

Integrity Verification in Embedded Systems

Bachelor's Thesis

By

Douglas Fjällrud

Axel Blomén

Department of Electrical and Information Technology

Faculty of Engineering, LTH

Lund University

SE-221 00 Lund, Sweden



LUND
UNIVERSITY

2026

Abstract

Linux-based embedded systems are often deployed for long periods, updated in the field, and expected to remain trustworthy under limited maintenance. Secure boot can protect the startup process by verifying software before execution, but practical secure boot depends on more than boot-time signature checks. Trust anchors, key material, provisioning steps, update mechanisms, and release artifacts must remain consistent throughout the device lifecycle.

This thesis analyzes secure boot, cryptographic key management, and provisioning in Linux-based embedded systems, with emphasis on practical maintainability. The work combines a literature and standards review, a comparative analysis of selected secure boot solutions, and a practical case study on an NXP i.MX 93 platform. The case study applies the resulting principles in a company-provided Atlas Yocto-based environment using Advanced High Assurance Boot (AHAB), dm-verity, the Robust Auto-Update Controller (RAUC), and release artifact traceability.

The practical implementation added read-only root filesystem behavior, separated writable runtime and configuration state, integrated dm-verity for A/B root filesystem slots, adapted RAUC to install raw dm-verity-protected images, synchronized slot-specific dm-verity metadata after updates, and implemented an AHAB signing and provisioning workflow. The AHAB work used Code Signing Tool (CST)-generated Super Root Key (SRK) material, programmed and read back the corresponding Super Root Key Hash (SRKH) fuse words, and validated AHAB behavior in Original Equipment Manufacturer (OEM) Open state with both mismatched and correctly signed boot artifacts.

The validation showed that the storage layout, read-only root filesystem behavior, dm-verity root mounting, RAUC A/B update flow, metadata synchronization, SRKH fuse programming, and AHAB OEM Open event behavior operated as intended under the tested conditions. Both RAUC update directions succeeded while preserving `/dev/mapper/rootfs` as the mounted root source. For AHAB, a mismatched boot image produced an EdgeLock Enclave (ELE) key-hash mismatch indication after SRKH programming, while the correctly signed `flash_singleboot` artifact produced no remaining ELE authentication events under the tested OEM Open conditions.

The results show that secure boot integration in embedded Linux depends on consistency across several artifact boundaries: signed boot artifacts, programmed trust-anchor material, root filesystem images, dm-verity metadata, update state, verification material, and provisioning records must refer to the same system state. The study does not claim full production deployment or OEM Closed enforcement, since the device lifecycle was not transitioned to OEM Closed and a complete adversarial test campaign was outside scope. Instead, the contribution is a method-oriented analysis and case study that connects general secure boot principles to concrete embedded Linux engineering decisions.

Keywords: secure boot; embedded Linux; NXP i.MX 93; AHAB; dm-verity; secure updates.

Sammanfattning

Linux-baserade inbyggda system används ofta under lång tid, uppdateras efter driftsättning och förväntas förbli pålitliga trots begränsat underhåll. Säker uppstart kan skydda startförloppet genom att verifiera programvara innan den körs, men en praktiskt användbar lösning kräver mer än signaturkontroller vid uppstart. Tillitsankare, nyckelmaterial, provisioneringssteg, uppdateringsmekanismer och releaseartefakter måste vara konsekventa genom hela enhetens livscykel.

Detta examensarbete analyserar säker uppstart, kryptografisk nyckelhantering och provisionering i Linux-baserade inbyggda system, med särskilt fokus på praktisk underhållbarhet. Arbetet kombinerar en litteratur- och standardstudie, en jämförande analys av utvalda secure boot-lösningar och en praktisk fallstudie på en NXP i.MX 93-plattform. Fallstudien tillämpar de identifierade principerna i en av uppdragsgivaren tillhandahållen Atlas-miljö baserad på Yocto, med Advanced High Assurance Boot (AHAB), dm-verity, Robust Auto-Update Controller (RAUC) och spårbarhet för releaseartefakter.

Den praktiska implementationen införde ett skrivskyddat rotfilssystem, separerade skrivbar runtime- och konfigurationsdata, integrerade dm-verity för A/B-slotar för rotfilsystemet, anpassade RAUC för installation av råa dm-verity-skyddade images, synkroniserade slotspecifik dm-verity-metadata efter uppdateringar och implementerade ett AHAB-baserat signerings- och provisioneringsflöde. AHAB-arbetet använde Code Signing Tool (CST)-genererat Super Root Key-material (SRK), programmerade och läste tillbaka motsvarande Super Root Key Hash-fuseord (SRKH), och validerade AHAB-beteende i Original Equipment Manufacturer (OEM) Open-läge med både felmatchade och korrekt signerade bootartefakter.

Valideringen visade att lagringslayouten, det skrivskyddade rotfilsystemet, dm-verity-baserad montering av rotfilsystemet, RAUC:s A/B-uppdateringsflöde, metadata-synkronisering, SRKH-fuseprogrammering och AHAB:s OEM Open-händelser fungerade som avsett under de testade förhållandena. Båda RAUC-uppdateringsriktningarna lyckades samtidigt som `/dev/mapper/rootfs` förblev den monterade källan för rotfilsystemet. För AHAB gav en felmatchad boot image en EdgeLock Enclave (ELE)-indikation om nyckelhashfel efter SRKH-programmering, medan den korrekt signerade `flash_singleboot`-artefakten inte gav några kvarvarande ELE-autentiseringshändelser under de testade OEM Open-förhållandena.

Resultaten visar att integration av säker uppstart i inbyggd Linux beror på konsekvens över flera artefaktgränser: signerade bootartefakter, programmerat tillitsankarmaterial, rotfilsystemimages, dm-verity-metadata, uppdateringstillstånd, verifieringsmaterial och provisioneringsdokumentation måste hänvisa till samma systemtillstånd. Studien gör inte anspråk på att vara en fullständig produktionsdriftsättning eller på att visa verkställning i OEM Closed, eftersom enhetens livscykel inte överfördes till OEM Closed och en fullständig angriprioriterad testkampanj låg utanför arbetets omfattning. Bidraget är i stället en metodorienterad analys och fallstudie som kopplar generella secure boot-principer till konkreta ingenjörsbeslut i inbyggd Linux.

Nyckelord: säker uppstart; inbyggd Linux; NXP i.MX 93; AHAB; dm-verity; säkra uppdateringar.

Contents

Abstract	1
Sammanfattning	3
Acknowledgements	9
Preface	11
List of Acronyms	13
Popular Science Summary	15
1. Introduction	17
1.1 Background	17
1.1.1 Industrial Context and Motivation	17
1.2 Problem Formulation	18
1.3 Aim and Research Questions	19
1.4 Scope and Delimitations	20
1.5 Contributions	20
1.6 Thesis Outline	21
2. Background and Theoretical Foundation	23
2.1 Secure Boot in Embedded Systems	23
2.2 Root of Trust, Key Hierarchies, and Provisioning	23
2.3 Key Lifecycle and Rotation	24
2.4 Relevant Standards and Frameworks	25
2.5 Embedded Linux Context	25
2.6 NXP i.MX 93 and AHAB	26
2.7 Yocto, RAUC, and Update Security	26
2.8 Technical Terms	27
2.8.1 Build, Artifacts, and Update Terminology	27
2.8.2 Yocto, RAUC, and Update Workflow Terms	28
2.8.3 Linux Storage, Filesystems, and Early Boot	28
2.8.4 Device Mapper and dm-* Targets	29
2.8.5 AHAB, Fuses, and Provisioning Terms	30
2.8.6 UEFI, TI, and Rockchip Comparison Terms	31
2.8.7 How These Terms Relate in the Case Study	32
3. Related Work	33
3.1 Standards-Oriented Work	33
3.2 Research on Secure Boot and Key Management	33
3.3 Industrial Practices and White Papers	34
3.4 Research Gap	35
4. Methodology	37

4.1 Research Design	37
4.2 Literature and Standards Review	37
4.2.1 Search Strategy and Selection	38
4.2.2 Source Criticism	39
4.3 Comparative Analysis Framework	39
4.4 Selection of Platforms / Solutions	40
4.5 Case Study Design	41
4.6 Validity, Limitations, and Threats to Validity	42
5. Comparative Analysis of Existing Solutions	45
5.1 Comparison Criteria	45
5.2 Cryptographic Design Choices	46
5.3 Provisioning Models	47
5.4 Key Storage and Protection	48
5.5 Key Rotation and Long-Term Maintenance	48
5.6 Tooling, Openness, and Usability	49
5.7 Qualitative Synthesis and Design Suitability	50
5.7.1 Suitability by Engineering Purpose	51
5.7.2 Common Design Principles	53
6. Practical Case Study on NXP i.MX 93	55
6.1 Hardware, Baseline Project, and Evaluation Environment	55
6.2 Implemented Boot and Integrity Handover	57
6.2.1 RAUC Slot Selection in U-Boot	59
6.2.2 Kernel and Initramfs Handover	59
6.2.3 Root Filesystem Integrity Boundary	59
6.3 Storage Layout and Writable State Separation	59
6.3.1 WIC Layout and Partition Roles	60
6.3.2 Read-only Root Filesystem	60
6.3.3 Data and Config Partitions	61
6.3.4 Writable Runtime State	61
6.4 dm-verity Integration for A/B Root Filesystem Slots	61
6.4.1 meta-security and Image Configuration	61
6.4.2 Raw Verity Images in the WIC Layout	62
6.4.3 Boot Script Changes for A/B Verity Boot	62
6.4.4 Initramfs Mount of /dev/mapper/rootfs	62
6.5 RAUC Integration with dm-verity Slots	62
6.5.1 Existing Atlas RAUC Baseline	63
6.5.2 RAUC Data Directory and Raw Slot Configuration	63
6.5.3 Raw RAUC Bundle Contents for dm-verity Slots	63
6.5.4 dm-verity Metadata Synchronization After Slot Installation	64
6.5.5 Failure and Recovery Assumptions	67
6.6 AHAB Signing and Provisioning	67
6.6.1 AHAB Signing Inputs and Artifacts	68
6.6.2 CST-generated SRK Material and SRKH Fuse Tables	68
6.6.3 SRKH Fuse Programming and Readback	69

6.6.4 OEM Open AHAB Validation After SRKH Programming	69
6.6.5 Provisioning Boundary and Lifecycle State	70
6.7 Key and Trust Material Placement	70
6.7.1 Implemented Key and Metadata Inventory	71
6.7.2 Host-side and Target-side Separation	72
6.7.3 Device-local Configuration Storage	72
6.8 Build and Release Artifact Traceability	72
6.8.1 Build-time Inputs and Generated Artifacts	73
6.8.2 Signing Authority and Artifact Traceability	74
6.8.3 Release and Provisioning Records	76
7. Results	79
7.1 Results from the Comparative Analysis	79
7.2 Results from the Practical Case Study	80
7.2.1 Overview of Validation Cases	80
7.2.2 Storage Layout and Read-only Mount Results	82
7.2.3 dm-verity Boot Results	82
7.2.4 RAUC A/B Update Results with dm-verity	83
7.2.5 Slot-specific Metadata Synchronization Results	84
7.2.6 AHAB Signing, SRKH Programming, and OEM Open Validation Results	85
7.2.7 Summary of Practical Case Study Results	88
8. Discussion	89
8.1 Interpretation of the Results	89
8.2 Trade-Offs Between Security, Usability, and Maintainability	89
8.3 Transferability Beyond NXP i.MX 93	91
8.4 Limitations of the Study	92
8.5 Societal and Ethical Considerations	94
9. Conclusions	95
9.1 Fulfillment of the Aim	95
9.2 Answers to the Research Questions	95
RQ1: Which key management and provisioning principles are most important for secure boot in Linux-based embedded systems?	95
RQ2: How do selected secure boot implementations differ with respect to cryp- tographic design, tooling, openness, usability, and long-term maintain- ability?	96
RQ3: How can secure boot be practically integrated on an NXP i.MX 93 using AHAB in a way that supports structured development and deployment workflows?	96
RQ4: How do supporting build, update, and release workflows, such as Yocto, RAUC, and artifact-traceability routines, affect the practical security and maintainability of such a system?	96
RQ5: Which aspects of the studied solutions can be transferred into practical design principles and workflow guidance for similar embedded Linux systems?	97

9.3 Main Contributions	97
10. Future Work	99
10.1 OEM Closed and Hardware-Enforced AHAB Validation	99
10.2 Negative Update, Recovery, and Tamper Testing	99
10.3 dm-integrity, dm-crypt, and Device-Local Secret Storage	100
10.4 Production Key Management and Release Automation	100
10.5 Cross-Platform and Industrial Baseline Validation	100
References	103
Appendix	107
Appendix A: WIC Storage Layout	107
Appendix B: Image Recipe Configuration	107
Appendix C: Filesystem Mount Setup	107
Appendix D: RAUC Target Configuration Patch	108
Appendix E: U-Boot A/B Boot Script	108
Appendix F: RAUC Bundle Configuration for dm-verity Slots	109
F.1 Original Atlas RAUC Bundle Recipe	109
F.2 DevSec RAUC Bundle Append	110
F.3 Raw Slot Post-Install Hook	111
Appendix G: Initramfs dm-verity Metadata Override	112
Appendix H: AHAB Signing and SRKH Programming Workflow	113
H.1 U-Boot AHAB Configuration	113
H.2 CST-based AHAB Signing Configuration	113
H.3 CST-based SRK and Fuse Material Generation Script	114
H.4 SRKH Fuse Programming and Readback Commands	117
H.5 Manual AHAB Signing and ELE Validation Commands	117

Acknowledgements

The authors would like to thank their supervisors and examiner for their guidance and feedback throughout this thesis work. The authors also thank the company and its team for providing access to the development environment, hardware platform, and technical context used in the case study. Finally, the authors thank everyone who contributed with discussions, feedback, and practical support during the work.

Declaration: The authors hereby affirm that this Bachelor thesis was composed by the authors themselves and that the work herein is their own, except where explicitly stated otherwise in the text. This work has not been submitted for any other degree or professional qualification, except as specified, and it has not been published. Where other people's work has been used, whether from printed, online, or other sources, it has been carefully acknowledged and referenced.

During the preparation of this thesis, the authors used ChatGPT to support language revision, translation support for the Swedish front matter, readability, flow, tense consistency, and high-level feedback on text clarity and structure. The tool was not used to generate experimental results, execute implementation steps, fabricate references, or independently determine the technical analysis, discussion, or conclusions. Prompts were mainly phrased as requests for language revision, structural feedback, identification of unclear passages, and suggestions for how reviewer comments could be addressed. All output was reviewed, edited, and accepted or rejected by the authors, who take full responsibility for the content of the thesis.

Preface

In this thesis work, Douglas Fjällrud was primarily responsible for Chapters 2, 3, and 5. Chapters 1, 4, and 6–10 were completed jointly by Douglas Fjällrud and Axel Blomén. The division of implementation, writing, research, and design work is summarized in Table 1.

Table 1: Workload distribution between the authors

	Implementation	Writing	Research	Design
Douglas Fjällrud	30%	70%	65%	30%
Axel Blomén	70%	30%	35%	70%

List of Acronyms

Acronym	Meaning
A/B	A/B root filesystem slot update model
AHAB	Advanced High Assurance Boot
API	Application Programming Interface
BSP	Board Support Package
CI	Continuous Integration
CI/CD	Continuous Integration / Continuous Delivery
CST	Code Signing Tool
dm-crypt	Device-mapper crypt target
dm-integrity	Device-mapper integrity target
dm-verity	Device-mapper verity target
DTB	Device Tree Blob
EDK II	EFI Development Kit II
ELE	EdgeLock Enclave
eFUSE	Electronic Fuse
eMMC	embedded MultiMediaCard
FDT	Flattened Device Tree
GUI	Graphical User Interface
HAB	High Assurance Boot
HS-FS	High Security - Field Securable
HS-SE	High Security - Security Enforced
HSM	Hardware Security Module
IIoT	Industrial Internet of Things
IoT	Internet of Things
KEK	Key Exchange Key
NIST	National Institute of Standards and Technology
NXP	NXP Semiconductors
OEM	Original Equipment Manufacturer
OS	Operating System
OTP	One-Time Programmable
PARTUUID	Partition Universally Unique Identifier
PK	Platform Key
PSA	Platform Security Architecture
QFPROM	Qualcomm Fuse Programmable Read-Only Memory
RAUC	Robust Auto-Update Controller
RFC	Request for Comments
ROM	Read-Only Memory
RQ	Research Question
SDK	Software Development Kit
SEC	MCUXpresso Secure Provisioning
SLSA	Supply-chain Levels for Software Artifacts

Acronym	Meaning
SoC	System-on-Chip
SPL	Secondary Program Loader
SPSDK	Secure Provisioning Software Development Kit
SRK	Super Root Key
SRKH	Super Root Key Hash
TEE	Trusted Execution Environment
TI	Texas Instruments
TISCI	Texas Instruments System Control Interface
TPM	Trusted Platform Module
U-Boot	Bootloader used in embedded Linux systems
UUU	Universal Update Utility
UEFI	Unified Extensible Firmware Interface
VFAT	Virtual File Allocation Table
WIC	Yocto/OpenEmbedded image creation tool

Popular Science Summary

Many products that people use every day contain small computers. Examples include routers, industrial controllers, vehicles, gateways, sensors, and connected appliances. These devices often run for many years and may receive software updates after they have already been installed. This creates an important security question: how can the device know that the software it starts and later updates has not been changed by an attacker?

This thesis studies that question for Linux-based embedded systems. The central idea is secure boot, where each part of the startup process checks the next part before it is allowed to run. However, secure boot is not only one technical switch. It also depends on how cryptographic keys are created, how trust is stored in the device, how software updates are handled, and how the system can be maintained over time.

The work combines a study of standards, research, and vendor documentation with a practical case study on an NXP i.MX 93 platform. The practical system used Yocto to build the embedded Linux image, Robust Auto-Update Controller (RAUC) to handle A/B software updates, dm-verity to verify the root filesystem, and Advanced High Assurance Boot (AHAB) to protect the early boot process.

In the case study, the root filesystem was made read-only, while writable data and configuration were moved to separate partitions. dm-verity was used to check that the root filesystem had not been changed, and Robust Auto-Update Controller was adapted so that updates could still be installed into inactive A/B slots. This required careful synchronization of update metadata, because each updated root filesystem slot needs matching verification data before it can boot correctly.

The AHAB part of the work was also extended beyond a simple signing test. New key and certificate material was generated, the corresponding Super Root Key Hash fuse words were programmed into the board, and the programmed values were read back to confirm that they matched. After this, a wrongly signed boot image produced a key-hash mismatch indication, while the correctly signed boot image produced no remaining authentication events in the tested Original Equipment Manufacturer (OEM) Open state. The board was not moved to the final OEM Closed state, so the work does not prove final production enforcement, but it does show that the programmed trust material was used during AHAB authentication.

The main lesson is that secure boot should be treated as an engineering chain rather than a single boot-time check. Build artifacts, signing keys, fuse programming, update bundles, dm-verity metadata, and validation records all need to stay aligned for the system to remain secure, understandable, and maintainable.

1. Introduction

1.1 Background

Integrity and authenticity are fundamental requirements in modern embedded systems. In this thesis, an embedded system is understood as a computing system integrated into a larger product or technical environment, typically designed for a specific function rather than for general-purpose desktop use. An embedded device refers to the physical product or platform that contains such a system, including hardware, boot firmware, operating system components, application software, and persistent storage.

Many embedded devices are deployed for long periods, updated in the field, and expected to operate reliably under limited maintenance. This makes the software supply chain and the early boot process critical attack surfaces rather than simple implementation details. Secure boot addresses this problem by establishing a chain in which each stage verifies the next before execution. In current guidance, this protection is not limited to verification alone, but also includes resilience, detection of unauthorized changes, and support for recovery after compromise [1].

Cryptographic key management is central to whether such a boot chain is secure in practice. A secure boot design is only as strong as the generation, storage, provisioning, use and retirement of the keys on which it depends. National Institute of Standards and Technology (NIST) frames key management as a full lifecycle problem, while system-level guidance emphasizes that the design of the key management architecture must be aligned with the system it is meant to protect [2], [3]. In embedded systems, this is further complicated by limited resources, long product lifecycles, and vendor-specific hardware mechanisms, which makes generalized solutions difficult to apply directly [4]. Key rotation and long-term maintenance therefore become operational concerns rather than secondary implementation details [5].

These challenges are especially relevant in Linux-based embedded systems, where secure boot, provisioning, build systems, and update mechanisms must work together as a coherent whole. In this thesis, Yocto and Robust Auto-Update Controller (RAUC) are treated as part of the operational security model because they determine how system images are assembled, packaged, verified, installed, and related to boot-state decisions [6], [7]. At the same time, lower layers in the chain of trust are often vendor-specific and closely tied to hardware roots of trust, which creates a gap between general standards and real implementation practice. Recent work shows that secure boot overhead depends strongly on the chosen architecture and protection mechanism, while stronger isolation of cryptographic key material, for example Trusted Execution Environment (TEE)-backed key protection, changes implementation assumptions and system complexity [8], [9]. Long-term maintenance concerns are especially visible in key rotation for deployed Industrial Internet of Things (IIoT) devices [5]. For this reason, secure boot in embedded Linux should be studied as a combined problem of trust anchors, key management, provisioning, and long-term system maintenance rather than as an isolated boot-time feature.

1.1.1 Industrial Context and Motivation

This thesis was carried out in collaboration with an industrial partner working with embedded systems. The company develops and maintains embedded solutions in contexts where customers

may have different product requirements, hardware choices, deployment environments, and security expectations. In such settings, secure boot and integrity verification cannot be treated only as isolated implementation tasks for a single board or a single customer project. They also become part of how the company reasons about reusable engineering practices, customer guidance, long-term maintainability, and the boundary between generic security principles and project-specific adaptation.

The industrial motivation for the thesis is therefore the need for a more defined and reusable security baseline for Linux-based embedded systems. Without such a baseline, similar customer needs may lead to separate bespoke solutions, where key handling, provisioning, update design, and validation evidence are redesigned for each project. This increases engineering effort and may make it harder to argue consistently that a proposed solution is sufficient for a given class of embedded Linux products. The thesis addresses this by studying secure boot, key management, provisioning, update integrity, and artifact traceability as connected parts of one workflow-oriented framework.

The purpose is not to deliver a complete commercial security product or a universally applicable certification framework. Instead, the industrial value lies in identifying principles, trade-offs, implementation patterns, and validation evidence that can support future customer projects. The practical NXP i.MX 93 case study is used as a concrete example of how such principles appear in an embedded Linux environment, while the broader analysis is used to separate platform-specific details from more transferable design considerations.

1.2 Problem Formulation

Secure boot in embedded systems is well motivated in theory, but considerably harder to design and maintain in practice. The core difficulty is not only to verify software at boot time, but to decide how trust should be established, which keys should be used at different stages, how those keys should be provisioned, how software updates should interact with the boot chain, and how the system should remain maintainable over a long operational lifetime. Existing standards provide important guidance on cryptographic key management and platform protection, but they do not remove the practical complexity introduced by vendor-specific hardware, limited resources, and long-lived deployed devices [1]–[3]. As a result, developers must make security-critical design decisions in environments where the available mechanisms are often heterogeneous and only partly transferable between platforms [4].

This challenge is also visible in the industrial setting of the thesis. The industrial partner encounters embedded-system projects where customers may request different security properties, hardware platforms, update models, and deployment assumptions. If each project is handled as a fully bespoke secure boot solution, decisions about trust anchors, signing authority, key storage, provisioning records, update metadata, and validation evidence risk becoming inconsistent or difficult to reuse. The practical problem is therefore not only how to implement one secure boot chain, but how to structure the surrounding decisions in a way that can support repeated engineering work across similar embedded Linux projects.

This creates a practical gap between general security principles and implementation-oriented engineering decisions in Linux-based embedded systems. In particular, questions remain regarding key hierarchies, provisioning workflows, key rotation, signing authority, artifact traceability,

and the relation between secure boot and secure update mechanisms over time [5]. These issues become especially important when secure boot is integrated with real development and deployment workflows involving build systems, update frameworks, and Continuous Integration (CI) processes.

The problem addressed in this thesis is therefore how secure boot, key management, and provisioning can be analyzed and structured in a way that is consistent with documented platform mechanisms, established cryptographic key-management principles, and reproducible validation evidence, while still being practically useful for embedded Linux development. The practical case study focuses on implementation realities on an NXP i.MX 93, while the broader analysis considers which design principles and workflow structures may transfer to similar Linux-based embedded systems.

