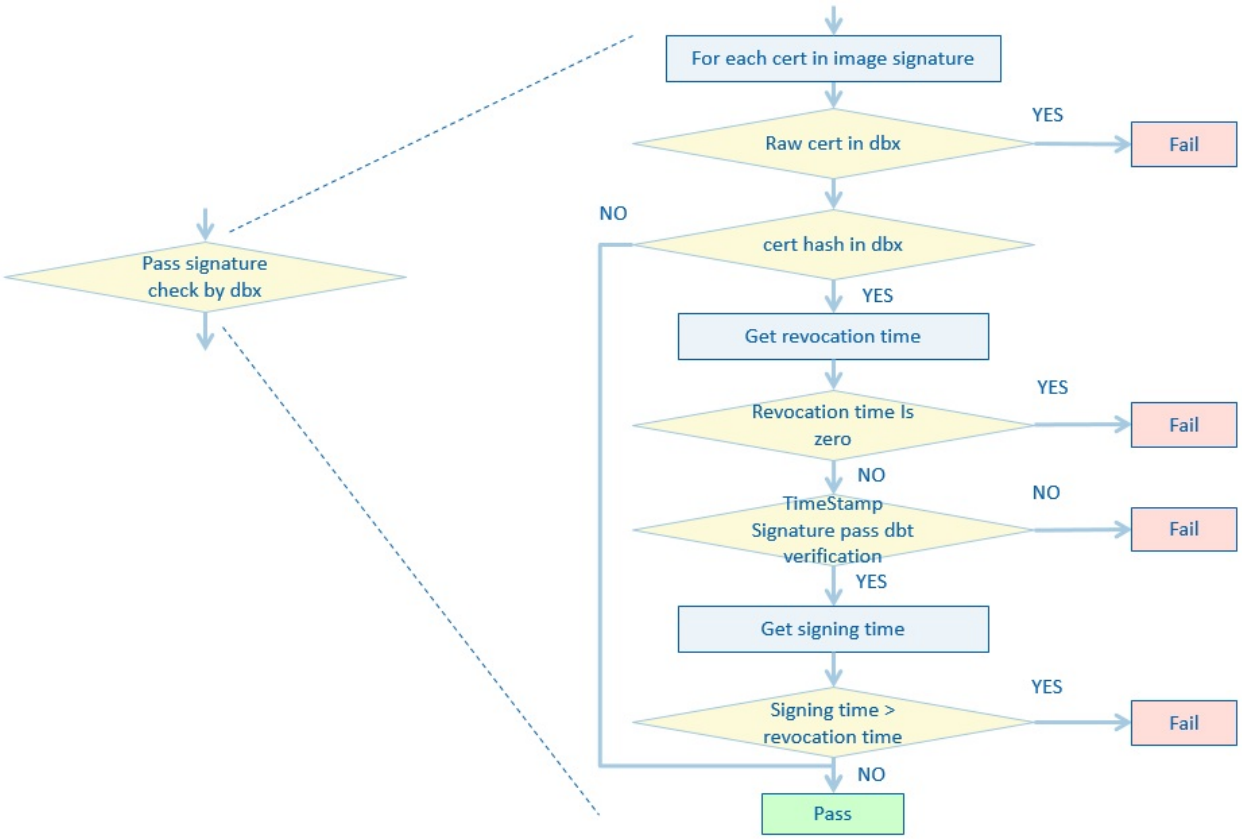



Figure 2-3: Image Verification with timestamp signature database

## UEFI Authenticated Variable Verification (Policy Update)

Table 2-3: UEFI Authenticated Variable Verification

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | UEFI Authenticated Variable Verification | OEM | Originally on flash, loaded into SMRAM |
| **CDI** | Manufacture Firmware Code in SMM. | OEM | Originally on flash, loaded into SMRAM |
| | UEFI Secure Boot Image Security Database (Policy) | End user (or OEM default) | Originally on flash, loaded into SMRAM |
| **UDI** | New UEFI Secure Boot Image Security Database | End user | Originally in normal DRAM, loaded into SMRAM |

In Table 2-2, the CDI (UEFI Secure Boot Image Security Database) is updatable. The database itself is in the UEFI Authenticated Variable region. Table 2-3 shows the component involved in the UEFI Authenticated Variable Verification.

## Signing

To update the existing Image Security Database (CDI), the new Image Security Database (UDI) needs to be signed if UEFI Secure Boot is enabled.

## Public Key Storage

The signer's public key must be enrolled in system firmware. It is the same as the public key used for UEFI Secure Boot Image Verification. The database is stored in a UEFI Authenticated Variable region.

## Verification

During runtime update, the TP (Authenticated Variable Verification Procedure) verifies the UDI (new Image Security Database), according to the CDI (UEFI Secure Boot Image Security Database) as policy. If verification passes, then the UDI is transformed into a CDI, and the new Image Security Database takes effect on the next boot. If verification fails, the new Image Security Data is discarded.

For details on the authenticated variable flow, please refer to the "Implementing

UEFI Authenticated Variables in SMM with EDK II" whitepaper.

# Intel® Boot Guard

UEFI Secure Boot assumes the OEM platform firmware is a Trusted Computing Base (TCB) and trusts it implicitly. A better implementation relies on a smaller TCB to verify the OEM platform firmware. A solution can be implemented using Intel® Boot Guard. This feature verifies the entire OEM platform firmware image using two components:

- Authenticated Code Module (ACM) Initial Boot Block (IBB) Verification
- Microcode ACM Verification.

Figure 2-4 shows the components involved in Intel® Boot Guard. Table 2-4 shows the key usage in Intel® Boot Guard.

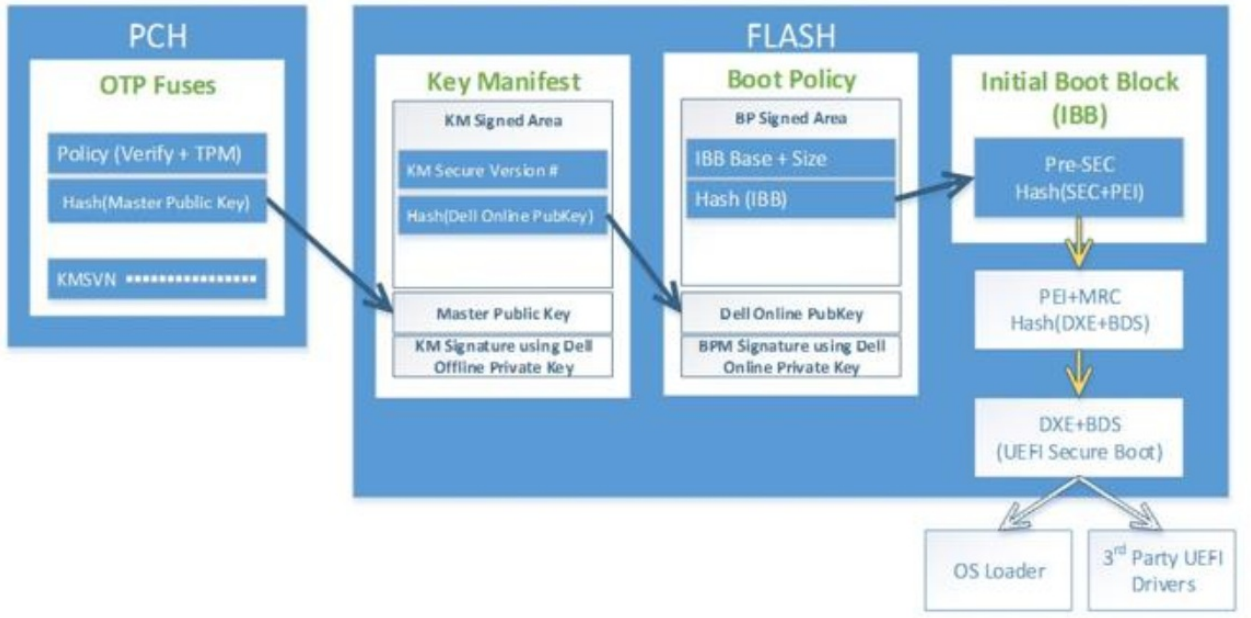



**Figure 2-4: Intel® Boot Guard diagram (credit: "CYBER-RESILIENCY IN CHIPSET AND BIOS" by Dell EMC)**

Table 2-4: Key Usage in Intel® Boot Guard

| Key | Verifies | Storage | Verified By |
|---|---|---|---|
| ACM Key | ACM | CPU | Microcode |
| Key Hash | Key Manifest | PCH | ACM |
| BP Key | Boot Policy Manifest | Key Manifest (Flash) | ACM |
| IBB Hash | IBB | Boot Policy Manifest (Flash) | ACM |

Please note that Intel Boot Guard is not the only solution available for OEM platform firmware verification. This document uses it as an example to illustrate the concept.

Table 2-5 shows how to reduce TCB from OEM platform firmware to ACM.

## ACM IBB Verification

Table 2-5: ACM IBB Verification

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | ACM IBB Verification | Intel | Originally on flash, loaded into Authenticated Code RAM (AC-RAM) |
| **CDI** | ACM Code | Intel | Originally on flash, loaded into AC-RAM |
| | Boot Policy Manifest (Policy) | OEM | Originally on flash, loaded into cache |
| | Key Manifest (Policy) | OEM | Originally on flash, loaded into cache |
| | Key Hash (Policy) | OEM | Write once PCH register, programmed in manufacture fuse. |
| **UDI** | Firmware Initial Boot Block, aka IBB | OEM | Originally on flash, loaded into cache |

Intel introduced the Intel® Boot Guard Authenticated Code Module (ACM), which is a module signed by Intel. The ACMs modules assume responsibility to verify OEM platform firmware before the host CPU transfers control to OEM firmware. Because verifying the entire image is time-consuming, the ACM only verifies the initial boot block (IBB) code. The IBB is then responsible for verifying the OEM boot block (OBB).

## Signing

The UDI here is the firmware IBB, so only the IBB needs to be signed.

## Public Key Storage

Intel® Boot Guard defines a set of Manifests to record the signature information.

1. Boot Policy Manifest – It records the hash of IBB and is signed by the Key Manifest Key.
2. Key Manifest – It records a set of hashes for the public key pair which signs the Boot Policy Manifest, and it is signed Boot Guard Key.
3. Key Hash - It records the hash for the public key pair which signs the Key Manifest. It is provisioned into the PCH hardware.

The Key Hash is read-only. It cannot be updated.

The Boot Policy Manifest and Key Manifest can be updated in the firmware.

## Verification

During runtime update, the TP – ACM IBB Verification gets the CDI - Key Hash from PCH - and verify the first UDI – the Key Manifest. If the verification passes, the Key Manifest is transformed into a CDI. Then ACM continues to get the key hash from the CDI - Key Manifest - and verify the UDI - Boot Policy Manifest. If the verification passes, the Boot Policy Manifest is transformed into a CDI. Then the ACM gets the final UDI – Firmware IBB code - and verify it according to the CDI – Boot Policy Manifest. If the final verification passes, then the Firmware IBB is transformed into a CDI, and the ACM transfers control to the IBB.

## Microcode ACM Verification

The ACM binary is signed by Intel. Now the question becomes who verifies the ACM binary. The answer is the CPU Microcode.

Table 2-6: Microcode ACM Verification

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Microcode ACM Verification | Intel | CPU |
| **CDI** | Microcode | Intel | CPU |
| | ACM Key hash | Intel | CPU |
| **UDI** | ACM Code | Intel | Originally on flash, loaded into AC-RAM |

## Signing

The UDI is the ACM binary. As such, the ACM needs to be signed with the Intel key.

## Public Key Storage

The hash of the ACM public key is inside of the CPU. A debug ACM is signed with the debug key. A production ACM is signed with the production key.

The policy can NOT be updated.

## Verification

During the ACM launch, the CPU Microcode loads the UDI - ACM to authenticated code execution area. Then the TP – ACM verification performs the verification. If the verification passes, then the UDI is transformed to CDI, the ACM starts executing. If the verification fails, the TXT shutdown is signaled.

The Intel® Boot Guard is one implementation to support boot ROM verification. Some other projects may have similar functions, such as Cerberus.

## OBB Verification

Intel® Boot Guard only verifies the initial boot block (IBB) of the whole OEM Firmware. To make sure the whole OEM Firmware is unmodified, the IBB needs to verify the reset OEM boot block (OBB).

