Leveraging U-Boot Binman with Hardware Security Modules (HSM) for Secure Boot

Date: 2026-05-18

Riya Aysola
Judith Mendez

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT NORTH AMERICA

Texas Instruments

Agenda

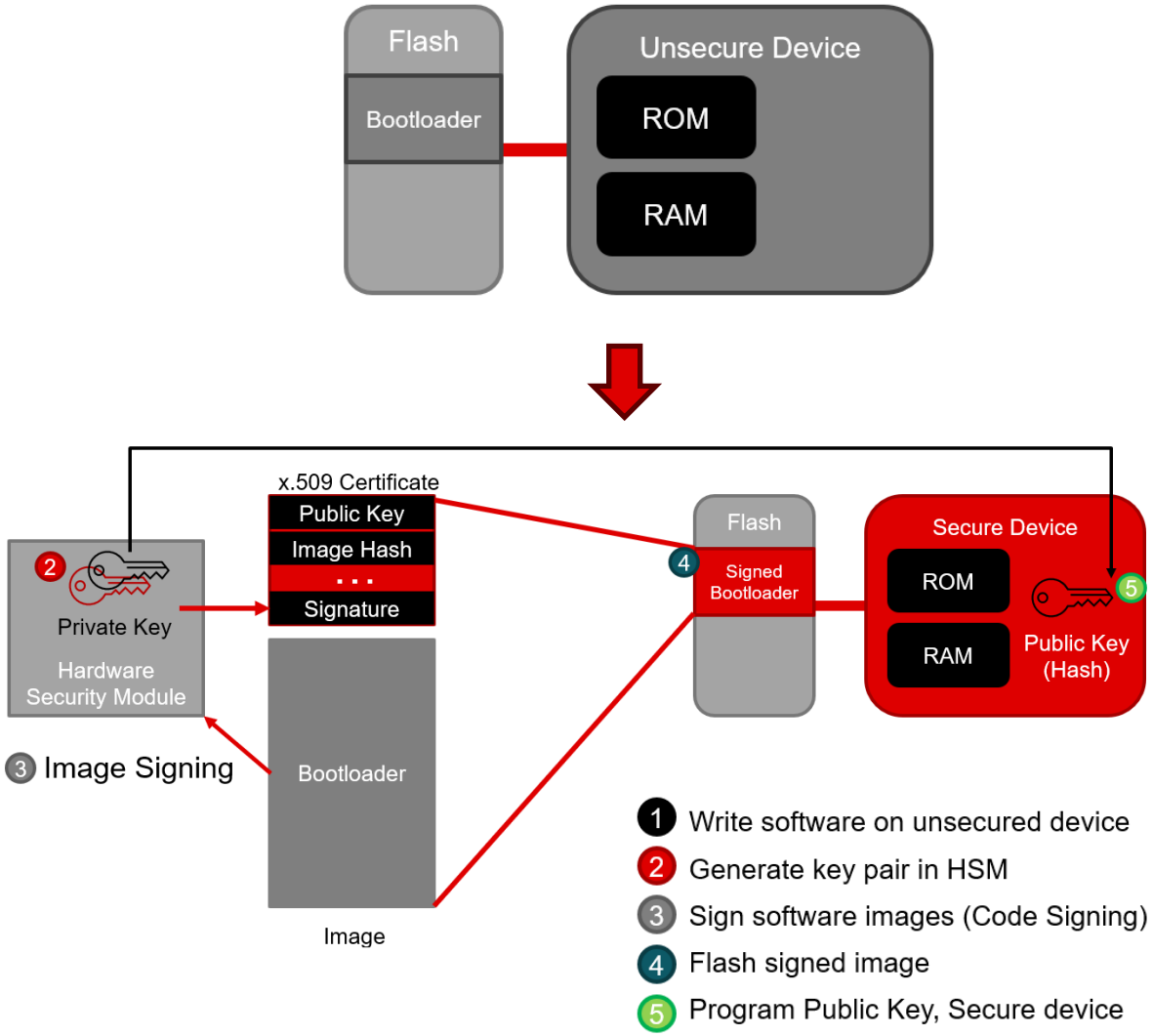
- Secure Boot
- Hardware Security Modules (HSMs)
- U-Boot Binman for image creation
- Integrating HSMs with Binman
- Best practices

Secure Boot overview

What is secure boot?

