Linaro connect
San Francisco 2017

Secure Boot on Arm systems

Matteo Carlini (Arm)

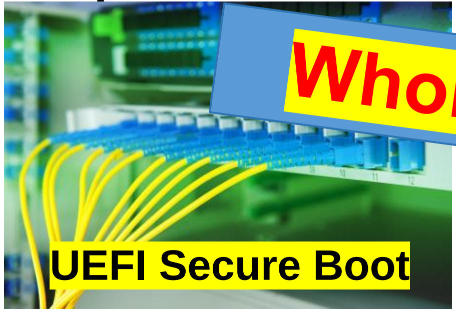
Agenda

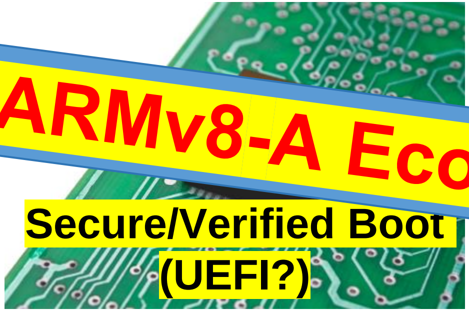
- Introduction & Scope of work
- Arm Trusted Board Boot (PKI, CoT, Authentication Flow)
- Arm Trusted Firmware implementation
- UEFI Secure Boot (PKI, CoT, Authentication Flow)
- UEFI Secure Boot on Arm – EDK2 recap
- Complete CoT
- Secure Variable Storage
- Other OSS Solutions (Android, U-Boot)
- Next steps

Introduction

- Secure Boot → a mechanism to build (and maintain!) a complete Chain of Trust on all the software layers executed in a system, preventing malicious code to be stored and loaded in place of the authenticated one
- Security through existing specifications, industry standards & OSS
  - Interoperability (same OS/Software on different Platforms/Firmware)
  - Common Secure Boot and Secure Firmware Update Interfaces → Reduced integration effort
  - Stability, frequent updates, wide usage → Reduced maintenance cost