Table 2-7: OBB Verification

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | OBB Verification | OEM | Originally on flash, loaded into DRAM |
| **CDI** | Firmware Initial Boot Block, aka IBB | OEM | Originally on flash, loaded into DRAM |
| | OBB Hash, OBB public key hash (Policy) | OEM | Originally on flash, loaded into DRAM |
| **UDI** | Firmware OEM Boot Block, aka OBB | OEM | Originally on flash, loaded into DRAM |

## Signing

The UDI is OBB, which is not verified by IBB. Since both IBB and OBB are provided by OEM, the OEM may define a separate specific format to sign the OBB.

## Public Key Storage

The OBB public key hash must be stored into the IBB region to make sure it is validated by ACM. As implementation choice, OEM may store the OBB hash directly to the IBB without using the public key.

## Verification

During Firmware boot, the TP is the OBB verification code inside of IBB. If the OBB passes the verification, the OBB is installed by IBB. If the OBB fails the verification, the OBB is skipped.

# Boot Chain – Putting it all together

Figure 2-5 shows a complete secure boot chain constructed using Intel® Boot Guard, OBB Verification, and UEFI Secure Boot.

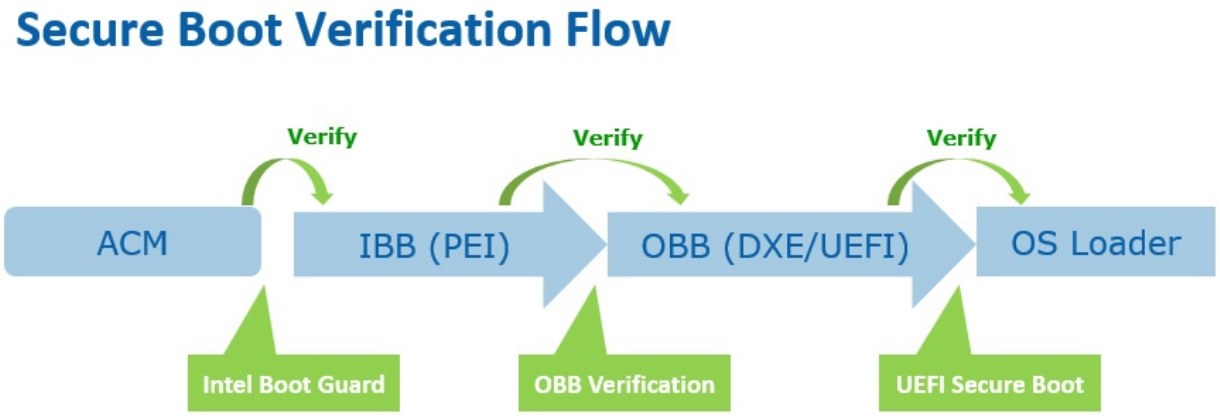



Figure 2-5: Secure Boot Verification Flow

## Signed Capsule Update

Platform firmware often requires an update. NIST provides multiple guidelines for authenticated updates (SP800-147, SP800-147B, SP800-193). EDK II implements authenticated updates based on Signed UEFI Capsule Updates and Capsule Recovery. Table 2-8 describes firmware update verification.

Table 2-8: Firmware Update Verification

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Firmware Update Verification | OEM | Originally on flash, loaded into flash unlockable environment. (It could be DRAM before the flash is locked, or SMRAM.) |
| **CDI** | Firmware Update TCB Code | OEM | Originally on flash, loaded into flash unlockable environment. |
| | Firmware Update Signature Database (Policy) | OEM | Originally on flash, loaded into flash unlockable environment. |
| **UDI** | Firmware Update Package | OEM | Originally on external storage (e.g. Hard drive, USB, Memory, or Read-Write Flash), loaded into flash unlockable environment. |

## Signing

The UDI is the whole new firmware image. As such, the whole firmware binary needs to be signed by the OEM private key.

## Public Key Storage

The OEM public key should be embedded in the original firmware. As such it can be used to verify the new firmware binary.

A policy may be updated along with the new Firmware image.

## Verification

During the firmware update process, TP is inside of the original firmware image. TP will load the new firmware image from external storage into memory. The memory can be normal DRAM (if the update happens before any 3rd party code is executed) or flash (in an unlocked state). If the update must occur after 3rd party code execution, the update must occur in an isolated execution environment (example: SMRAM). Care must be taken that both verification and update occur in the same environment, and there is no TOC-TOU threat (example: DMA attack). If TP passes verification, the new firmware image is programmed into flash. If verification fails, the flash update process is aborted.

## Intel® BIOS Guard

The implementation above assumes any code in the execution environment is secure. Reality shows that this is difficult to implement due to the number of drivers present in this environment. Intel provides the Intel® BIOS Guard solution which only allows the flash device to be programmed by the Intel® BIOS Guard AC module. This module performs firmware verification and updates in an Authenticated Code RAM (AC-RAM) environment. This is designed to prevent issues early in the firmware boot process or SMM from impacting the verification and update flow.

Figure 2-6 describes Intel® BIOS Guard components. Table 2-9 described firmware update verification using Intel® BIOS Guard.

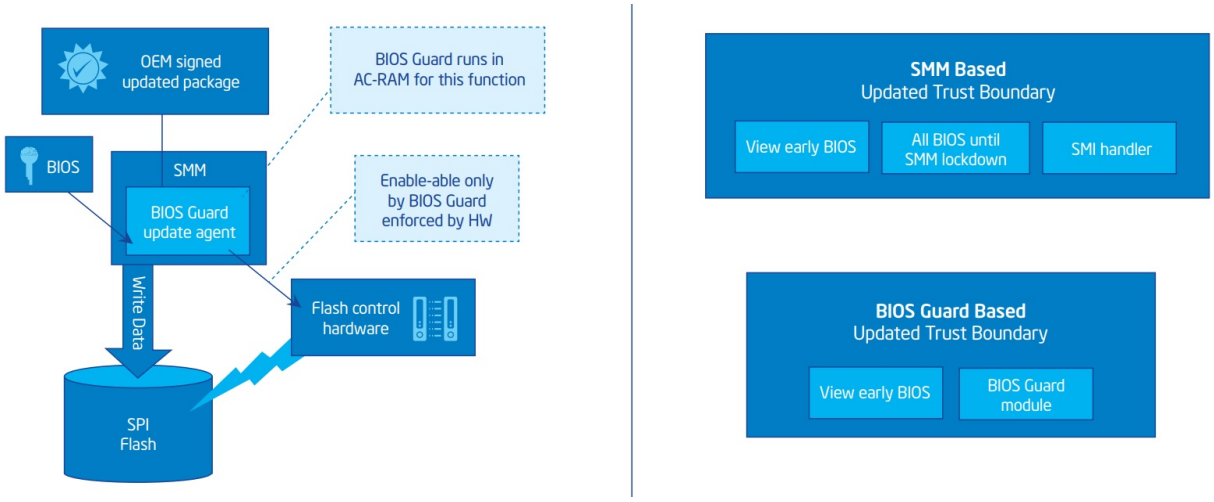



Figure 2-6: Intel® BIOS Guard

Table 2-9: Firmware Update Verification

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | ACM FU Verification | Intel | Original on the flash, loaded into AC-RAM |
| **CDI** | Intel® BIOS Guard ACM | Intel | Original on the flash, loaded into AC-RAM |
| | PubKey Hash (Policy) | OEM | Calculated during Firmware Boot early phase, and write to the CPU register. |
| **UDI** | Firmware Update Package | OEM | External Storage (e.g. Hard drive, USB, Memory, or Read-Write Flash), loaded into SMRAM. |

## Signing

The UDI is provided a new firmware image, the same as the UEFI Capsule Update implementation. The entire firmware binary must be signed using the OEM private key.

## Public Key Storage

The OEM public key should be embedded in the original firmware. During boot, the early BIOS needs to program the public key hash into the CPU BIOS Guard register. This is used by the BIOS Guard module during the verification. The policy may be updated along with the new BIOS image.

## Verification

During the firmware update process, a SMM module will load the firmware image and trigger the BIOS Guard module. TP is inside of the BIOS Guard module. TP first verifies if the OEM public key in the new firmware image matches the CPU BIOS Guard register, then verifies if the signature of the new firmware image. If TP passes verification, the BIOS Guard module writes the new firmware image into flash. If the verification fails, BIOS Guard returns with a failure.

# Signed Recovery

NIST SP800-193 defines three principles supporting platform resiliency:

- Protection,
- Detection
- Recovery

Signed UEFI capsule update and Intel® BIOS Guard provide these protections. Intel® Boot Guard and OBB verification provide detection. If firmware corruption is detected, the firmware can perform recovery to prevent a permanent denial of service (PDOS) attack. EDK II implements a signed recovery (see Table 2-10).

Table 2-10: Firmware Recovery Verification

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Firmware Recovery Verification | OEM | Originally on flash, loaded into DRAM. |
| **CDI** | Firmware Recovery TCB Code | OEM | Originally on flash, loaded into DRAM. |
| | Firmware Recovery Signature Database (Policy) | OEM | Originally on flash, loaded into DRAM. |
| **UDI** | Firmware Recovery Package | OEM | Originally on external storage (e.g. Hard drive, USB, Memory, or Flash), loaded into DRAM |

## Signing

The UDI is provided a new firmware image, the same as the UEFI Capsule Update implementation. The entire firmware binary must be signed using the OEM private key.

## Public Key Storage

The OEM public key should be embedded in the original firmware & recovery launcher module.

## Verification

If firmware corruption is detected during boot, the recovery boot path is triggered. In this scenario, TP is the firmware recovery launcher module. This module loads the recovery image from a known source and verifies the signature. If TP passes verification, the recovery image is loaded and the recovery launcher module transfers control to the recovery image. If recovery verification fails, the recovery image is discarded and the recovery launcher attempts to locate additional recovery images. If all recovery images fail verification, the recovery process is aborted.

NOTE: The signed recovery image itself may be updatable even if it is on the flash region.

# S3 Resume

S3 resume is a special boot path defined by the ACPI specification. During normal boot, firmware may save the system configuration and place the system in the S3 “sleep” state. During S3 resume phase, firmware loads the resume state to quickly “wakeup” the system and return to an operational state.

During S3 resume, firmware should not accept untrusted external inputs. The system configuration, referred to as the S3 Boot Script, should be stored in a secure place. EDK II implements a lockbox for the S3 resume state. This implementation provides no UDI for S3 resume, so all components should be treated as CDI.

# SMM Runtime Communication

System Management Mode (SMM) is a special highly privileged processor execution mode. One usage of SMM is that the Firmware may provide some special service in SMM, which is referred to as an SMI handler. The SMI handler uses a shared buffer (SMM Communication Buffer), to convey information to the service consumer during OS runtime. Table 2-11 describes SMM Runtime Communication Verification.

Table 2-11: SMM Runtime Communication Verification

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | SMM Communication Verifier Code | OEM | Originally on flash, loaded in SMRAM |
| **CDI** | SMI handler | OEM | Originally on flash, loaded in SMRAM |
| **UDI** | SMM communication buffer | Any | DRAM |

The SMM communication buffer is not signed because any program may use the buffer to invoke SMM services. SMM communication is treated as an attack surface, so the SMI handler must verify the contents of the SMM communication buffer. Since there is no signature, common verification is limited to prevent SMM attacks since it cannot verify the originator.

# ADDITIONAL SECURE BOOT CHAIN IMPLEMENTATIONS

Overview of Secure Boot in Other Areas including:

- Machine Owner Key (MOK)
- coreboot
- Android verified boot

# Machine Owner Key (MOK)

Multiple Linux distributions have implemented UEFI Secure Boot, but this creates problems deploying 3rd party modules and custom-built kernels alongside components signed by the UEFI certificate Authority (CA). The Machine Owner Key MOK concept can be used with a signed shim loader to enable key management at the user/sysadmin level.

Figure 3-1 and Table 3-1 provide an overview of MOK.