Security standard that ensures a device only runs software that is trusted by the hardware manufacturer or system owner

- Prevents execution of tampered or unauthorized code
- Secure boot is built on a chain of trust – the cryptographic verification where each stage verifies the next before execution

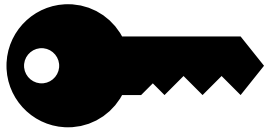


Challenges of Secure Boot at Scale

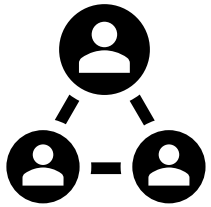
- Secure boot ensures only trusted firmware runs on a device
- Works well in controlled environments, but runs into difficulties at scale

Challenges in traditional systems:

Manual signing workflows
→ Error prone



Private keys stored locally
→ Security risk



Complexity due to multiple users
→ Not auditable

At scale:

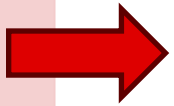
- Multiple images, multiple teams
- Key management becomes the bottleneck

To manage this complexity, we use U-Boot Binman and HSMs

Introduction to HSM

Traditional approach

- Private keys stored in developers' machines
- Scripts handle signing
- Risks ⚠
  - No key protection
  - No auditability
  - Not scalable



HSM approach

- Private keys never leave secure boundary
- Signing operations happen inside the secure boundary, HSM
- Controlled, auditable and more scalable
- Keys protected

Key features of an HSM

Key Management

- Streamline key management processes, allowing organizations to create, store, and manage cryptographic keys securely, reducing the risk of key compromise

Data Encryption

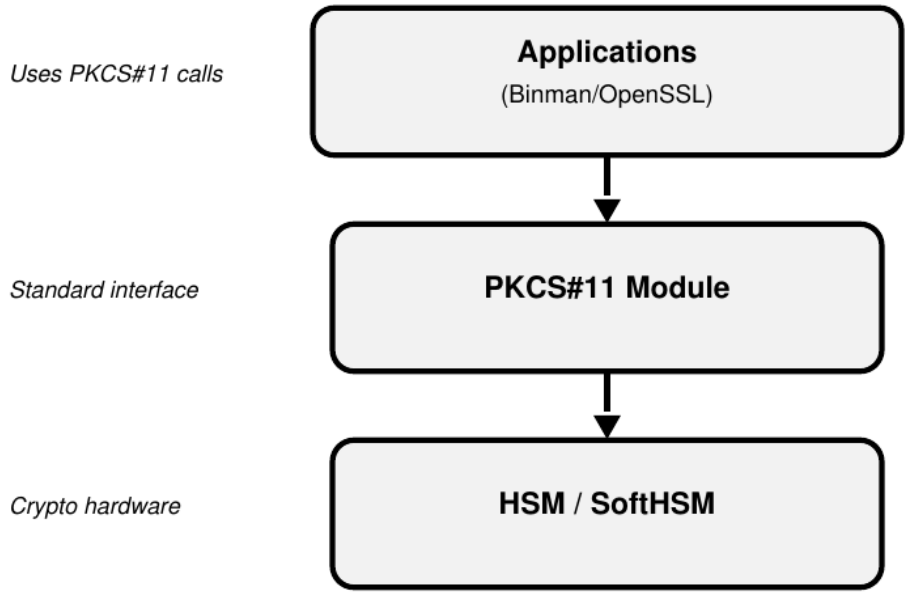
- Provide encryption techniques to protect sensitive data, ensuring it remains confidential and secure from unauthorized access