1.3 Aim and Research Questions

The aim of this thesis is to analyze how secure boot, cryptographic key management, and provisioning can be designed and applied in Linux-based embedded systems in a way that is both secure and practically maintainable. In the industrial context of the thesis, the aim is also to support a reusable security baseline for embedded Linux projects, where secure boot, update integrity, key handling, provisioning, and artifact traceability can be reasoned about consistently rather than redesigned independently for each customer case. The study combines a literature and standards review, a comparative analysis of selected solutions, and a practical case study on an NXP i.MX 93. The purpose is not only to describe existing mechanisms, but also to identify common design principles, important trade-offs, and practically useful working methods. Particular attention is given to how trust is established, how key material is managed over time, and how secure boot interacts with build, update, and deployment workflows. The thesis therefore seeks to connect general security principles with implementation-oriented engineering practice.

To address this aim, this thesis is guided by the following research questions:

1. **RQ1.** Which key management and provisioning principles are most important for secure boot in Linux-based embedded systems?
2. **RQ2.** How do selected secure boot implementations differ with respect to cryptographic design, tooling, openness, usability, and long-term maintainability?
3. **RQ3.** How can secure boot be practically integrated on an NXP i.MX 93 using Advanced High Assurance Boot (AHAB) in a way that supports structured development and deployment workflows?
4. **RQ4.** How do supporting build, update, and release workflows, such as Yocto, RAUC, and artifact-traceability routines, affect the practical security and maintainability of such a system?
5. **RQ5.** Which aspects of the studied solutions can be transferred into practical design principles and workflow guidance for similar embedded Linux systems?

1.4 Scope and Delimitations

The work is limited to secure boot, cryptographic key management, and provisioning in Linux-based embedded systems. The study focuses on principles and implementation-oriented questions that are relevant to long-lived deployed devices, rather than on embedded security in general. The work also considers how secure boot interacts with supporting build, update, and release workflows, especially Yocto, RAUC, and artifact-traceability routines. CI-related security is considered only at the level of workflow integrity, provenance, and release-process design; a complete production Continuous Integration / Continuous Delivery (CI/CD) pipeline is outside the implementation scope.

The industrial partner is kept anonymous in the report. The thesis therefore describes the company only at the level needed to explain the motivation, namely as an industrial actor working with embedded systems and recurring customer-driven security requirements. No customer-specific products, internal business processes, production credentials, private signing keys, or sensitive provisioning material are disclosed.

The thesis does not aim to provide a complete survey of all secure boot implementations or all embedded platforms. Other platforms, such as Unified Extensible Firmware Interface (UEFI) Secure Boot and solutions from other vendors, are primarily used as comparison points in the theoretical analysis rather than as targets for practical implementation. The practical work is further limited to the integration and evaluation of the selected mechanisms on the chosen platform, not to a full product-level deployment process. The practical implementation focuses on dm-verity for root filesystem integrity, while mechanisms such as dm-integrity and dm-crypt are only discussed where they clarify related design alternatives. The study is also limited by available documentation, tooling, hardware access, and the time frame of the thesis project.

Although the work is motivated by the need for a reusable industrial security baseline, the thesis does not deliver a complete customer-ready framework, certification package, or production security policy. The scope is limited to analysis, comparison, implementation-oriented workflow design, and validation of selected mechanisms in the case-study environment. The resulting principles should be interpreted as structured engineering guidance rather than as a complete security guarantee for all embedded systems.

The practical AHAB work includes generation of Code Signing Tool (CST)-based Super Root Key (SRK) and Super Root Key Hash (SRKH) material, programming and readback of SRKH fuse words on the test board, and Original Equipment Manufacturer (OEM) Open validation with both matching and mismatching AHAB-signed boot artifacts. However, the device lifecycle is not transitioned to OEM Closed. Closed-state rejection of incorrectly signed images is therefore outside the evaluated scope, and the AHAB result should be interpreted as SRKH-programmed OEM Open validation rather than as final production secure boot enforcement.

1.5 Contributions

The study makes four main contributions. First, it provides a structured analysis of secure boot, cryptographic key management, and provisioning in Linux-based embedded systems, with emphasis on the relation between general security principles and practical engineering decisions. Second, it provides a comparative evaluation of selected solutions with respect

to cryptographic design, tooling, openness, usability, and long-term maintainability. Third, it includes a practical case study on an NXP i.MX 93 using AHAB, CST-generated SRK and SRKH material, SRKH fuse programming, dm-verity, Yocto, and RAUC. The case study includes OEM Open validation of both matching and mismatching AHAB-signed boot artifacts, while leaving the final OEM Closed lifecycle transition outside scope. Fourth, it provides a workflow-oriented structure for separating private signing material, target-side verification material, dm-verity metadata, update state, and release/provisioning records. These contributions aim to support a practical and method-oriented understanding of secure boot in embedded Linux, and to provide a structured baseline that can inform future embedded Linux security work in similar industrial settings.

1.6 Thesis Outline

The remainder of this thesis is organized as follows:

1. **Chapter 2** presents the theoretical background and introduces the main concepts related to secure boot, cryptographic key management, provisioning, Linux-based embedded systems, and the implementation terminology used in the case study.
2. **Chapter 3** reviews related work, including relevant standards, research contributions, and industrial perspectives in the area.
3. **Chapter 4** describes the methodology of the thesis, including the literature and standards review, source criticism, the comparative analysis framework, platform-selection rationale, and the case study design.
4. **Chapter 5** provides a comparative analysis of selected solutions with respect to cryptographic design, provisioning, tooling, openness, usability, and long-term maintainability, and synthesizes the comparison into qualitative design-suitability guidance.
5. **Chapter 6** presents the practical case study on the NXP i.MX 93, with focus on the Atlas Yocto baseline, storage layout, dm-verity integration, RAUC-compatible A/B updates, AHAB signing, key handling, and release artifact traceability.
6. **Chapter 7** reports the main results from both the comparative analysis and the practical case study.
7. **Chapter 8** discusses the results, their implications, relevant trade-offs, transferability, limitations, and societal and ethical considerations.
8. **Chapter 9** concludes the thesis by relating the findings to the overall aim, answering the research questions, and summarizing the main contributions.
9. **Chapter 10** outlines possible directions for future work.

Accordingly, the thesis follows the problem from general principles to practical validation. Chapters 2 and 3 establish the security and workflow concerns that motivate the study. Chapter 4 turns these concerns into a research design and comparison framework. Chapter 5 applies the framework to selected secure boot solutions. Chapter 6 then applies the resulting principles in the NXP i.MX 93 case study, while Chapter 7 reports the validation results. Chapters 8 and 9 interpret these results and answer the research questions.

2. Background and Theoretical Foundation

This chapter introduces the theoretical and technical concepts needed to understand the later comparison and case study. Sections 2.1–2.7 describe the main security principles, standards, platform context, and update mechanisms. Section 2.8 then collects implementation-oriented terms used throughout the practical case study, so that later chapters can refer to them without repeatedly redefining them.

2.1 Secure Boot in Embedded Systems

Secure boot is a security mechanism that ensures that only authorized software is executed during system startup. It does so by anchoring trust in a root of trust and extending trust through a chain of trust in which each stage verifies the next before control is transferred. In NIST’s platform firmware resiliency guidance, this is part of a broader protection model in which firmware and critical data should be protected against unauthorized changes, support detection of corruption, and enable secure recovery after compromise [1]. This matters because firmware initializes the platform and a successful attack on it can disrupt normal operation or introduce malicious functionality before higher software layers are active. Secure boot should therefore be understood as an early enforcement point for platform integrity rather than as an isolated feature.

The effectiveness of secure boot depends on how cryptographic trust is established and maintained. Verification is typically based on digital signatures, or in some designs trusted hashes, which means that the platform must have access to trusted key material before mutable firmware is allowed to run [1]. This ties secure boot directly to provisioning, because public keys, key hashes, certificates, or equivalent metadata must be introduced and protected in a controlled way. NIST treats this as part of a broader key management problem that spans the full key life-cycle, including storage, use, replacement, recovery, and destruction, rather than as a one-time boot configuration task [2], [3]. For that reason, secure boot cannot be analyzed in isolation from key management and provisioning decisions.

Secure boot becomes harder in embedded systems because the same security goal must be achieved under stronger practical constraints. Embedded devices often have limited memory and compute budgets, long service lifetimes, and vendor-specific hardware security mechanisms, which makes generic solutions difficult to transfer between platforms [4], [10]. Many such devices also depend on long-term cryptographic keys for remote management and firmware updates, which means that compromise of key material can undermine the trust chain even when the verification logic itself is sound [5], [9]. Experimental work further shows that secure boot can introduce measurable boot-time overhead, although the cost depends strongly on the chosen architecture and protection mechanism [8]. Secure boot in embedded systems should therefore be treated as a design problem that spans trust anchors, cryptographic verification, provisioning, and long-term maintenance.

2.2 Root of Trust, Key Hierarchies, and Provisioning

A secure boot design ultimately depends on where trust begins and how that trust is anchored in the device. NIST defines a root of trust as an element that provides security-critical functions

such as verification, update, recovery, or storage, and that must always behave as expected because its misbehavior cannot be reliably detected [1]. In practice, this means that the first trust anchor must be protected more strongly than the mutable software it is meant to verify. For embedded systems, this anchor is commonly placed in immutable or tightly protected hardware or firmware, such as Read-Only Memory (ROM), One-Time Programmable (OTP)-backed storage, or an isolated trusted subsystem. Arm’s trusted boot guidance follows the same logic by requiring an immutable boot stage and a root of trust public key that is stored, or represented by a protected hash, in immutable storage [11]. The root of trust is therefore not only the beginning of the chain, but also the security boundary on which the rest of the chain depends.

Key hierarchies are used because not all keys in the system should have the same role, or operational lifetime. NIST’s key-management guidance distinguishes between several key types, including trust anchors, key-establishment keys, key-wrapping keys, and derived operational keys, which reflects the need to separate long-lived authority from frequently used keys [2], [3]. In practice, a secure boot design often places a small number of highly protected root keys at the top of the hierarchy and uses them to validate certificates or delegated signing keys further down the chain. NIST explicitly notes that when the keys are stored, there is often a key-encrypting or key-wrapping hierarchy above them, with a top level key that typically requires physical protection [3]. Arm’s trusted boot specification similarly recommends separate signing keys when multiple actors are involved and treats the root key as the first link that authorizes later certificates and images [11]. A key hierarchy therefore limits the impact of key compromise and makes key replacement or delegation possible without redesigning the entire trust model.

Provisioning is the process that binds trust anchors, key material, and security policy to a specific device or product line. This step is critical because the security of the boot chain depends not only on which keys exist, but also on when they are introduced, who is allowed to introduce them, and how their integrity is protected during storage and transport [2], [3]. NIST’s framework for cryptographic key management emphasizes that trust anchors must be securely distributed and managed, while keys and metadata submitted for storage or establishment must be authenticated, integrity protected, and access controlled [3]. In embedded systems, provisioning is further complicated by supply-chain structure, since keys may be introduced by the silicon vendor, the Original Equipment Manufacturer (OEM), or another actor later in the lifecycle. Arm explicitly discusses these multi-actor cases and notes that OEM root keys may be provisioned at different stages of the supply chain, while factory provisioning methods remain highly implementation-dependent [11]. For this reason, provisioning should be treated as a core security mechanism in secure boot systems rather than as a one-time manufacturing detail.

2.3 Key Lifecycle and Rotation

Key lifecycle management determines whether a secure boot design remains trustworthy after the initial provisioning step. In NIST’s terminology, cryptographic keys are not static artifacts, but managed objects that move through defined states such as activation, deactivation, revocation, destruction, and in some cases archival [2], [3]. A central concept in this lifecycle is the cryptoperiod, which defines the time during which a key is authorized for use. NIST further notes that a key may need to be deactivated because of time, number of uses, or amount of protected data, and that revocation is required when authorized use must end before the planned cryptoperiod has expired [2], [3]. In secure boot systems, this means that key management

must include not only trusted storage and use, but also clear rules for replacement, retirement, and compromise recovery. Secure boot is therefore sustained over time by key lifecycle policy rather than by provisioning alone.

Key rotation is important because long-lived signing keys create long-lived consequences when they are exposed or misused. NIST notes that limiting a key’s cryptoperiod reduces the amount of material affected by compromise, and recommends that private signature keys should be destroyed at the end of their cryptoperiod under the stated assumption in the guidance [2]. More recent work focused on IIoT firmware and secure boot makes the same point from an implementation perspective: rotation reduces the impact of compromised keys and narrows the attacker’s window of opportunity, but it is harder to realize on devices with limited memory, limited processing capability, and long deployment lifetimes [5]. In practice, rotation also requires continuity, since devices must still verify valid software while trust is being transitioned from one signing authority to another. This makes overlap strategies, revocation handling, and compromise-recovery planning central parts of a maintainable secure boot architecture rather than exceptional edge cases.

2.4 Relevant Standards and Frameworks

The study uses a small set of standards and security frameworks as reference points for the later analysis. Their role is not to prescribe exact implementation, but to provide common vocabulary and a structured basis for reasoning about trust establishment, key management, provisioning, firmware verification, and recovery. In this work, NIST SP 800-57 Part 1 is used primarily for key-management principles and lifecycle considerations, NIST SP 800-130 for the design perspective on cryptographic key management systems, and NIST SP 800-193 for platform firmware protection, detection, and recovery [1]–[3]. These documents are complemented by Arm Platform Security Architecture (PSA) Trusted Boot and Firmware Update, which is particularly relevant for embedded platforms because it translates security goals into an architecture for trusted boot, measured boot, root-of-trust design, and firmware update handling [11].

The purpose of including these standards and frameworks is to anchor the thesis in established guidance before moving to platform-specific mechanisms and practical workflows. They make it possible to compare different solutions against common design concerns rather than only against vendor documentation or implementation convenience. This is important in embedded systems, where secure boot, provisioning, and update handling are often tightly coupled but implemented differently across platforms. For the update perspective, the thesis also draws on the architecture in Request for Comments (RFC) 9019, which highlights the relation between firmware verification, update metadata, and recovery strategies in long-lived devices [12]. Together, these sources provide the conceptual baseline for the comparative analysis and the practical case study.

2.5 Embedded Linux Context

Linux-based embedded systems provide a broader and more layered context for secure boot than many simpler embedded platforms. In such systems, trust must typically be maintained

across several interacting components, including boot firmware, bootloaders, the kernel, device-specific configuration, the root filesystem, and the update mechanism. This makes secure boot closely connected to how software artifacts are built, packaged, signed, deployed, and replaced over time rather than to startup verification alone. In this context, the build system becomes security-relevant because it defines how system images are assembled, while the update framework becomes security-relevant because it defines how authenticated software is delivered and installed in the field. For this reason, Linux-based embedded systems are an especially relevant setting for studying secure boot, provisioning, and long-term key management in a practical and operationally realistic way [6], [7], [12]. This is especially relevant when the root filesystem itself becomes part of the trusted update path, since bootloader decisions, initramfs behavior, filesystem layout, and update metadata must remain consistent across system versions.

2.6 NXP i.MX 93 and AHAB

NXP i.MX 93 is used in this thesis as the main practical case because it provides a Linux-capable embedded platform in which secure boot, provisioning, and long-term deployment concerns can be studied together. NXP presents the i.MX 93 family as an applications-processor platform for edge, industrial and Internet of Things (IoT) systems, and highlights both Linux-oriented use cases and advanced security with an integrated EdgeLock secure enclave [13]. The current i.MX Linux User's Guide also shows that i.MX 93 is part of NXP's actively supported Linux Board Support Package (BSP) environment and that Yocto is the preferred framework for building system images [14]. This makes the platform relevant not only from a hardware-security perspective, but also from a workflow perspective, since build, boot, and update concerns meet in the same software stack. The platform is therefore suitable as a bridge between the general theory in the previous sections and the implementation-oriented analysis later in the thesis.

AHAB is included in this section because it is the secure-boot architecture that structures trust establishment on i.MX 93. NXP's documentation explicitly states that AHAB is used in i.MX 93 and that it replaces High Assurance Boot (HAB) from earlier families, while the Linux BSP documentation for i.MX 93 refers to AHAB-related boot artifacts such as the `mx93a1-ahab-container.img` and the EdgeLock Enclave (ELE) firmware container [14], [15]. NXP's provisioning tooling also supports secure programming, device provisioning, and the generation of AHAB bootable images for i.MX 9 devices [16]. For this reason, AHAB is not only a platform detail, but the main mechanism through which root-of-trust design, signing, provisioning, and practical boot-image handling become visible on the chosen case-study platform.

2.7 Yocto, RAUC, and Update Security

In Linux-based embedded systems, secure boot must be understood together with the workflows that build and update the system over time. Yocto is relevant in this context because it structures how images, packages, and configuration data are assembled into deployable system artifacts, which makes the build process part of the overall trust chain rather than a neutral background activity [6]. RAUC is relevant because it focuses directly on authenticated software delivery and installation on the target system. Its documented core functions include update bundles, signing and verifying those bundles, performing robust installation, and interfacing with the

boot process [7]. Together, these tools make it possible to study secure boot not only as startup verification, but as a part of a broader operational workflow for building, signing, deploying, and maintaining embedded Linux systems.

Update security is included because a secure boot design is incomplete if the system cannot securely transition from one valid software state to another. RFC 9019 explicitly connects secure boot to firmware update by stating that the firmware verifier must verify the image it boots and that the verification metadata may be the same manifest that was obtained during the update process [12]. The same document also emphasizes that devices need a recovery strategy when update or invocation fails, for example by retaining alternative firmware images or invoking a recovery image [12]. This aligns with RAUC’s focus on robust installation and boot-process integration, and with the broader NIST view that firmware security should include protection, detection and secure recovery rather than verification alone [1], [7]. Update security is treated here as a natural extension of secure boot rather than as a separate operational concern.

2.8 Technical Terms

The practical case study uses several terms from embedded Linux, Yocto, RAUC, Linux storage, NXP AHAB provisioning, and secure boot validation. Many of these terms are short in the acronym list, but need a longer explanation because they describe concepts that later affect implementation decisions. This section therefore provides a compact reference for the terminology used in Chapters 5–7.

2.8.1 Build, Artifacts, and Update Terminology

An **artifact** is an output produced by a build, signing, update, or provisioning step. In this thesis, artifacts include boot images, FIT images, root filesystem images, RAUC bundles, dm-verity metadata files, SRK tables, fuse-material files, checksums, logs, and release records. An artifact is security-relevant when a later step assumes that it belongs to a specific build, key set, or target configuration.

An **artifact boundary** is a point where separately generated artifacts must remain consistent with each other. For example, a signed boot image must match the key material represented by the programmed SRKH fuses, and a root filesystem image must match the dm-verity metadata used to verify it. If artifacts from different builds or key sets are combined incorrectly, each artifact may be valid in isolation while the assembled system still fails to boot or fails authentication.

A **tarball** is an archive file, commonly compressed, that packages several files into one distributable object. In the case study, tarballs are relevant when key material, certificates, or generated build outputs need to be moved as a controlled package rather than as loose files.

Upstream refers to the original source project or provider from which a component, layer, recipe, or patch originates. For example, an upstream Yocto layer or upstream RAUC component is the version maintained by the original project rather than a locally modified version. Keeping track of upstream origin is important for maintainability, security updates, and reproducibility.

Workflow integrity means that the process producing and handling software artifacts is itself controlled and traceable. It concerns whether build inputs, generated images, signing steps,

checksums, provisioning records, and release outputs can be connected to the same release state. In this thesis, workflow integrity is treated as a complement to secure boot because secure boot can only verify the artifacts it is given [17], [18].

2.8.2 Yocto, RAUC, and Update Workflow Terms

BitBake is the task execution engine used by Yocto and OpenEmbedded. It reads recipes, resolves dependencies, and executes tasks that fetch, configure, compile, install, package, and deploy software components. In the case study, BitBake is relevant because it produces the root filesystem image, boot artifacts, dm-verity metadata, and RAUC bundle inputs [6].

A **recipe** is a Yocto/OpenEmbedded build description that tells BitBake how to build or package a component. A **layer** is a collection of recipes, configuration files, classes, and patches that can be combined with other layers to form a complete Yocto build. A **Board Support Package (BSP)** environment provides board-specific bootloader, kernel, device tree, firmware, and configuration support. In this thesis, the Linux BSP environment is the platform-specific foundation on which the Yocto-based case study is built [6], [14].

An **A/B root filesystem slot update model** uses two root filesystem partitions, often called slot A and slot B. The device boots from one slot while updates are installed into the inactive slot. After installation, the bootloader or update framework switches the active slot. This makes it possible to update the system without overwriting the currently running root filesystem and can support fallback if the new slot fails to boot [7].

A **RAUC bundle** is an update package used by RAUC. It normally contains one or more images, a manifest, and signature-related data. In this thesis, the bundle is important because it carries the raw dm-verity-protected root filesystem payload and the matching `dm-verity.env` metadata needed by the next boot [7].

A **hook** is a script or callback executed by a framework at a defined point in a workflow. In this thesis, the most important example is the RAUC post-install hook, which runs after a slot has been written and updates the slot-specific dm-verity metadata in the U-Boot environment [19].

A **marker file** is a deliberately placed file used as simple validation evidence. For example, if a marker file is added to an updated root filesystem image and appears after reboot, it helps show that the expected updated slot was actually booted. A marker file is not a security mechanism; it is only a practical validation aid.

2.8.3 Linux Storage, Filesystems, and Early Boot

The **root filesystem**, often shortened to **rootfs**, is the filesystem mounted at `/` after the system boots. It contains the operating-system userspace, libraries, configuration defaults, applications, and service files. In this thesis, the rootfs is made read-only and is verified through dm-verity.

Writable-state separation means separating mutable runtime data from the otherwise immutable root filesystem. Instead of allowing the rootfs to be modified during normal operation, writable data is placed on separate partitions or volatile filesystems. This reduces accidental or malicious modification of the verified rootfs and makes updates easier to reason about.

ext4 is a common Linux filesystem format. In the case study, ext4 is relevant as the filesystem type used for root filesystem images before dm-verity protection is applied [20].

VFAT, or Virtual File Allocation Table, is a filesystem format often used for boot partitions because it is simple and widely supported by firmware and bootloaders. In embedded Linux systems, a VFAT boot partition may contain kernel images, device tree files, initramfs images, or boot metadata.

fstab refers to the `/etc/fstab` file, which describes filesystems that should be mounted during boot. It is relevant when persistent partitions such as `/data` or `/config` must be mounted separately from the read-only rootfs [21].

findmnt is a Linux command used to inspect mounted filesystems. In the validation, it is useful because it can show whether `/` is mounted from `/dev/mapper/rootfs`, confirming that the dm-verity mapping is active [22].

A **tmpfiles rule** is a systemd-tmpfiles configuration entry that creates or cleans files and directories at boot. In this thesis, tmpfiles rules are relevant because writable directories under separate partitions may need to exist before services or update tools use them [23].

An **initramfs** is an initial RAM filesystem loaded early in the Linux boot process before the real root filesystem is mounted. It can contain scripts and tools needed to prepare the final rootfs. In the case study, the initramfs creates the dm-verity device-mapper target and mounts the verified root filesystem.

The **initramfs metadata override path** refers to the case-study mechanism where dm-verity metadata passed through the kernel command line overrides the fallback metadata built into the initramfs. This is needed because RAUC-installed root filesystem slots may have different dm-verity root hashes from the initially flashed image. The override path keeps U-Boot slot selection, RAUC-installed payloads, and initramfs verification metadata aligned.

A **Device Tree Blob (DTB)** is a compiled hardware-description file passed to the Linux kernel. It describes board-specific hardware such as memory, buses, peripherals, and device configuration. In a verified boot flow, the DTB is security-relevant because an attacker-modified hardware description can affect how the kernel initializes the platform.