Enterprise/Networking



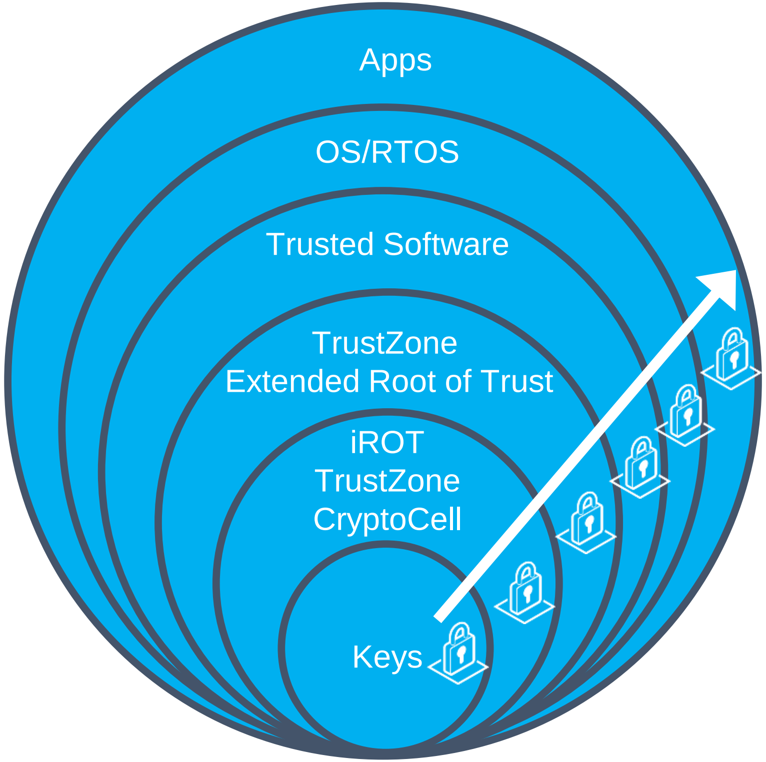


Mobile/Client

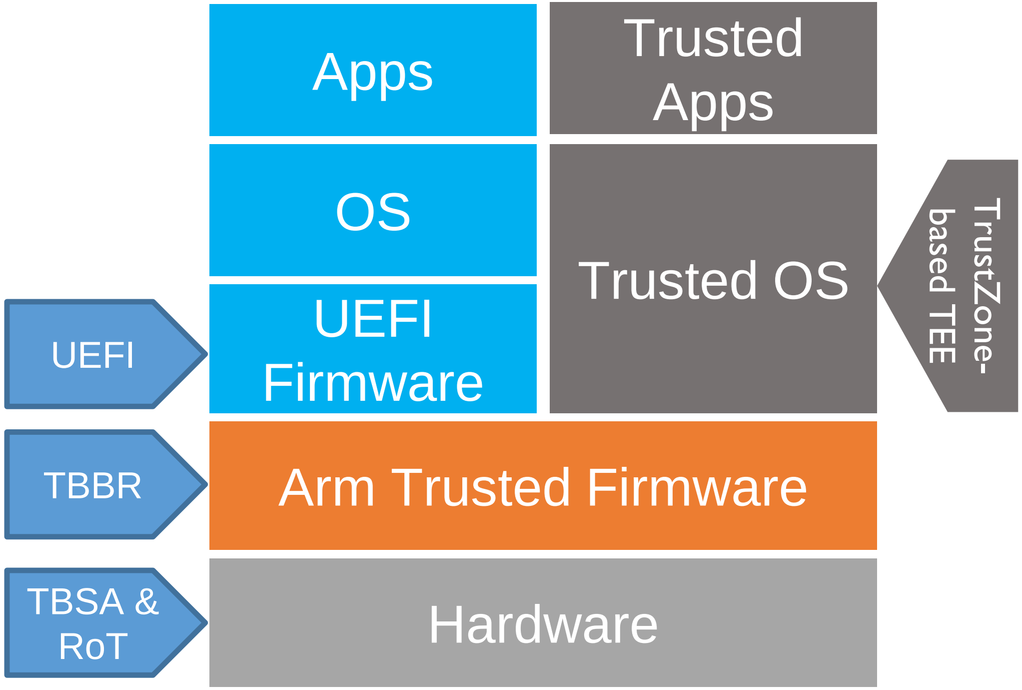
Whole ARMv8-A Ecosystem

Embedded

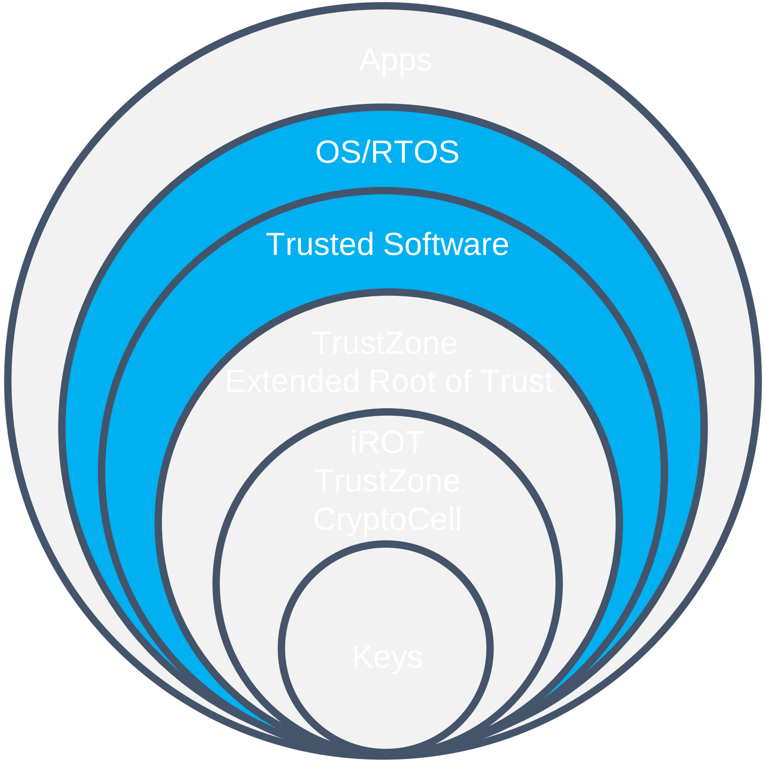
Scope of Work

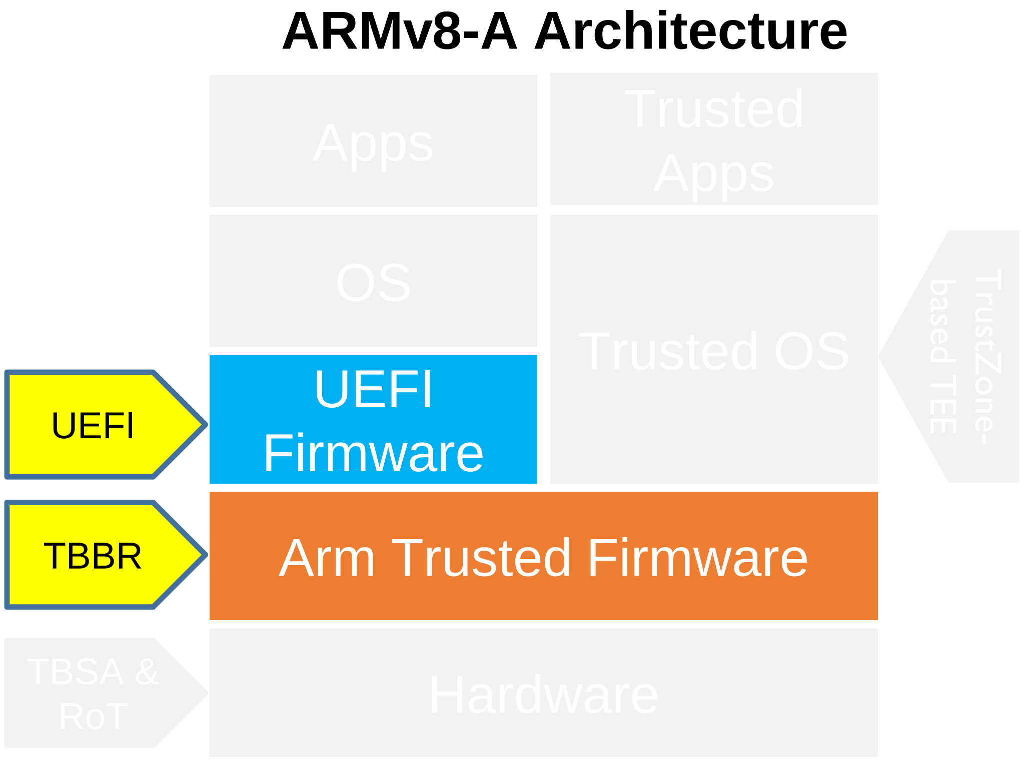


ARMv8-A Architecture



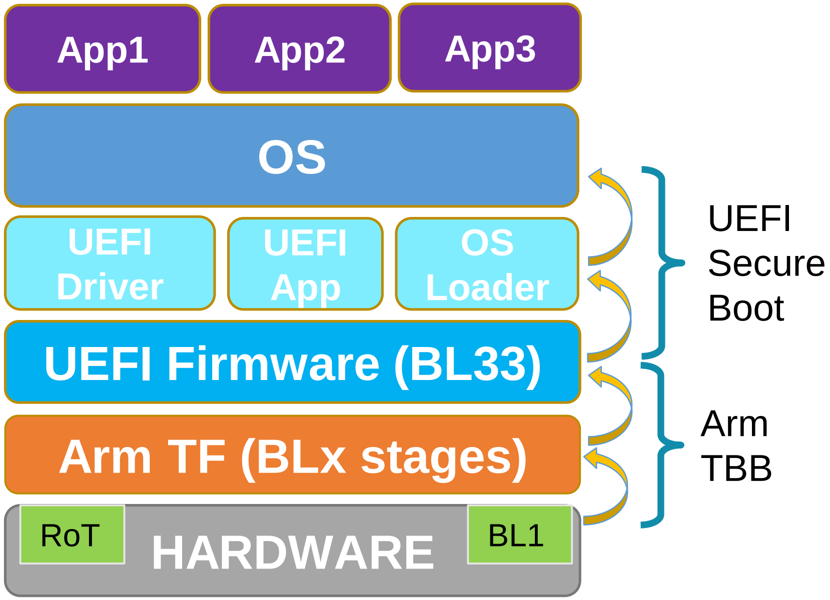
Scope of Work





Arm Trusted Board Boot vs UEFI Secure Boot

- TWO DISTINCT MECHANISMS : different Key/Certificates & PKI
- SAME GOAL : verifying the authenticity and integrity of a software/firmware image before allowing its runtime execution
- DIFFERENT TARGET IMAGES
- **Combined together they enable a full Secure Boot establishing a complete Chain Of Trust (despite different PKI) from the very first firmware executed up to the OS**



Arm Trusted Board Boot

- Based on Arm TBSA/TBRR documents (available under NDA)
  - TBBR-Client specification (DEN0006C) reference for Arm Trusted Firmware implementation

- Arm TBB: a reference example on how to build a CoT from the very first ROM firmware executed (BL1) up to the first normal world firmware (BL33)

- **SBBR recent implications (ARMServerAC):**
  - **v1.1 will generically mandate the use of a "*complete cascading Chain of Trust from the initial firmware up to the first normal world firmware*"**
  - Arm TBB and Arm Trusted Firmware provide a reference implementation
  - Other 3rd party solutions (BL1/BL2) will also be accepted as long as they start from an HW RoT and allow a complete verification up to the UEFI compliant firmware (BL33)