Digital Signatures

- Generate and verify digital signatures to enhance transaction integrity, enable secure communications, and authenticity in electronic transactions

PKCS#11 Standard Interface

Standard interface for apps to talk to cryptographic devices (HSMs, tokens, smartcards)



Cryptographic Token Interface Standard

Hierarchy

- Physical device (HSM)
  - **Slot 0: Token "Super_secret_keys"**
    - Session (authenticated, PIN=1234)
      - Private key object: custMpk (never extractable)
      - Public key object: custMpk (extractable)
      - Other objects...
  - Slot 1: Token "Backup_super_secret_keys"
- .....

URI:pkcs11:token=Super_secret_keys;object=custMpk;type=private;pin-value=1234

Types of HSMs

No HSM

- Security: Not certified (software based)
- Use case: development/testing with **no key protection**

Software HSMs

- SoftHSM
- Security: Not certified (software based)
- Use case: development/testing with HSM emulation

Hardware HSMs

- Nitrokey HSM 2, Yubico YubiHSM 2
- Security: FIPS 140-2 Level 3 (production HSMs)
- Use case: hardware security production signing

Cloud HSMs

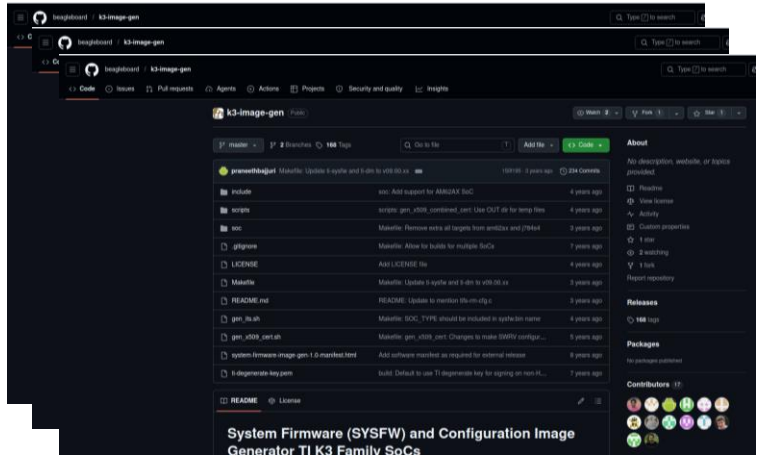
- AWS CloudHSM, Google Cloud HSM
- Security: provider dependent (FIPS 140-2 Level 2-3)
- Use case: cloud production signing

Server HSMs

- Thales Luna 7, Utimaco Security Server
- Security: FIPS 140-2 Level 3 (hardware-based)
- Use case: locally owned production signing

What is U-Boot Binman?

Firmware image packing tool in U-Boot source by Simon Glass in 2016 for secure boot