2.8.4 Device Mapper and dm-* Targets

Device mapper is a Linux kernel framework for creating virtual block devices from one or more underlying block devices. It is used by several storage features, including integrity verification, encryption, and logical volume management. The case study uses device mapper through dm-verity [24].

dm-verity is a device-mapper target that provides transparent integrity verification for read-only block devices. It verifies blocks against a hash tree and a trusted root hash. If the data read from storage does not match the expected hash tree, verification fails. In this thesis, dm-verity extends the integrity chain from the verified boot artifacts to the mounted root filesystem [25].

dm-crypt is a device-mapper target for block-device encryption. It protects confidentiality by encrypting data at rest. It is discussed as a related mechanism, but it is not the main

implementation target in this thesis because the practical focus is root filesystem integrity rather than confidentiality [26].

dm-integrity is a device-mapper target that provides integrity metadata for block devices, especially where writable block-level integrity is needed. It differs from dm-verity because dm-verity is primarily suited to read-only verified images, while dm-integrity can support writable integrity use cases. In this thesis, dm-integrity is discussed only as a related design alternative [27].

2.8.5 AHAB, Fuses, and Provisioning Terms

A **trust anchor** is the protected starting point for a chain of trust. In secure boot, it is usually an immutable public key, a key hash, or hardware-rooted verification mechanism that later software cannot freely modify. All later verification decisions depend on this starting point [1], [3].

A **Super Root Key (SRK)** is an NXP AHAB root-key concept used to authorize later boot authentication material. In this thesis, the SRK material is generated as part of the AHAB signing and provisioning workflow. The SRK should be understood as high-authority signing material near the top of the AHAB key hierarchy, not as an ordinary runtime key.

An **SRK table** is a data structure containing one or more SRK public keys or certificates used by AHAB. A boot image can refer to a selected entry in this table. The table is tied to the trust material represented by the SRKH fuses.

An **SRK index** identifies which entry in the SRK table is used for signing or authentication. For example, using SRK index 0 means that the first SRK entry is selected. The selected index must match the intended signing key and certificate material.

Operational keys are keys used for regular operations after the root authority has been established. They may include signing keys, update keys, communication keys, or device-local credentials. They should be separated from long-lived root authority where possible, because operational keys may need to be rotated or revoked more often [2], [3].

A **fuse**, in this thesis context, is a one-time-programmable hardware storage element used by the SoC to store security-relevant configuration or key-hash material. Fuses are commonly irreversible or difficult to change after programming, which makes them suitable for anchoring trust but risky to handle incorrectly.

Fuse material is the generated data intended to be programmed into fuses. In the AHAB case study, this includes SRKH material derived from the generated SRK certificate set. The signed boot image and the programmed fuse material must be based on the same SRK material.

Fuse programming is the process of writing fuse material into the device. Because it can be irreversible, it must be done only after the generated material, signing configuration, and boot artifacts have been checked.

A **fuse table** is a representation of which fuse values should be written and where they should be written. It may be represented as generated binary data, a U-Boot command file, or a tool-specific batch file. The important point is that the table connects generated trust material to concrete fuse locations.

Fuse words are addressable units within fuse storage. In U-Boot-style notation, fuse locations may be described by bank and word numbers. A **bank** is a group of fuse words, and a word is one programmable unit inside that bank. In tool-specific notation, the same fuse words may instead be represented by linear fuse indices.

nxpele notation refers to how NXP's EdgeLock Enclave tooling addresses and programs security-related values. In the case study, the same SRKH fuse material can be viewed through U-Boot bank/word notation or through **nxpele** fuse indices. The notations differ, but they refer to the same underlying programmed fuse words [16].

A **batch command file** is a file containing a sequence of tool commands that can be executed together. In the AHAB provisioning workflow, a batch command file can be used by tooling such as **nxpele** to write several fuse words in one controlled operation.

OEM Open state is a device lifecycle state in which authentication events can be observed, but authentication failure does not necessarily prevent the board from continuing to boot. In this thesis, OEM Open validation is used to check whether the programmed SRKH material participates in AHAB authentication without permanently moving the device to the final closed lifecycle state.

OEM Closed state is a later lifecycle state in which the device is expected to enforce the programmed secure boot policy more strictly. In this state, an incorrectly signed or mismatched boot artifact should be rejected by the hardware-rooted boot process. The practical case study does not transition the device to OEM Closed.

Closed-state boot rejection means that an invalid or incorrectly signed boot artifact is rejected when the device is in a closed enforcement state. This is stronger than observing an authentication event in OEM Open. In this thesis, closed-state rejection is outside the evaluated scope.

An **open event** is an authentication or status event observed while the device remains in OEM Open. Such an event can provide evidence that AHAB checked a boot artifact, but it is not the same as final production enforcement.

Positive OEM Open validation means that the correctly signed boot artifact produces no remaining authentication failure under the tested OEM Open conditions. **Negative OEM Open validation** means that a mismatched or wrong-key boot artifact produces an authentication failure indication while the device remains in OEM Open. A **negative boot test** is a broader validation test where an intentionally invalid or inconsistent artifact is used to confirm that the system detects or rejects the failure condition.

2.8.6 UEFI, TI, and Rockchip Comparison Terms

In UEFI Secure Boot, **db** is the allowed signature database and **dbx** is the forbidden or revoked signature database. The **db** database contains certificates or hashes that may authorize boot components, while **dbx** contains revoked or forbidden entries. Together with Platform Key (PK) and Key Exchange Key (KEK), these databases make UEFI Secure Boot a policy-driven trust model [28].

HS-FS means High Security - Field Securable in the Texas Instruments secure boot context.

It describes a lifecycle state where the device supports security provisioning but is not yet in final enforced mode. **HS-SE** means High Security - Security Enforced, where the secure boot policy is enforced by the device. These terms are used in the comparative analysis to contrast TI lifecycle handling with NXP OEM Open and OEM Closed concepts [29].

rkbin refers to Rockchip-provided binary boot components and related tooling used in Rockchip boot flows. In this thesis, it is relevant mainly as an example of platform-specific boot material where public documentation and implementation details can be more fragmented than for some other comparison targets [30].

2.8.7 How These Terms Relate in the Case Study

The main implementation chain in the case study can be summarized as follows. Yocto and BitBake produce boot, root filesystem, and update artifacts. AHAB signing and SRKH fuse programming bind the early boot artifact to hardware-rooted trust material. U-Boot selects a RAUC A/B slot and passes slot-specific dm-verity metadata to the kernel. The initramfs uses this metadata to create `/dev/mapper/rootfs` through dm-verity. The verified root filesystem is then mounted read-only, while writable runtime and configuration state are kept outside the rootfs. RAUC updates the inactive rootfs slot and uses a hook to synchronize the matching dm-verity metadata for the next boot.

This terminology is important because many validation results in the later chapters depend on consistency across these boundaries. A successful update is not only a question of whether RAUC writes an image to a partition. The updated rootfs, dm-verity metadata, U-Boot environment, initramfs behavior, signed boot artifacts, and provisioning records must all refer to the same intended system state.

3. Related Work

3.1 Standards-Oriented Work

Standards-oriented work has established much of the conceptual foundation on which secure boot, key management, and provisioning are discussed today. In particular NIST SP 800-57 frames cryptographic key management as a lifecycle problem, NIST SP 800-130 treats key management as a system-design problem, and NIST SP 800-193 extends the discussion from secure verification to broader platform firmware resiliency, including protection, detection, and recovery [1]–[3]. For embedded and device-oriented settings, Arm PSA Trusted Boot and Firmware Update provides a more implementation-facing framework that connects root-of-trust design, secure boot, measured boot, and firmware update requirements [11]. RFC 9019 complements this perspective by describing firmware update as an architectural problem that includes verification, manifest handling, and recovery strategy [12]. Taken together, these documents define a strong normative baseline for how trust, key material, and firmware integrity should be managed over time.

At the same time, standards-oriented work is necessarily general and does not remove the practical complexity of real embedded platforms. The standards clarify what properties a secure system should provide, but they do not fully determine how those properties should be realized across vendor-specific boot architectures, provisioning models, build systems, and update workflows. This creates a gap between normative guidance and implementation-oriented engineering practice, especially in Linux-based embedded systems where secure boot must interact with image construction, signing pipelines, deployment, and long-term maintenance. For this reason, standards-oriented work is highly valuable in this thesis as a reference framework, but not sufficient on its own as a guide for practical system integration. It therefore serves as a foundation for the more implementation-focused related work and comparative analysis that follow.

3.2 Research on Secure Boot and Key Management

Research on secure boot and key management has mainly focused on how general security principles can be realized under the practical constraints of embedded systems. One recurring theme is performance, where secure boot must strengthen integrity protection without introducing unacceptable startup overhead. Profentzas et al. show that this trade-off is real but highly dependent on the selected design, with relatively low overhead for a software-based approach and significantly higher overhead when additional hardware-backed cryptographic storage is introduced [8]. A second recurring theme is key protection, where long-lived device credentials must remain usable for remote management and firmware updates without becoming easily extractable from compromised devices. Papp, Zsombor, and Buttyán address this problem by proposing TEE-based protection of cryptographic keys on embedded IoT devices, aiming to provide stronger protection without requiring a separate secure co-processor [9].

A third line of research focuses on structure and maintainability rather than on a single mechanism. Jank, Schmidt, and Koschuch argue that secure key management in embedded systems is difficult to generalize because the area is shaped by heterogeneous platforms, constrained resources, and differing application requirements [4]. More recent work by Joshi et al. shows

that this complexity becomes especially visible in key rotation, where long-lived firmware-signing trust must be updated without breaking the secure boot chain or making the system operationally unmanageable [5]. Taken together, the literature shows that secure boot and key management are not solved by one preferred architecture or one preferred storage mechanism. Instead, the research points to a set of recurring trade-offs between security strength, performance, deployability, and long-term maintenance, which motivates the comparative and implementation-oriented approach taken in this thesis.

3.3 Industrial Practices and White Papers

Industrial white papers are relevant in this thesis because they capture implementation-facing concerns that are often less visible in standards and more fragmented in academic research. In particular, they tend to emphasize provisioning workflows, operational key handling, organizational responsibilities, and the practical interaction between secure boot, firmware signing, and long-term maintenance. The HCLTech whitepaper used in this work is useful in this regard because it presents embedded key management as a practical engineering problem spanning key generation, storage, provisioning, rotation, and disposal [31]. Community- and industry-oriented secure boot documents are also valuable when they explain how these principles are realized in common embedded boot chains rather than only at a conceptual level. Such sources therefore help contextualize the gap between general security guidance and the realities of deployed embedded systems.

At the same time, industrial and community-oriented documents should not be treated in the same way as standards or peer-reviewed research. Their main value lies in showing how practitioners structure the problem, which trade-offs they consider important, and which implementation assumptions recur in real systems. For example, the prpl Foundation secure boot whitepaper describes a typical embedded boot sequence built around hardware root of trust, ROM code, Secondary Program Loader (SPL), a bootloader, and authenticated Linux startup, which makes it especially relevant as a Linux-oriented implementation perspective [32]. In this thesis, such sources are therefore used to supplement the standards-oriented and research-oriented literature rather than to replace it. They strengthen the practical framing of the study, but the core analytical claims still rest primarily on standards, official documentation, and peer-reviewed research.

Recent community- and industry-driven software supply-chain frameworks are also relevant to the practical security context of this thesis. In particular, Supply-chain Levels for Software Artifacts (SLSA) provides a structured way to reason about build integrity, provenance, and progressively stronger supply-chain guarantees [17], [18]. The SLSA framework is not a secure boot mechanism in itself, but it is relevant because it strengthens the workflow that produces, signs, and distributes the software artifacts that secure boot and secure update mechanisms ultimately rely on. It therefore complements the secure boot literature by making the build and release process itself more transparent and verifiable. In this thesis, SLSA is used as a supporting reference for workflow integrity, provenance, and CI-related security considerations rather than as a core theoretical source.

3.4 Research Gap

The reviewed literature provides a strong foundation for understanding secure boot, cryptographic key management, and provisioning but it does not fully close the gap between general security guidance and practical system integration. Standards and frameworks define important principles for key lifecycle management, trust establishment, firmware verification, and recovery, while research studies address specific problems such as performance overhead, key protection, and key rotation [1]–[3], [5], [8], [9]. Industrial and community-oriented sources further clarify how these issues appear in practice, especially in embedded Linux environments [31], [32]. However, the existing material is still fragmented across standards, academic studies, and implementation-oriented documentation. As a result, it remains difficult to derive a coherent and practically useful approach for secure boot and key management in Linux-based embedded systems.

This gap is especially visible when secure boot is considered together with provisioning, build workflows, secure updates, and long-term maintenance on a concrete platform. Existing work gives useful principles and practical solutions, but offers limited guidance on how these elements should be combined in a structured and maintainable way in a real embedded Linux setting. This is particularly relevant for platform-specific environments such as NXP i.MX 93, where vendor-specific boot architectures, provisioning mechanisms, and tooling must be understood together with broader security principles. The gap addressed in this thesis is therefore not the absence of secure boot theory, but the lack of an integrated and practically grounded analysis that connects standards, prior research, comparative evaluation, and implementation-oriented experience. This motivates the thesis approach of combining literature and standards review, comparative analysis, and a practical case study.

4. Methodology

4.1 Research Design

This thesis uses a combined qualitative research design to study secure boot, cryptographic key management, and provisioning in Linux-based embedded systems. The study is structured around three complementary parts: a literature and standards review, a comparative analysis of selected solutions, and a practical case study on NXP i.MX 93. This design was chosen because the problem addressed in the thesis is both conceptual and implementation-oriented. On the one hand, it requires an understanding of existing standards, security principles, and prior research. On the other hand, it requires practical examination of how these principles can be realized in a concrete embedded Linux environment. The research design therefore aims to connect general guidance with platform-specific engineering practice.

The literature and standards review establishes the theoretical and normative foundation of the study. The comparative analysis identifies recurring design principles, important trade-offs, and practical differences between selected secure boot and key-management approaches. Finally, the case study examines how these ideas can be applied on a specific platform, using an NXP i.MX 93 and AHAB, with attention to related workflows involving Yocto, RAUC, and operational security considerations. The three parts are intended to support each other rather than to function as isolated activities. Together, they make it possible to study both what secure boot systems should achieve and how such systems can be integrated and maintained in practice. The overall design is therefore intended to produce a result that is both analytically grounded and practically useful.

4.2 Literature and Standards Review

The literature and standards review was used to establish the conceptual and normative basis of the thesis. Its purpose was not only to identify how secure boot, cryptographic key management, and provisioning are described in prior work, but also to determine which principles and design concerns recur across different types of sources. The review therefore included standards and security frameworks, peer-reviewed research, official vendor documentation, tool documentation, and selected industrial or community-oriented material. These source types were included because the topic spans both normative guidance and implementation-oriented practice. In particular, standards were used to identify security principles, peer-reviewed research was used to capture focused technical findings and recurring trade-offs, and official platform and tool documentation was used because the thesis addresses practical integration on a concrete embedded Linux platform.

The review was qualitative and synthesis-oriented rather than systematic in the sense of an exhaustive database review. The intention was to survey the sources most relevant to the research questions and the selected case-study context, not all available literature on secure boot or embedded security. This was important because the field includes a mix of general standards, platform-specific mechanisms, and workflow-specific tools that are not always discussed within the same body of literature. The review was consequently used as a method for building a coherent analytical framework across otherwise fragmented material.

4.2.1 Search Strategy and Selection

Sources were identified through targeted searches using combinations of secure boot, embedded Linux, key management, provisioning, update security, platform names, and tool names. The searches were not limited to one database because the thesis required both academic literature and implementation-oriented documentation. Academic and standards-oriented material was searched for using general technical search tools and reference lists from relevant papers and standards. Platform-specific material was searched for through official vendor documentation portals, tool documentation, and targeted web searches for the selected platforms and mechanisms. The search terms were combined and refined during the review. Examples of search terms and term combinations are included in Table 2.

Table 2: Search focus and example search terms

Search focus	Example search terms
General secure boot and embedded systems	secure boot embedded Linux, trusted boot embedded systems, secure boot key management embedded systems
Key management and provisioning	cryptographic key management embedded systems, secure boot provisioning, trusted boot key hierarchy, key rotation secure boot IIoT
Firmware update and recovery	firmware update security embedded Linux, secure firmware update recovery, RFC 9019 firmware update architecture
Yocto, RAUC, and root filesystem integrity	Yocto secure boot, RAUC A/B update dm-verity, dm-verity embedded Linux, dm-verity RAUC Yocto
NXP i.MX 93 and AHAB	NXP i.MX 93 AHAB secure boot, i.MX 93 SRKH fuse programming, SPSDK AHAB provisioning, NXP AHAB SRK table
Comparison platforms	UEFI Secure Boot PK KEK db dbx, TI AM64x secure boot HS-FS HS-SE, Qualcomm Linux secure boot QFPROM, Rockchip secure boot rkbin eFUSE
Supply-chain and workflow integrity	SLSA provenance software supply chain, build artifact provenance, software artifact traceability secure boot

Sources were selected when they were directly relevant to one or more research questions, provided authoritative or technically useful information, and helped explain either general principles or concrete implementation mechanisms. Priority was given to primary and authoritative sources, especially standards, official documentation, and peer-reviewed publications. Industrial and community-oriented documents were used more selectively, mainly to contextualize practical workflows and recurring implementation assumptions. Sources were excluded or treated cautiously when they were only indirectly related, lacked technical detail, were not traceable to a clear publisher, or could not support claims beyond general context.

4.2.2 Source Criticism

The sources used in the thesis differ in authority, purpose, and level of implementation detail. For that reason, they were not treated as equivalent. Standards and security frameworks, such as NIST guidance and RFC 9019, provide a strong basis for terminology, security goals, and lifecycle reasoning, but they are intentionally general and do not prescribe all platform-specific implementation choices [1]–[3], [12]. They are therefore most useful for defining principles and evaluation criteria, not for proving that a specific embedded Linux implementation is complete.

Peer-reviewed research was used to support claims about secure boot overhead, embedded key protection, key management challenges, and key rotation [4], [5], [8], [9]. These sources strengthen the academic basis of the thesis, but they often focus on specific mechanisms or experimental settings. They therefore cannot fully replace platform documentation when analyzing vendor-specific boot chains or provisioning workflows.

Official vendor and tool documentation was necessary for the practical and comparative parts of the work because mechanisms such as AHAB, SRKH fuse programming, TI secure boot, Qualcomm secure boot, UEFI Secure Boot, Yocto, RAUC, and dm-verity are defined through platform or project-specific documentation [6], [7], [14], [16], [25], [28], [29], [33]. Such documentation is authoritative for how a tool or platform is intended to work, but it may also omit assumptions, simplify workflows, or differ in depth between vendors. The thesis therefore uses vendor documentation primarily to describe documented mechanisms and engineering workflows, while avoiding claims about undocumented behavior.

Industrial and community-oriented material was used as supporting context where it clarified practical secure boot workflows, key-management practices, or software supply-chain concerns [17], [18], [31], [32]. These sources can be valuable because they reflect practical engineering concerns, but they are generally less independent than peer-reviewed research or formal standards. They are therefore used to complement the standards, research literature, and official documentation rather than to carry the main analytical claims.

Some web-based sources, especially product pages, project documentation, and vendor documentation pages, may change over time. For this reason, web references are used mainly when they are the official or most relevant source for a platform, project, or tool. Where possible, the bibliography includes access dates, and long-term traceability can be improved further by adding archived URLs for web references. This is especially relevant for documentation-heavy topics where future readers may need to verify which version of a source was used.

4.3 Comparative Analysis Framework

The comparative analysis framework was developed to enable a structured evaluation of selected secure boot and key-management solutions. Its purpose is to ensure that the comparison is guided by explicit analytical criteria rather than by platform-specific description alone. This is important because the solutions considered in this thesis differ not only in implementation details, but also in how they establish trust, manage cryptographic material, support provisioning, and remain maintainable over time. The framework was therefore designed to connect the research questions with the recurring themes identified in the literature, standards, and platform documentation. In this way, the comparative analysis functions as an intermediate step between the general

background of the thesis work and the practical case study on NXP i.MX 93.

The comparison is based on six main criteria: cryptographic design, trust establishment and provisioning, key lifecycle and maintainability, tooling and workflow integration, openness and documentation, and usability and implementation complexity. Cryptographic design covers the use of recognized mechanisms and how they are applied in the boot chain. Trust establishment and provisioning address how the root-of-trust is anchored and how key material is introduced into the system. Key lifecycle and maintainability concern rotation, replacement, revocation, and the long-term handling of trust relationships. Tooling and workflow integration cover how well the solution fits with practical development and deployment processes, including build and update workflows. Openness, documentation, and usability are included because they affect how accessible, understandable, and realistically adoptable a solution is in engineering practice. Together, these criteria provide a structured basis for comparing different solutions in a way that is both security-relevant and practically oriented.

The comparative analysis is qualitative rather than score-based. The purpose is not to rank the selected solutions by a single numerical value, because the platforms differ in scope, documentation depth, lifecycle model, and intended deployment context. A numerical score would require explicit weights and comparable empirical measurements that are outside the scope of the thesis. Instead, the comparison evaluates each solution against the same criteria and identifies relative strengths, limitations, and suitable use cases. Where documentation is incomplete or fragmented, this is treated as an analytical finding under openness, documentation, and usability rather than as missing data to be hidden. The later comparative synthesis therefore uses qualitative categories and explanatory reasoning rather than a weighted benchmark.

4.4 Selection of Platforms / Solutions

The selected platforms and solutions were chosen to support the aim of the thesis rather than to provide a survey of secure boot implementations. NXP i.MX 93 with AHAB was selected as the main case-study platform because it combines a Linux-based deployment context with a vendor-specific secure boot architecture, provisioning support, and publicly available development documentation [13], [14], [16]. In addition to this main case, a smaller set of comparison points was included to represent different secure boot models and practical design choices in embedded and adjacent system environments. These comparison points include UEFI Secure Boot and selected vendor-specific approaches from Texas Instruments (TI), Rockchip, and Qualcomm [28]–[30], [33]. Together they provide a relevant basis for comparing differences in trust establishment, provisioning, tooling, openness, workflow integration and long-term maintainability.

The selection was guided by four main considerations. First, each selected solution had to be relevant to the research questions and to the practical problem of secure boot in Linux-based or closely related systems. Second, the selection was intended to include both broadly adopted reference models and more vendor-specific secure boot architectures, in order to capture meaningful variation rather than only superficial differences. Third, official documentation and sufficiently clear technical material had to be available, since the comparison in this thesis is based on documented mechanisms rather than reverse engineering or undocumented behavior [28]–[30], [33]. Fourth, the number of selected solutions had to remain limited enough to allow

analytical depth while still providing a broader comparative context around the NXP i.MX 93 case. The selection is therefore intentional and comparative in character, not exhaustive.

Rockchip is retained in the comparison for a narrower methodological purpose than the other comparison points. The public Rockchip material used in this thesis is more fragmented and less complete as a secure-boot lifecycle source than the documentation available for NXP, UEFI, TI, and Qualcomm. It is therefore not used as an equally deep case for provisioning workflow, key lifecycle, or end-to-end secure boot maintenance. Instead, Rockchip functions mainly as a contrast case for public documentation visibility, tooling transparency, and the practical effect of limited public material on engineering usability. This makes Rockchip relevant to the openness and documentation criterion, but limits the strength of any conclusions about one consolidated Rockchip secure-boot model.

4.5 Case Study Design

The case study was designed to examine how secure boot, key management, provisioning, and update-related integrity protection can be structured in a real Linux-based embedded system. It uses an NXP i.MX 93 as the practical platform because the platform combines a vendor-specific secure boot architecture with Linux-oriented tooling, Yocto-based image generation, and documented provisioning support [13], [14], [16]. The case study is not a greenfield implementation of an entire embedded product. Instead, it starts from Atlas, a company-provided Yocto-based embedded Linux baseline supplied by the industrial partner for this thesis. In this report, Atlas refers to that internal integration baseline rather than to a public framework, external standard, or open-source project. It already provides a bootable image, U-Boot integration, and a basic RAUC A/B update flow. The methodological purpose is therefore to study how additional security properties can be added to an existing embedded Linux baseline without losing maintainability. Since Atlas is company-internal case-study material, it is described where needed for reproducibility and context, but it is not cited as an external source. The case study therefore operationalizes the comparison criteria from Chapter 4 by examining how trust establishment, maintainability, workflow integration, and usability appear in a concrete Yocto-based implementation.