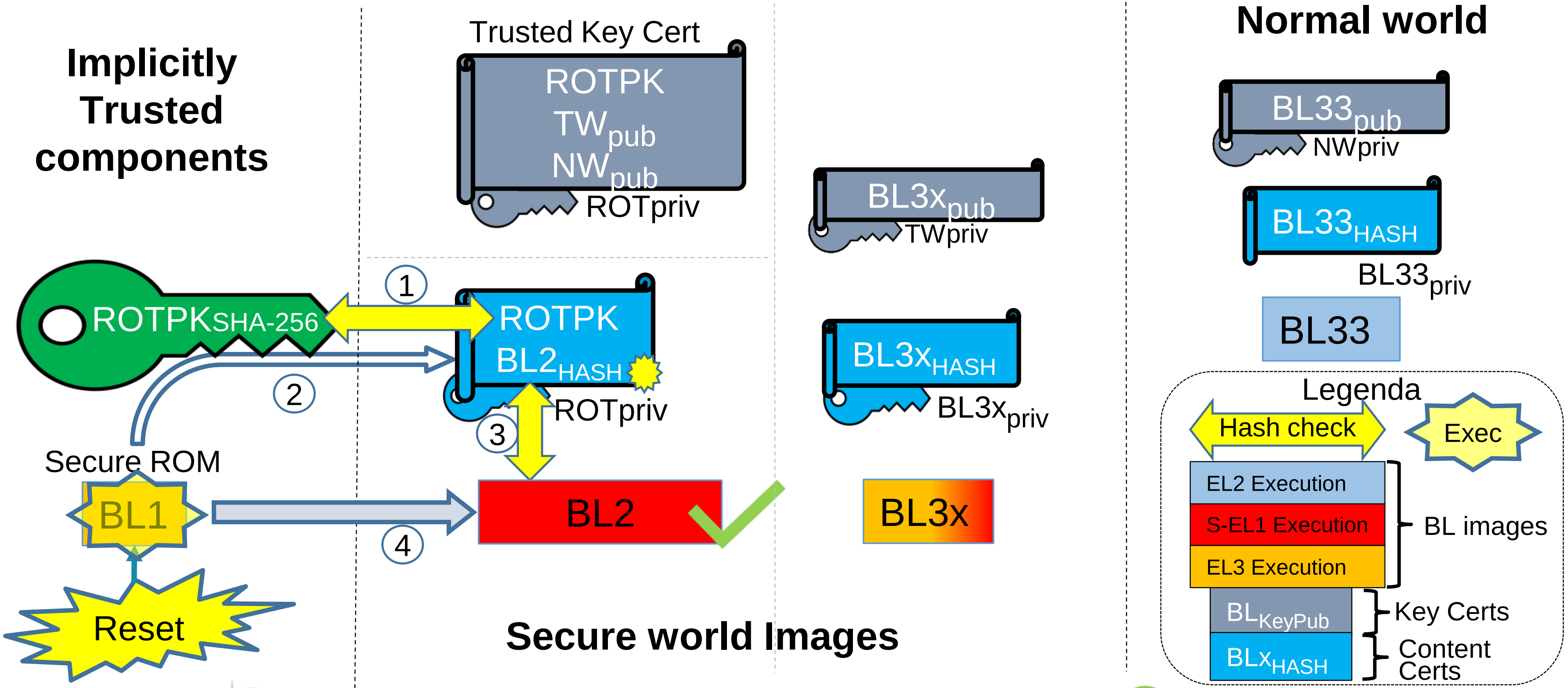
Arm Trusted Firmware TBB – PKI Details

- 2 implicitly trusted components (tamper proof)
  1. Root Of Trust Public Key (ROTPK) with SHA-256 hash stored on trusted registers
  2. Boot Loader Stage 1 (BL1) stored on trusted ROM
- 2 Certificates pairs for each BL3x image
  1. Key Certificate
     - Holds the $BL3x_{pub}$ key needed to validate the corresponding Content Certificate
  2. Content Certificate
     - Holds the BL3x image hash to be verified against the hash of the loaded image
- 2 Key pairs used to sign/validate Key Certificates
  1. Trusted World Key pair ($TW_{pub/priv}$) used for BL31 & BL32 Key Certificates
  2. Normal World Key pair ($NW_{pub/priv}$) used for BL33 Key Certificate
- Public Keys and hashes are included as extensions to X.509 certificates
- Certificates are self-signed: no need for a valid CA

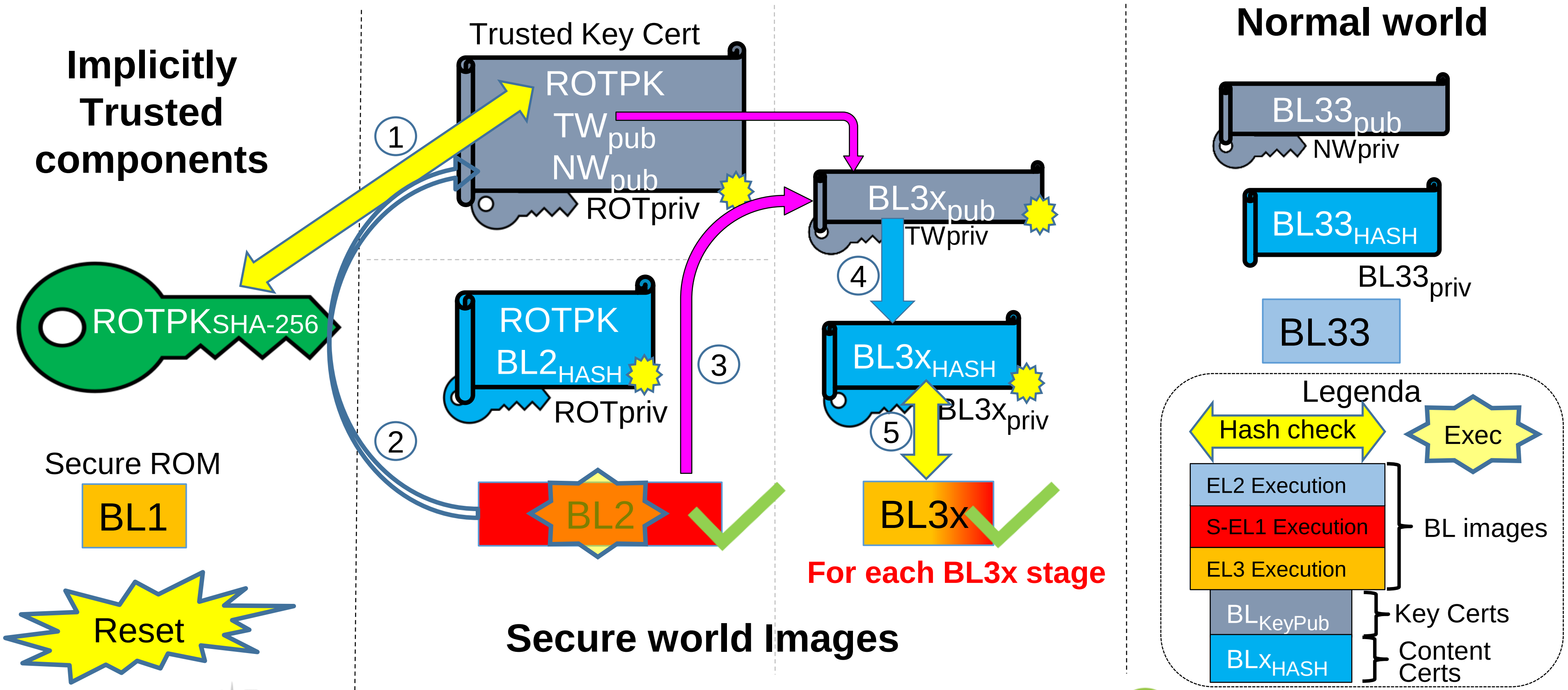
Arm Trusted Firmware TBB – Authentication Flow