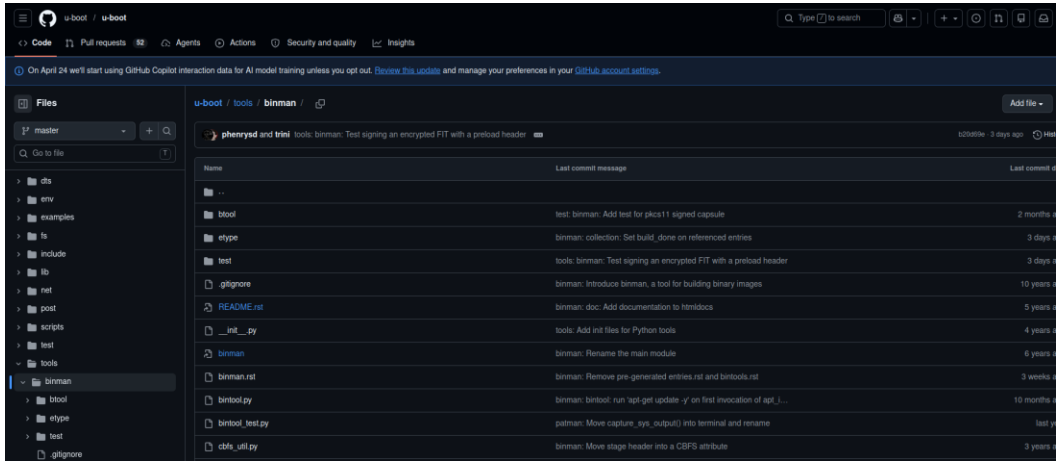


The Problem:

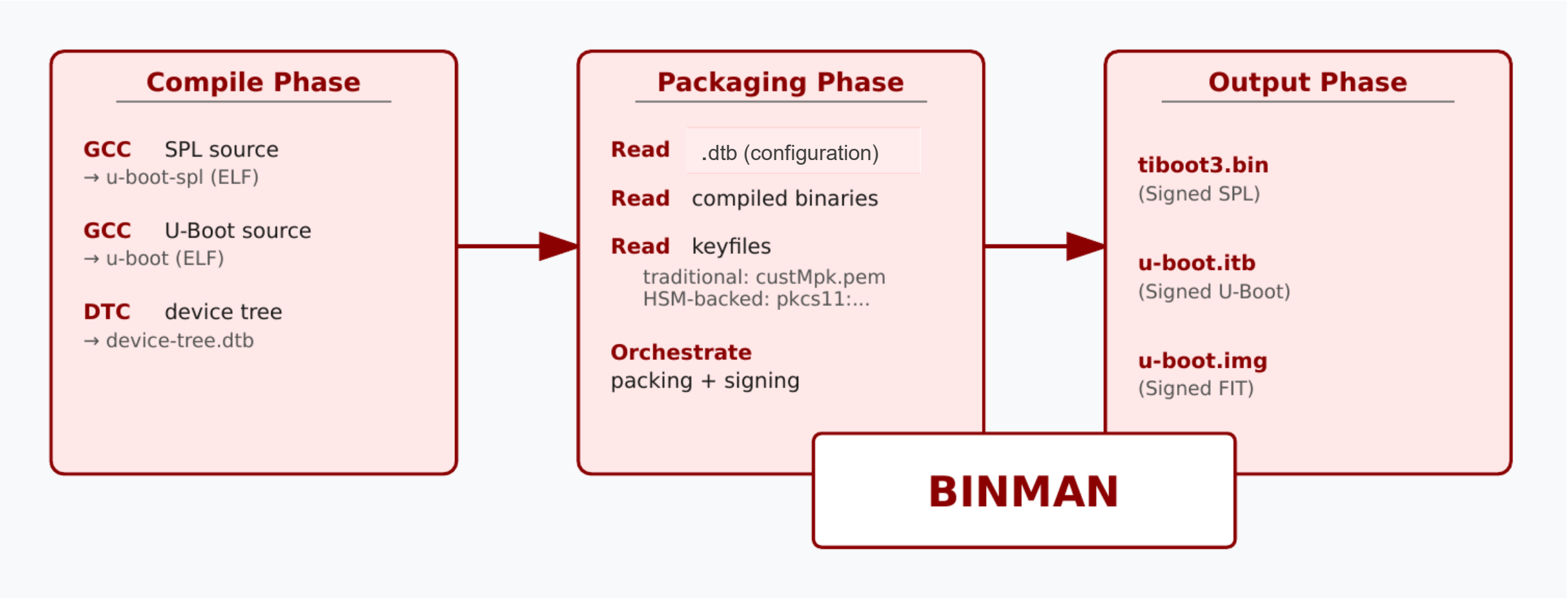
- Multiple sources to produce bootable images