The implementation work is organized around three practical work packages that correspond to the main case-study concerns. The first work package concerns the secure boot and verified root filesystem chain: AHAB-based early boot authentication, the bootloader handover, dm-verity-protected root filesystem slots, and the interaction between these mechanisms during slot selection. The second work package concerns key handling and provisioning: which keys are created during the build, which trust anchors or verification artifacts are placed on the target, which keys may be fleet-wide or device-specific, and how SRKH fuse programming and final lifecycle closure should be separated from ordinary build and update steps. The third work package concerns writable state: how the system separates immutable operating-system content from persistent runtime data, private configuration, and possible future overlay or encryption mechanisms. Together, these work packages connect the case study to the thesis questions about trust establishment, key lifecycle, update workflows, and practical maintainability.

The dm-verity and RAUC parts of the case study were evaluated through incremental implementation in the Atlas Yocto build. First, the root filesystem was made read-only, persistent

runtime state was moved to a dedicated data partition, and a separate configuration partition was introduced for private or device-local configuration. The root filesystem slots were then changed to use dm-verity-protected images, and the RAUC configuration was adapted so that update state was stored on the writable data partition while rootfs slots were treated as raw update targets. This made it possible to examine how verified root filesystems and A/B update logic interact in practice. The detailed implementation steps are described in Chapter 6, while the observed boot behavior, update behavior, and validation evidence are reported in Chapter 7.

Validation evidence in the case study was collected from implementation artifacts, boot logs, command output, target-side inspection, and release/provisioning records. For the storage and update parts, the evidence includes filesystem mount information, dm-verity mapping state, RAUC status, U-Boot environment variables, slot-specific metadata, and update logs. For the AHAB and provisioning parts, the evidence includes generated SRK/SRKH material, SRKH fuse readback, the signed boot artifact used in the flashing workflow, and AHAB/ELE event inspection in OEM Open. These outputs are used as functional integration evidence rather than as performance measurements or as a complete adversarial security evaluation.

The AHAB and key-management work is treated as a parallel but connected part of the case study. Development signing, CST-based key generation, SRK table generation, SRKH fuse material, and SRKH fuse programming are part of the implementation scope because they determine how trust is anchored on the i.MX 93. The case study therefore includes a separate provisioning checkpoint: the generated SRKH fuse words are programmed on the target, read back from U-Boot, and then used as the basis for OEM Open validation with both matching and mismatching signed boot artifacts. The device lifecycle is not transitioned to OEM Closed, since lifecycle closure is an irreversible production boundary that should only be performed after the rest of the chain has been validated. This distinction keeps the case study practically testable while still aligning the implementation with the final secure boot model.

The case study does not aim to deliver a complete production deployment process. It is limited to the integration and analysis of the mechanisms that are most relevant to the thesis: AHAB-oriented signing, SRKH provisioning, and OEM Open validation, dm-verity root filesystem integrity, RAUC-compatible A/B updates, writable-state separation, and key-management decisions that affect long-term maintainability. Questions such as full production CI, large-scale device provisioning, storage encryption for all persistent data, and final product recovery policy are considered only to the extent that they influence the secure boot and provisioning design.

4.6 Validity, Limitations, and Threats to Validity

The validity of this thesis depends on how well the chosen methods support the research questions and how consistently the different parts of the study are connected. A strength of the research design is that it combines a literature and standards review, a comparative analysis, and a practical case study. This makes it possible to examine the problem from more than one perspective and to relate general security principles to implementation-oriented observations. The use of standards, official documentation, and peer-reviewed research further strengthens the analytical basis of the study. At the same time, the practical case study provides a way to test whether concepts that appear clear in documentation remain workable in real embedded Linux systems. The overall validity of the thesis therefore relies on the interaction between conceptual

analysis and platform-specific practical examination.

The study also has clear limitations. It is limited to Linux-based embedded systems and is centered on an NXP i.MX 93 as the practical case platform. Other secure boot architectures are included primarily as comparison points and are not implemented or evaluated at the same depth. The comparative analysis is therefore intentionally selective rather than exhaustive, and the practical findings should not be interpreted as representative of all embedded platforms. The study is further limited by available documentation, available hardware, and the time frame of the thesis project. As a result, some workflow-related or operational aspects may be analyzed at a design level rather than through full end-to-end deployment.

Several threats to validity must also be considered. Vendor documentation can be detailed but may reflect assumptions, abstractions, or omissions that are not fully visible until implementation. Documentation depth also differs between the selected comparison points. This creates an asymmetry in the comparative analysis, especially for platforms where public material is fragmented or incomplete. The thesis mitigates this by basing the comparison on documented mechanisms, by treating limited documentation as part of the openness and usability analysis, and by avoiding strong claims where the available material does not support them.

Practical observations from a single case study may reduce the direct generalizability of some conclusions, especially where vendor-specific boot mechanisms or tooling strongly influence the outcome. The NXP i.MX 93, AHAB, SPSDK/CST tooling, U-Boot environment handling, Yocto image structure, RAUC configuration, and dm-verity metadata flow all shape the implementation. The study therefore distinguishes between transferable design principles, such as artifact traceability and writable-state separation, and platform-specific implementation details, such as exact fuse handling, command syntax, and AHAB lifecycle behavior.

There is also a source-related validity threat because some relevant references are web-based and may change over time. Official documentation and stable standards were prioritized where possible, but product pages, tool documentation, and community-oriented sources may still be updated or moved. This affects long-term traceability rather than the immediate technical reasoning of the thesis. Access dates and, where possible, archived URLs should therefore be used in the final bibliography for web-based references.

Finally, implementation priorities and time constraints may affect how deeply some supporting topics, such as update-failure recovery, writable storage handling, encryption, or complete production CI/CD, can be explored. There is also a risk that usability and workflow assessments are partly shaped by the specific development environment used in the case study. For this reason, the thesis does not aim to claim a universally optimal solution, but to provide a structured and practically grounded analysis whose conclusions should be interpreted within the stated scope.

5. Comparative Analysis of Existing Solutions

5.1 Comparison Criteria

The comparative analysis in this chapter is guided by the criteria defined in Chapter 4. These criteria are used to examine how the selected solutions differ with respect to cryptographic design, trust establishment and provisioning, key lifecycle and maintainability, tooling and workflow integration, openness and documentation, and usability and implementation complexity. The purpose of this structure is to make the comparison systematic and transparent rather than descriptive only. The analysis is qualitative in nature and focuses on identifying recurring design choices, meaningful differences, and practical trade-offs across the selected solutions.

Before the criteria are applied in detail, Table 3 summarizes the comparison cases used throughout the chapter. The table is intended as a reading aid: it collects the different boot and trust models before the following sections compare them from separate perspectives.

Table 3: Overview of comparison cases used in Chapter 5

Comparison case	Main model	Role in the comparison
UEFI Secure Boot [28]	Standardized firmware-policy model based on PK, KEK, db/dbx, and authenticated variable handling.	Reference model for open, updateable trust policy.
NXP i.MX 93 / AHAB [13], [14], [16], [34]	Hardware-rooted signed-image model based on AHAB, SRK material, ELE interaction, and provisioning tooling.	Main practical case and example of vendor-specific SoC secure boot.
TI AM64x secure boot [29], [35], [36]	SoC boot-chain model with HS-FS to HS-SE provisioning, eFUSE/OTP material, signed boot stages, and key-revision support.	Example of documented provisioning, lifecycle handling, and anti-rollback support.
Qualcomm secure boot [33], [37], [38]	Hardware-fused root-certificate model extended through Linux and UEFI-related secure boot guidance.	Example of layered hardware-rooted trust with Linux/UEFI workflow support.
Rockchip public material [30], [39], [40]	Publicly documented secure boot/eFUSE support distributed across different Rockchip SoC documents.	Used mainly to assess documentation visibility and openness limits.

Because the selected platforms differ in how much public documentation is available, the depth of the comparison is necessarily asymmetric in some sections. This asymmetry is treated as part of the comparison of openness, documentation, and usability, rather than as a hidden assumption. The analysis therefore focuses on documented mechanisms and on what can be reasonably compared from public material. Rockchip is the clearest example of this asymmetry: because

the public Rockchip material used here comes from different official documents and System-on-Chip (SoC) contexts, the Rockchip comparison should be read mainly as an assessment of public documentation visibility rather than as a complete evaluation of one consolidated secure-boot lifecycle model.

5.2 Cryptographic Design Choices

Cryptographic trust is structured differently across the selected solutions. UEFI Secure Boot is based on an authenticated key hierarchy in which a Platform Key (PK) authorizes Key Exchange Keys (KEKs), which in turn authorize updates to signature databases such as db and dbx [28]. By contrast, NXP AHAB is centered on a signed boot-image model in which an SRK-based chain is used to authenticate the boot image and the associated key material on AHAB-enabled devices [13], [14]. TI's documented AM64X flow is also more tightly bound to the SoC boot chain, since SPL verifies U-Boot proper and U-Boot then verifies and decrypts later boot artifacts such as the kernel, Device Tree Blobs (DTBs), and initramfs [29]. Qualcomm follows a similarly device-anchored model, where secure boot begins from hardware-fused trust and is extended through the software stack, with additional support for UEFI Secure Boot in later stages [33]. The solutions therefore differ not only in cryptographic primitives, but also in whether trust is primarily expressed through firmware-managed key databases or through vendor-specific signed-image chains rooted in hardware.

The solutions also differ in the scope and flexibility of their authenticated objects. UEFI Secure Boot is comparatively policy-oriented because it is designed around authorized and forbidden signature databases that can be updated through authenticated variable handling [28]. This makes the mechanism relatively flexible at the firmware-policy level, even though the underlying trust still depends on strongly protected non-volatile key storage [28]. NXP AHAB, TI secure boot, and Qualcomm secure boot are more appliance-like in the sense that they tie trust more directly to signed boot artifacts and hardware-rooted authorization decisions [13], [29], [33]. In addition, some of these platforms explicitly combine authentication with cryptographic confidentiality features such as encrypted boot or decryption of later boot artifacts, which extends the design beyond authenticity alone [13], [29]. The comparison in this section therefore concerns both what is authenticated and how much cryptographic policy can be changed without redesigning the trust model.

The reviewed sources also vary in how clearly they expose each cryptographic design publicly. UEFI provides a comparatively explicit and standardized model of key roles, signature databases, and update authorization [28]. TI and Qualcomm also document their boot chains in a way that makes the cryptographic trust path visible, although the solutions remain vendor-specific and closely coupled to platform firmware and SoC security architecture [29], [33]. NXP provides a documented AHAB-based approach for i.MX 93, but much of the practical detail is distributed across product pages, user guides, and security tooling rather than one single compact specification [13], [14], [16]. Public Rockchip material is more limited at this level, although official documentation does indicate support for secure boot and security-oriented platform features [30], [39]. This still makes Rockchip's cryptographic design harder to compare in equal depth from openly available official documentation alone than the models documented for UEFI, TI, Qualcomm, or NXP. For this reason, the most meaningful comparison in this section is not which solution is "best," but how differently the platforms structure trust, signature

authority, and authenticated boot artifacts at the cryptographic level.

5.3 Provisioning Models

Provisioning is one of the clearest points of separation between the selected solutions. UEFI Secure Boot uses a firmware-managed enrollment model in which the PK, KEKs, and signature databases such as db and dbx are stored as authenticated variables, with update rules that depend on whether the platform is in setup mode or user mode [28]. By contrast, the SoC-oriented solutions from NXP, TI and Qualcomm tie provisioning directly to hardware-rooted trust. NXP describes secure programming and device provisioning for i.MX 9 devices through Secure Provisioning Software Development Kit (SPSDK) and MCUXpresso Secure Provisioning (SEC) tooling, including an i.MX 9x workflow in which signed AHAB boot images are prepared and device fuses can be written as part of trust establishment [16], [34]. TI documents secure boot for AM64x secure devices as tied to a High Security - Security Enforced (HS-SE) provisioning model, where High Security - Field Securable (HS-FS) devices are converted to HS-SE by programming customer keys into Electronic Fuse (eFUSE)-backed device state using an OTP keywriter. TI's Texas Instruments System Control Interface (TISCI) documentation further describes OTP/eFUSE handling for root-of-trust and customer-programmable material [29], [35], [41]. Qualcomm similarly describes secure boot as enabled by blowing Qualcomm Fuse Programmable Read-Only Memory (QFPROM) fuses containing the hash of the root certificate [33]. Taken together, the reviewed material shows that the provisioning models differ not only in mechanism, but also in where authority is anchored and how early it becomes fixed in the device lifecycle.

The provisioning model directly affects how much control developers retain after the device has entered operation. UEFI's authenticated-variable model is comparatively update-oriented because key enrollment and signature-database management are built into the firmware model itself [28]. The vendor-specific SoC models are less flexible in that sense, since trust is strongly tied to production-time provisioning decisions and hardware-bound authorization state [16], [29], [33]. At the same time, these hardware-rooted models can provide a stronger boundary between mutable software and trust-anchor configuration, which is valuable in long-lived embedded devices. NXP's provisioning model is particularly relevant to this thesis because it combines hardware-rooted trust establishment with practical image-generation and deployment workflows [16], [34]. In this thesis, provisioning is therefore treated as both a security mechanism and a workflow design choice.

The clarity of provisioning documentation also varies across the compared platforms. UEFI provides a standardized and openly specified enrollment model through authenticated variables and signature databases [28]. TI and Qualcomm also describe hardware-rooted provisioning in an implementation-visible way, especially through eFUSES and QFPROM-based trust establishment [29], [33], [42]. NXP provides comparatively strong tooling support and public workflow documentation for i.MX 9 provisioning, but the relevant details are distributed across tooling pages and processor-specific workflow documentation rather than one single compact provisioning specification [16], [34]. For this reason, the comparison in this section is most useful when read as a comparison of provisioning models and operational assumptions rather than as a uniform feature checklist.

5.4 Key Storage and Protection

Key storage is implemented under different protection assumptions across the selected solutions. UEFI Secure Boot stores its trust anchors as firmware-managed variables, and the specification explicitly requires the PK to be kept in non-volatile storage that is tamper and delete resistant, while KEKs must be stored in non-volatile tamper-resistant storage [28]. The vendor-specific SoC solutions instead rely more heavily on hardware-rooted storage. TI's documentation describes customer root-of-trust keys as programmed through the keywriter flow into extended OTP / eFUSE-backed device state [29], [35]. Qualcomm similarly describes QFPROM fuses as storing the cryptographic keys that authenticate software images during secure boot [33]. For the i.MX 93, the storage model is more layered, since the platform combines fuse-based trust establishment with an integrated EdgeLock Secure Enclave that supports protected key storage and security services [13], [43], [44]. As mentioned before, the public Rockchip material reviewed here is more limited. However, the official RK1808 documentation does indicate secure eFuse access in secure mode and support for secure boot [40].

The comparison shows that key protection must be evaluated separately from signature-verification correctness. A secure boot chain may be cryptographically sound, but still fragile if long-lived keys or trust anchors are stored with weak isolation or broad software exposure. UEFI's model is comparatively explicit and standardized, but it is primarily centered on protecting public trust databases in firmware storage rather than on exposing a dedicated hardware keystore for private operational keys [28]. For the i.MX 93, the storage model is more layered, since the platform combines hardware-rooted trust establishment with an integrated EdgeLock Secure Enclave. The EdgeLock Hardware Security Module (HSM) Application Programming Interface (API) describes a keystore in which symmetric and private keys are stored, which makes the platform relevant for protected key-handling discussions [13], [44]. This makes the NXP model particularly relevant in the context of this thesis, because the reviewed public material points to a combination of hardware-rooted trust establishment and a more isolated protection model for sensitive key material. In a broader embedded context, this also aligns with prior research that motivates TEE-based or similarly isolated handling of long-lived device keys [9].

The reviewed public material exposes these protection boundaries with different levels of clarity. UEFI specifies its storage requirements in a clear and standardized way, which makes the trust-boundary assumptions easy to identify [28]. TI and Qualcomm also document hardware-rooted protection with fairly visible references to OTP, eFUSE, or QFPROM-based key storage [33], [35]. NXP provides a stronger public story for layered protection than many vendor-specific solutions, because it documents both secure provisioning workflows and secure-enclave-backed key handling, although the details are spread across several documents rather than a single compact specification [16], [34], [43], [44]. For this reason, the comparison in this section is most informative when it focuses not only on where keys are stored, but also on how strongly each solution appears to separate trust-critical key material from ordinary mutable software.

5.5 Key Rotation and Long-Term Maintenance

Long-term maintenance exposes some of the clearest differences between the compared trust models. From a key-management perspective, this matters because trust relationships are

not static over the lifetime of a deployed system. NIST treats revocation as a lifecycle function through which a key is removed from operational use before the planned end of its cryptoperiod, and recent embedded-security research highlights key rotation as a central requirement for long-lived devices rather than an operational refinement [2], [5]. UEFI Secure Boot provides an explicitly maintainable model in this respect, since PK, Key Exchange Key (KEK), and the signature databases such as db and dbx are managed as authenticated variables that can be updated under defined authorization rules [28]. TI's AM64x security documentation provides an even more explicit hardware-rooted maintenance model, since HS-SE devices can be provisioned with backup keys and support runtime key-revision updates that switch the root-of-trust from the primary key set to the programmed backup keys [35], [36]. TI also documents software-revision checks against eFUSE-backed values, which provide a mechanism commonly used to prevent rollback to older images [45].

These differences have direct operational consequences. UEFI's model is comparatively flexible because revocation and trust updates are built into the firmware-policy layer itself through authenticated updates to PK, KEK, db, and dbx [28]. TI's model is less flexible in day-to-day administration, but more explicit in lifecycle planning, since key count, key revision, backup keys, and anti-rollback checks are all tied to hardware-rooted state and documented runtime transition procedures [35], [36], [41], [45]. Qualcomm's Linux security documentation indicates a layered approach in which lower secure boot remains tied to QFPROM-rooted trust, while Linux-image authentication can additionally use a UEFI Secure Boot model with PK, KEK, db and dbx management [33], [37]. In the public material reviewed for the NXP i.MX 93, the maintenance story is more strongly centered on provisioning, secure programming, and hardware-rooted trust establishment than on one compact, publicly documented post-deployment key-transition procedure [13], [16], [34].

Another important difference lies in how clearly long-term trust maintenance is exposed in public documentation. UEFI and TI provide the clearest public descriptions of how trust can be updated, constrained, or migrated without redefining the entire platform security model [28], [36], [45]. Qualcomm also presents an understandable layered model, but the relevant details are distributed across several Linux security and UEFI-related documents rather than one compact lifecycle specification [33], [37]. For the NXP i.MX 93, the public documentation reviewed here is strong on provisioning and platform security features, but less explicit on lifecycle-oriented trust transition in one consolidated public workflow [13], [34]. Considering the reviewed public material, this section is best read as a comparison of maintenance models rather than as a yes-or-no checklist for key rotation support. In practical terms, the comparison suggests that the most mature solutions are not only those that establish trust well at manufacturing time, but also those that provide a clearer path for trust change, revocation, and recovery over the operational lifetime of the device.

5.6 Tooling, Openness, and Usability

From an engineering perspective, the compared solutions are separated as much by their tooling ecosystems as by their security models. UEFI Secure Boot benefits from an openly available specification and a mature open-source implementation ecosystem through TianoCore EFI Development Kit II (EDK II) [28], [46]. NXP provides a more guided vendor toolchain through the SEC Tool, with support for AHAB image generation, key and certificate handling, trust

provisioning, and secure programming [16]. TI exposes a broad set of public security guides and utilities through its TISCI documentation, including signing, keywriter, extended OTP, key revision, and related security topics [47]. Qualcomm likewise provides public Linux security documentation and SecTools v2 for image signing, fuse-blower generation, and secure debug policy generation [33], [38]. Rockchip is more open in a source-availability sense than in a tooling-guidance sense: the boot documentation points to source code such as upstream or Rockchip U-Boot and rkbin, but gives a less complete public secure-boot tooling workflow than the other ecosystems [30].

Workflow integration is where the practical differences become most visible. The Yocto Project is explicitly designed to build custom Linux systems through BitBake, layers, metadata, and image generation, while RAUC expects integration with both host-side bundle generation and target-side update handling and explicitly recommends a Linux build system such as Yocto/OpenEmbedded or PTXdist for well-defined software states [6], [48]. In that context, NXP's public tooling is relevant because it offers both a Graphical User Interface (GUI) path and scriptable Software Development Kit (SDK) path that fit naturally into structured build and provisioning workflows [16]. TI and Qualcomm also expose practical tooling for secure boot enablement, but their workflows are more tightly coupled to their own SDK and platform-security environments [33], [38], [47]. UEFI is highly portable as a model, but in practice its usability depends on which firmware implementation, signing tools, and platform conventions are available around it [28], [46]. This means that tooling usability in this comparison is less about whether a mechanism exists and more about how naturally it fits repeatable engineering workflows.

Documentation quality and openness ultimately shape how approachable each solution is for development teams. UEFI/TianoCore, Yocto, and RAUC are comparatively strong in openness and community visibility, but they also assume a broader firmware or Linux build background from the user [6], [46], [48]. NXP offers a guided public experience through SEC, SPSDK, and processor-specific workflows, although the relevant details are spread across several pages and tool documents rather than one consolidated integration guide [16]. TI's public material is detailed and task-oriented, but is also strongly SoC- and SDK-specific, which can make transfer between platforms less straightforward [47]. Qualcomm's public security guidance is useful and broad, but it remains more fragmented and ecosystem-specific than the UEFI/TianoCore or Yocto/RAUC documentation reviewed here [33], [38]. For this reason, the comparison in this section is most useful as an assessment of how much engineering support, openness, and workflow clarity each solution appears to offer in practice.

The comparison above shows that the selected solutions differ along several dimensions at the same time. A solution may be strong in trust-anchor design but less transparent in public tooling, or flexible in policy management but dependent on a suitable firmware ecosystem. For this reason, the following section synthesizes the comparison qualitatively and relates the different designs to the types of engineering situations in which they are most appropriate.

5.7 Qualitative Synthesis and Design Suitability

The preceding sections compare the selected solutions from separate perspectives: cryptographic design, provisioning, key storage, lifecycle support, tooling, openness, and usability.

This section synthesizes those observations into a qualitative matrix and a purpose-oriented design interpretation. The matrix is not a numerical benchmark. The selected solutions differ in scope, documentation depth, hardware assumptions, and ecosystem maturity, so adding weighted scores would imply a degree of precision that the available material does not support.

The qualitative categories used in Table 4 should be read as follows. ++ means that the reviewed material shows strong and clearly documented support for the criterion. + means that the criterion is supported, but with relevant assumptions, platform coupling, or workflow limitations. 0 means that the criterion is context-dependent or not a primary strength of the model. - means that the reviewed public material gives limited support or limited visibility for the criterion. These categories are not additive and should not be summed into an overall ranking.

Table 4: Qualitative synthesis matrix for the reviewed secure boot solutions

Comparison case	Trust-anchor clarity	Provisioning guidance	Lifecycle support	Workflow integration	Public documentation	Transferability
UEFI Secure Boot	++	+	++	0/+	++	++
NXP i.MX 93 / AHAB	++	++	+	++	+	0/+
TI AM64x secure boot	++	++	++	+	+	0/+
Qualcomm secure boot	++	+	+	+	0/+	0/+
Rockchip public material	0	-	0/-	-	-	0/-

The matrix shows that there is no single best solution across all criteria. UEFI Secure Boot is strongest as an open and policy-oriented trust model, especially where updateable firmware trust policy is important. NXP i.MX 93 / AHAB is strongest for the practical case-study context because it combines hardware-rooted boot authentication with platform-specific provisioning tooling that can be integrated into a structured embedded Linux workflow. TI AM64x is comparatively strong where a documented device lifecycle, provisioning transition, and key-revision model are central requirements. Qualcomm provides a layered hardware-rooted model with Linux and UEFI-related secure boot support, but the publicly reviewed material is more ecosystem-specific. Rockchip is not treated as an equally complete lifecycle model in this thesis; its main value is as a contrast case showing how limited public tooling and lifecycle documentation reduce engineering usability.

5.7.1 Suitability by Engineering Purpose

The question of which design is preferable therefore depends on the engineering purpose rather than on a universal ranking. Table 5 summarizes the most defensible interpretation based on

the comparison in Sections 5.2–5.6.