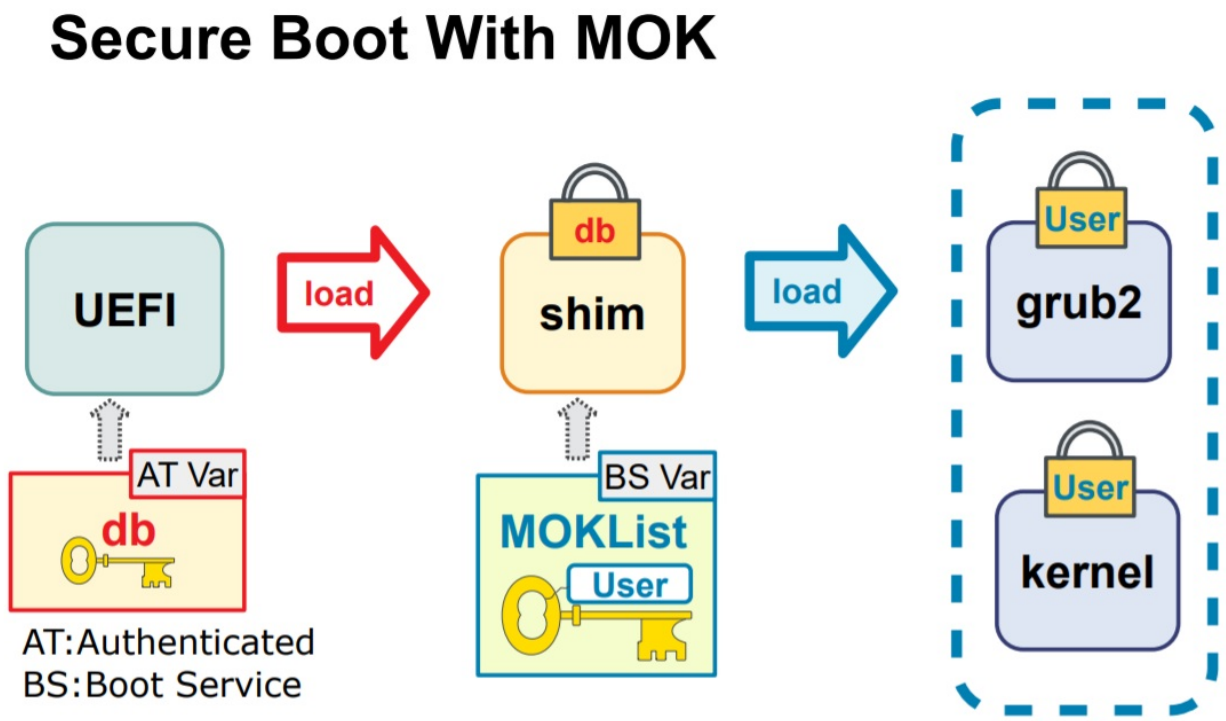



**Figure 3-1: Linux MOK Boot, (source: "UEFI Secure Boot Webinar")**

Table 3-1: Linux MOK Boot

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | OS Kernel Verification | OSV | External storage |
| **CDI** | Shim | OSV | External storage |
| | MOK list | User | Variable |
| **UDI** | OS Kernel | User | External storage |

# coreboot

The open source coreboot firmware project implements verified boot, which is similar to a combination of OBB verification and UEFI Secure Boot.

Figure 3-2 shows the verified boot flow. Table 3-2 shows keys used in the verified boot flow.

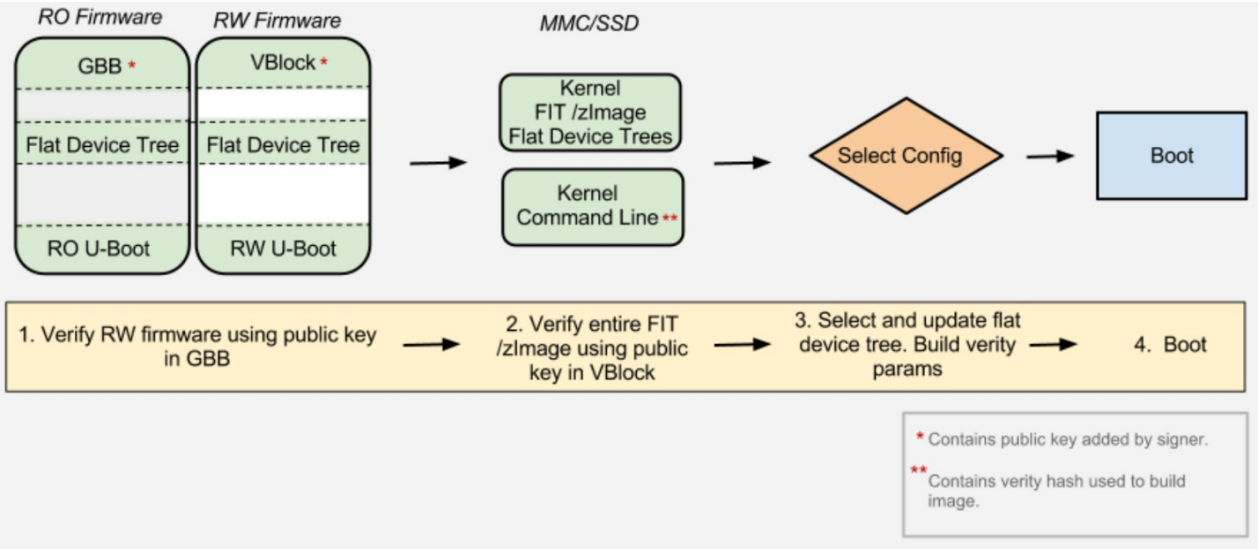



**Figure 3-2: coreboot Verified Boot (source: "Verified Boot in Chrome OS and how to make it work for you")**

Table 3-2: Keys used by coreboot verified boot (source: "Verified Boot: Surviving in the Internet of Insecure Things")

| Key | Verifies | Stored in | Versioned | Notes |
|---|---|---|---|---|
| Root Key | Firmware Data Key | RO Firmware | NO | Private key in a locked room guarded by laser sharks; N of M present. RSA4096+ |
| Firmware Data Key | RW Firmware | RW FW Header | YES | Private key on signing server. RSA4096. |
| Kernel Subkey | Kernel Data Key | RW Firmware | YES (as FW) | Private key only needed to sign new kernel data key. RSA4096. |
| Kernel Data Key | OS Kernel | OS kernel Header | YES | Private key on signing server. RSA2048. |
| Recovery Key | Recovery OS Kernel | RO Firmware | NO | Locked room and laser sharks. RSA4096+.<br>Different than all keys above. Signs recovery installer, not payload. |

Table 3-3: coreboot Verified Boot (for firmware)

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Read/Write Firmware Verification | OEM | Flash (Read Only Region) |
| **CDI** | Read-Only Firmware | OEM | Flash (Read Only Region) |
| | Root key | OEM | RO firmware, Google Binary Blob (GBB) |
| **UDI** | Read/Write Firmware | OEM | Flash (Read Write Region) |

Table 3-4: coreboot Verified Boot (for OS)

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | OS Kernel Verification | OEM | Flash (Read Write Region) |
| **CDI** | Read-Write Firmware | OEM | Flash (Read Write Region) |
| | Kernel Subkey | OSV | R/W firmware, Google Binary Blob (GBB) |
| **UDI** | OS Kernel | OSV | External storage |

# Android Verified Boot

The Android verified boot solution, like UEFI Secure Boot, is used to verify the integrity of an OS image.

*"Verified Boot strives to ensure all executed code comes from a trusted source (usually device OEMs), rather than from an attacker or corruption. It establishes a full chain of trust, starting from a hardware-protected root of trust to the bootloader, to the boot partition and other verified partitions including system, vendor, and optionally OEM partitions. During device boot up, each stage verifies the integrity and authenticity of the next stage before handing over execution."*