The Binman solution: ***One packaging solution to rule them all***

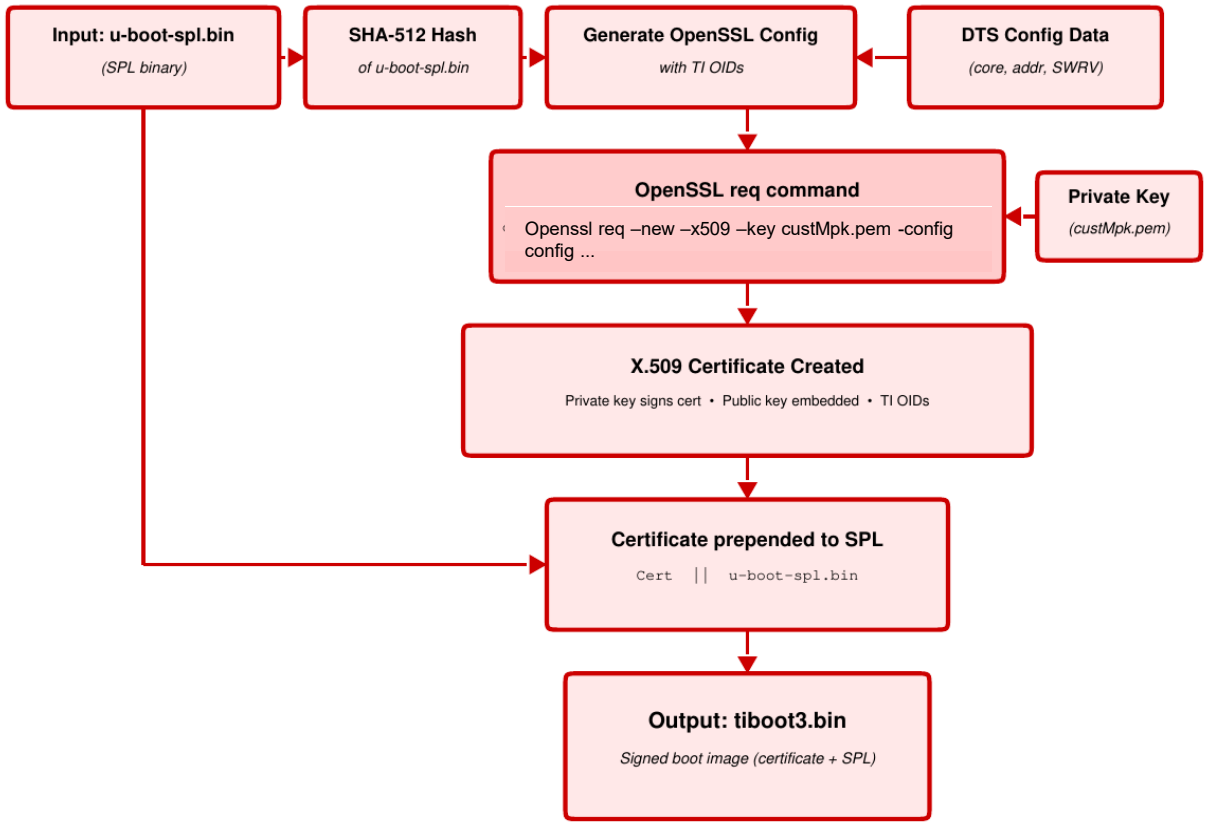
- Standardization
- Open Source
- Modular architecture
- Easy to expand on



Binman's role: binaries to signed images



Binman: example traditional signing flow

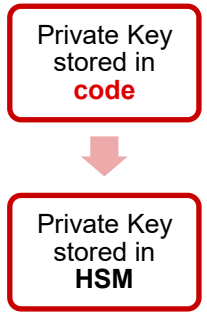


Binman

- Takes multiple inputs
- Puts together OpenSSL CMD
- Generates x509 Cert
- Prepends Cert to tiboot3.bin

Project focus:

- **Openssl req CMD**
- Important change:

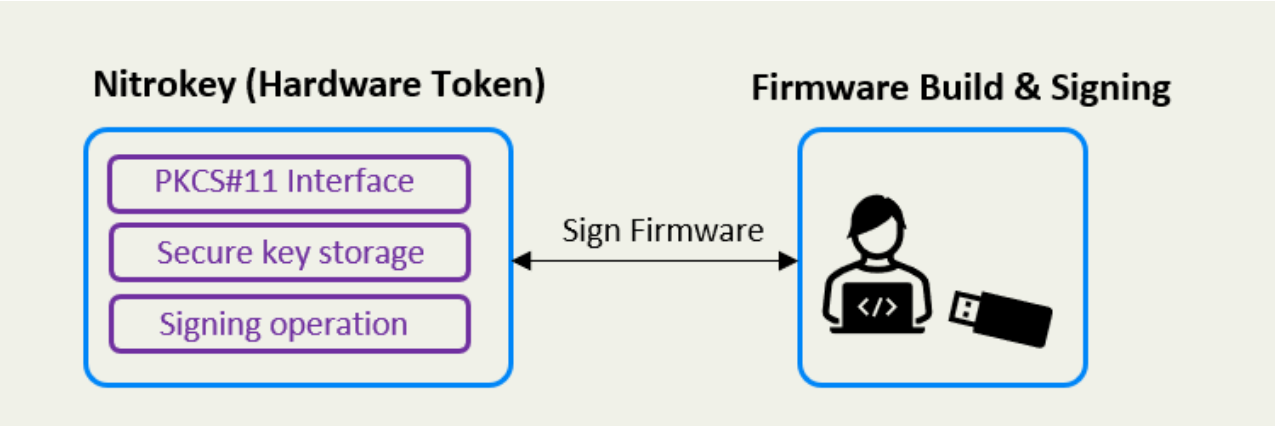


Software based HSM

- What is a Software based HSM:
  - Cryptographic software mimicking HSM behavior on general purpose CPU
- How it works:
  - Run cryptographic operations using software libraries
  - Store keys in files & expose standard interfaces (often PKCS#11) to mimic hardware HSM
- Advantages:
  - No specialized hardware cost
  - Fast & simple to reset for testing
- Challenges:
  - Keys stored in disk, vulnerable to theft/extraction
  - No tamper detection nor auto destroy if compromised

Hardware based HSM

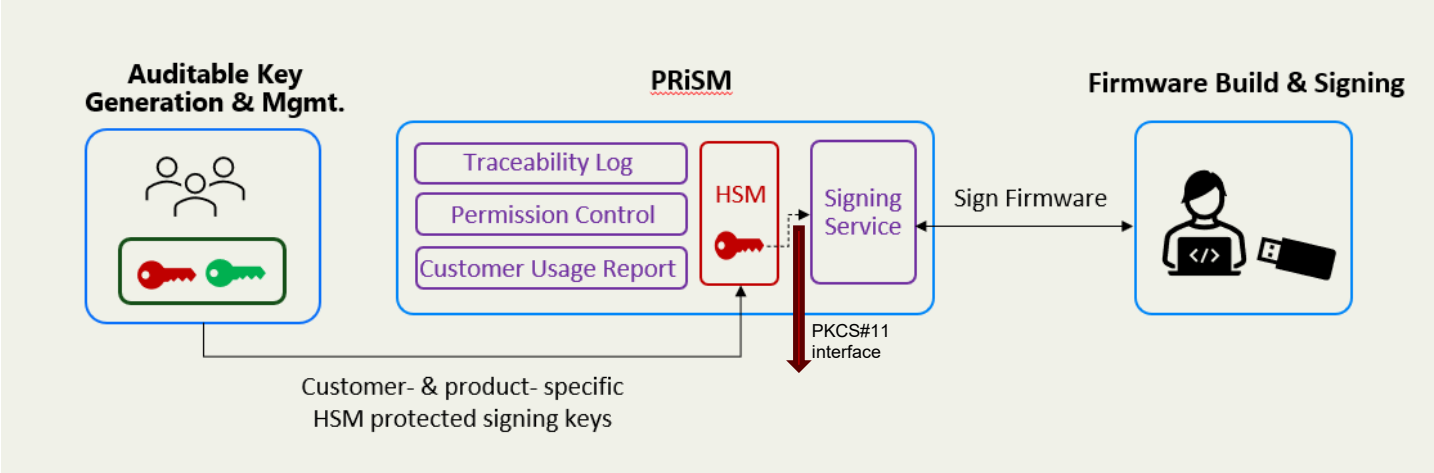
- What is a Hardware based HSM:
  - Dedicated hardware device for secure key storage and cryptographic operations
  - Generates and stores private keys securely
- How it works:
  - Connected locally to signing system
  - Signing happens inside the device
- Advantages:
  - Strong key isolation
  - Simple to get started
  - No network dependency
- Challenges:
  - Hard to automate in CI/CD
  - Physical access required