Table 5: Purpose-oriented interpretation of the compared designs

Engineering purpose	Preferable design direction	Why it fits	Main caveat
Open firmware-policy management and updateable trust databases	UEFI Secure Boot	It provides a standardized model with PK, KEK, db, and dbx, and is well suited when the platform already uses a UEFI-style firmware environment.	Practical usability depends on the available firmware implementation, signing tools, and platform conventions.
NXP i.MX 9 based embedded Linux products	NXP AHAB with structured provisioning, Yocto integration, and update integrity mechanisms	It fits the case-study platform, supports hardware-rooted signed boot artifacts, and can be connected to Yocto, RAUC, dm-verity, and release-artifact traceability.	The implementation details are vendor-specific, and OEM Closed enforcement was outside the evaluated scope of this thesis.
Products requiring a clearly documented SoC lifecycle transition and key-revision support	TI AM64x secure boot	The reviewed TI material is comparatively explicit about HS-FS to HS-SE provisioning, eFUSE/OTP handling, and key-revision concepts.	The workflow is strongly tied to TI devices, TI tooling, and TI’s platform-security environment.
Qualcomm-based Linux or UEFI-related products	Qualcomm secure boot	It is relevant when the target product is already inside the Qualcomm ecosystem and needs hardware-fused root-certificate trust combined with Linux or UEFI-related boot support.	The public material is useful but fragmented and less transferable outside Qualcomm-specific workflows.
Assessing the effect of limited public documentation	Rockchip public material	It illustrates how source availability or scattered documentation is not the same as a complete secure-boot engineering workflow.	It should not be used in this thesis as the basis for strong conclusions about one consolidated Rockchip secure-boot lifecycle model.

Engineering purpose	Preferable design direction	Why it fits	Main caveat
Reusable embedded Linux security baseline across customer projects	Hardware-rooted secure boot combined with signed updates, read-only verified rootfs, writable-state separation, and artifact/provisioning traceability	This combines the transferable principles visible across several solutions with the practical needs of long-lived Linux-based embedded systems.	Exact signing, fuse, lifecycle, and recovery mechanisms remain platform-specific.

For the industrial motivation of this thesis, the final row is the most important interpretation. A reusable embedded Linux security baseline should not be defined as one vendor mechanism alone. It should instead define the properties that a customer project needs to preserve: a clear trust anchor, separated signing authority, controlled provisioning, authenticated update flow, root filesystem integrity, writable-state separation, and traceable release artifacts. Vendor mechanisms such as AHAB, UEFI Secure Boot, TI secure boot, or Qualcomm secure boot can then be evaluated by how well they support that baseline in a specific project.

5.7.2 Common Design Principles

The comparison points to three recurring design principles across the selected solutions. First, secure boot consistently depends on an explicit and strongly protected trust anchor. The specific mechanisms differ, but all reviewed solutions rely on an early point in the system that is treated as more trustworthy than the mutable software it authorizes. This is consistent with the broader guidance in NIST SP 800-193 and NIST SP 800-130, but the comparison in this thesis also shows that the practical expression of this principle varies substantially across platforms [1], [3]. The first common principle is therefore not a single preferred implementation, but the need for a clearly defined and well-protected trust anchor that structures the boot chain.

Second, secure boot only becomes practically meaningful when provisioning and lifecycle handling are designed together. Across the reviewed solutions, trust depends not only on initial verification, but also on how key material is introduced, maintained, constrained, and, when necessary, replaced or revoked over time [2], [5], [12]. The comparison also shows that long-term maintainability varies significantly between solutions, especially in how clearly they support updates to trust relationships after deployment. In this thesis, lifecycle support is therefore treated as part of the security model itself rather than as a secondary operational concern.

Third, practical secure boot quality depends heavily on workflow support. Secure boot is easier to adopt, evaluate, and maintain when image generation, signing, provisioning, update handling, recovery planning, and release evidence are supported by sufficiently clear tooling and documentation [6], [7], [31]. This does not mean that openness alone determines security, but it does mean that engineering transparency affects how reliably a solution can be integrated into repeatable workflows.

Taken together, these findings provide the comparative basis for the practical case study in

the following chapter. The case study therefore does not attempt to prove that NXP AHAB is universally preferable. Instead, it uses NXP i.MX 93 / AHAB as a concrete platform for applying the transferable principles identified in the comparison: hardware-rooted trust, controlled provisioning, update-aware integrity verification, and traceable artifact handling.

6. Practical Case Study on NXP i.MX 93

This chapter describes the implementation work carried out on the NXP i.MX 93 case-study platform. The chapter focuses on source-level integration, storage layout, boot flow, update configuration, and key-related implementation structure. Observed behavior, validation logs, and pass/fail results are reported separately in Chapter 7. The chapter first describes the Atlas-side integrity and update chain, since this was the part most tightly integrated into the Yocto image layout and RAUC workflow. AHAB signing and hardware-rooted provisioning are then described separately because they form a parallel trust-establishment layer around the early boot image.

6.1 Hardware, Baseline Project, and Evaluation Environment

The implementation was developed on an NXP i.MX 93 target platform using Atlas as the starting point. Atlas is an internal Yocto-based embedded Linux baseline provided by the industrial partner for this thesis. It is not treated as a public framework or external source in this report; instead, it is the company-provided baseline on which the case-study implementation builds. The Atlas baseline already contained a bootable Linux image, U-Boot integration, an embedded MultiMediaCard (eMMC) image layout, and a working RAUC-based A/B update flow. The practical work in this thesis therefore extends an existing embedded Linux baseline rather than constructing a complete boot and update architecture from scratch. This distinction is important for the case study, because many implementation choices concern how security properties can be added without breaking an existing build, boot, and update workflow. Table 6 summarizes the case-study environment and integration baseline.

The company-provided Atlas baseline provides the main product image with `atlas-demo-image` and the update bundle with `atlas-demo-bundle`. The work in this thesis adds a separate layer, `meta-devsec`, for the security- and integrity-related changes. This layer is used for the modified Yocto/OpenEmbedded Image Creator (WIC) layout, image-level configuration, read-only root filesystem behavior, dm-verity integration, RAUC configuration changes, and target-side mount configuration. The external `meta-security` layer is added to supply the dm-verity image class and initramfs support used by the implementation. Keeping these additions in a separate layer makes the changes easier to inspect and keeps most security-specific integration separate from the baseline Atlas layer.

Table 6: Case-study environment and integration baseline

Area	Configuration or role in the case study
Target platform	NXP i.MX 93 based target board, booting from eMMC
Company-provided Yocto baseline	Atlas baseline with <code>meta-atlas-demo</code> , <code>atlas-demo-image</code> , and <code>atlas-demo-bundle</code>
Added project layer	<code>meta-devsec</code> , used for storage, image, RAUC, and dm-verity integration
Added external layer	<code>meta-security</code> , used for dm-verity image generation and initramfs support

Area	Configuration or role in the case study
Existing update mechanism	RAUC A/B root filesystem update flow
Implemented storage layout	Boot partition, rootfs A, rootfs B, writable data partition, and read-only config partition
Implemented rootfs protection	Yocto read-only rootfs and dm-verity-protected A/B rootfs images
Runtime writable state	Persistent runtime data under <code>/data</code> ; device-local configuration under <code>/config</code>
Boot handover	U-Boot slot selection, boot partition loading, and initramfs-based dm-verity root mounting
AHAB provisioning state	OEM Open; CST-based SRKH fuses programmed and verified by U-Boot readback; OEM Closed lifecycle transition not performed

For the AHAB part of the case study, the target was kept in OEM Open while SRKH fuse programming was nevertheless performed and verified. This means that the board was bound to the generated SRKH material for authentication checking, but the lifecycle state was not closed. The implementation therefore evaluates SRKH programming, readback, and OEM Open AHAB/ELE event behavior, while final closed-state rejection of incorrect boot images remains outside the tested scope.

The resulting Atlas-side implementation uses a shared boot partition, two root filesystem slots, a persistent `/data` partition, and a separate `/config` partition. The root filesystem is mounted read-only, `/data` is used for persistent runtime state and RAUC state, and `/config` is mounted read-only during normal operation. The two root filesystem slots are written as complete dm-verity-protected images. During boot, U-Boot selects the active RAUC slot, loads the kernel with bundled initramfs from the boot partition, passes the selected root filesystem as a Partition Universally Unique Identifier (PARTUUID), and the initramfs mounts the selected slot through `/dev/mapper/rootfs`. Baseline Atlas configuration and recipes were treated as pre-existing project material. The contribution of this thesis lies in the security-related modifications, integration decisions, and analysis described in the following sections. Figure 1 shows the physical cabling, debug setup, and FRDM i.MX 93 target board.

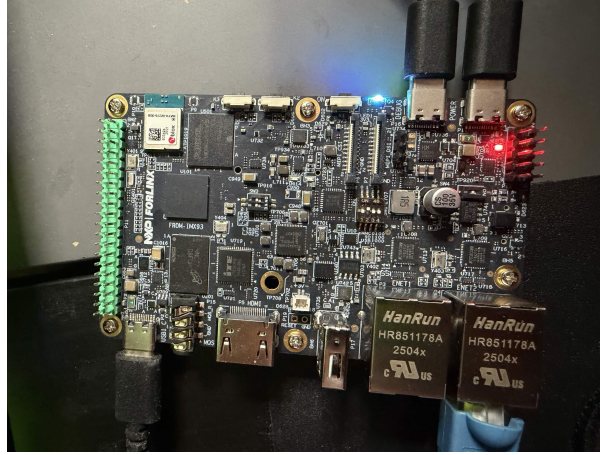


Figure 1: Physical cabling and debug setup

6.2 Implemented Boot and Integrity Handover

The case-study implementation should be read as a chain of coupled integrity boundaries rather than as a set of independent mechanisms. AHAB signing protects the early boot artifact, U-Boot selects the active RAUC slot and passes the selected root filesystem identity, the initramfs establishes the dm-verity mapping for that slot, and RAUC maintains the A/B root filesystem images and their corresponding metadata. A successful boot therefore depends on the signed boot artifact, selected rootfs image, slot-specific dm-verity metadata, and update state remaining consistent with one another.

The implemented Atlas-side boot flow is structured around a separation between boot artifacts and root filesystem contents. This section focuses on the software-level handover after early boot; AHAB signing and hardware-rooted provisioning are treated separately in Section 6.6. The boot partition contains the U-Boot boot script, kernel image with bundled initramfs, and device tree files. The root filesystem partitions contain the operating system images and their appended dm-verity hash trees. This separation allows both RAUC slots to use the same boot partition while still selecting a different root filesystem partition for A or B.

Figure 2 summarizes the implemented handover chain. AHAB authenticates the early boot artifact, U-Boot verifies the signed FIT configuration using the documented FIT signature verification model [49], the FIT authorizes the kernel, DTB, and initramfs, and the initramfs creates `/dev/mapper/rootfs` using slot-specific dm-verity metadata. RAUC is not part of the early boot authentication decision, but it affects the next boot by installing the inactive rootfs slot and updating the metadata that U-Boot later passes to the initramfs.

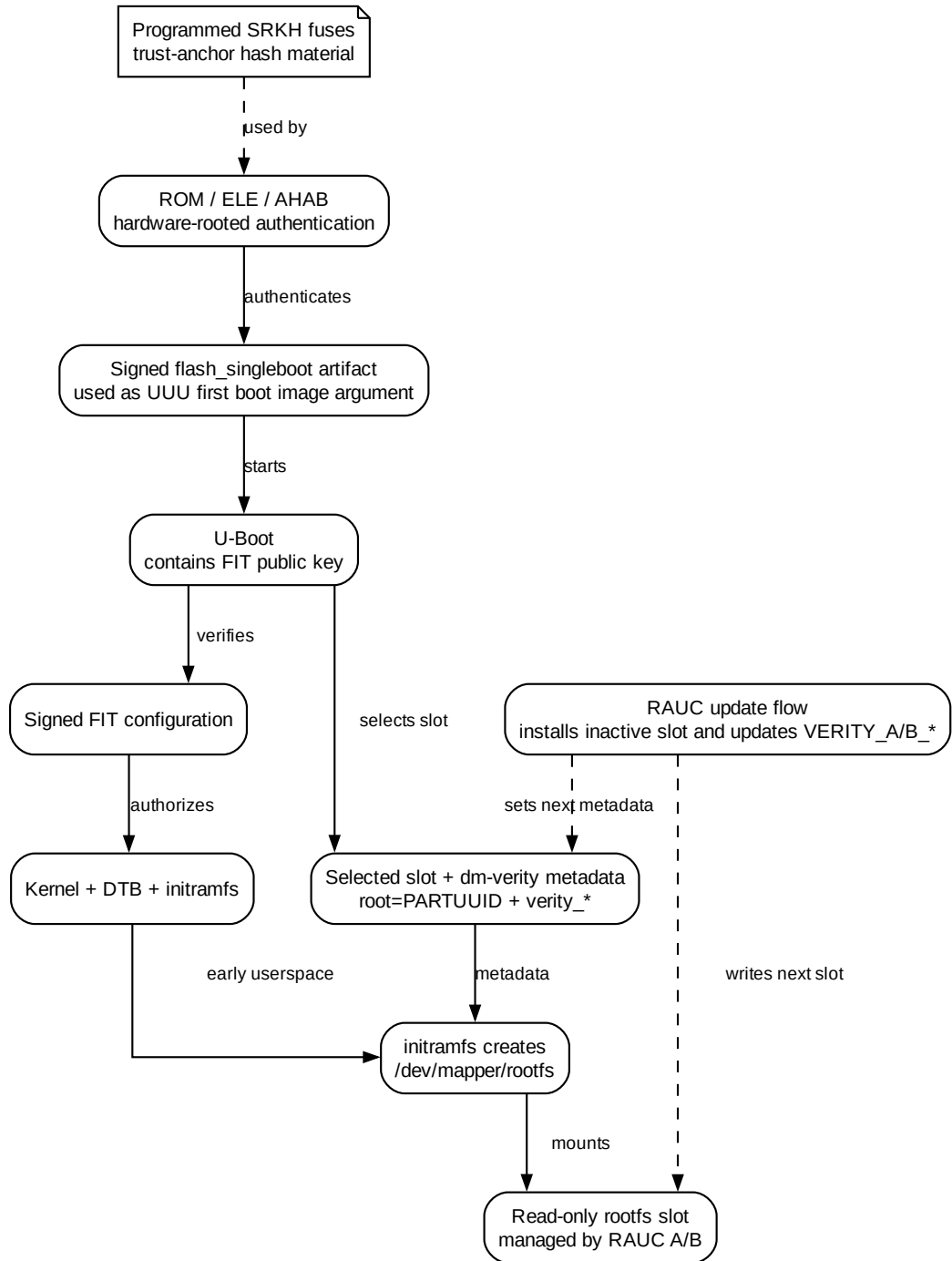


Figure 2: End-to-end secure boot, root filesystem integrity, and update chain in the case study.

6.2.1 RAUC Slot Selection in U-Boot

U-Boot is responsible for selecting which RAUC root filesystem slot is attempted at boot. The existing Atlas boot script used `BOOT_ORDER`, `BOOT_A_LEFT`, and `BOOT_B_LEFT` to choose between slot A and slot B. The implementation keeps this slot-selection model, but changes what is passed to Linux. Instead of passing a fixed block device such as `/dev/mmcblk0p2`, the boot script resolves the selected rootfs partition to its `PARTUUID` and passes it as `root=PARTUUID=...`. Slot A maps to partition 2 and slot B maps to partition 3. This gives the `initramfs` a stable identifier for the root filesystem partition selected by U-Boot.

6.2.2 Kernel and Initramfs Handover

The kernel image used for the `dm-verity` flow includes a bundled `initramfs`. The image recipe installs the `Image-initramfs-${MACHINE}.bin` artifact on the boot partition as `Image`, so that the boot script can load one kernel image that already contains the early userspace needed for `dm-verity` setup. The U-Boot boot script loads this image and the selected device tree from the boot partition using `fatload`, then starts the kernel with `booti`. The device tree load address is moved above the larger `initramfs` kernel image to avoid overlap between the kernel image and the Flattened Device Tree (FDT) in memory.

6.2.3 Root Filesystem Integrity Boundary

After the kernel starts, the `initramfs` becomes the handover point between the bootloader-selected rootfs partition and the mounted Linux root filesystem. The `initramfs` reads the `dm-verity` metadata generated at build time, creates a device-mapper target named `rootfs`, and mounts `/` from `/dev/mapper/rootfs`. The selected root filesystem partition is therefore not mounted directly as `/`. Reads from the root filesystem pass through `dm-verity`, while the root filesystem itself remains read-only.

6.3 Storage Layout and Writable State Separation

The storage layout is part of the security design because `dm-verity` requires stable root filesystem images, while RAUC requires clear slot boundaries and persistent state outside the verified rootfs. The WIC layout in `meta-devsec` therefore keeps the existing A/B update structure but adds a separate configuration partition and changes the root filesystem slots to support byte-for-byte `dm-verity` images.

Figure 3 shows how the storage layout relates to the mounted filesystem hierarchy. The active root filesystem is not mounted directly from the raw `rootfs` partition. Instead, the selected `rootfs A` or `rootfs B` partition is opened through `dm-verity` and mounted as `/` through `/dev/mapper/rootfs`. Persistent writable state is separated onto `/data`, device-local configuration is kept on `/config`, and volatile runtime state is handled outside the read-only root filesystem.

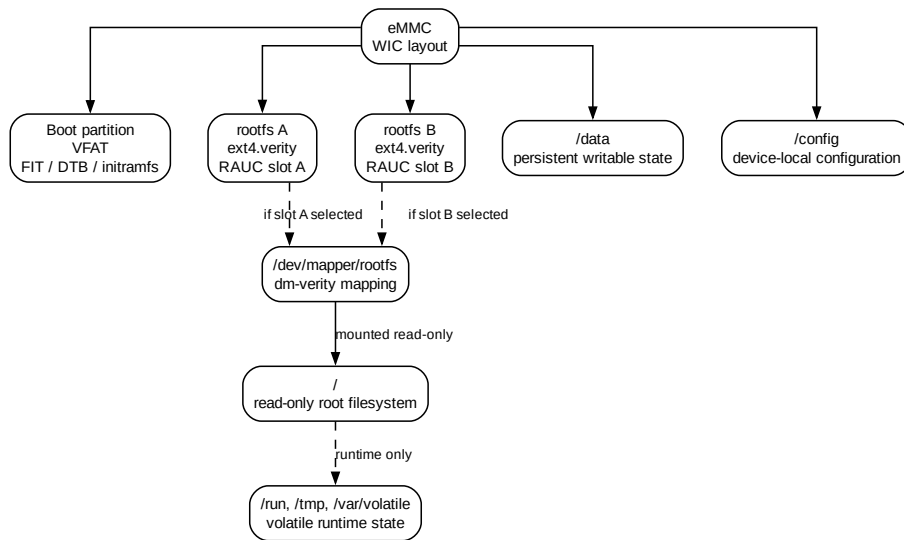


Figure 3: Storage and filesystem tree used by the practical case study.

6.3.1 WIC Layout and Partition Roles

The WIC layout defines a raw bootloader area, a boot partition, rootfs A, rootfs B, a data partition, and a config partition. The boot partition is shared by both slots and contains boot artifacts loaded by U-Boot. The rootfs A and rootfs B partitions are the RAUC-managed operating system slots. The data partition is the normal persistent writable location for runtime state. The config partition is a separate device-local configuration area mounted at `/config`.

The rootfs partitions are treated as complete image slots rather than filesystems that are modified after creation. This matters for dm-verity because the hash tree and root hash are generated over the exact block image. Recreating, relabeling, or resizing the filesystem after the hash tree has been generated would change what is verified. For that reason, WIC uses raw copying for the generated `.ext4.verity` image when populating the A/B rootfs slots. The WIC file that defines the final eMMC partition layout is included in Appendix A. The important property of this layout is that the rootfs slots are written as raw verity images, while mutable runtime state is kept outside the verified root filesystem.

6.3.2 Read-only Root Filesystem

The image is configured with Yocto’s `read-only-rootfs` image feature. This changes the root filesystem from a normal writable runtime filesystem into an immutable system image during normal operation. Runtime writes that must persist are moved out of `/` and into dedicated writable locations. This step is separate from dm-verity: read-only mounting reduces accidental or normal runtime modification, while dm-verity adds block-level integrity verification. The image-level Yocto configuration that enables the read-only root filesystem and selects the custom WIC layout is included in Appendix B. This appendix shows which image-level settings make the immutable root filesystem behavior reproducible in the build, rather than only a runtime convention.

6.3.3 Data and Config Partitions

The `/data` partition is mounted read-write and is used for persistent runtime state. RAUC state is also stored under `/data/rauc`, which keeps update state outside the read-only root filesystem and outside the verified rootfs slots. The directory is created at boot with a `tmpfiles` rule so that it exists after the separate data partition has been mounted.

The `/config` partition is mounted read-only during normal operation. It is separate from the root filesystem so that device-local or private configuration can be kept outside the Operating System (OS) image while still being protected from ordinary runtime modification. The mount configuration is installed through a `base-files` append in `meta-devsec`, which adds the `/config` mount point and the corresponding `fstab` entry. This partitioning choice provides integrity-oriented separation of configuration and runtime data, but does not by itself provide confidentiality for private material. The filesystem setup used to create the read-only `/config` mount point and the runtime-created `/data/rauc` directory is included in Appendix C. This configuration is relevant because it shows how mutable update state and device-local configuration are kept outside the verified root filesystem.

6.3.4 Writable Runtime State

Writable runtime state is handled outside the verified root filesystem. Persistent data belongs under `/data`, volatile runtime files remain in temporary filesystems such as `/run` or other RAM-backed locations, and device-local configuration belongs under `/config`. This keeps the root filesystem suitable for read-only mounting and `dm-verity` verification while still allowing the system to store update state, application state, and runtime files in controlled writable locations.

6.4 dm-verity Integration for A/B Root Filesystem Slots

`dm-verity` verifies block-device reads against a hash tree and relies on a trusted root hash to detect tampering during access [25]. The `dm-verity` integration is implemented through Yocto image generation, the WIC layout, and the U-Boot boot script. The build first produces a normal `ext4` root filesystem image, then `meta-security` generates a corresponding `.ext4.verity` image containing the root filesystem and appended `dm-verity` data. This image is used as the raw content for both rootfs A and rootfs B in the full-flash image.

6.4.1 meta-security and Image Configuration

The `meta-security` layer supplies the `dm-verity-img` image class and `dm-verity-image-initramfs` image used by the implementation. The image configuration sets the Atlas image as the `dm-verity` input image and uses `ext4` as the root filesystem type. The generated `initramfs` contains the tools and early boot logic needed to create the `dm-verity` device-mapper target before the real root filesystem is mounted.

The image recipe also installs the bundled `initramfs` kernel on the boot partition. This is required because the normal kernel image does not contain the early userspace logic needed to create `/dev/mapper/rootfs`. By placing the bundled `initramfs` kernel on the boot partition as `Image`, the existing boot flow can load a single kernel artifact while still entering the `dm-verity` setup

path. The image recipe append that selects the custom WIC layout and installs the bundled dm-verity initramfs kernel on the boot partition is shown in Appendix B. This makes the appendix part of the reproducibility evidence for the dm-verity boot path, since the initramfs must be present before the real root filesystem can be mounted.

6.4.2 Raw Verity Images in the WIC Layout

The WIC file copies the generated `.ext4.verity` image into both A and B rootfs slots using `rawcopy`. This keeps the slot contents byte-for-byte identical to the image for which the verity metadata was generated. The rootfs slots are not assigned new filesystem labels by WIC at this stage, because modifying the filesystem after verity generation would invalidate the metadata. The WIC changes that copy the generated `.ext4.verity` image into the A/B root filesystem slots are shown in Appendix A. The appendix therefore documents the build-time mechanism that prevents post-generation filesystem modification from invalidating the dm-verity metadata.

6.4.3 Boot Script Changes for A/B Verity Boot

The U-Boot boot script is changed so that both slots load the kernel and device tree from the shared boot partition while passing the selected rootfs slot by `PARTUUID`. Slot A resolves partition 2 and slot B resolves partition 3. Since the boot partition is Virtual File Allocation Table (VFAT), the boot script uses `fatload` for the kernel and device tree when booting through the dm-verity path. The script uses `booti` because the boot partition provides an ARM64 Image rather than the older `zImage` path used by the original script.

The larger initramfs kernel image also required a memory layout adjustment. The original FDT load address overlapped with the larger kernel image that includes the initramfs. The boot script therefore sets a higher FDT address before loading the device tree. This keeps the kernel image and device tree separate in memory before `booti` transfers control to the kernel. The U-Boot boot script used to select the RAUC slot, load the kernel and device tree from the boot partition, and pass the selected root filesystem by `PARTUUID` is included in Appendix E. This script is central to the implementation because it connects RAUC slot selection with the dm-verity parameters consumed later by the initramfs.