- BL1 responsible for authenticating BL2 stage
  1. BL1 verifies ROTPK in BL2 Content Certificate against ROTPK stored hash
  2. BL1 verifies BL2 Content Certificate using enclosed ROTPK
  3. BL1 loads BL2 and performs its hash verification
  4. Execution is transferred to BL2
- BL2 responsible for authenticating BL3x stages (BL31, BL32, BL33)
  1. BL2 verifies ROTPK in Trusted Key Certificate against ROTPK stored hash
  2. BL2 verifies Trusted Key Certificate using enclosed ROTPK and saves TW_{pub} /NTW_{pub}
  3. BL2 verifies BL3x (BL31/BL32) Key Certificate using TW_{pub}
  4. BL2 verifies BL3x (BL31/BL32) Content certificate using enclosed $BL3x_{pub}$ key
  5. BL2 extracts and saves BL3x hash used for BL3x (BL31/BL32) image verification
  6. BL2 verifies BL33 Key Certificate using NTW_{pub}
  7. BL2 verifies BL33 Content certificate using enclosed $BL33_{pub}$ key
  8. BL2 extracts and saves BL33 hash used for BL33 image verification
  - Execution is transferred to verified BL3x → BL33 images

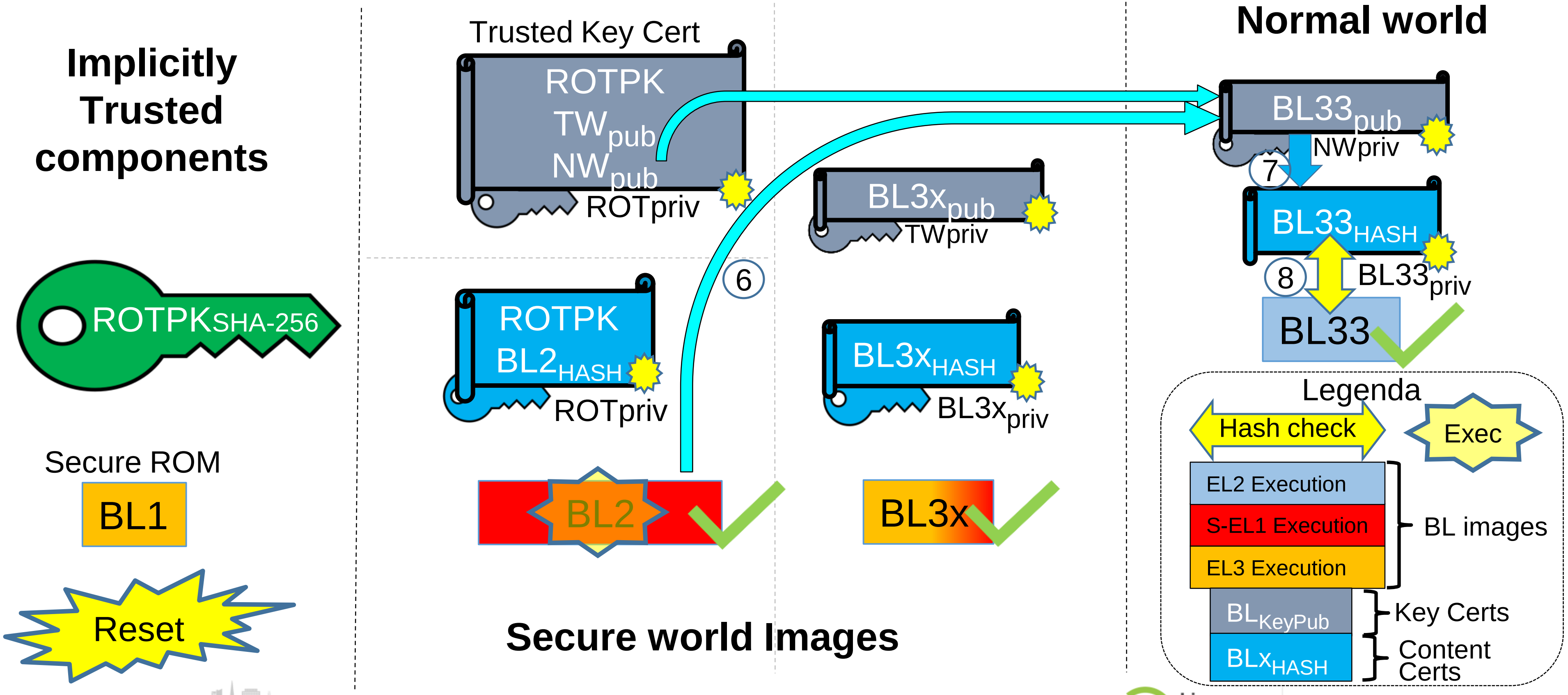
Arm Trusted Firmware TBB – How it works (BL1)



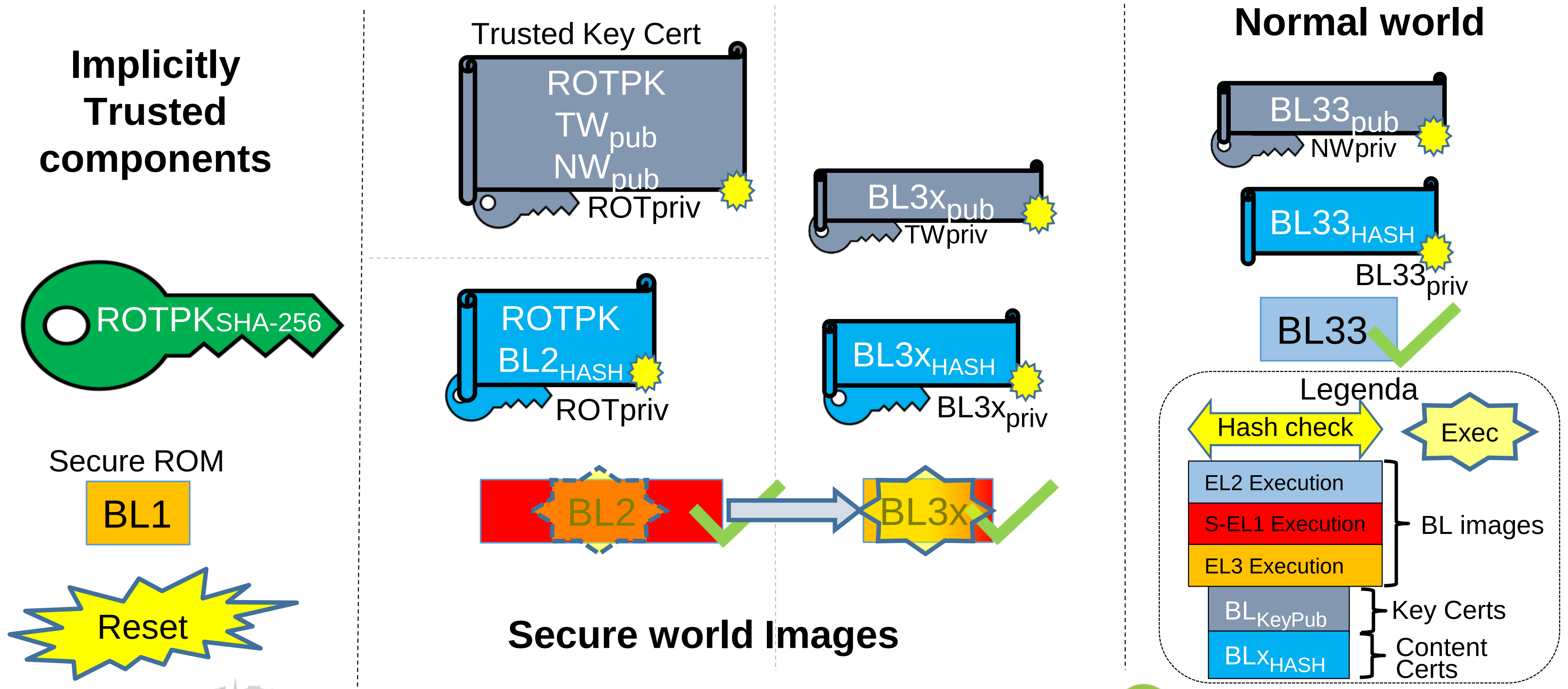
Arm Trusted Firmware TBB – How it works (BL2)