**Evaluation performed using a hardware token solution (NitrokeyHSM2)*

Server based HSM

- What is a Server based HSM:
  - Centralized HSM system designed for multi-user access and high-throughput signing used in production signing pipelines
- How it works:
  - Build system sends signing request
  - HSM performs signing remotely
  - Private keys are never exposed
- Advantages:
  - Scalable for larger teams
  - Easy integration with CI/CD
  - Centralized Key control
- Challenges:
  - Setup complexity
  - Latency/connectivity considerations



**Evaluation performed using a TI partner remote HSM service (Aurora Networks)*

Why PKCS#11 matters?

WITHOUT PKCS#11

Thales Luna:
- thales_sign()
- thales_connect()

Switch to Yubico:
- yubico_sign()
- yubico_connect()

Changes needed:

- Rewrite signing code
- New error handling
- Test everything again
- Rebuild Binman

WITH PKCS#11

Thales Luna:
module_path=/opt/luna

Switch to Yubico:
module_path=/yubihsm

Changes needed:

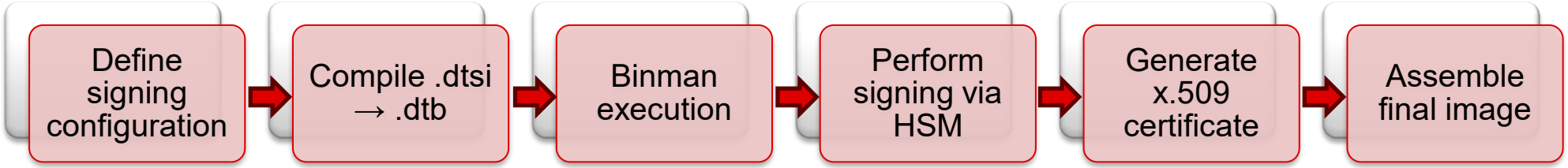
- Update config
- Update PKCS#11 URI

Integration of Binman with different HSMs

- **Bridging Image creation and secure signing:**
  - Binman determines what needs signing & prepares components for signing
  - HSMs perform signature generation via secure key access

- **Consistent Flow across HSM types:**
  - Image structure and boot flow remain unchanged
  - Signing backend can vary depending on the type of HSM

- **Standardized interaction layer:**
  - Binman interfaces with HSMs through a common PKCS#11 interface layer
  - Enables portability across different HSM implementations