6.4.4 Initramfs Mount of `/dev/mapper/rootfs`

The dm-verity initramfs uses the selected `root=PARTUUID=...` argument to resolve the backing block partition for the active root filesystem slot. The dm-verity parameters, including the root hash and block layout values, are taken from the metadata generated during image construction. The initramfs then creates the `rootfs` device-mapper target and mounts the root filesystem from `/dev/mapper/rootfs`. The root filesystem is mounted read-only. The selected rootfs slot remains the storage backing for the mapping, but it is not mounted directly as `/`.

6.5 RAUC Integration with dm-verity Slots

The Atlas baseline already contains RAUC support and a working A/B update model. The implementation adapts the RAUC target-side configuration so that it matches the dm-verity slot layout. The central change is that RAUC state is kept on `/data`, while the rootfs slots are treated

as raw image targets rather than ext4 filesystems that RAUC mounts, resizes, or modifies as filesystems.

6.5.1 Existing Atlas RAUC Baseline

The existing Atlas baseline defines two RAUC root filesystem slots, `rootfs.0` and `rootfs.1`, mapped to the A and B boot names used by U-Boot. Before the dm-verity integration, these slots were configured as ext4 slots and RAUC could install an update to the inactive slot while the system continued running from the active one. This existing slot naming and bootloader integration is retained. The partition-level definition of the A/B root filesystem slots is included in Appendix A, while the RAUC target configuration patch that makes the slots raw update targets is included in Appendix D. Together, these appendices show how the physical A/B partition layout is matched to RAUC's logical slot configuration.

6.5.2 RAUC Data Directory and Raw Slot Configuration

The RAUC system configuration is patched by a `rauc-conf` append in `meta-devsec` when the target root filesystem is generated. The patch adds `data-directory=/data/rauc`, changes both root filesystem slots from `type=ext4` to `type=raw`, and removes `resize=true`. The data directory keeps RAUC state on the writable data partition. The raw slot type matches the dm-verity image layout, because the slot contents are complete block images that include both the ext4 filesystem and verity data. This prepares the target-side slot configuration for dm-verity-protected updates. The bundle contents and metadata synchronization are described in the following subsections. The patch is included in Appendix D because it changes RAUC from filesystem-level slot handling to raw image installation, which is required for dm-verity-protected rootfs slots.

6.5.3 Raw RAUC Bundle Contents for dm-verity Slots

The RAUC bundle was adapted so that the root filesystem payload matches the dm-verity slot layout. The original Atlas bundle recipe used the normal ext4 root filesystem image as its rootfs payload. For dm-verity slots, this was changed in `meta-devsec` so that the bundle uses the generated `.ext4.verity` image instead. This image contains both the ext4 root filesystem and the appended dm-verity data, and must therefore be written to the inactive slot as a complete block image.

The bundle append overrides the rootfs image file selected by the original bundle recipe. The source artifact remains the generated `.ext4.verity` image from the Yocto deploy output. Inside the RAUC bundle, however, the payload is renamed to a generic `.img` filename. This avoids treating `.ext4.verity` as an unsupported image extension during installation and allows the target-side `type=raw` slot configuration to install the payload as raw image data. The resulting bundle manifest therefore points to the renamed rootfs image while still packaging the dm-verity-protected artifact generated by the build.

The original bundle hook was also replaced for the dm-verity case. The previous hook copied selected files from the running root filesystem into the newly installed slot and modified files such as `/etc/shadow`. This behavior is not compatible with dm-verity, because any modification

to the installed root filesystem after the hash tree has been generated would make the image no longer match its verification metadata. The replacement hook is therefore intentionally minimal: it accepts the `slot-post-install` hook call but does not mount or modify the installed rootfs slot. Device-local state and runtime configuration are instead kept outside the verified root filesystem, mainly under `/data` and `/config`. The source-level bundle changes are included in Appendix F. Appendix F documents the boundary between update installation and verified image immutability. The slot-specific metadata synchronization needed for updates that introduce new dm-verity metadata is treated separately in the following subsection.

6.5.4 dm-verity Metadata Synchronization After Slot Installation

Each dm-verity-protected root filesystem image has metadata that must match the exact block image installed in the slot. This includes the root hash, data size, and block size values used by the initramfs when creating the `/dev/mapper/rootfs` device-mapper target. When RAUC installs a new `.ext4.verity` image to the inactive slot, the corresponding metadata must therefore be updated for that specific slot before the next boot. Otherwise, U-Boot may select the updated slot while the initramfs still verifies it with metadata from an older image.

The bundle includes the `dm-verity.env` file generated for the root filesystem payload. After RAUC installs the raw rootfs image, the post-install hook reads this metadata file from the bundle and determines which boot slot was updated. The hook uses `RAUC_SLOT_BOOTNAME` when available, with a fallback mapping from `rootfs.0` to A and `rootfs.1` to B. It then writes slot-specific U-Boot environment variables such as `VERITY_A_ROOT_HASH`, `VERITY_A_DATA_SIZE`, and `VERITY_A_DATA_BLOCK_SIZE`, or the corresponding `VERITY_B_*` variables. The hook does not mount or modify the installed root filesystem slot; it only updates bootloader metadata. RAUC supports post-install hooks after slot installation, which is used here to synchronize slot-specific dm-verity metadata without modifying the installed root filesystem [19]. The bundle append and post-install hook are included in Appendix F. This hook is central to the update flow because it ensures that each installed slot is paired with matching dm-verity metadata before the next boot.

The U-Boot boot script uses the selected slot to choose the matching metadata. For slot A, it resolves the PARTUUID of partition 2 and checks whether `VERITY_A_*` values exist in the environment. For slot B, it resolves partition 3 and checks the corresponding `VERITY_B_*` values. If the metadata is present, U-Boot adds it to the kernel command line as `verity_root_hash`, `verity_data_size`, and `verity_data_block_size`. These arguments are passed together with `root=PARTUUID=...`, which identifies the backing root filesystem partition for the selected slot. The slot-selection and command-line construction logic is shown in Appendix E.

The dm-verity initramfs was also adapted to consume this slot-specific metadata. The initramfs still contains a built-in `dm-verity.env` file, which acts as a fallback for full-image flashing and first boot. A `dm-verity-image-initramfs` append patches the initramfs script so that metadata passed on the kernel command line overrides the built-in values. This allows the same boot path to support both the initial flashed image and later RAUC-installed root filesystem images with different dm-verity root hashes. The initramfs patch is included in Appendix G. Appendix G defines how update-time metadata passed by U-Boot overrides the initramfs fallback

metadata during later boots. This synchronization flow keeps the installed root filesystem image, the bootloader environment, and the initramfs verification parameters aligned. RAUC writes the raw dm-verity image to the inactive slot, the hook updates metadata for that slot, U-Boot passes the selected metadata at boot, and the initramfs creates `/dev/mapper/rootfs` using values that match the selected slot. The validation output for this flow is reported in Chapter 7.

Figure 4 shows why the RAUC hook is part of the integrity handover. RAUC writes the raw dm-verity-protected rootfs image to the inactive slot, but the next boot also needs metadata that matches that exact image. The post-install hook therefore reads the bundle-provided `dm-verity.env` file and updates the slot-specific `VERITY_A_*` or `VERITY_B_*` variables used by U-Boot and the initramfs.

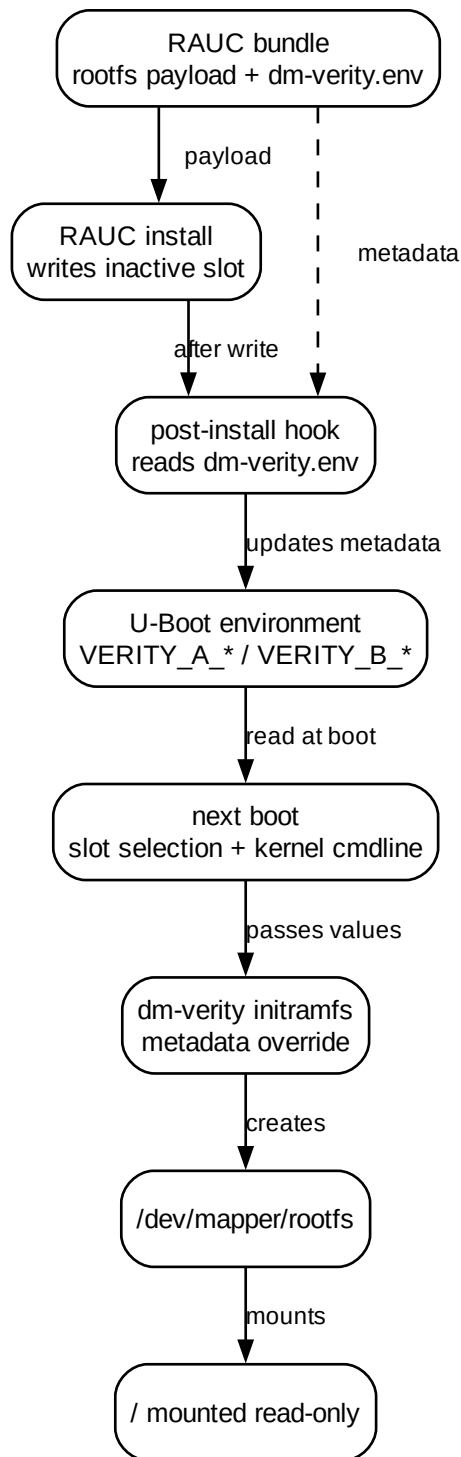


Figure 4: RAUC and dm-verity metadata synchronization flow after slot installation.

6.5.5 Failure and Recovery Assumptions

The implemented RAUC and dm-verity handover assumes that the selected slot, the U-Boot environment, the slot-specific dm-verity metadata, and the PARTUUID reference remain consistent. If these inputs are inconsistent, the prototype is expected to fail during boot or root filesystem mounting rather than silently mount an unverified or mismatched root filesystem. Table 7 summarizes the main failure assumptions. These cases were not implemented as a complete production recovery policy. They are included to clarify the boundary between the validated prototype behavior and the additional recovery design that would be required in a deployed product.

Table 7: Failure and recovery assumptions for the RAUC and dm-verity handover

Failure condition	Expected prototype behavior	Production design need
Selected PARTUUID cannot be resolved	Boot or root mounting is expected to fail because the backing rootfs partition cannot be identified.	Redundant boot metadata, explicit fallback slot policy, or recovery media.
Slot-specific VERITY_A_* or VERITY_B_* values are missing	The boot flow cannot construct complete dm-verity parameters for the selected updated slot.	Controlled fallback to known-good metadata only when it belongs to the selected image.
dm-verity metadata does not match the selected rootfs image	dm-verity verification or root mounting is expected to fail.	Recovery to previous slot or factory recovery path.
U-Boot environment is corrupted or inconsistent	Slot selection and metadata handover may fail or select stale values.	Redundant environment storage, validation of environment state, or signed boot metadata.
RAUC bundle payload and dm-verity.env do not belong together	The installed rootfs and verification metadata may no longer describe the same image.	Release-time checks that bind bundle payload, metadata, manifest, and signature together.
Rootfs data is modified after installation	dm-verity is expected to detect block-level mismatch when affected blocks are read.	Defined error handling, logging, and recovery policy.

The current implementation therefore validates the positive integration path and selected negative AHAB behavior, but it does not claim a complete recovery architecture for all boot and update failures. A production design would need to define which failures should trigger fallback to the previous slot, which should enter a recovery image, and which should require manual service intervention.

6.6 AHAB Signing and Provisioning

The AHAB part of the implementation adds signing, SRKH provisioning, and OEM Open validation of the early boot artifact used by the i.MX 93 boot flow. This complements the dm-verity and RAUC work described in the previous sections. While dm-verity protects the selected root filesystem after the kernel has started, AHAB applies to the earlier boot artifact that

is processed before U-Boot takes over. The implementation therefore treats AHAB as a separate trust-establishment layer around the existing Atlas boot image and the later dm-verity-protected root filesystem chain.

The final AHAB validation was performed after SRKH fuse programming, but while the device remained in the OEM Open lifecycle state. This means that the case study validates generation of CST-based SRK material, SRKH fuse programming, U-Boot readback, and ELE event behavior for both matching and mismatching signed boot artifacts. It does not claim full hardware-enforced secure boot, since that would require transitioning the device lifecycle to OEM Closed and then verifying closed-state boot rejection of incorrectly signed images.

6.6.1 AHAB Signing Inputs and Artifacts

The final AHAB validation used the boot artifact loaded by the Universal Update Utility (UUU) flashing flow. During the implementation work, this was found to be the `flash_singleboot` artifact passed as the first argument to the UUU `emmc_all` command, rather than only `imx-boot.tagged` or the WIC rawcopy artifact. This distinction is important because the AHAB-signed artifact must be the artifact that is actually loaded and authenticated during the flashing and boot process. Signing only the WIC rawcopy boot artifact is not sufficient for validating AHAB behavior in this UUU workflow. For the UUU `emmc_all` flashing workflow used in the case study, the AHAB-relevant boot artifact was the `flash_singleboot` image passed as the first UUU argument, not only the boot artifact stored inside the WIC image. The signed `flash_singleboot` artifact therefore had to be generated from the same CST SRK material as the programmed SRKH fuses.

Before signing, U-Boot was configured with AHAB support by enabling `CONFIG_AHAB_BOOT`. This was added through a small configuration fragment in `meta-devsec`, so that the target U-Boot build contained the AHAB/ELE support needed for event inspection. The U-Boot configuration fragment is shown in Appendix H.1. The final signing flow then used `imx-signer` together with CST-generated AHAB key material and the signing configuration shown in Appendix H.2. The signed `flash_singleboot` artifact was used as the first boot image argument in the UUU flashing command, which kept the signing artifact, flashing input, and target-side AHAB authentication path aligned.

The WIC boot artifact remains part of the release artifact structure, since it records how the full image is assembled and which boot artifacts are included in the generated image. However, the final AHAB validation should be interpreted in relation to the signed `flash_singleboot` artifact. This is the artifact that was matched against the programmed SRKH material during the OEM Open validation. The corresponding manual signing and validation commands are summarized in Appendix H.5.

6.6.2 CST-generated SRK Material and SRKH Fuse Tables

The AHAB key and fuse material used in the final validation was generated with the `ahab_gen_keys.sh` script, which replaced the earlier SPSDK-only helper script used during development. The script uses CST tooling to generate ECC P-384 / SHA-384 AHAB key and certificate material with `ahab_pki_tree.sh`, and then uses `srktool` to generate the SRK

table and SRKH fuse material. The script is included in Appendix H.3, while the sanitized signing configuration that uses the generated material is shown in Appendix H.2.

The generated CST-style key package contains key and certificate material under `ahab/dev/iMX9/keys/` and `ahab/dev/iMX9/crts/`. The most important generated artifacts are `SRK1234table.bin`, `SRK1234fuse.bin`, `uboot-fuses.cmd`, and `nxpele-fuses.bcf`. These artifacts are generated from the same SRK certificate set, which is necessary because the signed boot image and the programmed SRKH fuses must refer to the same trust anchor. The signing configuration uses the i.MX 93 family, the OEM SRK set, SRK index 0, the first SRK private key as signer, and an SRK table based on the four CST-generated SRK certificates.

The SRK table hash configuration was set to match the CST/AHAB certificate material. In the final configuration, the SRK table uses `hash_algorithm: sha384`. This matters because a mismatch between the signing metadata and the programmed SRKH material can cause AHAB/ELE to report a key-hash mismatch even when a boot artifact has been signed. Private keys, passphrase files, generated signed binaries, and exact fuse-word values are not included in the main report. They are treated as controlled provisioning material rather than normal report content.

6.6.3 SRKH Fuse Programming and Readback

After the CST-generated SRK table and SRKH fuse material had been created, the SRKH fuse words were programmed on the target using the generated `nxpele-fuses.bcf` batch file. The programming was performed through the U-Boot serial interface using `nxpele`. The command wrote the eight SRKH fuse words corresponding to the generated CST SRK material. The command structure is shown in Appendix H.4, while exact fuse-word values are omitted from the report.

After programming, the fuse values were read back from U-Boot using `fuse read` for bank 16, words 0 through 7. These words correspond to SRKH fuse indices 128 through 135 in the `nxpele` notation. The readback values matched the generated fuse material from `SRK1234fuse.bin`, `uboot-fuses.cmd`, and `nxpele-fuses.bcf`. This confirmed that the board had been programmed with the expected SRKH material before the AHAB event validation was performed.

The exact readback values are treated as provisioning records and are not reproduced in the main text. The important implementation result is that fuse programming completed successfully and that the programmed values were verified by readback. This moves the case study beyond a purely prepared provisioning workflow, while still keeping the lifecycle state open for development recovery and inspection.

6.6.4 OEM Open AHAB Validation After SRKH Programming

The AHAB validation was repeated after SRKH fuse programming while the device remained in the OEM Open lifecycle state. Two boot-image cases were evaluated. First, a mismatched boot image was used to check whether AHAB/ELE detected that the boot image did not correspond to the programmed SRKH material. In this case, the ELE event output reported a key-hash

mismatch indication. This shows that the programmed SRKH material was used during AHAB authentication.

Second, the `flash_singleboot` artifact was signed with the CST-generated key material matching the programmed SRKH fuses. When the correctly signed artifact was used, the ELE event output did not report a remaining authentication failure. In the observed output, the event count was zero after booting with the correctly signed image. The command sequence for signing and inspecting ELE events is summarized in Appendix H.5, and the validation evidence is reported in Section 7.2.6.

These results validate SRKH programming, readback, and open-state AHAB authentication behavior. They do not constitute proof of closed-state enforcement, because the device was not transitioned to OEM Closed. In OEM Open, a mismatched image can still allow the system to reach U-Boot while recording an ELE authentication indication. Closed-state rejection of incorrectly signed images is therefore not evaluated in this thesis. The exact SRKH fuse-word values are not reproduced in the main report. They are treated as provisioning records because they are device- and key-material-specific. The important implementation result is that programming completed successfully and that U-Boot readback for bank 16, words 0–7 matched the generated CST fuse material.

6.6.5 Provisioning Boundary and Lifecycle State

The implementation separates SRKH fuse programming from final lifecycle closure. Programming the SRKH fuses binds the device to the selected SRK material for AHAB authentication checking, but the device lifecycle was still left in OEM Open after the validation. This means that the work validates the provisioning material, the programming operation, readback, and AHAB/ELE mismatch detection, but not final OEM Closed secure boot enforcement.

This distinction is important for the interpretation of the case study. The system reached a stronger validation state than a purely prepared provisioning workflow, because the SRKH fuses were actually programmed and checked. However, the final production transition remains outside the scope of the thesis. A production workflow would need to combine SRKH programming with lifecycle closure, controlled key handling, release approval, and negative boot tests in the closed state.

6.7 Key and Trust Material Placement

The implementation uses several different forms of security-relevant material, but they do not all have the same purpose. AHAB uses CST-generated signing material, SRK certificate material, and programmed SRKH fuse material for early boot authentication, RAUC uses signing and verification material for update bundles, and dm-verity uses hash metadata for root filesystem integrity. The implementation separates private signing material, public verification material, programmed trust-anchor material, provisioning records, and runtime integrity metadata. This section describes where these artifacts are placed in the implemented development setup and how they are connected to the boot and update flow.

6.7.1 Implemented Key and Metadata Inventory

The implemented system contains both cryptographic key material and non-secret integrity metadata. The AHAB private signing material and RAUC private bundle signing key are host-side artifacts and are not installed on the target filesystem. The target instead contains public or non-secret material required for verification and boot-time operation, such as the RAUC verification certificate and the slot-specific dm-verity metadata stored in the U-Boot environment. Table 8 summarizes the material used in the implementation and shows which part of the system uses each artifact.

Table 8: Key, verification, and integrity material in the case study

Material	Role in the implementation	Location in the implementation	Target-side?
AHAB private signing key	Signs the AHAB boot artifact during host-side signing	Host-side CST/AHAB key workspace under <code>ahab/dev/IMX9/keys/</code>	No
AHAB SRK certificate material	Input for SRK table, signing configuration, and SRKH fuse material generation	Host-side CST/AHAB certificate workspace under <code>ahab/dev/IMX9/crts/</code>	No
AHAB SRKH fuse material	Programmed trust-anchor hash material verified by U-Boot readback	Generated from <code>SRK1234fuse.bin</code> and programmed on the target; records kept host-side	Yes, as programmed fuse state
AHAB-signed <code>flash_singleboot</code> artifact	Early boot artifact used as the first UUU boot image argument for AHAB validation	Host-side <code>deploy/signing</code> output and UUU flashing input	Yes, as boot artifact
RAUC private key	Signs RAUC update bundles	Build/signing side	No
RAUC certificate / keyring	Verifies RAUC bundles on the target	Target root filesystem	Yes
dm-verity root hash	Anchors root filesystem integrity verification	Bundle metadata and U-Boot environment	Yes
<code>VERITY_A_*</code> / <code>VERITY_B_*</code>	Slot-specific dm-verity metadata	U-Boot environment	Yes
<code>dm-verity.env</code>	Carries rootfs metadata in the RAUC bundle	RAUC bundle and <code>initramfs</code> fallback	Yes, as metadata
Device-local configuration	Stores local configuration outside the rootfs image	<code>/config</code> partition	Yes

The table reflects the implemented development arrangement rather than a final production key-management policy. Its purpose is to make the boundaries between host-side signing ma-

terial, target-side verification material, and update-time metadata explicit. The most important distinction is that private signing keys are used before deployment, while the target receives only the artifacts needed to verify or boot signed and integrity-protected images. The dm-verity values are included in the table because they are part of the boot trust flow, even though they are hash metadata rather than signing keys.

6.7.2 Host-side and Target-side Separation

The implementation keeps private signing material outside the target system. The AHAB signing key, CST-generated SRK certificate material, and RAUC private bundle signing key are used on the host or signing side to generate signed boot artifacts and signed update bundles. The target receives the resulting signed artifacts, public verification material, slot-specific metadata, and the programmed SRKH fuse state, but not the private keys used to create them. This separation is visible in the final AHAB workflow: the `flash_singleboot` boot artifact is signed on the host side with CST-generated material, used as the first UUU boot image argument, and then validated through AHAB/ELE events on the target.

The target-side material is limited to what the running system needs for verification and boot selection. RAUC uses its installed certificate or keyring to verify update bundles before installation. U-Boot stores `VERITY_A_*` and `VERITY_B_*` variables so that the selected A/B slot can be matched with the dm-verity metadata generated for that image. The `initramfs` receives this metadata through the kernel command line and uses it to create `/dev/mapper/rootfs`. In this way, target-side material supports verification and slot selection without requiring private signing keys on the device.

6.7.3 Device-local Configuration Storage

The implementation adds a separate `/config` partition for device-local configuration and mounts it read-only during normal operation. This partition is separate from the immutable root filesystem and from the writable `/data` partition used for persistent runtime state. The purpose of this layout is to avoid placing device-local configuration inside the dm-verity-protected rootfs image, while still preventing ordinary runtime writes to `/config`. The mount point and `fstab` entry are installed through the `base-files` append in `meta-devsec`.

In the implemented layout, `/config` is a storage separation mechanism rather than part of the dm-verity rootfs. The operating system image remains immutable and verified through `/dev/mapper/rootfs`, while `/data` and `/config` provide controlled locations for state that must exist outside that image. RAUC state is stored under `/data/rauc`, whereas `/config` is reserved for configuration that should not be regenerated as part of every rootfs update. This makes the storage layout compatible with A/B updates, since the rootfs slots can be replaced as complete raw dm-verity images without overwriting persistent runtime state or device-local configuration.

6.8 Build and Release Artifact Traceability

The implemented boot and update chain produces several artifacts that must remain synchronized across the build, signing, flashing, update, and provisioning workflow. The final AHAB

validation depends on the signed `flash_singleboot` artifact used by UUU, the programmed SRKH fuse material, and the ELE event inspection after boot. The WIC image and RAUC bundle remain important release artifacts, while the dm-verity-protected root filesystem image and slot-specific metadata must remain aligned with the bootloader and initramfs handover. For this reason, the implementation treats artifact traceability as part of the practical integration work rather than as a separate administrative concern.