*-- "Verified Boot" (source.android.com)*

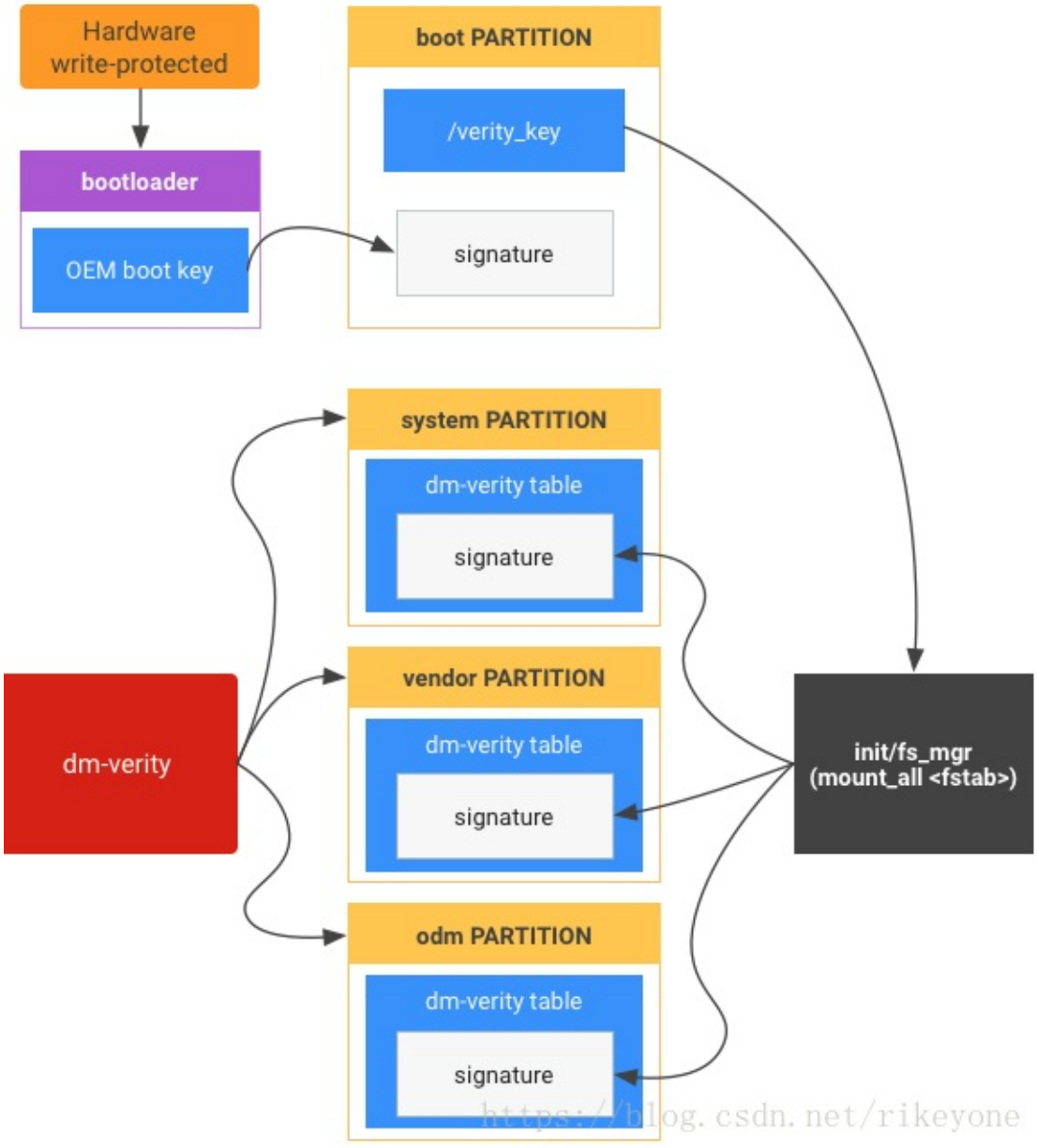


**Figure 3-3: Android Verified Boot 1.0 without A/B (source: Android Verified Boot 2.0)**

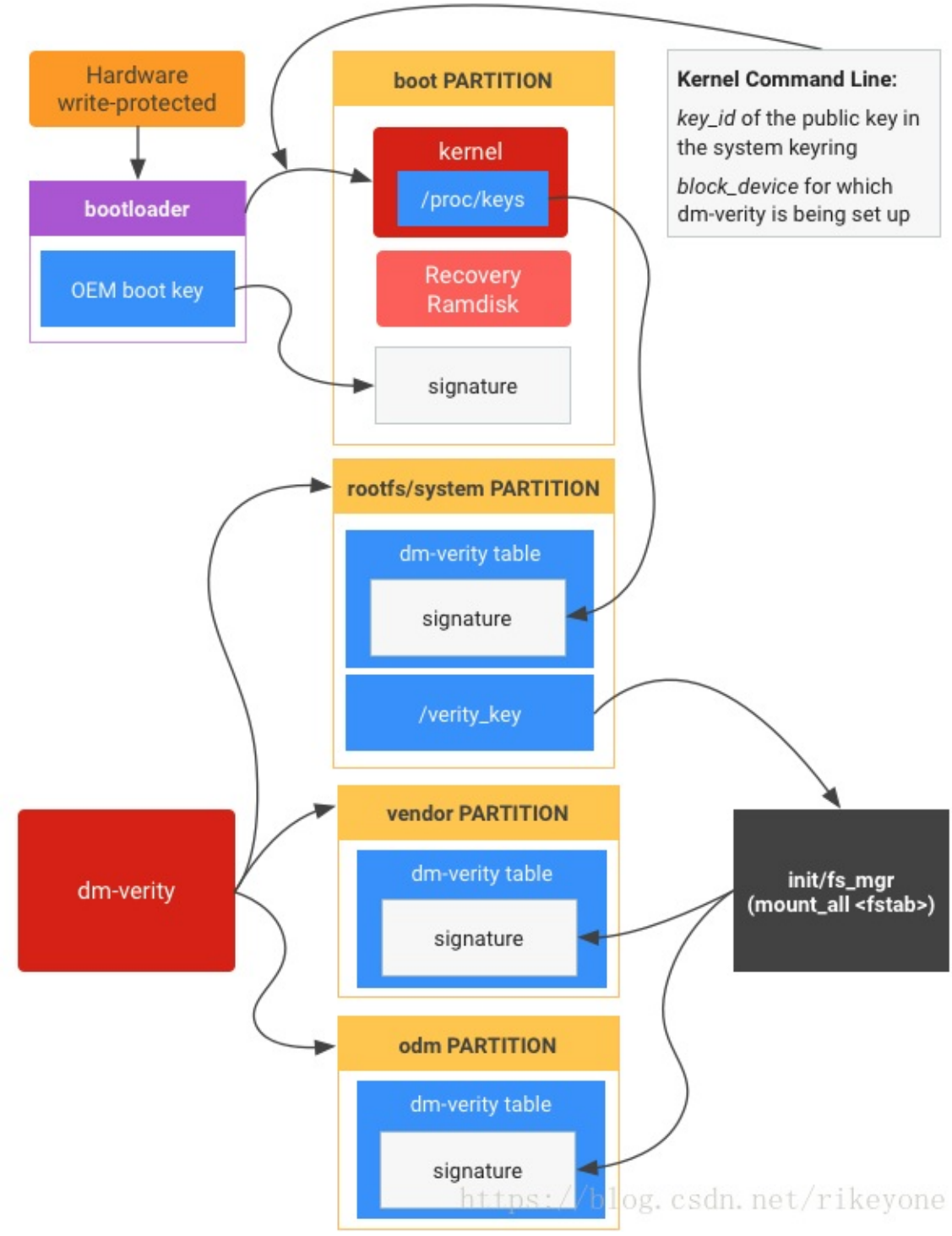



**Figure 3-4: Android Verified Boot 1.0 with A/B (source: Android Verified Boot 2.0)**

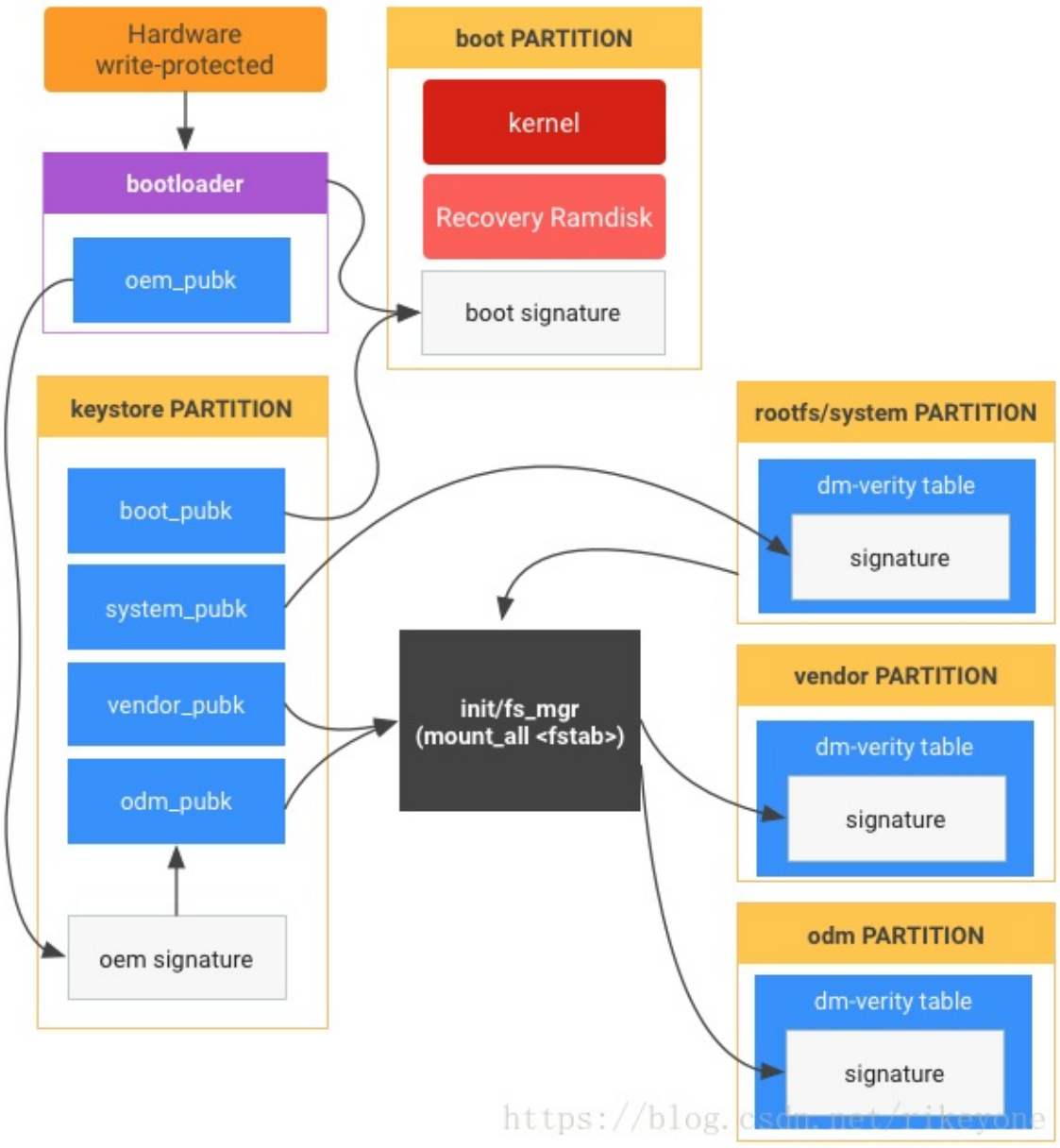



**Figure 3-5: Android Verified Boot 2.0 (source: Android Verified Boot 2.0)**

For additional information on OS kernel verification, see the following:

- https://source.android.com/security/verifiedboot
- https://android.googlesource.com/platform/external/avb/+/master/README.md
  https://blog.csdn.net/rikeyone/article/details/80606147

# *LOOKING FORWARD – PLATFORM FIRMWARE RESILIENCY*

NIST SP800-193 provides guidelines on using protection, detection, and recovery to implement platform firmware resiliency (PFR).

# Platform Firmware Resiliency

In modern platforms, system firmware is only one of multiple firmware images. Most system components rely on some form of device firmware. The scope of PFR covers both system firmware and device firmware images, so the trust chain is maintained for all boot firmware components. See Figure 4-1 for an overview diagram.

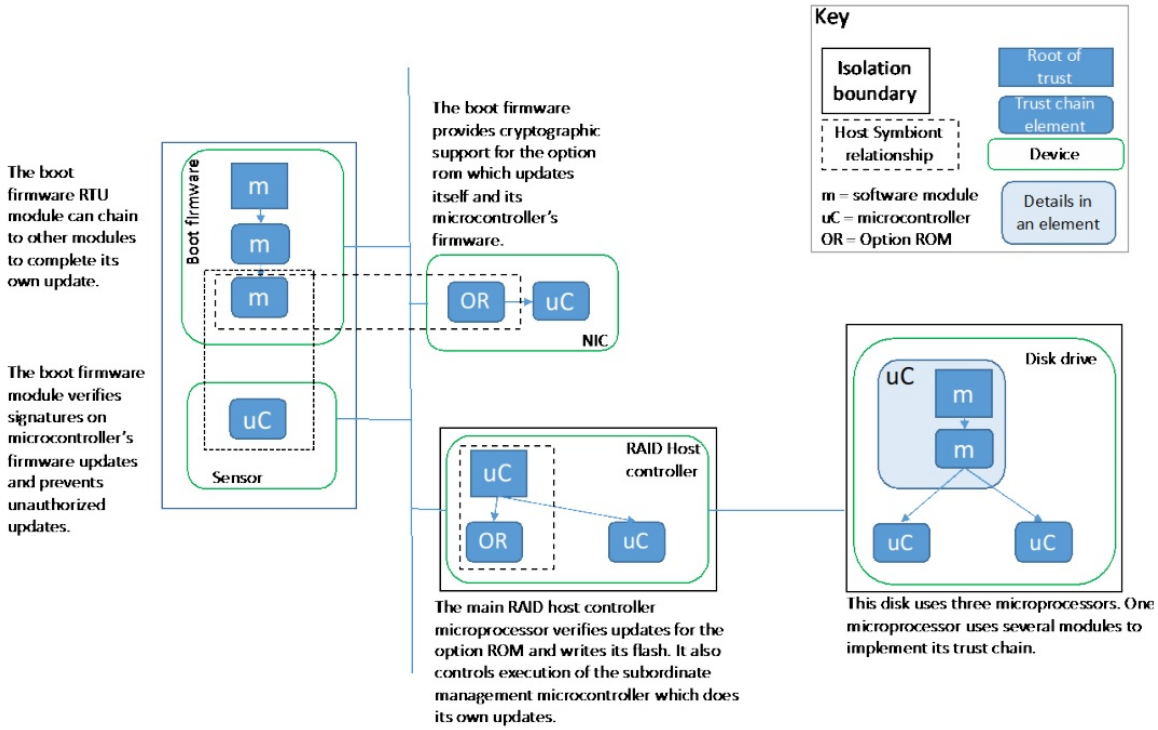



**Figure 4-1: Component and Trust Chain, from NIST SP800-193**

Device firmware may exist in a device-specific region that is managed by the device. In some cases, device firmware may reside in the same location as the system firmware, such as Serial Peripheral Interface (SPI) attached to flash, and the system firmware is responsible for loading the device firmware into a device firmware region.

Most device firmware initializes the hardware so it is functional at runtime. Examples include:

- Network Interface Card (NIC)
- Solid State Drive (SSD)
- Universal Serial Bus (USB)
- Battery management

Some device firmware is involved in the system boot process and may play an important role in system firmware verification. Examples include:

- Embedded Controller (EC) firmware

- Baseboard Management Controller (BMC) firmware
- Intel® Converged Security and Management Engine (Intel® CSME)
- Glue logic in a Field Programmable Gate Array (FPGA) or Complex Programmable Logic Device (CPLD)

There are multiple existing standards describing device authentication, including:

- PCIe Device Security
- USB Authentication
- Security Protocol and Data Model (SPDM)

Figure 4-2 shows a high-level view of an authentication protocol.

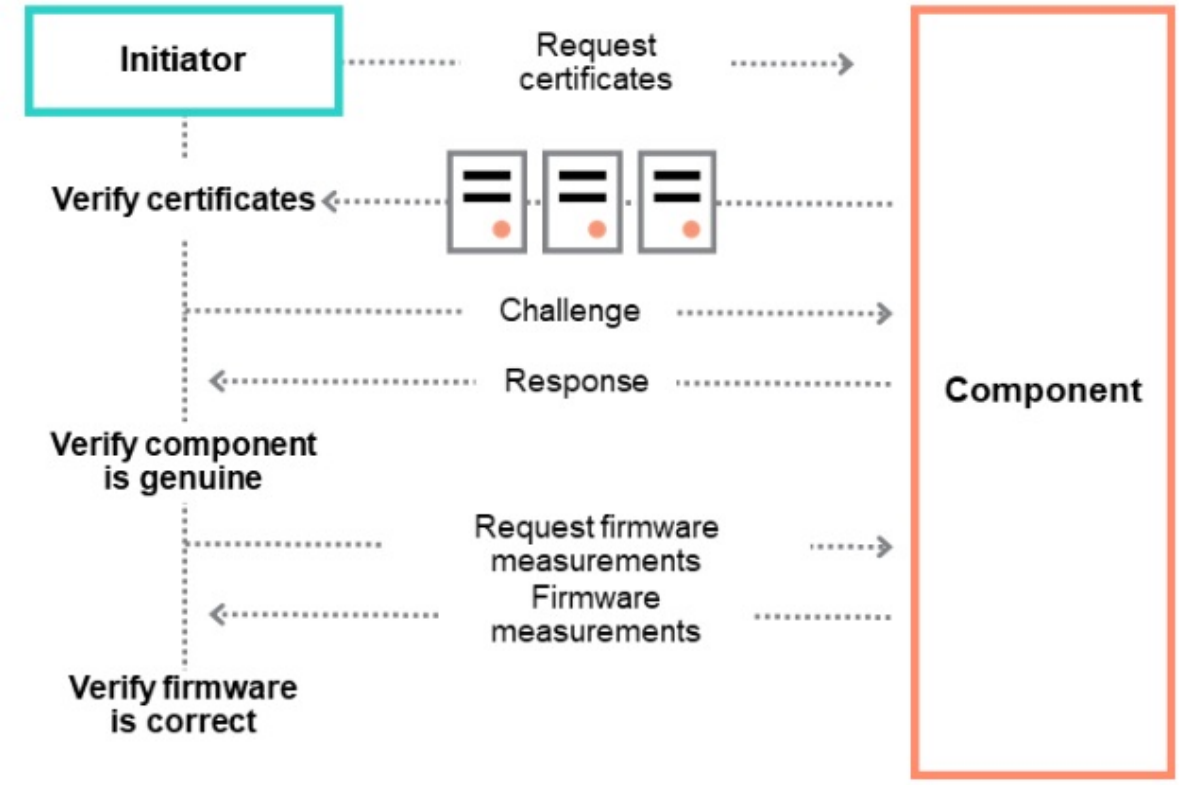



**Figure 4-2: High-level View of PCIe® Component Authentication (source: PCIe® Component Authentication)**

# Device Firmware Boot

If device firmware is not in TCB, it must be verified by the system firmware or device firmware in TCB.