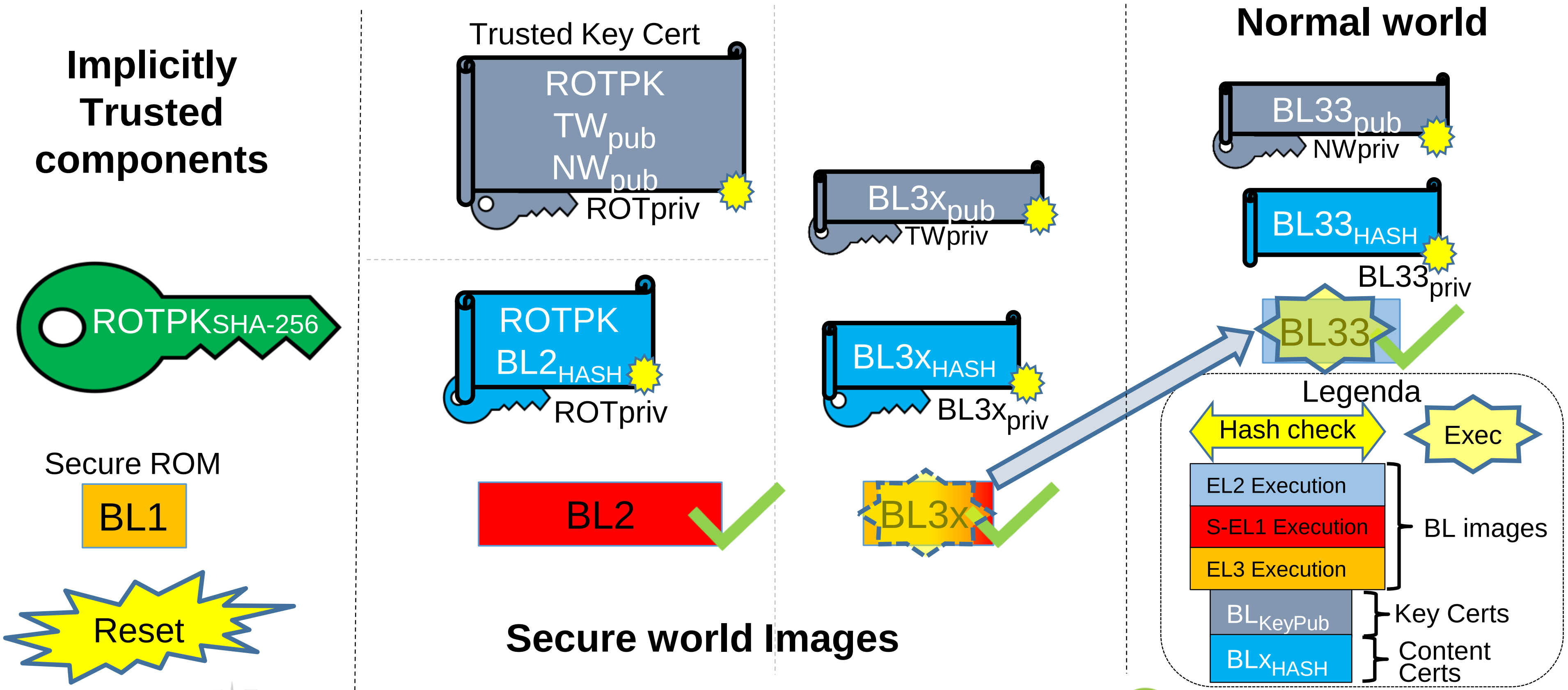
Arm Trusted Firmware TBB – How it works (BL2)



Arm Trusted Firmware TBB – How it works (BL3x)



Arm Trusted Firmware TBB – How it works (BL33)



Arm Trusted Firmware Implementation Overview

- TBB working properly on BL1/BL2 on both AArch64 & AArch32!
  - **JUNO and FVP Platforms TBB example running in AArch32 state on GitHub!**
- Build flags (summary)
  - *TRUSTED_BOARD_BOOT=1* to enable BL1+BL2 TBB support
  - *GENERATE_COT=1* build and execute cert_create tool (see below)
  - *XXX_KEYS=[path]* used to specify location of keys in PEM format
  - Have a look at the user guide(1)!
- Tools:
  - cert_Create too: BL images and Keys as input → Certificates as output
  - Fiptool: Certificates as input → FIP (Firmware Image Package)
- Pre-integration of TBB with the Arm TrustZone CryptoCell product (CC-712) to take advantage of its HW RoT and crypto acceleration services

UEFI Secure Boot

1. A platform ownership model for establishing a trust relationship among:
        - Platform Owner (ODM/OEM/EndUser) – PO
        - Platform Firmware (EDK2 / U-Boot / 3rd party BIOS) – PF
        - OS / 3rd party software vendors – OSV/ISV → SV
    - Uses standard PKI, X.509 certificates and PE images digital signature, based on PE digest/hash calculation described in Microsoft Authenticode PE Signature Format
    - Signature database (white/black list) update mechanism from trusted sources

2. A generic framework, based on the above model, to allow:
    1. The firmware to authenticate UEFI executable images before allowing their execution, preventing pre-boot malwares to be run
    2. The Platform Owner and/or SV to securely update the signature databases into PF with new/known allowed/forbidden image signatures