The purpose of this section is not to report validation results. Successful boot logs, RAUC update behavior, `nxpele get-events` output, and dm-verity mount observations are reported in Chapter 7. Instead, this section describes the structure used to keep the build and signing outputs connected to their inputs. This distinction matters because a system can fail later if a valid artifact is combined with metadata from another build, even when each individual artifact was generated correctly. The release-record structure used in this case study is not a full SLSA implementation, but it follows the same general concern: making build and release relationships explicit enough to be inspected and reproduced.

6.8.1 Build-time Inputs and Generated Artifacts

The implementation uses the company-provided Atlas Yocto baseline as the build baseline and adds security-specific behavior through `meta-devsec` and `meta-security`. The main build inputs are the selected machine configuration, the Atlas image and bundle recipes, the custom WIC layout, the RAUC configuration append, the dm-verity image configuration, the AHAB signing configuration, and the generated development SRK material. These inputs define how the final boot image, root filesystem slots, update bundle, and provisioning records are produced. Keeping these inputs identifiable is important because the final flash image is not produced by one tool alone, but by a sequence of Yocto, RAUC, dm-verity, and AHAB-related steps.

The main generated artifacts are summarized in Table 9. The table separates boot artifacts, root filesystem artifacts, update artifacts, and provisioning artifacts because they are consumed at different stages of the system lifecycle. The final AHAB validation depends on the signed `flash_singleboot` artifact used by UUU and on the SRKH material programmed into the target. The WIC image, `.ext4.verity` root filesystem image, and `dm-verity.env` metadata remain separate release artifacts used by the full image and update flow. This separation makes it easier to check that each artifact is used in the correct place and that generated metadata belongs to the image it is intended to verify.

Table 9: Build-time inputs and generated artifacts

Artifact or input	Role in the implementation	Consumed by
<code>atlas-demo-image</code>	Main target image recipe	Yocto image build
<code>atlas-demo-bundle</code>	RAUC bundle recipe	RAUC bundle generation
<code>devsec-ab-storage.wks.in</code>	Defines boot, A/B rootfs, data, and config partitions	WIC image generation
<code>ahab_gen_keys.sh</code>	Generates CST-style AHAB key material, SRK table, SRKH fuse material, and fuse command files	Host-side AHAB provisioning workflow

Artifact or input	Role in the implementation	Consumed by
SRK1234table.bin	SRK table generated from the CST SRK certificate set	AHAB signing configuration
SRK1234fuse.bin	SRKH fuse material generated from the same CST SRK certificate set	Fuse programming validation records
nxpele-fuses.bcf	Batch command file for programming SRKH fuse words with nxpele	Target-side SRKH fuse programming
flash_singleboot boot artifact	Boot artifact used as the first UUU <code>emmc_all</code> argument	AHAB signing and UUU flashing
Signed flash_singleboot artifact	AHAB-signed boot artifact generated with CST-based signing material	Target boot and AHAB/ELE validation
Image-initramfs artifact	Kernel with bundled dm-verity initramfs, installed as Image on the boot partition	Boot partition
*.ext4.verity rootfs image	dm-verity-protected rootfs image	WIC rootfs slots and RAUC bundle
dm-verity.env	Metadata for the generated rootfs image	RAUC bundle and initramfs fallback
atlas-demo-bundle-*.raucb	Signed update bundle	Target-side RAUC installation

The implementation also records checksums for selected handoff artifacts. For the final AHAB validation, the most important handoff is the signed `flash_singleboot` artifact used as the first UUU boot image argument. This artifact is generated through a host-side signing workflow and must match the CST-generated SRK material used to program the SRKH fuses. The WIC image is still recorded as a full-image release artifact, but it should not be treated as the only AHAB validation boundary in this flashing workflow.

6.8.2 Signing Authority and Artifact Traceability

The implementation uses three different mechanisms to bind artifacts to a trusted or expected state. AHAB signing binds the early boot image to the development SRK/signing material. RAUC signing binds the update bundle to the RAUC certificate installed on the target. dm-verity binds a root filesystem image to hash metadata that is later passed to the initramfs. These mechanisms operate at different points in the chain, but the release workflow must keep their outputs aligned.

For the AHAB part, the final validation records distinguish between the Yocto-produced boot artifacts, the `flash_singleboot` artifact used by UUU, the CST-generated SRK/SRKH material, and the programmed target state. The signed `flash_singleboot` artifact is the critical authentication input in the final UUU flashing workflow. It must be signed with material derived from the same SRK certificate set as the programmed SRKH fuses. The record therefore connects the CST key-generation step, the SRK table, the SRKH fuse material, the `nxpele`

programming batch file, the signed `flash_singleboot` artifact, and the ELE event output collected after boot.

For the RAUC and dm-verity part, the generated `.ext4.verity` image is used as the update payload and is renamed to a generic `.img` file inside the RAUC bundle. The bundle also includes the `dm-verity.env` file generated for that root filesystem payload. During installation, the post-install hook reads that metadata and writes slot-specific `VERITY_A_*` or `VERITY_B_*` values to the U-Boot environment. This connects the bundle payload, the inactive rootfs slot, the bootloader environment, and the initramfs metadata override path.

The signing and metadata relationships are summarized in Table 10. The table does not report whether validation succeeded; it only describes which artifact is bound to which verification or integrity mechanism. This distinction keeps the section focused on implementation traceability rather than results. The corresponding runtime evidence, such as ELE event output, RAUC slot transitions, and `/dev/mapper/rootfs` mounts, is treated as validation material in Chapter 7.

Table 10: Artifact bindings in the implemented workflow

Input artifact	Binding or transformation step	Later consumer
CST SRK certificate set	<code>ahab_gen_keys.sh</code> and <code>srktool generate</code> <code>SRK1234table.bin</code> , <code>SRK1234fuse.bin</code> , and <code>nxpele-fuses.bcf</code>	AHAB signing and SRKH programming
<code>flash_singleboot</code> boot artifact	<code>imx-signer</code> signs the artifact with CST-generated AHAB material	UUU flashing and AHAB authentication
<code>nxpele-fuses.bcf</code>	<code>nxpele batch</code> programs SRKH fuse words over U-Boot serial access	AHAB/ELE authentication path
Mismatched boot artifact	Boot and ELE event inspection after SRKH programming	Negative OEM Open validation
Correctly signed boot artifact	Boot and ELE event inspection after SRKH programming	Positive OEM Open validation
<code>atlas-demo-image.ext4</code>	Processed during dm-verity image generation to produce <code>.ext4.verity</code> and <code>dm-verity.env</code>	WIC, RAUC bundle, and initramfs
<code>.ext4.verity</code> rootfs image	Added to the RAUC bundle as the raw slot payload, renamed as an <code>.img</code> file inside the <code>.raucb</code> bundle	RAUC raw slot installation
<code>dm-verity.env</code>	Parsed by the RAUC post-install hook to update either <code>VERITY_A_*</code> or <code>VERITY_B_*</code>	U-Boot boot script

Input artifact	Binding or transformation step	Later consumer
VERITY_A_* / VERITY_B_*	Used by U-Boot to construct the kernel command line, including the <code>verity_root_hash=...</code> arguments	dm-verity initramfs
RAUC bundle contents	Signed during RAUC bundle generation to produce the final <code>.rauchb</code> file	Target-side RAUC verification

6.8.3 Release and Provisioning Records

The implementation uses release and provisioning records to keep generated artifacts understandable after the build has completed. These records are not treated as validation results. Instead, they describe which files were produced, which artifacts were used as inputs to later steps, and which checksums or references are needed to connect the boot, update, dm-verity, and AHAB provisioning flows. This is useful because the implementation contains artifacts that are generated in different environments and then combined during image creation.

The AHAB signing and provisioning flow is a clear example of why such records are needed. The CST key-generation step produces the SRK certificate material, SRK table, SRKH fuse material, and fuse-programming command files. The signed `flash_singleboot` artifact must then be generated from the same SRK material as the programmed SRKH fuses. The record for this step therefore identifies the CST key-generation date, the SRK certificate set, the SRK table artifact, the SRKH fuse material artifact, the `nxpele-fuses.bcf` file used for programming, the U-Boot readback evidence, the signed `flash_singleboot` artifact, and the ELE event inspection results for both mismatched and correctly signed boot images. The record does not include private keys, passphrase files, or exact fuse-word values in the report.

The dm-verity and RAUC update flow requires similar traceability. The `.ext4.verity` root filesystem image, the corresponding `dm-verity.env` metadata, the renamed rootfs payload inside the RAUC bundle, and the slot-specific `VERITY_A_*` or `VERITY_B_*` values all belong to the same update chain. Recording these artifacts makes it possible to understand which root filesystem image a bundle installed and which metadata was expected to be used at the next boot. The observed update behavior and validation output are reported in Chapter 7, while this section only describes the record structure used to keep the generated artifacts connected.

Table 11: Release and provisioning record fields

Record field	Purpose
Build identifier	Gives the build or release a human-readable identifier.
Target machine	Identifies the machine configuration used for the generated artifacts.
WIC image name and checksum	Identifies the full image used for flashing.
Unsigned AHAB boot artifact	Records the <code>flash_singleboot</code> artifact used as signing input for the final AHAB validation.

Record field	Purpose
Signed AHAB boot artifact and checksum	Records the signed <code>flash_singleboot</code> artifact used as the first UUU boot image argument.
SRK table reference	Identifies the <code>SRK1234table.bin</code> artifact used by the signing configuration.
SRKH fuse material reference	Identifies the <code>SRK1234fuse.bin</code> , <code>uboot-fuses.cmd</code> , and <code>nxpele-fuses.bcf</code> artifacts.
SRKH fuse programming evidence	Points to programming output and U-Boot readback records for bank 16, words 0–7.
ELE event evidence	Points to mismatched-image and correctly signed-image ELE event inspection logs.
Lifecycle/provisioning state	Records that SRKH fuses were programmed and verified, while the device remained in OEM Open.
Evidence file references	Points to saved validation logs that are reported in Chapter 7.

7. Results

7.1 Results from the Comparative Analysis

The comparative analysis in this thesis shows that the reviewed solutions pursue the same fundamental security objective, but differ substantially in how trust is established, maintained, and exposed to developers. At the highest level, all reviewed solutions rely on an early trust anchor and a chain of verified handovers, which indicates that the core structure of secure boot remains consistent across otherwise different implementation models [1], [11]. The main differences instead appear in how trust metadata is organized, how tightly trust is bound to platform-specific hardware, and how much of the security model remains operationally flexible after provisioning. The result of the comparison is therefore not one universally preferable model, but a clearer picture of recurring design patterns shaped by platform architecture, lifecycle assumptions, and workflow requirements [4], [10].

A second result is that provisioning and lifecycle management are decisive for the long-term quality of a secure boot solution. The reviewed material indicates that the security of a boot chain depends not only on verification, but also on how trust material is introduced, protected, revoked, and recovered over time [2], [3]. This is especially important in embedded Linux and IIoT contexts, where devices remain active over long periods and where compromise of signing authority can have lasting consequences [5]. The comparison further shows that solutions are different in how they support such long-term maintenance in public documentation and operational workflows. As a result, lifecycle support should be treated as a primary evaluation dimension rather than as a secondary implementation detail. In practical terms, the analysis shows that maintainability is part of the security model itself.

A third result is that tooling quality, workflow integration, and documentation clarity strongly influence the practical value of a secure boot design. Solutions that can be more easily connected to structured build, signing, provisioning, and update workflows are also easier to evaluate, reproduce, and maintain in engineering practice [6], [7], [17], [18]. This does not mean that openness alone determines security, but it does mean that poorly supported workflows increase the risk of fragile integration and inconsistent operational handling. The analysis also shows that publicly available documentation varies enough to affect how confidently different solutions can be compared, which places a limit on the depth of cross-platform evaluation. Taken together, these results show that a strong secure boot design is characterized not only by sound cryptographic structure, but also by explicit lifecycle handling and a workflow model that supports repeatable and controlled system integration.

The qualitative synthesis in Table 4 further shows that the comparison should not be interpreted as a single ranking. UEFI Secure Boot is strongest as an open and policy-oriented firmware trust model, NXP AHAB is the most directly applicable design for the selected i.MX 93 case study, TI AM64x is strong where explicit lifecycle transition and key-revision support are central, Qualcomm is most relevant inside its own platform ecosystem, and Rockchip is mainly useful in this thesis as a contrast case for documentation visibility. The practical implication is that design suitability depends on engineering purpose rather than on a universal secure-boot score.

7.2 Results from the Practical Case Study

The practical case study evaluated the implemented Atlas-based boot and update chain on the NXP i.MX 93 target. This section documents observed behavior after flashing, booting, updating, and inspecting the target, while the implications of these results are discussed in Chapter 8. The evaluated areas were the storage layout, read-only root filesystem behavior, dm-verity root mounting, RAUC A/B update operation, slot-specific dm-verity metadata synchronization, SRKH fuse programming, and AHAB/ELE validation in OEM Open. These results correspond to the implementation described in Chapter 6.

The validation evidence is also connected to the release and provisioning record structure described in Table 11. This is important because the case study depends on relationships between generated artifacts rather than on isolated outputs: the signed `flash_singleboot` artifact must match the programmed SRKH material, and each RAUC-installed `.ext4.verity` payload must match the `dm-verity.env` metadata and slot-specific `VERITY_A_*` or `VERITY_B_*` values used at the next boot.

The validation was performed incrementally. First, the flashed storage layout and mount state were checked to confirm that the expected boot, rootfs, data, and config partitions were present. The dm-verity boot path was then verified by checking that the root filesystem was mounted from `/dev/mapper/rootfs`. After that, RAUC installation was tested in both A-to-B and B-to-A directions. Finally, the AHAB validation was extended to include CST-generated SRK/SRKH material, SRKH fuse programming, U-Boot readback, and ELE event inspection for both mismatched and correctly signed `flash_singleboot` artifacts while the device remained in OEM Open.

7.2.1 Overview of Validation Cases

The validation cases in Table 12 correspond to the main boundaries in the implemented chain: storage layout, root filesystem immutability, dm-verity root mounting, RAUC A/B updates, slot-specific metadata synchronization, and AHAB OEM Open validation. The table also separates the development-state validation from the closed-state AHAB enforcement case, which was outside the evaluated scope.

Table 12: Summary of practical validation cases and evidence

Validation case	Expected result	Observed result	Main evidence
Storage layout after full flash	Boot, rootfs A, rootfs B, data, and config partitions are present	Passed	WIC layout, <code>lsblk</code> , partition inspection
Read-only root filesystem	<code>/</code> is mounted read-only	Passed	<code>findmnt</code> , mount output
Writable data partition	<code>/data</code> accepts persistent writes	Passed	write/read persistence check
Separate config partition	<code>/config</code> is mounted separately according to the configured layout	Passed	mount output, <code>fstab</code> configuration

Validation case	Expected result	Observed result	Main evidence
dm-verity root mount	/ is mounted from /dev/mapper/rootfs	Passed	findmnt /, dmsetup, boot log
Initial dm-verity metadata provisioning	first boot has matching metadata for the flashed rootfs	Passed	verity.env, kernel command line, initramfs behavior
RAUC A to B update	system boots B after installing to inactive B slot	Passed	rauc status, reboot result, marker/evidence file
RAUC B to A update	system boots A after installing to inactive A slot	Passed	rauc status, reboot result, marker/evidence file
Slot-specific metadata synchronization	updated slot boots with matching dm-verity metadata	Passed	VERITY_A_* / VERITY_B_*, kernel command line, /dev/mapper/rootfs
AHAB signing artifact selection	UUU flash_singleboot artifact is signed with CST-generated AHAB material	Passed	signed artifact record, UUU command, AHAB signing output
SRKH fuse programming	eight SRKH fuse words are programmed on the target	Passed	nxpele programming output
SRKH fuse readback	programmed SRKH values match the generated fuse material	Passed	U-Boot fuse read bank 16, words 0–7
AHAB mismatched-image detection in OEM Open	mismatching boot image produces ELE key-hash mismatch indication	Passed	ELE event output
AHAB correctly signed image in OEM Open	correctly signed image reports no remaining authentication failure under tested OEM Open conditions	Passed	ahab_status / nxpele get-events output
AHAB closed-state enforcement	incorrectly signed image is rejected after lifecycle transition to OEM Closed	Not evaluated	outside scope; device remained OEM Open

The table separates validation evidence from release and provisioning records. The record structure in Table 11 identifies which artifacts, checksums, SRK/SRKH material, fuse-programming outputs, and ELE event logs belong to a release or provisioning step. Table 12, by contrast, summarizes the observed runtime and target-side validation results. The two tables should therefore be read together: Table 11 explains how evidence is traced to artifacts, while Table 12

reports whether the tested behavior passed.

7.2.2 Storage Layout and Read-only Mount Results

The flashed image produced the expected storage layout on the target. The eMMC contained a shared boot partition, two root filesystem slots, a writable data partition, and a separate config partition. The target-side mount output showed that `/data` and `/config` were mounted as separate filesystems rather than as directories inside the root filesystem. This confirmed that the WIC layout and `base-files` configuration produced the intended runtime separation between immutable operating-system content, persistent writable data, and device-local configuration.

The root filesystem was mounted read-only during normal operation. Attempts to write directly to `/` failed, while writes under `/data` succeeded. The `/config` partition was also mounted separately and read-only, which matched the intended storage model for device-local configuration. Figure 5 shows the observed partition and mount state after full flashing. The output confirms that the root filesystem, `/data`, and `/config` were treated as distinct runtime storage areas.

```
root@atlas-demo:~# lsblk -f
NAME                                FSTYPE FSVER LABEL UUID                                FSAVAIL FSUSE% MOUNTPOINTS
mmcblk0
|-mmcblk0p1 vfat                boot  0808-C809                                5.1M   95% /boot/vfat
|-mmcblk0p2 ext4                73e2743a-b194-47de-b428-75a9b7cfa9ca
| `--rootfs ext4                73e2743a-b194-47de-b428-75a9b7cfa9ca 82.6M   69% /
|-mmcblk0p3 ext4                73e2743a-b194-47de-b428-75a9b7cfa9ca
|-mmcblk0p4
|-mmcblk0p5 ext4                data  7a82ad48-f668-4ea2-bcc2-dd8fc71770e4 444.8M   0% /data
|-mmcblk0p6 ext4                config a87d532f-59cf-4bfa-b68d-29b6b399ce5c 52.2M   0% /config
mmcblk0boot0
mmcblk0boot1
root@atlas-demo:~# findmnt -R -o TARGET,SOURCE,FSTYPE,OPTIONS | grep -E 'TARGET| /$|/data|/config|/boot'
TARGET SOURCE FSTYPE OPTIONS
| `-/sys/kernel/config configfs configfs rw,nosuid,nodev,noexec,relatime
|-/boot/vfat /dev/mmcblk0p1 vfat rw,relatime,fmask=0022,dmask=0022,codepage=437,ioccharset=io
|-/config /dev/mmcblk0p6 ext4 ro,nosuid,nodev,noexec,relatime
`-/data /dev/mmcblk0p5 ext4 rw,relatime
```

Figure 5: Storage and mount evidence after full flash.

The storage validation result is important because later tests depend on this separation. `dm-verity` requires the root filesystem image to remain immutable after its hash tree has been generated, while `RAUC` requires writable state to be stored outside the verified rootfs slots. The observed mount layout showed that runtime state could be placed under `/data`, `RAUC` state could be stored under `/data/rauc`, and `/config` could remain separate from the operating-system image. This established the storage foundation used by the later `dm-verity` and update tests.

7.2.3 dm-verity Boot Results

The `dm-verity` boot path was validated by inspecting the kernel command line, the mounted root filesystem source, the device-mapper state, and the kernel log. The kernel command line showed that U-Boot passed the selected rootfs partition as a `PARTUUID`, rather than as a fixed block-device path. After the `initramfs` completed its setup, `findmnt` reported `/dev/mapper/rootfs` as the source for `/`. This was the main evidence that the selected rootfs slot was used as backing storage for a verified device-mapper target, instead of being mounted directly.

The device-mapper state and kernel log confirmed the same result. The `/dev/mapper/rootfs` entry pointed to a `dm-0` device, and `/sys/block/dm-*/dm/name` reported the mapping name

as `rootfs`. The kernel log included dm-verity messages using SHA-256 and showed the ext4 filesystem being mounted from the mapped device. Figure 6 shows the command-line, mount, device-mapper, and kernel-log evidence collected after boot.

```

root@atlas-demo:~# cat /proc/cmdline
console=ttyLP0,115200 earlycon rootfstype=ext4 rootwait ro root=PARTUUID=076c4a2a-02 rauc.slot=A
root@atlas-demo:~# findmnt -no SOURCE,FSTYPE,OPTIONS /
/dev/mapper/rootfs ext4 ro,relatime
root@atlas-demo:~# ls -l /dev/mapper
total 0
crw----- 1 root root 10, 236 Mar  5 15:13 control
lrwxrwxrwx 1 root root  7 Jan  1 1970 rootfs -> ../dm-0
root@atlas-demo:~# cat /sys/block/dm-*/dm/name 2>/dev/null
rootfs
root@atlas-demo:~# dmesg | grep -Ei 'verity|mapper|dm-|rootfs'
[ 0.000000] Kernel command line: console=ttyLP0,115200 earlycon rootfstype=ext4 rootwait ro root=PARTUUID=076c4a2aA
[ 1.234127] device-mapper: ioctl: 4.48.0-ioctl (2023-03-01) initialised: dm-devel@redhat.com
[ 4.584887] device-mapper: verity: sha256 using implementation "sha256-ce"
[ 4.687328] EXT4-fs (dm-0): write access unavailable, skipping orphan cleanup
[ 4.694609] EXT4-fs (dm-0): mounted filesystem 73e2743a-b194-47de-b428-75a9b7cfa9ca ro with ordered data mode. Quo.
[ 5.863026] systemd[1]: Reached target RPC Port Mapper.
[ 6.616080] EXT4-fs (dm-0): re-mounted 73e2743a-b194-47de-b428-75a9b7cfa9ca ro. Quota mode: none.
root@atlas-demo:~#

```

Figure 6: dm-verity boot evidence showing `/dev/mapper/rootfs` as the mounted root filesystem.

The observed boot result confirms that the initramfs-based dm-verity path was active. The selected rootfs partition remained the storage backing for the mapping, but it was not the filesystem mounted directly as `/`. The mounted root filesystem was read-only and passed through the `rootfs` device-mapper target. This verified the intended integrity boundary between the raw rootfs slot and the running Linux root filesystem.

7.2.4 RAUC A/B Update Results with dm-verity

RAUC update behavior was validated in both A-to-B and B-to-A directions. The target-side RAUC configuration reported both rootfs slots as raw, which matched the `.ext4.verity` image payload used in the bundle. When the system was booted from one slot, RAUC installed the bundle to the inactive slot. After reboot, the system selected the newly installed slot and still mounted `/` from `/dev/mapper/rootfs`. This showed that the A/B update mechanism remained compatible with the dm-verity rootfs format.

The first update direction installed the bundle to the B slot and rebooted into `rootfs.1`. The kernel command line showed `rauc.slot=B`, and `rauc status` reported B as the booted slot. The mounted root filesystem source remained `/dev/mapper/rootfs`, confirming that the system did not bypass dm-verity after the RAUC update. The reverse direction then installed the bundle back to the A slot and rebooted into `rootfs.0`, again with `/dev/mapper/rootfs` as the mounted root source. Figure 7 shows representative RAUC status, command-line, and mount output from the A/B validation. Table 13 summarizes the observed A/B update behavior. Both update directions reached the expected slot after reboot and preserved the dm-verity root mount. This result shows that converting the RAUC slots from ext4-handled slots to raw dm-verity image targets did not break the existing A/B update model. The slot transitions, bootloader selection, and verified root filesystem mount all remained aligned.

```

root@atlas-demo:~# rauc status
=== System Info ===
Compatible: ATLAS-DEMO
Variant:
Booted from: rootfs.1 (B)

=== Bootloader ===
Activated: rootfs.1 (B)

=== Slot States ===
x [rootfs.1] (/dev/mmcblk0p3, raw, booted)
  bootname: B
  boot status: good

o [rootfs.0] (/dev/mmcblk0p2, raw, inactive)
  bootname: A
  boot status: good

root@atlas-demo:~# cat /proc/cmdline
console=ttyLP0,115200 earlycon rootfstype=ext4 rootwait ro root=PARTUUID=076c4a2a-03 verity_root_hash=c7f3bb5dcafcf15B
root@atlas-demo:~# findmnt -no SOURCE,FSTYPE,OPTIONS /
/dev/mapper/rootfs ext4 ro,relatime
root@atlas-demo:~# rauc status
=== System Info ===
Compatible: ATLAS-DEMO
Variant:
Booted from: rootfs.0 (A)

=== Bootloader ===
Activated: rootfs.0 (A)

=== Slot States ===
o [rootfs.1] (/dev/mmcblk0p3, raw, inactive)
  bootname: B
  boot status: good

x [rootfs.0] (/dev/mmcblk0p2, raw, booted)
  bootname: A
  boot status: good

root@atlas-demo:~# cat /proc/cmdline
console=ttyLP0,115200 earlycon rootfstype=ext4 rootwait ro root=PARTUUID=076c4a2a-02 verity_root_hash=c7f3bb5dcafcf15A
root@atlas-demo:~# findmnt -no SOURCE,FSTYPE,OPTIONS /
/dev/mapper/rootfs ext4 ro,relatime

```

Figure 7: RAUC A/B update evidence after booting both dm-verity-protected slots.