Integration of Binman with different HSMs contd

	Software HSM	Hardware HSM	Server HSM
Initialize token	`softhsm2-util --init-token --slot 0 --label "TI_AM62x_Keys"`	`pkcs11-tool --module /usr/local/lib/opensc-pkcs11.so --init-token --init-pin --so-pin="pin" --new-pin=648219 --label="test" --pin=648219`	Done by provider
Generate signing key	`pkcs11-tool --keypairgen --key-type rsa:4096 --label "custMpk"`	`pkcs11-tool --module /usr/local/lib/opensc-pkcs11.so -l --pin 648219 --keypairgen --key-type rsa:4096 --id 10`	Done by provider
Configure OpenSSL for PKCS#11	In /etc/ssl/openssl.cnf •Add [pkcs11_section] •Set module_path = /usr/lib/softhsm/libsofthsm2.so	In /etc/ssl/openssl.cnf •Add [pkcs11_section] •Set module_path = /usr/libx86_64-linux-gnu/opensc-pkcs11.so	Done by provider
Modify Binman OpenSSL invocation	`-engine pkcs11 -keyform engine`	`-engine pkcs11 -keyform engine`	Uses PRISM plugin
Embed PKCS#11 URI in binman DTS	`pkcs11:token=TI_AM62x_Keys;object=custMpk;type=private;pin-value=1234`	`pkcs11:token=test%20%28UserPIN%29;id=%12;object=Private%20Key;type=private;pin-value=234567`	Hidden

- Primary change across different HSM's is the configuration, not the build flow

Before:

```
Keyfile = "localMpk.pem"
```

After: Same Binman flow – only the key reference changes

```
Keyfile = "pkcs11:token=<token>;id=<id>;object=Private;type=private;pin-value=<pin-value>"
```

Integration of Binman with different HSMs contd

	Software HSM	Hardware HSM	Server HSM
Initialize token	`softhsm2-util --init-token --slot 0 --label "TI_AM62x_Keys"`	`pkcs11-tool --module /usr/local/lib/opensc-pkcs11.so --init-token --init-pin --so-pin="pin" --new-pin=648219 --label="test" --pin=648219`	Done by provider
Generate signing key	`pkcs11-tool --keypairgen --key-type rsa:4096 --label "custMpk"`	`pkcs11-tool --module /usr/local/lib/opensc-pkcs11.so -l --pin 648219 --keypairgen --key-type rsa:4096 --id 10`	Done by provider
Configure OpenSSL for PKCS#11	In /etc/ssl/openssl.cnf •Add [pkcs11_section] •Set module_path = /usr/lib/softhsm/libsofthsm2.so	In /etc/ssl/openssl.cnf •Add [pkcs11_section] •Set module_path = /usr/libx86_64-linux-gnu/opensc-pkcs11.so	Done by provider
Modify Binman OpenSSL invocation	`-engine pkcs11 -keyform engine`	`-engine pkcs11 -keyform engine`	Uses PRISM plugin
Embed PKCS#11 URI in binman DTS	`pkcs11:token=TI_AM62x_Keys;object=custMpk;type=private;pin-value=1234`	`pkcs11:token=test%20%28UserPIN%29;id=%12;object=Private%20Key;type=private;pin-value=234567`	Hidden

- Primary change across different HSM's is the configuration, not the build flow

Before:

```
Keyfile = "localMpk.pem"
```

After: Same Binman flow – only the key reference changes

```
Keyfile = "pkcs11:token=<token>;id=<id>;object=Private;type=private;pin-value=<pin-value>"
```

- Acknowledgement: During development of this evaluation, related PKCS#11 integration patches for U-Boot Binman were also contributed by Sergio Prado.
- Related patches:
  - Patch v4

Best practices for production signing

Production signing is critical in secure boot, ensuring only trusted firmware is deployed to devices at scale.

Protect Root of Trust:

- Store private keys in HSMs
- Use separate keys for development and production
- Plan for key rotation and revocation

Secure CI/CD integration:

- Automate signing within the build pipeline
- Use authenticated access to signing services
- Avoid manual signing workflows

Access control and Auditability:

- Enforce role-based access control
- Maintain audit logs for all signing operations
- Monitor usage for anomalies

Firmware integrity and lifecycle:

- Implement versioning and rollback protection
- Ensure all boot stages are consistently signed
- Validate signatures through chain of trust

Thank you!

Questions?