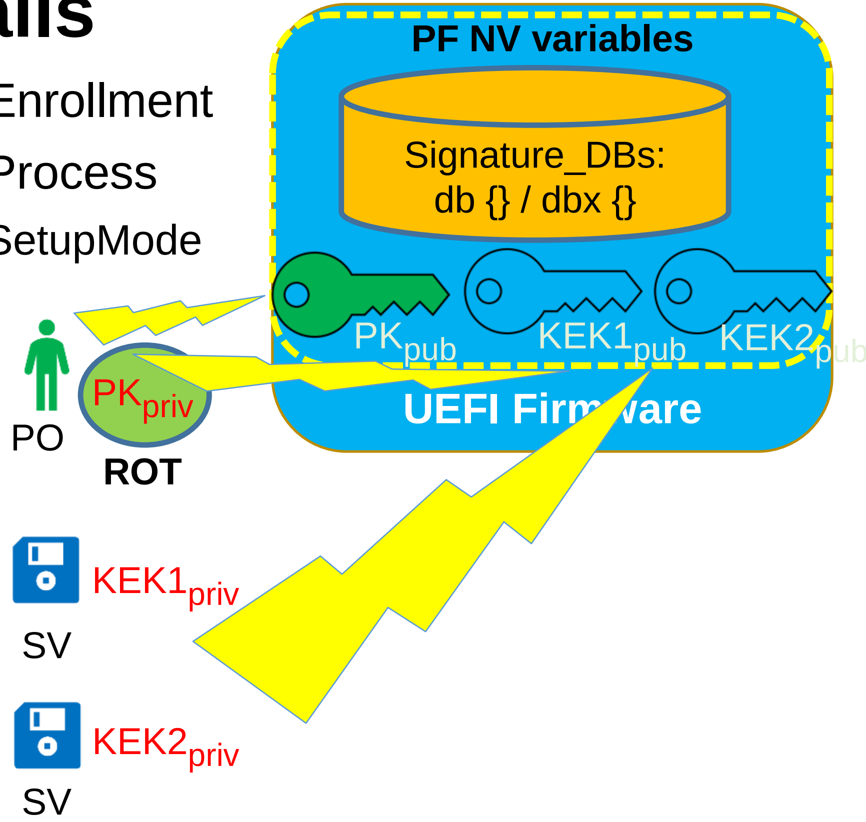
UEFI Secure Boot – PKI details

- 2 asymmetric key pairs:
    1. Platform Key (PK): Trust relationship between PO & PF
        - PK_{priv} owned by the PO
        - PK_{pub} enrolled into PF
    2. Key Exchange Key (KEK): Trust relationship between SV & PF
        - Different KEK_{priv} for each SV
        - Each SV enrolls KEK_{pub} into PF
- Platform firmware NV variables (**on tamper proof storage**) to hold:
    - PK_{pub} / KEK_{pub} list
    - Signatures DBs: signatures white/black lists (db/dbx)

Enrollment

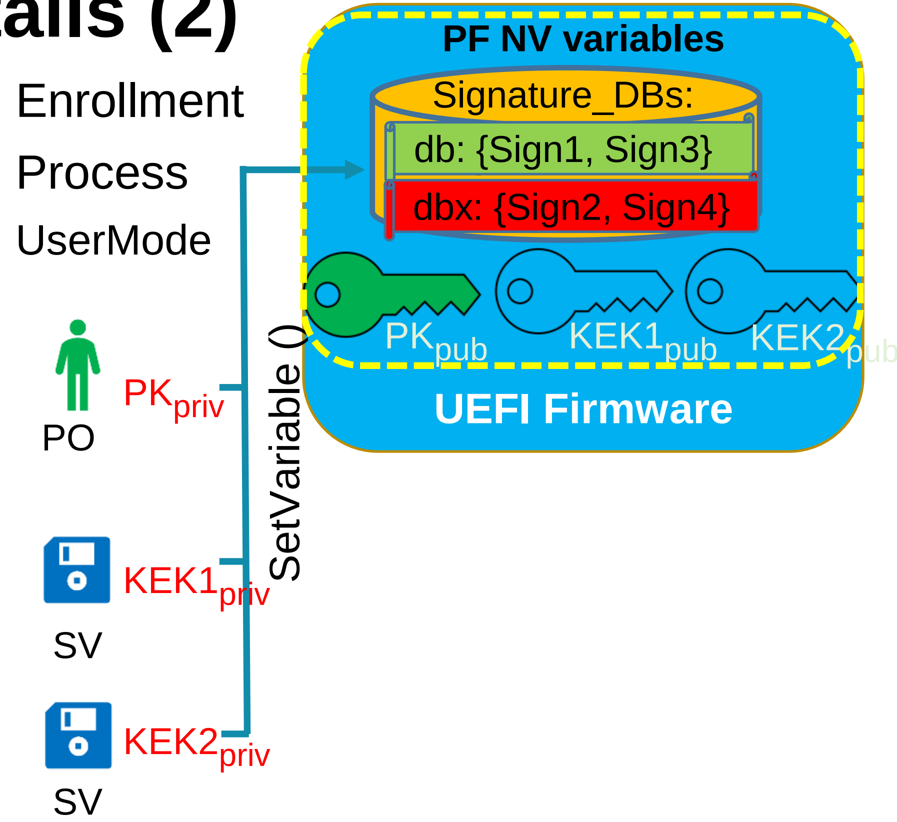
Process

SetupMode

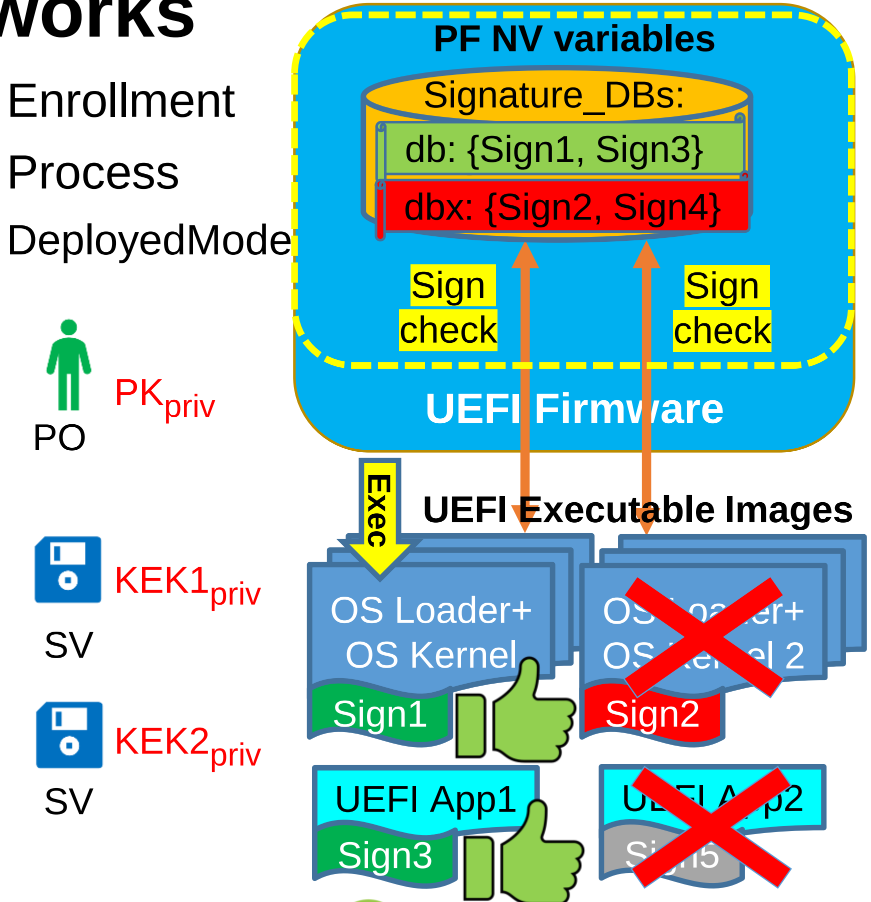


UEFI Secure Boot – PKI details (2)

- Using PK_{priv}/KEK_{priv} , Signature_DB is updated from **trusted sources** with allowed/forbidden image signatures, by means of UEFI SetVariable() Runtime service