During system boot, host firmware may choose to verify some device firmware components. For device firmware stored in the device's internal storage, verification may happen based upon device policy. For device firmware images in external storage loaded at runtime, verification is mandatory. Device firmware verification may follow the same rules as the system firmware verification. Device firmware is only loaded after it is verified.

Table 4-1: Device Firmware Boot Verification

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Device Firmware Verification | OEM or IHV | Flash (Read Only Code), Device ROM. |
| **CDI** | System Firmware or Device firmware TCB | OEM or IHV | Flash (Read Only Code), ROM |
| | Device Firmware Signature Database (Policy) | OEM or IHV | Flash (Read Only Data), ROM |
| **UDI** | Device Firmware | IHV | Device Internal Storage (or) External Storage (e.g. Hard drive, USB, Memory, or Read-Write Flash) |

# Device Firmware Update

If the device firmware is updatable, the update must be verified.

The verifier is determined by the entity with write access to the device firmware location. The entity performing verification must be the same entity performing the update.

For example, if the device firmware is in the device internal location, which is not accessible by the host firmware, such as TPM, then the device must do the verification and update. If the device firmware is in the device internal location, but it is accessible by the host firmware, such as EC, then the host firmware may do the verification and update. If device firmware is on the external storage and loaded by system firmware, then the system firmware must do the verification and update.

Table 4-2: Device Firmware Update Verification

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Firmware Update Verification | OEM or IHV | Depends |
| **CDI** | Firmware Update TCB Code | OEM or IHV | Depends |
| | Firmware Update Signature Database (Policy) | OEM or IHV | Depends |
| **UDI** | Device Firmware Update Package | IHV | Originally on external storage (e.g. Hard drive, USB, Memory, or Read-Write Flash), loaded into device firmware unlockable environment. |

# Project Cerberus

As part of the Open Compute Project (OCP), Project Cerberus defines a hierarchical Root of Trust (RoT) architecture. All active components are required to support both hardware and firmware combined identifing through the Device Identifier Composition Engine (DICE).

Figure 4-3 thru 4-6 describe the power on sequence, boot flow, recovery flow, and firmware update flow.

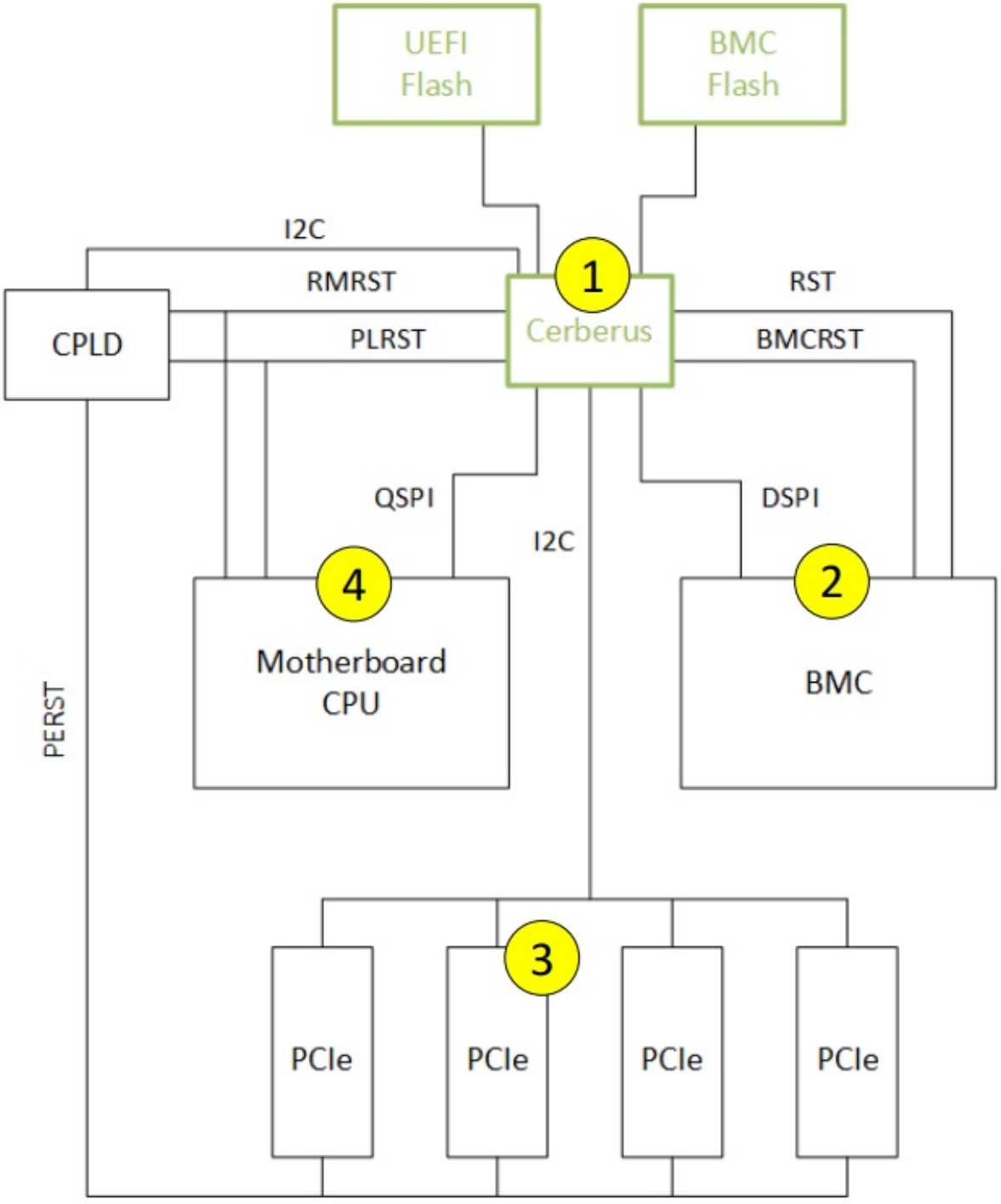
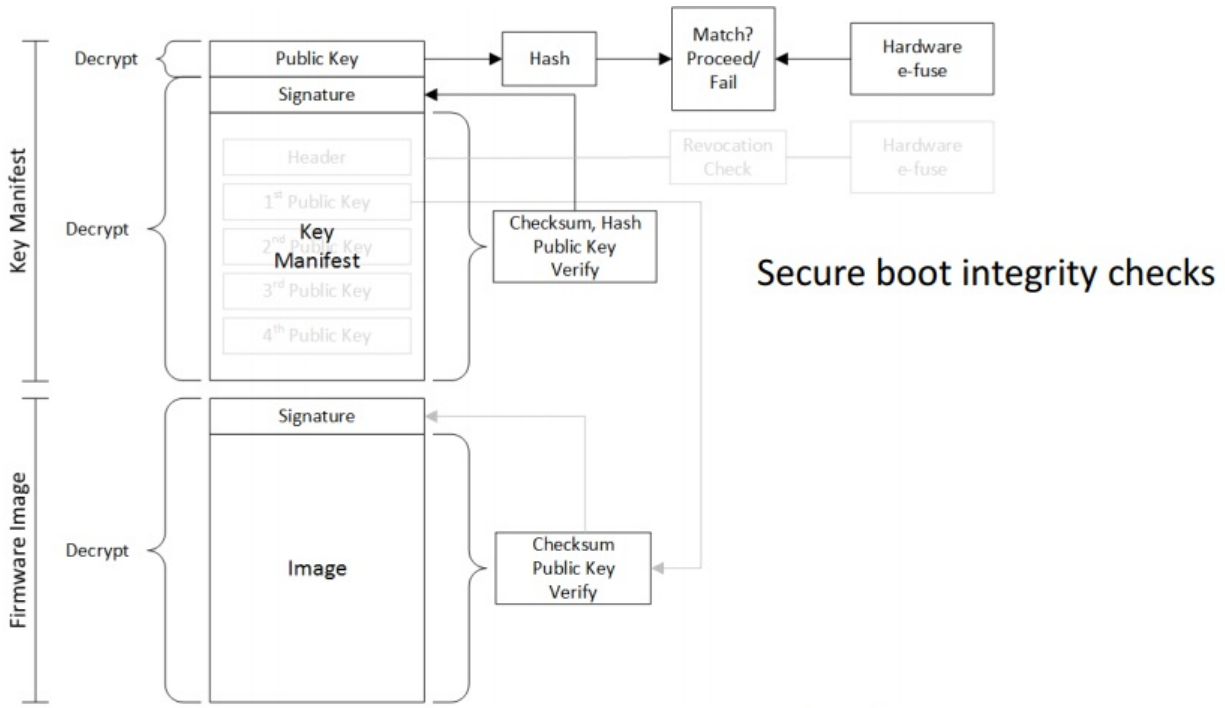