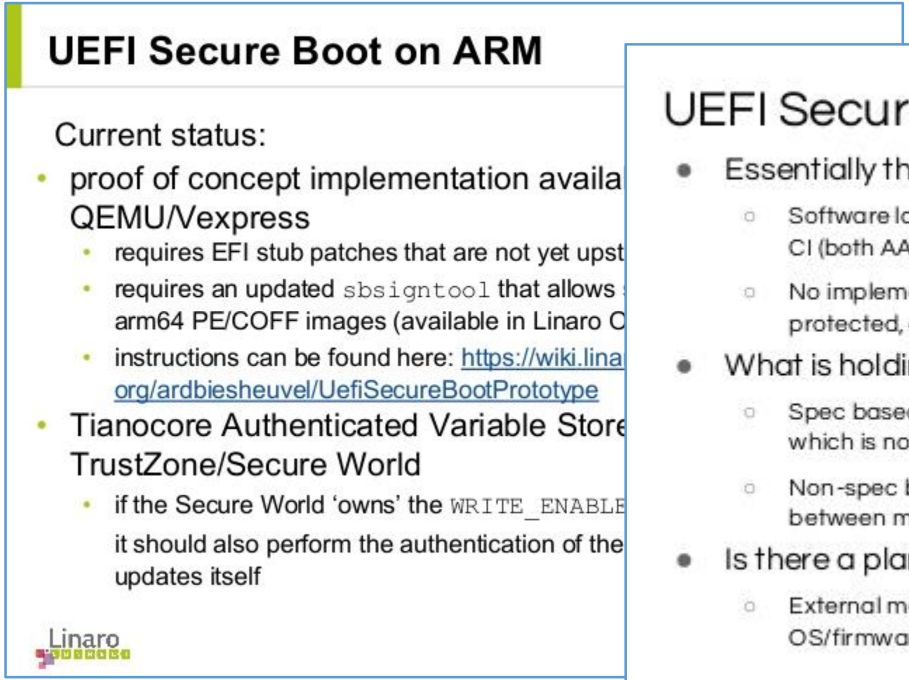
UEFI Secure Boot – How it works

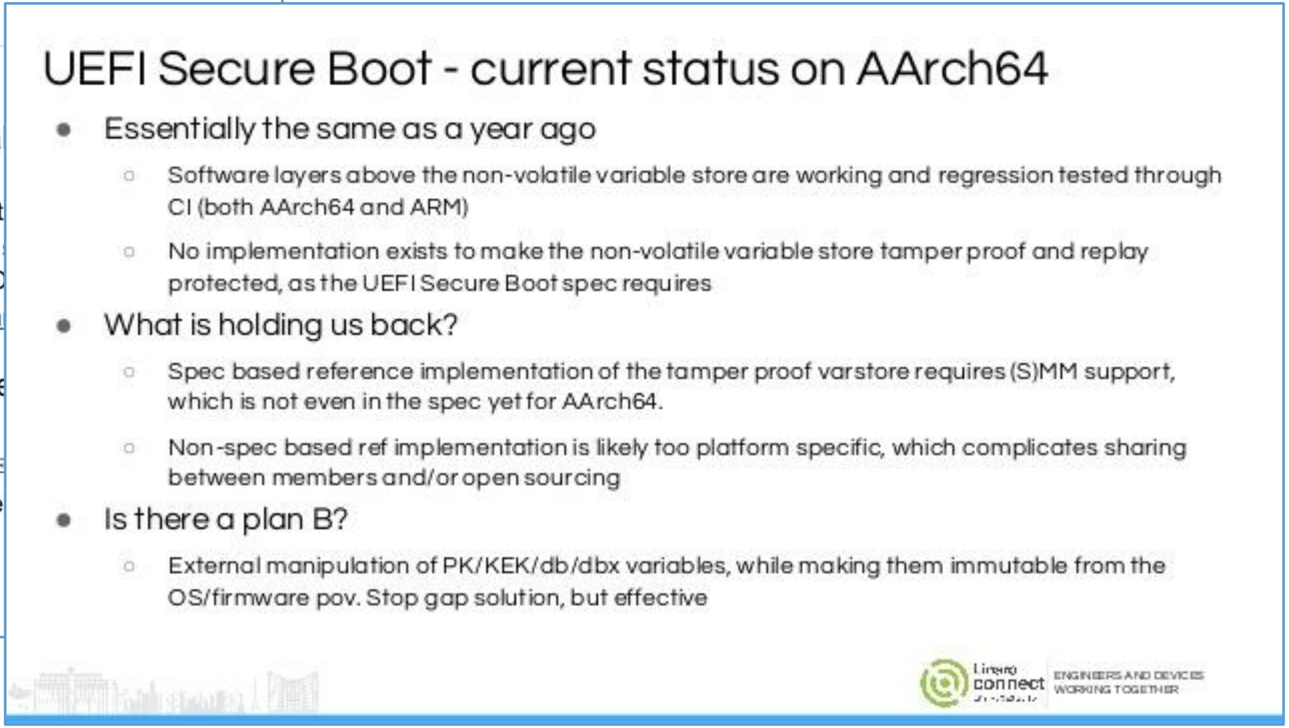


- UEFI executable images are verified against Signature_DBs found in the firmware

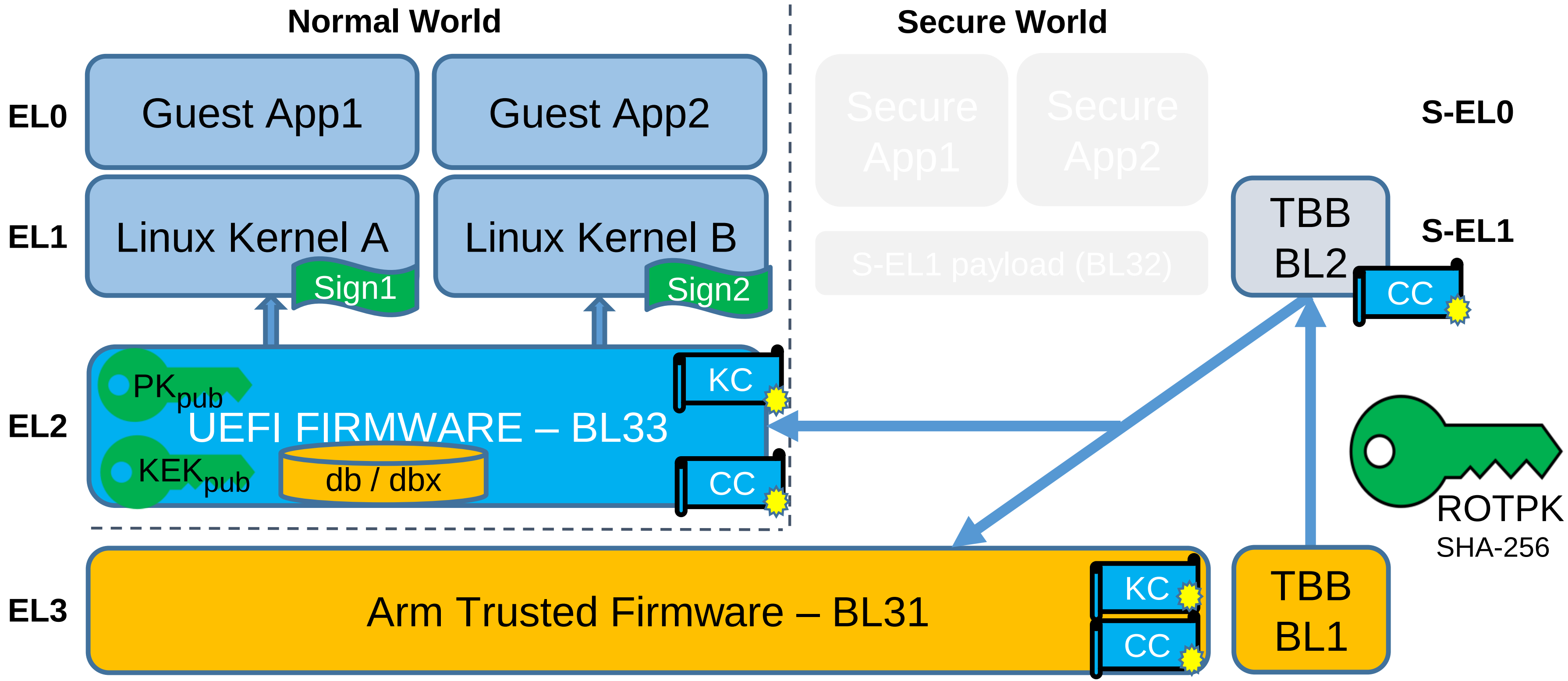
UEFI Secure Boot on Arm – EDK2 recap

- LCA14 (from Ard Biesheuvel)(2)
- LAS16 (Ard Biesheuvel)(3)

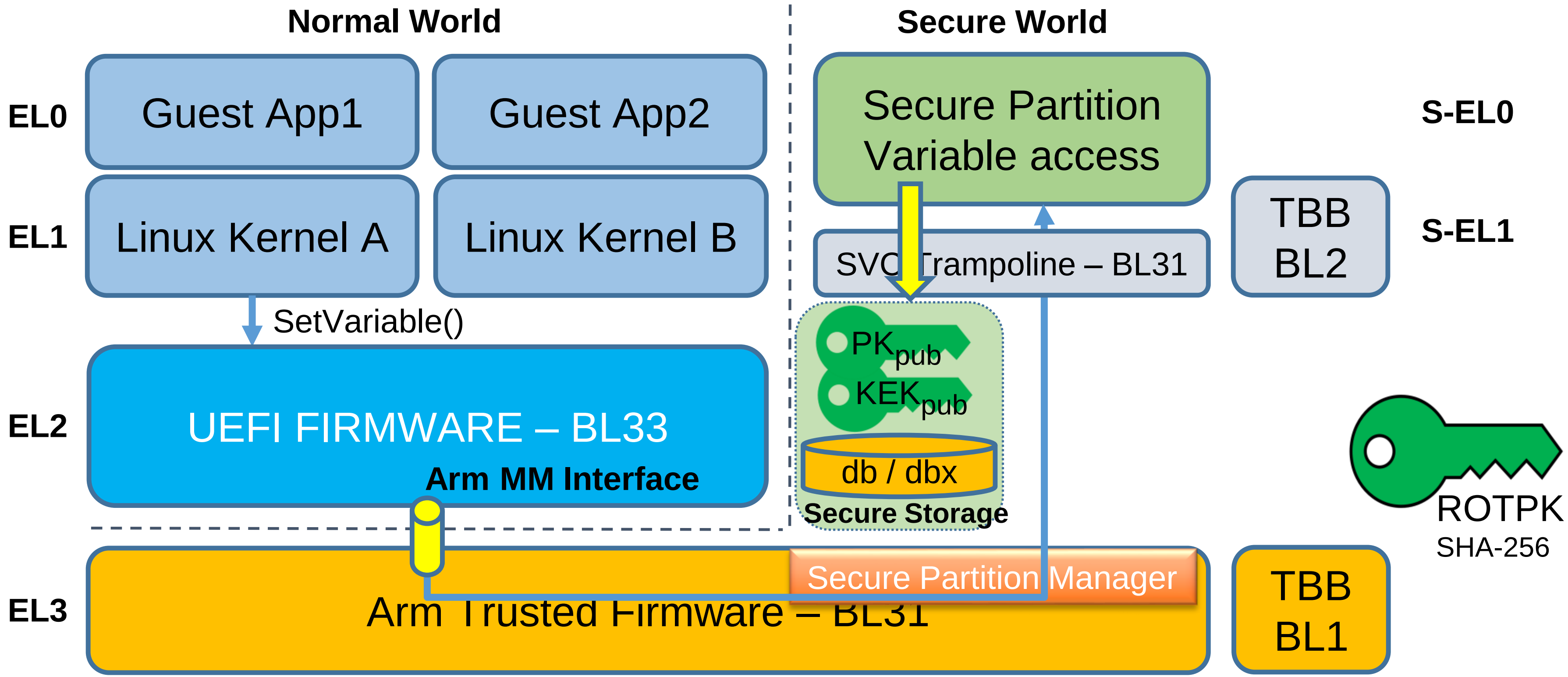




Complete CoT – Putting all together



Secure Variable access



Other OSS Solutions

- Android Verified Boot[4] on AOSP:
  - De-facto industry standard for Mobile secure boot path since Android 4.4/5.0
  - CoT starting from OEM public key (tamper proof) to verify android boot image
  - Device State (LOCKED/UNLOCKED) must be protected not to break the CoT
  - On newer versions (8.0) also Rollback protection available[5]
- U-Boot Verified Boot[6]
  - CoT starting from trusted U-Boot image (BL33) carrying initial public key (tamper proof)
  - Usual image verification chain then follows
  - No specified platform ownership model for updating keys in field
- U-Boot Secure Boot?
  - Leveraging "*UEFI on Top on U-Boot*"[7] work, with SetVariable extension?
  - Plugging shim over UEFI-enabled U-Boot to handle key management?

    → **Convergence of Embedded and Enterprise secure boot flows!**

Plans & Next Steps

- Software side:
  - Arm open-source reference platform software of TBB+UEFI Secure Boot with Secure Variable storage access from Secure Partition
  - Investigate U-Boot based solution for Embedded/Mobile
  - Future: Secure Firmware Update (FWU vs UEFI Signed Capsule Update)
- Specification side:
  - TBBR/SBBR updates & possible Server side TBBR/TBSA
  - Interactions with TCG TPM & Measured Boot
  - What level of standardization required on the Firmware side for a TBB solution?
    - A guidance on which authentication steps to be executed at each ELx/BLx to avoid arbitrary code execution at EL3[8]?
- Different HW solutions for the initial RoT (→ SFO17-304)

References

1) ARM Trusted Firmware TBB Documentation, Design Guide, User Guide
   - https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/trusted-board-boot.rst
   - https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/auth-framework.rst
   - https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/user-guide.rst
2) LCA14-105: UEFI secure boot
   - http://connect.linaro.org/resource/lca14/lca14-105-uefi-secure-boot/
3) LAS16-200: UEFI Secure Boot
   - http://s3.amazonaws.com/connect.linaro.org/las16/Presentations/Tuesday/LAS16-200%20-%20Firmware%20Summit%20-%20UEFI%20Secure%20Boot.pdf
4) Android Verified Boot: https://source.android.com/security/verifiedboot/
5) AVB Codebase and latest updates: https://android.googlesource.com/platform/external/avb/
6) U-Boot Verified Boot: https://lwn.net/Articles/571031/
7) UEFI on Top of U-Boot:
   - https://www.suse.com/docrep/documents/a1f0ledpbe/UEFI%20on%20Top%20of%20U-Boot.pdf
8) UCSB Mobile Boot Loaders Analysis and TEE implementation flaws
   - https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini

Thank You (matteo.carlini@arm.com)

matteo.carlini@arm.com

#SFO17

BUD17 keynotes and videos on: connect.linaro.org

For further information: www.linaro.org