**Figure 4-3: Cerberus power on sequence (source: "Project Cerberus Hardware Security")**



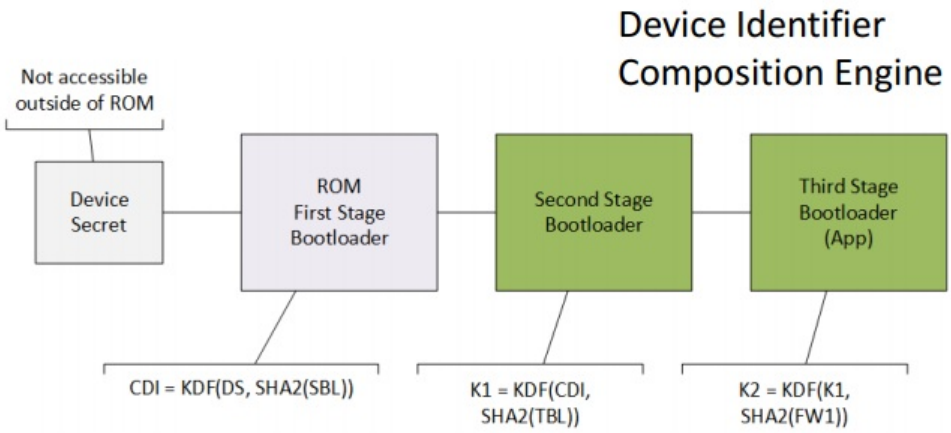




**Figure 4-4: Cerberus boot flow (source: "Project Cerberus Hardware Security")**

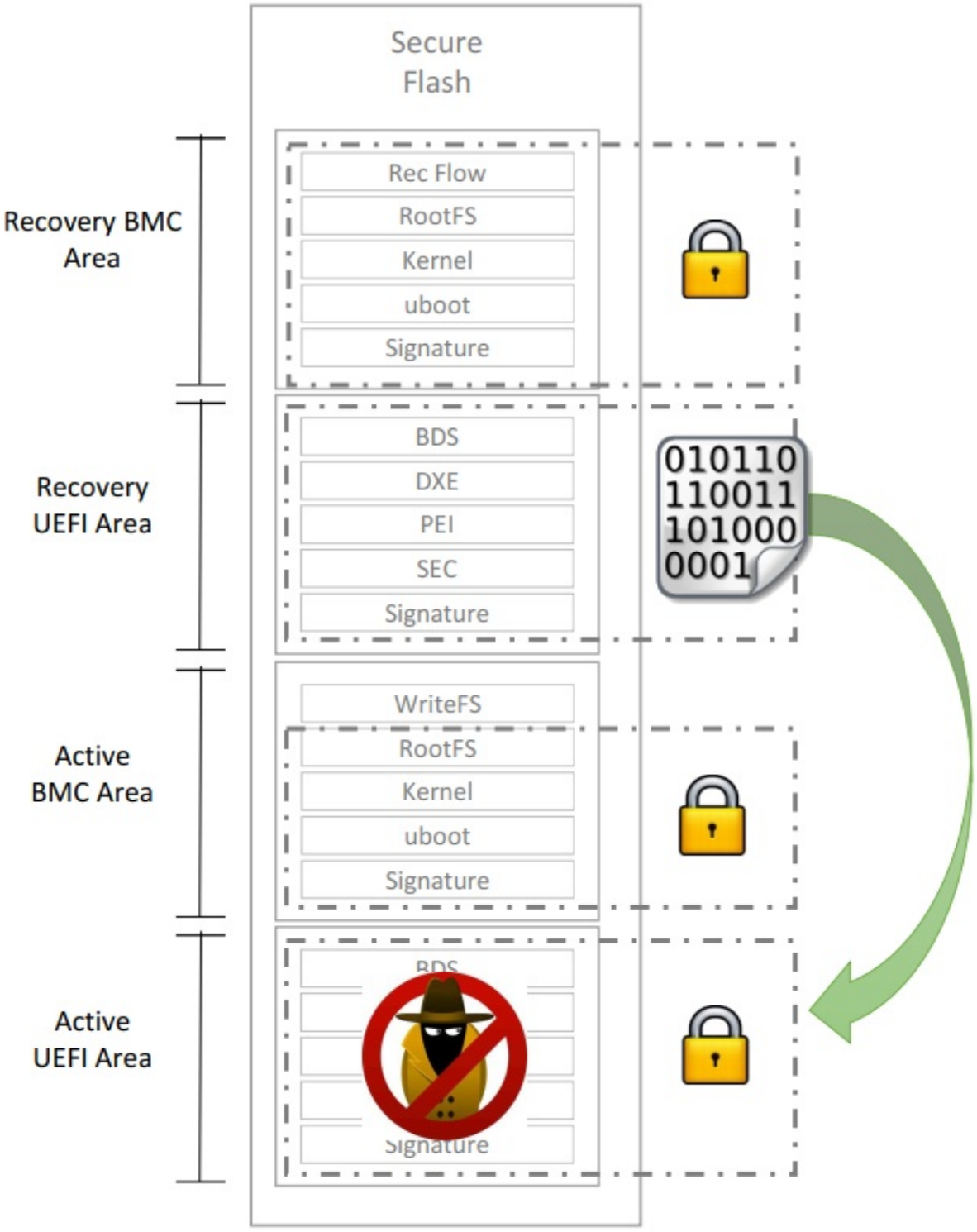
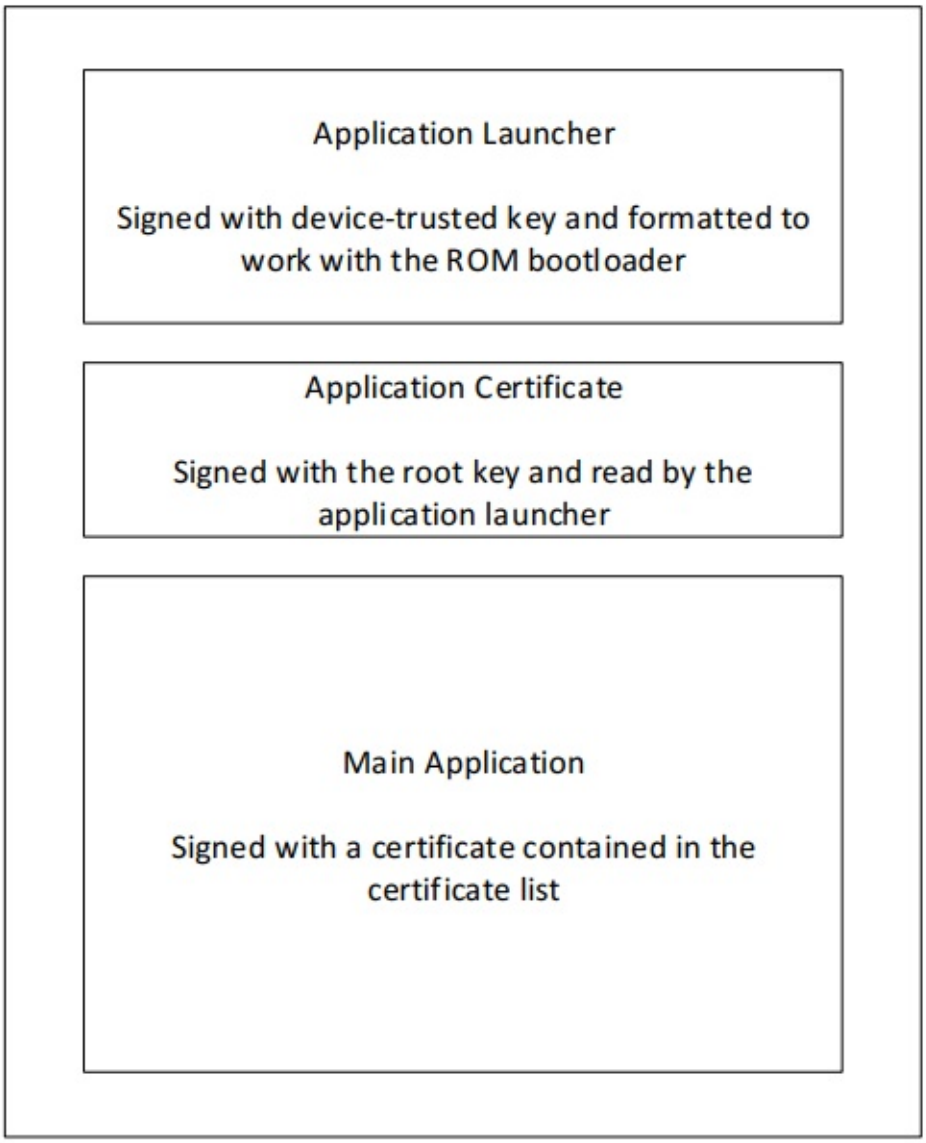
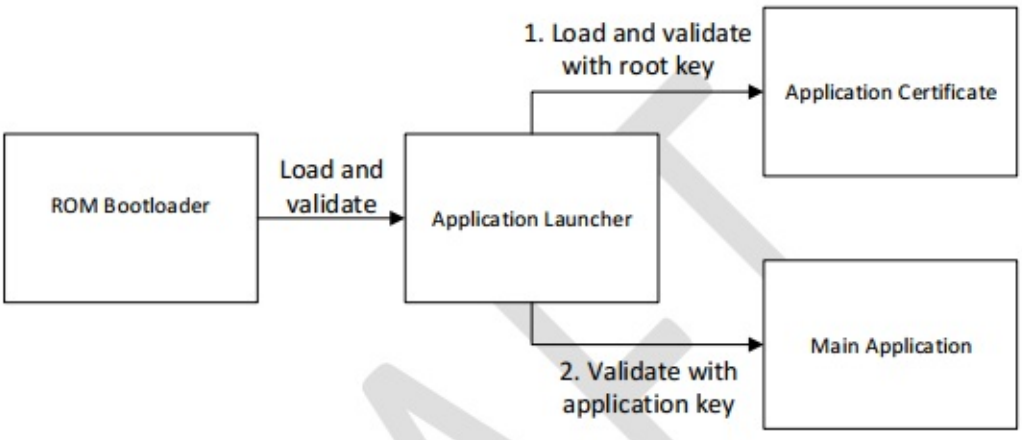



**Figure 4-5: Cerberus recovery flow (source: "Project Cerberus Hardware Security")**




1. Extract application launcher and

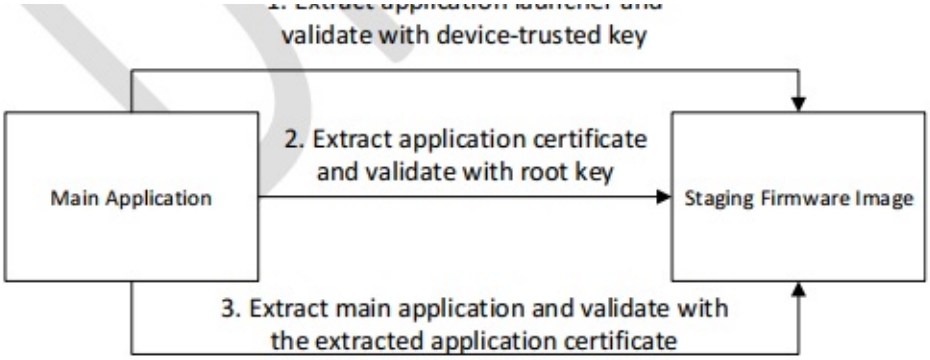



**Figure 4-6: Cerberus firmware update (source: "Project Cerberus Hardware Security")**

The concept of Cerberus is similar to Intel® Boot Guard., but there are several key differences:

1. Intel® Boot Guard uses Microcode as RoT, while Cerberus uses a dedicated RoT device.
2. Intel® Boot Guard can mitigate hardware bus attacks.
3. Intel® Boot Guard only verifies the host system firmware, while Cerberus verifies all boot firmware (platform firmware, BMC, etc.)
4. Cerberus defines a detailed flow for update and recovery.

Table 4-3: Cerberus Boot

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Boot Firmware Verification (in Cerberus Microcontroller) | OEM | Flash (Read Only Code), Device ROM. |
| **CDI** | Cerberus Microcontroller | OEM | Flash (Read Only Code), Device ROM. |
| | Boot Firmware Signature Database (Policy) | OEM | Flash (Read Only Data), ROM |
| **UDI** | Boot Firmware (BMC, Firmware) | OEM/IHV | Flash (Read Only Data) – active area |

Table 4-4: Cerberus Recovery

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Boot Firmware Verification (in Cerberus Microcontroller) | OEM | Flash (Read Only Code), Device ROM. |
| **CDI** | Cerberus Microcontroller | OEM | Flash (Read Only Code), Device ROM. |
| | Boot Firmware Signature Database (Policy) | OEM | Flash (Read Only Data), ROM |
| **UDI** | Boot Firmware Recovery (BMC, Firmware) | OEM/IHV | Flash (Read Only Data) - recovery area |

Table 4-5: Cerberus Firmware Update

| Item | Entity | Provider | Location |
|---|---|---|---|
| **TP** | Boot Firmware Verification (in Cerberus Microcontroller) | OEM | Flash (Read Only Code), Device ROM. |
| **CDI** | Cerberus Microcontroller | OEM | Flash (Read Only Code), Device ROM. |
| | Boot Firmware Signature Database (Policy) | OEM | Flash (Read Only Data), ROM |
| **UDI** | Boot Firmware (BMC, Firmware) | OEM/IHV | Flash (Read Only Data) – staging area |

# Intel® Platform Firmware Resilience (Intel® PFR)

To reduce firmware-related security risks, Intel developed Intel PFR for server platforms. This feature protects critical firmware from attacks during boot and runtime. It can be treated as an implementation of Project Cerberus or NIST SP800-193.

Intel PFR also enables a protect-in-transit feature, allowing customers to lock and unlock systems to guard against firmware changes during shipment. and “Intel transparent supply chain with platform certificate to create transparency in the supply chain to prevent counterfeit components from being used.”

Figure 4-7 shows the Intel PFR system diagram. Figure 4-8 shows the Intel PFR boot flow. Figure 4-9 shows the Intel PFR reset sequence.

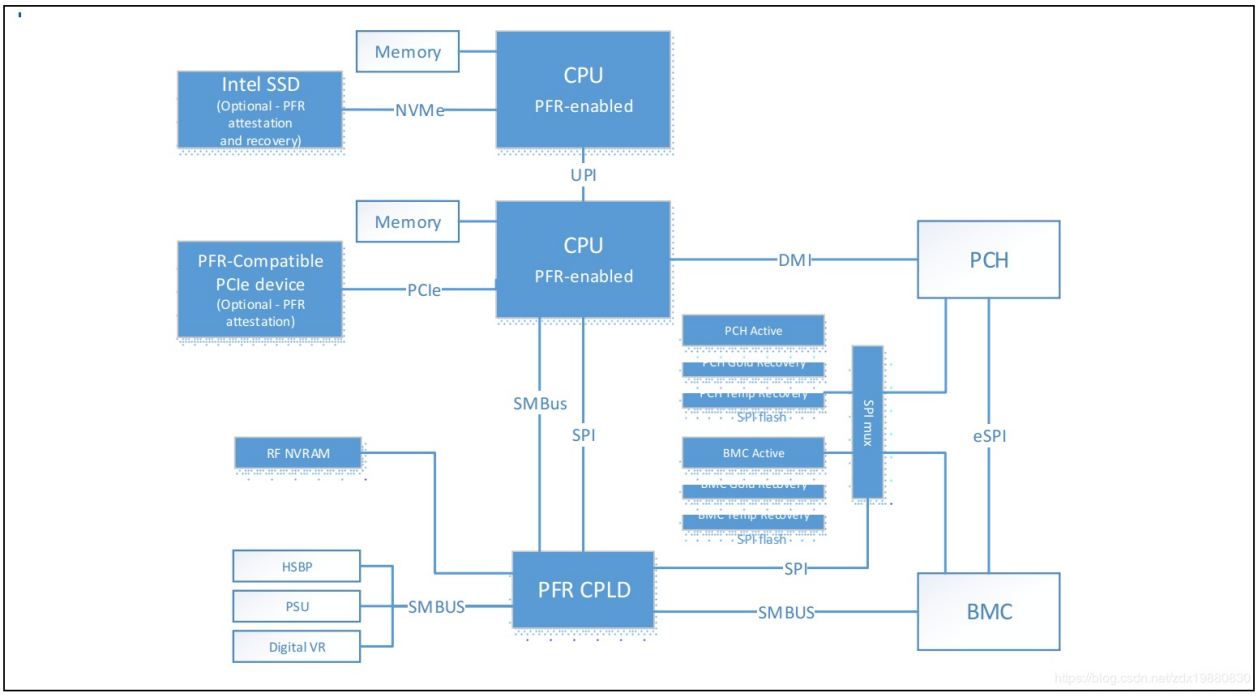



**Figure 4-7: Intel® PFR Overview (source: csdn.net)**

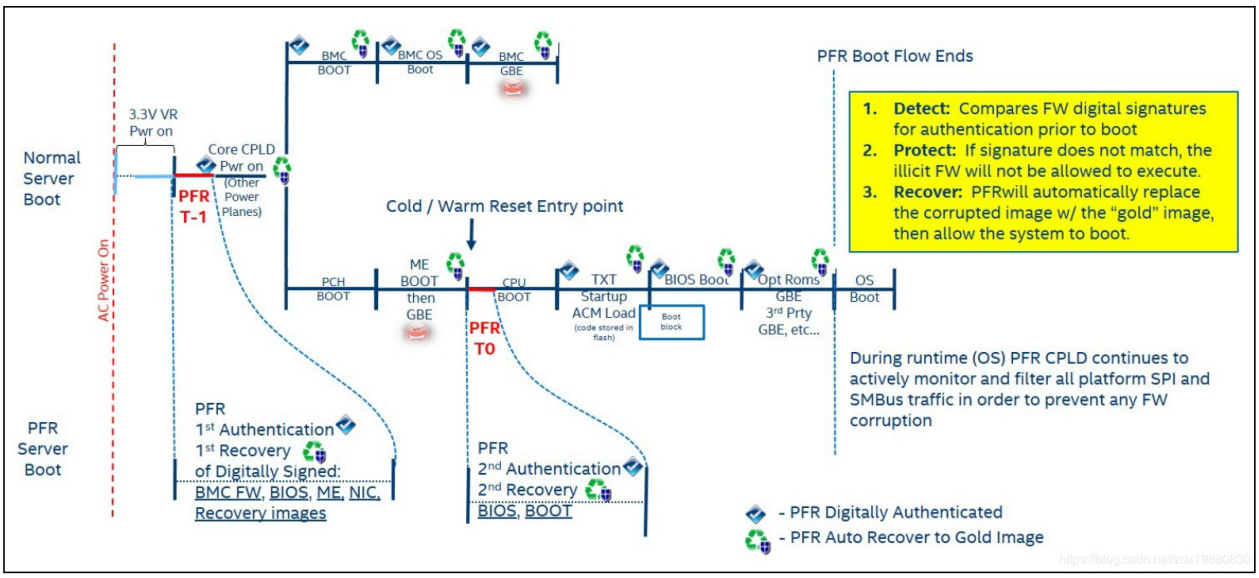



**Figure 4-8: Intel® PFR boot flow (source: csdn.net)**

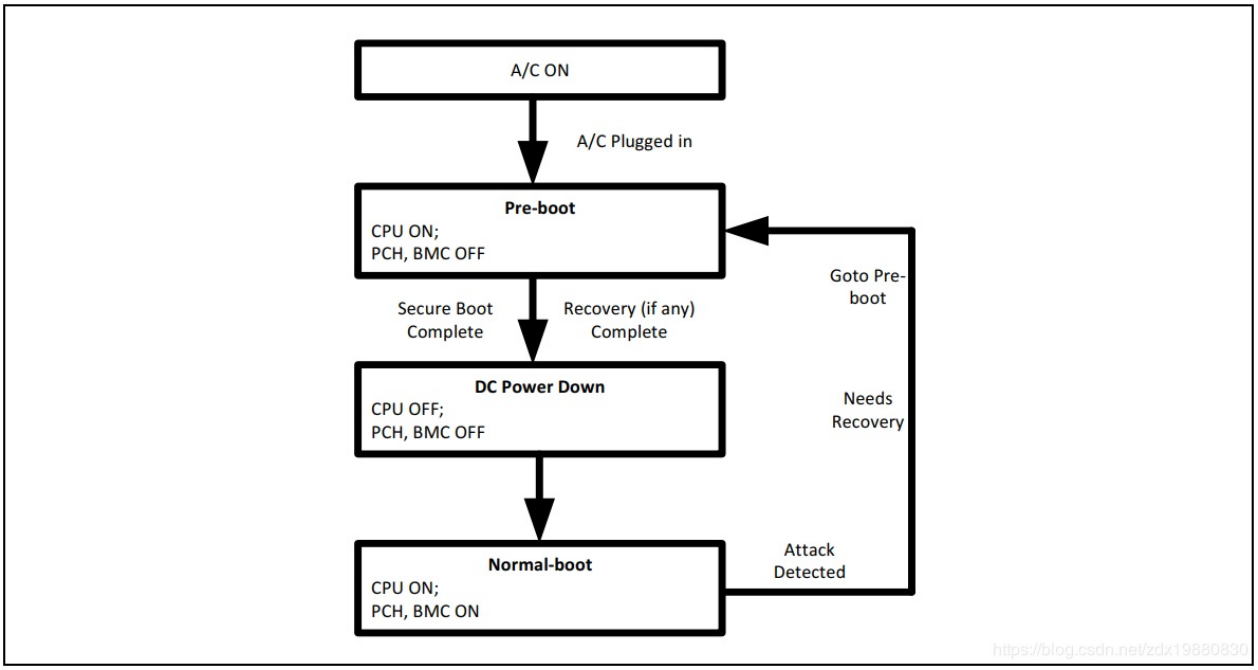



**Figure 4-9: Intel® PFR Reset Sequence (source: csdn.net)**

# Google Titan

Google developed Titan as a hardware root-of-trust solution for Google Cloud Platform (GCP). Aside from basic secure boot, Titan implements remediation and first-instruction integrity. These features are like functions found in Intel Boot Guard and Project Cerberus.

*"Trust can be re-established through remediation in the event that bugs in Titan firmware are found and patched, and first-instruction integrity allows the platform to identify the earliest code that runs on each machine's startup cycle."*

*-- "Titan in depth: Security in plaintext" (cloud.google.com)*

Figure 4-10 shows the Titan System Integration diagram. Figure 4-11 shows the Titan Verified Boot flow.

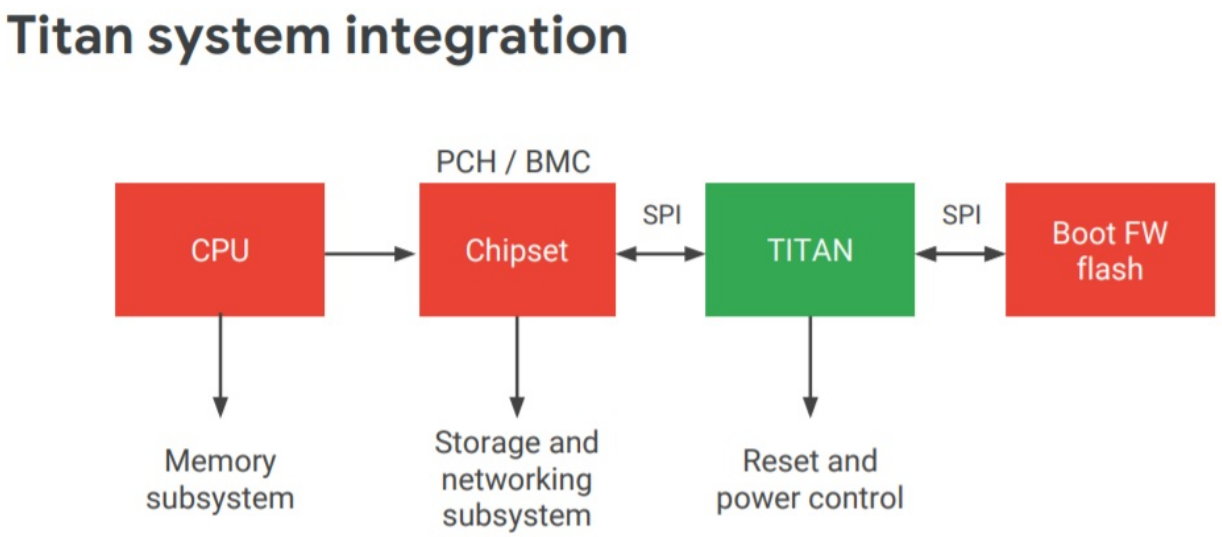
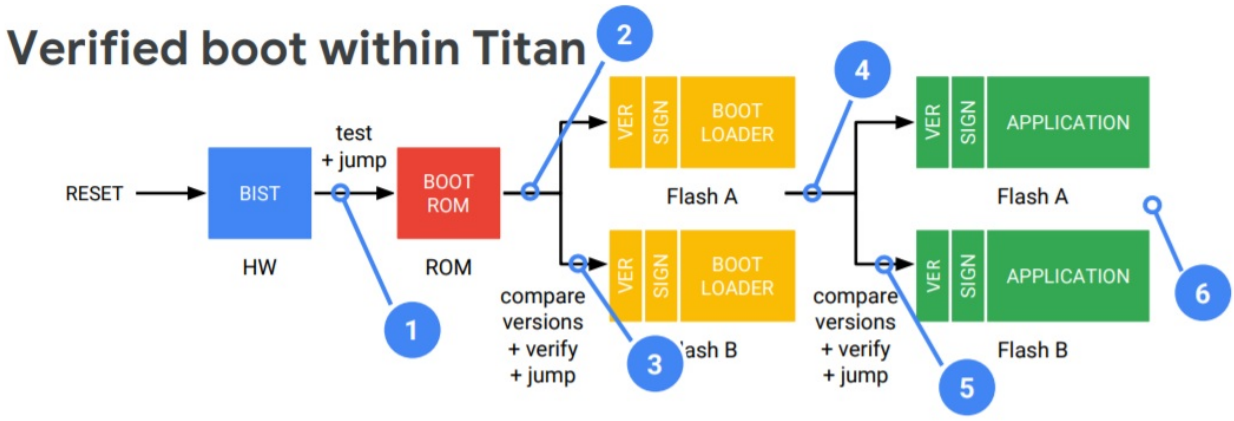



**Figure 4-10: Titan System Integration (source: "Titan silicon root of trust for Google Cloud")**




1. Test logic (LBIST) and ROM (MBIST); if fail ⇒ stay in reset; else jump to ROM
2. Compare bootloader (BL) versions A + B; choose most recent
3. Verify BL signature; if fail, retry with other BL; if fail, freeze
4. Compare firmware application (FW) versions A + B; choose most recent
5. Verify FW signature; if fail, retry with other FW; if fail, freeze
6. Execute successfully verified FW

**Figure 4-11: Titan Verified Boot(source: "Titan silicon root of trust for Google Cloud")**

# Other Platform Firmware Resiliency (PFR) Implementations

Additional PFR solutions are available for implementing NIST SP800-193, such as the Lattice Root of Trust FPGA solution (see Figure 4-12).

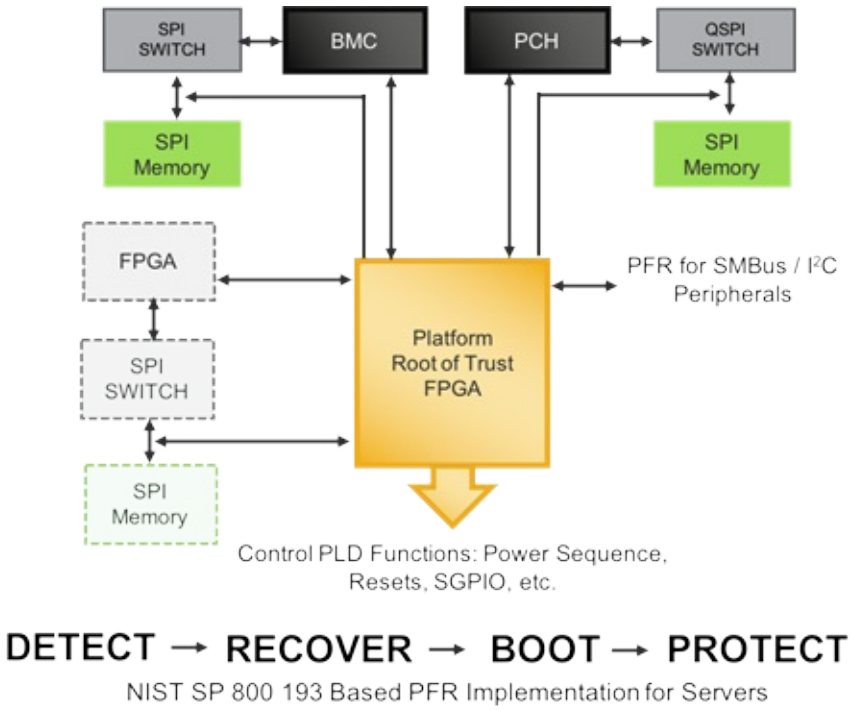



**Figure 4-12: Lattice PFR (source: latticesemi.com/pfr).**

The difference in implementations is hardware RoT device selection. Selections include processor microcode, CPLD devices, or FPGA devices. Each has its particular advantages and disadvantages. For example, processor microcode has a limited protection scope, which is why many customers use add-on devices for hardware RoT (CPLD, FPGA).

# GLOSSARY

ACM – Authenticated Code Module. See Intel® Boot Guard.

AC-RAM - RAM Authenticated Code RAM

IBB – Initial Boot Block. See Intel® Boot Guard.

OBB – OEM Boot Block. See Intel® Boot Guard.

CDI – Constrained Data Item. See Clark-Wilson.

UDI – Unconstrained Data Item. See Clark-Wilson.

TP – Transformation Procedure. See Clark-Wilson.

IVP – Integrity Verification Procedure. See Clark-Wilson.

CR – Certification Rule. See Clark-Wilson.

ER – Enforcement Rule. See Clark-Wilson.

OEM – Original Equipment Manufacturer

ODM – Original Design Manufacturer

IBV – Independent BIOS Vendor

IFV – Independent Firmware Vendor

IHV – Independent Hardware Vendor

ISV – Independent Silicon Vendor

OSV – Operating System Vendor

TCB – Trust Computing Base

RoT – Root of Trust

RTU – Root of Trust for Update

RTD – Root of Trust for Detection

RTRec – Root of Trust for Recovery

DICE - Device Identifier Composition Engine

PFR – Platform Firmware Resilience

MMIO – Memory Mapped I/O.

PI – Platform Initialization. Volume 1-5 of the UEFI PI specifications.

SMM – System Management Mode.

UEFI – Unified Extensible Firmware Interface. Firmware interface between the platform and the operating system. Defined by the UEFI Forum (uefi,org).

# REFERENCES

## Books and Papers

[Amoroso] Amoroso, Edward. Fundamentals of Computer Security Technology. Prentice Hall, 1994

[Bell–LaPadula] Looking Back at the Bell-La Padula Model, 2005, https://www.acsac.org/2005/papers/Bell.pdf

[Biba] Integrity Considerations for Secure Computer Systems, 1975, http://seclab.cs.ucdavis.edu/projects/history/papers/biba75.pdf

[Bishop] Bishop, Matt. Computer Security: Art and Science. Addison-Wesley Professional, 2018

[Blake] Blake, Sonya Q., “The Clark-Wilson Security Model”, SANS Institute Information, May 17, 2000. https://www.giac.org/paper/gsec/835/clark-wilson-security-model/101747

[Clark-Wilson] A Comparison of Commercial and Military Computer Security Policies, 1987, http://theory.stanford.edu/~ninghui/courses/Fall03/papers/clark_wilson.pdf

[CW-Lite] Toward Automated Information-Flow Integrity Verification for Security-Critical Applications, http://www.cse.psu.edu/~trj1/papers/ndss06.pdf, http://www.cse.psu.edu/~trj1/cse544-s10/slides/Info-Flow-NDSS-2006.pdf

[Lee] Prof. E. Stewart Lee, Director. “Essays about Computer Security.” Centre for Communications Systems Research, Cambridge, 1999. http://www.cl.cam.ac.uk/~mgk25/lee-essays.pdf

[NIST SP800-147] BIOS Protection Guidelines, 2011, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-147.pdf

[NIST SP800-147B] BIOS Protection Guidelines for Servers, 2014, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-147B.pdf

[NIST SP800-193] Platform Firmware Resiliency Guidelines, 2018, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-193.pdf

[Smith] Ned Smith, A Comparison of the trusted Computing Group Security Model with Clark-Wilson, 2014, https://www.semanticscholar.org/paper/A-Comparison-of-the-trusted-Computing-Group-Model-Smith/fa82426d99b86d1040f80b8bd8e0ac4f785b29a6

..

[Welch] Welch, Ian and Stroud, Robert. Supporting Real World Security Models in Java, 7th IEEE Workshop on Future Trends of Distributed Computing Systems, FTDCS'99. Cape Town, South Africa, December 20-22, 1999.

## Web

[AndroidVerifiedBoot] Android Verified Boot, https://source.android.com/security/verifiedboot

[AndroidVerifiedBoot2] Android Verified Boot 2.0, https://android.googlesource.com/platform/external/avb/+/master/README.md

[AndroidVerifiedBoot3] Android Verified Boot 2.0, https://blog.csdn.net/rikeyone/article/details/80606147

[BootGuard] Direct from Development – Cyber Resiliency In Chipset and BIOS, https://downloads.dell.com/solutions/servers-solution-resources/Direct%20from%20Development%20-%20Cyber-Resiliency%20In%20Chipset%20and%20BIOS.pdf

[CapsuleRecovery] Jiewen Yao, Vincent Zimmer, A Tour Beyond BIOS- Capsule Update and Recovery in EDK II, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Capsule_Update_and_Recovery_in_EDK_II.pdf

[Cerberus] Project Cerberus Architecture Overview, etc, https://github.com/opencomputeproject/Project_Olympus/blob/master/Project_Cerberus

[Cerberus2] Bryan Kelly, Project Cerberus Hardware Security, 2018,

https://f990335bdbb4aebc3131-b23f11c2c6da826ceb51b46551bfafdc.ssl.cf2.rackcdn.com/images/fbbdd5feceb6e6328373417e1ab7c06a13a2ef2c.pdf

[CorebootVerifiedBoot] vboot – Verified Boot Support, https://doc.coreboot.org/security/vboot/index.html?highlight=verified%20boot

[CorebootVerifiedBoot2] Simon Glass, Verified boot in Chrome OS and how to make it work for you, https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42038.pdf

[CorebootVerifiedBoot3] Randall Spangler, Verified boot surviving in the internet of insecure things, https://www.coreboot.org/images/c/ce/Verified*Boot*-_Surviving_in_the_Internet_of_Insecure_Things.pdf

[DICE] TCG DICE, https://trustedcomputinggroup.org/work-groups/dice-architectures/

[FirmwareSecurity] Dell Firmware Security, https://www.platformsecuritysummit.com/2018/speaker/johnson/PSEC2018-Dell-Firmware-Security-Justin-Johnson.pdf

[GoogleTitan] Titan in depth security in plaintext, https://cloud.google.com/blog/products/gcp/titan-in-depth-security-in-plaintext

[GoogleTitan2] Scott Johnson, https://keystone-enclave.org/workshop-website-2018/slides/Scott_Google_Titan.pdf

[HIRS] Host Integrity at Runtime and Startup https://github.com/nsacyber/hirs

[Intel Security] security-technologies-4th-gen-core, https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/security-technologies-4th-gen-core-retail-paper.pdf

[IntelPFR] PFR server blocks solution, https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/pfr-server-blocks-solution-brief.pdf

[IntelPFR2] Intel Platform Firmware Resilience, https://blog.csdn.net/zdx19880830/article/details/84190005

[LatticePFR] Universal Platform Firmware Resilience solution, http://www.latticesemi.com/zh-CN/Solutions/Solutions/SolutionsDetails02/PFR

[Linux MOK] Ubuntu Secure Boot, https://wiki.ubuntu.com/UEFI/SecureBoot

[Linux MOK2] Olaf Kirch, UEFI Secure Boot, https://www.suse.com/media/presentation/uefi_secure_boot_webinar.pdf

[PCIeAuth] PCIe* Component Authentication

https://pcisig.com/pcie%C2%AE-component-authentication

[PCIeSecurity] PCIe* Device Security Enhancements Specification, https://www.intel.com/content/www/us/en/io/pci-express/pcie-device-security-enhancements-spec.html

[S3Resume] Jiewen Yao, Vincent Zimmer, A Tour Beyond BIOS Implementing S3 Resume with EDK II, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Implementing_S3_resume_with_EDKII_V2.pdf

[SECURE1] Jacobs, Zimmer, "Open Platforms and the impacts of security technologies, initiatives, and deployment practices," Intel/Cisco whitepaper, December 2012, http://uefidk.intel.com/sites/default/files/resources/Platform_Security_Review_Intel_Cisco_W

hite_Paper.pdf

[SECURE2] Magnus Nystrom, Martin Nicholes, Vincent Zimmer, "UEFI Networking and Pre-OS Security," in Intel Technology Journal - UEFI Today: Boostrapping the Continuum, Volume 15, Issue 1, pp. 80-101, October 2011, ISBN 978-1-934053-43-0, ISSN 1535-864X

https://www.researchgate.net/publication/235258577_UEFI_Networking_and_Pre-OS_Security/file/9fcfd510b3ff7138f4.pdf

[SECURE3] Zimmer, Shiva Dasari (IBM), Sean Brogan (IBM), “Trusted Platforms: UEFI, PI, and TCG-based firmware,” Intel/IBM whitepaper, September 2009, http://www.cs.berkeley.edu/~kubitron/courses/cs194-24-S14/hand-outs/SF09_EFIS001_UEFI_PI_TCG_White_Paper.pdf

[SmmComm] Jiewen Yao, Vincent Zimmer, Star Zeng, A Tour Beyond BIOS Secure SMM Communication, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Secure_SMM_Communication.pdf

[SPDM] Security Protocol and Data Model Specification, https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_0.9.0a.pdf

[SPDMonMCTP] SPDM over MCTP Binding Specification, https://www.dmtf.org/sites/default/files/standards/documents/DSP0275_0.9.0a.pdf

[TXT] Intel TXT software development guide, https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.5

www.uefi.org

[UEFI Book] Zimmer, et al, “Beyond BIOS: Developing with the Unified Extensible Firmware Interface,” 2nd edition, Intel Press, January 2011

[UEFI Overview] Zimmer, Rothman, Hale, “UEFI: From Reset Vector to Operating System,” Chapter 3 of *Hardware-Dependent Software*, Springer, February 2009

[UEFI PI Specification] UEFI Platform Initialization (PI) Specifications, volumes 1-5, Version 1.3 www.uefi.org

[USBAuth] Universal Serial Bus Type-C™ Authentication Specification, https://www.usb.org/document-library/usb-authentication-specification-rev-10-ecn-and-errata-through-january-7-2019

[Variable] Jiewen Yao, Vincent Zimmer, Star Zeng, A Tour Beyond BIOS Implementing UEFI Authenticated Variables in SMM with EDK II, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Implementing_UEFI_Authenticated_Variables_in_SMM_with_EDKII_V2.pdf

# Figures