Table 13: RAUC A/B update validation with dm-verity

Update direction	Booted slot after reboot	RAUC slot name	Root filesystem source
A to B	B	rootfs.1	/dev/mapper/rootfs
B to A	A	rootfs.0	/dev/mapper/rootfs

7.2.5 Slot-specific Metadata Synchronization Results

The slot-specific dm-verity metadata path was validated by installing a RAUC bundle that changed the root filesystem image and therefore produced different dm-verity metadata. After installation, the updated slot received new `VERITY_A_*` or `VERITY_B_*` values in the U-Boot environment. The A and B slots were observed to have different root hash values after the test, which confirmed that the post-install hook maintained metadata per slot instead of overwriting one global root hash. This was necessary because A and B may contain different rootfs images after updates.

The marker-file test confirmed that the updated root filesystem image was the one that actually booted. A temporary marker file was added to the generated rootfs, a new RAUC bundle was built, and the bundle was installed to the inactive slot without reflashing the boot partition. After reboot, the marker file was present under `/etc/devsec-verity-marker`, while the root filesystem still mounted from `/dev/mapper/rootfs`. This showed that the booted rootfs came from the RAUC-installed image and that the updated dm-verity metadata path was sufficient for the slot to boot.

The kernel command line also contained the metadata arguments passed by U-Boot, including `verity_root_hash`, `verity_data_size`, and `verity_data_block_size`. These values were derived from the slot-specific `VERITY_A_*` or `VERITY_B_*` environment variables written by the RAUC post-install hook. Figure 8 shows the observed U-Boot metadata, kernel command line, marker file, and mount source after the metadata synchronization test.

```

root@atlas-demo:~# fw_printenv | grep '^VERITY_'
VERITY_A_DATA_BLOCK_SIZE=1024
VERITY_A_DATA_SIZE=394166272
VERITY_A_HASH_ALGORITHM=sha256
VERITY_A_HASH_BLOCK_SIZE=4096
VERITY_A_ROOT_HASH=c7f3bb5dcafcf15f16c50dde3889804de9cc4d455aeff2b90e4a9f7a6b1c0b18
VERITY_A_SALT=66b32b7d33f4eadeaad7eca0cf2f20442a0f03bda64c415d0b61d65a5d37a58d
VERITY_A_SEPARATE_HASH=0
VERITY_B_DATA_BLOCK_SIZE=1024
VERITY_B_DATA_SIZE=394166272
VERITY_B_HASH_ALGORITHM=sha256
VERITY_B_HASH_BLOCK_SIZE=4096
VERITY_B_ROOT_HASH=02989ee3bd5baf3e188540b6c39d8b0a1073328a2e79720ee9f3a243629d52a
VERITY_B_SALT=e8a2f1c8785714efa524d55899f30b1012666a878afa60022b26d8726f68e1d0
VERITY_B_SEPARATE_HASH=0
root@atlas-demo:~# cat /proc/cmdline
console=ttyLP0,115200 earlycon rootfstype=ext4 rootwait ro root=PARTUUID=076c4a2a-03 verity_root_hash=02989ee3bd5baf3B
root@atlas-demo:~# cat /etc/devsec-verity-marker
metadata-sync-test-v2
root@atlas-demo:~# findmnt -no SOURCE,FSTYPE,OPTIONS /
/dev/mapper/rootfs ext4 ro,relatime

```

Figure 8: Slot-specific dm-verity metadata and marker-file evidence after RAUC update.

This result provides the strongest end-to-end validation of the RAUC and dm-verity integration. It shows that the updated rootfs image, the bundle metadata, the U-Boot environment, the kernel command line, and the initramfs dm-verity setup all referred to the same slot-specific image state. The successful boot of the marked image through `/dev/mapper/rootfs` demonstrates that the system did not only support repeated installation of identical images, but could also handle a rootfs change with new dm-verity metadata.

7.2.6 AHAB Signing, SRKH Programming, and OEM Open Validation Results

The AHAB validation was extended from signed-image inspection to SRKH fuse programming and open-state authentication testing. The CST-generated `nxpele-fuses.bcf` file was used to program the SRKH fuse words on the target through the U-Boot serial interface. The programming command reported successful completion for all eight SRKH fuse words. The programmed values were then read back from U-Boot using bank 16, words 0 through 7. These readback values were compared against the generated CST fuse material and matched the expected values.

Figures 9 and 10 show the programming and readback evidence. The exact fuse-word values are treated as provisioning records, but the important validation result is that the programming step completed and that the target reported the expected programmed values. This confirmed that the board had been bound to the CST-generated SRKH material before the AHAB event validation was repeated.

```
ELE write fuse ends successfully.  
ELE write fuse ends successfully.  
ELE write fuse ends successfully.  
ELE write fuse ends successfully.  
ELE write fuse ends successfully.  
ELE write fuse ends successfully.  
ELE write fuse ends successfully.  
ELE write fuse ends successfully.
```

Figure 9: SRKH fuse programming output showing successful nxpele fuse programming.

```
u-boot=> fuse read 16 0  
Reading bank 16:  
  
Word 0x00000000:   
u-boot=> fuse read 16 1  
Reading bank 16:  
  
Word 0x00000001:   
u-boot=> fuse read 16 2  
Reading bank 16:  
  
Word 0x00000002:   
u-boot=> fuse read 16 3  
Reading bank 16:  
  
Word 0x00000003:   
u-boot=> fuse read 16 4  
Reading bank 16:  
  
Word 0x00000004:   
u-boot=> fuse read 16 5  
Reading bank 16:  
  
Word 0x00000005:   
u-boot=> fuse read 16 6  
Reading bank 16:  
  
Word 0x00000006:   
u-boot=> fuse read 16 7  
Reading bank 16:  
  
Word 0x00000007: 
```

Figure 10: U-Boot SRKH fuse readback from bank 16, words 0–7.

After fuse programming, a mismatched boot image was used as a negative OEM Open validation case. The ELE event output reported that the key hash verification did not match OTP. This shows that the boot image did not correspond to the programmed SRKH material and that the programmed fuse values were active in the AHAB authentication path. Because the device remained in OEM Open, the observed result should be interpreted as mismatch detection rather than closed-state boot rejection.

```

ELE get events ends successfully.
Event count:      2
Event[0]:         0x0287FAD6
  IPC ID:         Application Processor message unit
  Command:        OEM Container authenticate
  Indication:     The key hash verification does not match OTP
  Status:         The request was successful
Event[1]:         0x0287FAD6
  IPC ID:         Application Processor message unit
  Command:        OEM Container authenticate
  Indication:     The key hash verification does not match OTP
  Status:         The request was successful

```

Figure 11: ELE event output for a mismatched boot image after SRKH fuse programming.

The correctly signed `flash_singleboot` artifact was then used for the positive validation case. This artifact was signed with the CST-generated key material corresponding to the programmed SRKH fuses and used as the first UUU boot image argument. In this case, the ELE event inspection did not report a remaining authentication failure. The observed output reported an event count of zero, which indicates that no remaining ELE authentication events were reported under the tested OEM Open conditions.

```

ELE get events ends successfully.
Event count:      0

```

Figure 12: ELE event output for the correctly signed boot image after SRKH fuse programming.

The AHAB validation therefore confirms four points: the SRKH fuse words were programmed, the programmed values were verified by readback, a mismatched boot image produced an ELE key-hash mismatch indication, and the correctly signed `flash_singleboot` artifact produced no remaining ELE authentication events under the tested OEM Open conditions. The validation does not evaluate OEM Closed lifecycle behavior, since the device was not transitioned to OEM Closed.

Table 14: AHAB signing, SRKH programming, and OEM Open validation

Check	Expected result	Observed result
CST-based SRK/SRKH material generation	SRK table, SRKH fuse material, and fuse command files are generated from the same SRK certificate set	Passed

Check	Expected result	Observed result
Signing artifact selection	The <code>flash_singleboot</code> artifact used by UUU is signed with matching CST material	Passed
SRKH fuse programming	Eight SRKH fuse words are programmed on the target	Passed
SRKH fuse readback	U-Boot readback matches the generated CST fuse material	Passed
Mismatched-image ELE inspection	ELE reports a key-hash mismatch indication in OEM Open	Passed
Correctly signed-image ELE inspection	ELE reports no remaining authentication events under tested OEM Open conditions	Passed
Closed-state enforcement	Incorrectly signed image is rejected after lifecycle transition to OEM Closed	Not evaluated

7.2.7 Summary of Practical Case Study Results

The practical case study produced a working boot and update chain with a read-only root filesystem, separated writable state, dm-verity-protected A/B rootfs slots, RAUC raw-slot updates, slot-specific dm-verity metadata synchronization, and AHAB validation after SRKH fuse programming in OEM Open. The target-side evidence showed that the root filesystem mounted from `/dev/mapper/rootfs`, that both RAUC update directions succeeded, and that an updated rootfs image with different dm-verity metadata could boot after the metadata was synchronized through the RAUC hook, U-Boot environment, kernel command line, and `initramfs`. The AHAB evidence further showed that SRKH fuse programming completed, that the programmed values were verified by readback, that a mismatched boot image produced an ELE key-hash mismatch indication, and that the correctly signed `flash_singleboot` artifact produced no remaining ELE authentication events under the tested OEM Open conditions.

8. Discussion

8.1 Interpretation of the Results

The main implication of the practical results is that secure boot integration in an embedded Linux system depends on consistency across several artifact boundaries, not only on enabling one verification mechanism. The comparative analysis showed that secure boot solutions differ in how they anchor trust, expose provisioning workflows, and support long-term maintenance, while the practical case study showed how these concerns appear at implementation level. The AHAB-signed boot artifact, programmed SRKH material, RAUC slot state, dm-verity metadata, kernel command line, and initramfs setup must refer to the same system state. The case study therefore supports the view that build and update workflows are part of the practical security model, not merely supporting engineering infrastructure.

The case study results also show that trust-anchor binding and final lifecycle enforcement should be interpreted separately. SRKH fuse programming and readback, together with the OEM Open mismatch and matching-artifact results in Section 7.2.6, strengthen the evidence that the programmed SRKH material participated in AHAB authentication. However, as scoped in Section 1.4 and discussed further in Section 8.4, this remains SRKH-programmed OEM Open validation rather than OEM Closed enforcement.

The dm-verity and RAUC results are stronger with respect to runtime behavior. Figure 6 shows that the root filesystem was mounted through `/dev/mapper/rootfs`, while Table 13 shows that both A-to-B and B-to-A RAUC updates completed with the root filesystem still mounted through dm-verity. This indicates that the implementation preserved the existing A/B update model while changing the rootfs slots from ordinary ext4-managed slots into raw dm-verity image targets. The result supports the view that update mechanisms must be adapted when root filesystem integrity is introduced, rather than treated as independent infrastructure.

The metadata synchronization result is one of the most significant practical findings. Figure 8 shows that the updated slot could boot with slot-specific dm-verity metadata supplied through the RAUC hook, U-Boot environment, kernel command line, and initramfs. This demonstrates that the rootfs image and the metadata used to verify it must be treated as a coupled update unit. Without this synchronization, a valid rootfs image could still fail to boot because the initramfs verifies it against stale metadata. This finding connects directly to the artifact traceability structure described in Section 6.8.

Overall, the results in Table 12 should not be read as isolated test outcomes. They show that the implementation works when Yocto generates the expected storage layout and rootfs artifacts, RAUC installs the correct raw payload, AHAB tooling signs the early boot image, and release records keep the resulting artifacts aligned. In this sense, the practical security of the system depends on the alignment between build, signing, update, bootloader, and initramfs responsibilities across the chain.

8.2 Trade-Offs Between Security, Usability, and Maintainability

The implementation improves integrity protection but also increases the number of artifacts and relationships that must be maintained. A normal writable root filesystem and ext4-based RAUC

slots are easier to understand and update, but they do not provide the same block-level integrity boundary as dm-verity. Moving to dm-verity required raw rootfs slots, read-only operation, separate writable state, a dm-verity initramfs, and slot-specific metadata synchronization. This improves resistance to unintended or unauthorized rootfs modification, but it also makes the update flow less forgiving. The system becomes more secure only if the metadata, payload, and boot arguments remain consistent.

The RAUC integration illustrates a practical trade-off between preserving an existing update model and enforcing immutability. Keeping the A/B slot model was valuable because it preserved the baseline Atlas update behavior and allowed rollback-oriented slot switching to remain in place. However, the previous hook behavior, where files were copied into the newly installed rootfs, was no longer compatible with dm-verity. The replacement hook had to avoid modifying the rootfs and instead update boot metadata outside the verified image. This is a useful trade-off: the update framework remains usable, but mutable device state must be moved out of the rootfs.

The AHAB part shows a different trade-off between validation strength and production risk. Programming and reading back the SRKH fuses made the trust-anchor validation stronger than a host-side signing test, but keeping the device in OEM Open preserved recoverability during thesis work. This was appropriate for a development case study, but production use would still require lifecycle closure, controlled release approval, and closed-state negative boot tests.

The `/config` partition also reflects a trade-off between practical separation and stronger secret protection. As described in Section 6.7.3, `/config` separates device-local configuration from the immutable root filesystem and is mounted read-only during normal operation. This supports maintainability because rootfs updates can replace A/B slots without overwriting local configuration. It also supports integrity-oriented separation, since ordinary runtime writes do not modify `/config`. However, this does not make `/config` a confidential keystore. If the final product needs to protect private device credentials against raw storage access, additional hardware-backed or encrypted storage mechanisms would be needed.

Key handling presents a similar balance. The case study separates host-side signing material from target-side verification material, as summarized in Table 8. This is an important security property because private AHAB and RAUC signing keys are not needed on the target. At the same time, the implementation still uses development keys and manually coordinated signing steps for AHAB. That is acceptable for integration validation, but it would need stronger access control, signing policy, and release procedure in production. The trade-off is therefore between development flexibility and the stricter operational discipline required for deployed signing authority.

Maintainability also depends on artifact traceability. The implementation creates a chain of artifacts where a signed boot image, WIC image, RAUC bundle, `.ext4.verity` payload, `dm-verity.env`, and slot-specific U-Boot variables must match. This increases the documentation burden compared with a simpler system, but it also makes failures easier to reason about. The release-record structure in Table 11 is therefore not administrative overhead alone. It is a practical structure that helps developers identify which artifacts belong together and reduces the risk of incorrectly combining valid components from different builds.

8.3 Transferability Beyond NXP i.MX 93

The findings from the case study should be described as transferable design principles and workflow guidance rather than as a fully general method. The practical implementation was carried out on one NXP i.MX 93 platform and one existing Yocto-based baseline, which means that the results cannot by themselves prove that the same implementation structure works unchanged on other platforms. However, several of the design concerns are not unique to NXP AHAB and can still inform similar embedded Linux projects. Table 15 summarizes which parts of the work are most transferable and which parts are platform-specific.

Table 15: Transferability of case-study findings beyond the NXP i.MX 93 platform

Finding or design element	Transferability	Explanation
Clear trust-anchor definition	High	Secure boot designs generally need an explicit root of trust or trust anchor, even though the concrete mechanism differs between platforms.
Separation of private signing material and target-side verification material	High	Keeping private signing keys off the target is a general key-management principle, not an NXP-specific detail.
Writable-state separation	High	Systems with immutable or verified root filesystems must decide where persistent runtime data and device-local configuration should be stored.
Artifact traceability	High	Any workflow that combines separate build, signing, update, and provisioning artifacts needs a way to record which artifacts belong together.
A/B update model with verified rootfs slots	Moderate to high	The principle transfers to other update frameworks, but the exact slot handling, metadata storage, and bootloader integration differ.
RAUC and dm-verity metadata synchronization	Moderate	The need to synchronize verification metadata with the installed image is general, but the implemented hook, U-Boot variables, and initramfs override path are specific to this stack.
Yocto image, WIC, and recipe changes	Conditional	The structure is useful for similar Yocto-based projects, but not directly portable to non-Yocto build systems.

Finding or design element	Transferability	Explanation
AHAB, ELE events, SRK tables, SRKH fuses, and OEM lifecycle states	Platform-specific	These mechanisms are specific to NXP AHAB and cannot be assumed to exist on other SoCs.
UUU flash_singleboot signing boundary	Workflow-specific	This finding is important for the tested flashing workflow, but other deployment flows may use different boot artifacts.

The most transferable result is therefore not a command sequence, but a way of structuring the problem. Similar embedded Linux systems can benefit from first identifying the trust anchor, then separating signing authority from target-side verification material, then deciding how immutable operating-system content and writable state should be separated, and finally ensuring that update metadata and release records remain synchronized with the artifacts they describe.

The RAUC and dm-verity work is transferable mainly at the workflow level. A system with A/B updates and verified rootfs images must ensure that the update framework installs complete image artifacts and that the next boot uses verification metadata belonging to the selected slot. In this thesis, that was implemented through RAUC hooks, U-Boot environment variables, and an initramfs metadata override path. Other systems may store this information differently, but the underlying requirement remains the same: the rootfs image and the metadata used to verify it must belong to the same release state.

The AHAB-specific work is less portable. AHAB, ELE event reporting, SRK tables, SRKH fuse words, and OEM Open/OEM Closed lifecycle handling are specific to the NXP platform. The broader pattern can still transfer: generated trust-anchor material must match the signed early boot artifact, provisioning must be recorded, and development validation should be separated from final production enforcement. However, the exact commands, fuse locations, lifecycle states, and event outputs should not be treated as general embedded Linux behavior.

The qualitative comparison in Chapter 5 supports the same interpretation. UEFI Secure Boot, NXP AHAB, TI secure boot, Qualcomm secure boot, and Rockchip public material differ in documentation depth, lifecycle model, tooling, and trust-anchor structure. The case study therefore provides a concrete example of applying transferable principles on one platform, not evidence that one implementation recipe is universally applicable across all embedded Linux systems.

8.4 Limitations of the Study

The most important implementation limitation is that final AHAB closed-state enforcement was not evaluated. As already scoped in Section 1.4, the device remained in OEM Open after SRKH fuse programming and readback. The study can therefore support claims about programmed SRKH material, OEM Open mismatch detection, and correct-artifact OEM Open behavior, but not about hardware rejection of invalid boot images in OEM Closed.

A second limitation is that the case study is based on one target platform and one existing Yocto project. The Atlas baseline already provided a bootable image, U-Boot integration, and a RAUC A/B update model, as described in Section 6.1. The work therefore evaluates how security properties can be added to an existing project, not how a complete secure boot and update architecture should be designed from zero. This makes the results practically relevant for similar projects, but it also means that some design decisions were shaped by the existing Atlas structure and may not transfer directly to other baselines.

The validation scope is also limited to selected functional and integration properties. The study validated read-only mounting, `/data` and `/config` separation, dm-verity root mounting, RAUC A/B updates, slot-specific metadata synchronization, AHAB OEM Open events, and artifact handoff checks. It did not perform a complete adversarial test campaign. For example, it did not include systematic fault injection, physical storage tampering, power-loss testing during RAUC installation, or long-duration update cycling. The results therefore show that the implemented chain works under the tested conditions, not that all failure modes or attack scenarios have been exhausted. These missing failure cases are therefore treated as future work in Section 10.2 rather than as validation results. In particular, dm-verity tamper detection, metadata mismatch handling, interrupted RAUC updates, and corrupted boot/update metadata should only be reported as passed after they have been tested and documented with explicit evidence.

The key-management evaluation is similarly bounded. The implementation separates private signing material from target-side verification material and documents the CST-generated SRK/SRKH material, SRKH programming evidence, and ELE event evidence as provisioning records. However, it does not implement a complete production key-management system with HSM-backed signing, formal access control, backup procedures, key rotation, revocation, or emergency recovery. These topics are discussed conceptually through the literature and comparative analysis, but they are not fully implemented in the case study. This is consistent with the scope in Section 1.4, but it means that production key operations remain an important area for later work.

The treatment of `/config` is another limitation. The implementation introduces `/config` as a separate read-only partition for device-local configuration, but this should not be interpreted as confidential storage. It provides separation from the root filesystem and protects against ordinary runtime writes while mounted read-only, but it does not by itself protect secrets against raw storage access or a physical attacker. If the final system requires device-private credentials, an additional storage-protection mechanism would be needed. This limitation is important because secure boot and dm-verity provide integrity properties, not general confidentiality for all persistent data.

A final limitation concerns release automation. The case study defines artifact traceability and demonstrates how the relevant artifacts can be connected through checksums and records, but it does not implement a complete production CI/CD or secure release pipeline. AHAB signing was performed through a host-side SPSDK/imx-signer workflow, while Yocto image generation ran in the Docker-based build environment. This is sufficient for the evaluated development workflow, but production use would require stronger automation, policy enforcement, and auditability. The study therefore demonstrates a maintainable integration pattern, while leaving

full release-process hardening outside the tested implementation.

8.5 Societal and Ethical Considerations

Secure boot and integrity verification have societal relevance because embedded systems are often deployed in products and infrastructure that users, companies, and public organizations rely on for long periods. A stronger boot and update chain can reduce the risk that devices run modified or unauthorized software, which can support reliability, safety, and trust in connected products. This is especially important for long-lived IoT and industrial systems, where insecure update handling or weak integrity protection can create risks that remain present after deployment. From this perspective, the mechanisms studied in the thesis can contribute positively by making unauthorized modification harder to hide and by making release and provisioning evidence easier to inspect.

At the same time, stronger technical enforcement also increases responsibility for the actors controlling the keys, update infrastructure, and lifecycle decisions. If signing keys are mishandled, if update support ends too early, or if recovery paths are not planned, secure boot can make devices difficult to repair or recover. A device that only accepts vendor-authorized software may become more resistant to malicious modification, but it may also become dependent on the vendor or integrator continuing to provide valid updates. Long-term key management, recovery planning, and clear ownership of signing authority are therefore ethical as well as technical concerns.

There is also a tension between integrity protection and repairability. Secure boot can prevent unauthorized software from running, but the same mechanism can restrict legitimate maintenance, third-party repair, research, or owner modification if policy decisions are too closed. This thesis does not define a final product policy for who should be allowed to sign software for a deployed device. However, the results show that this policy must be considered early, because trust anchors, provisioning records, and lifecycle transitions can become difficult or impossible to change after deployment.

The technology can also be used in systems with privacy-sensitive or surveillance-related purposes. Secure boot does not determine whether a product is socially beneficial or harmful; it strengthens the integrity of whatever software the product is designed to run. In a beneficial setting, that may help protect critical devices against tampering. In a harmful or poorly governed setting, the same mechanisms could make monitoring or control systems harder for users or auditors to inspect or modify. Engineers should therefore consider not only whether the secure boot chain works, but also what type of system it protects, who controls the update authority, and what accountability mechanisms exist.

Finally, the mechanisms implemented in the case study mainly address integrity, not confidentiality or broader cybersecurity governance. dm-verity can detect modification of the root filesystem, and AHAB can authenticate early boot artifacts, but these mechanisms do not by themselves protect all private data, guarantee secure application behavior, or define responsible incident handling. A production system should therefore combine secure boot with vulnerability management, update support, key-rotation planning, recovery procedures, and clear organizational responsibility for security failures.

9. Conclusions

9.1 Fulfillment of the Aim

The aim of the thesis was to analyze how secure boot, cryptographic key management, and provisioning can be designed and applied in Linux-based embedded systems in a way that is secure, practically maintainable, and useful as structured engineering guidance. This aim is addressed through the combination of the literature and standards review, the comparative analysis, and the practical NXP i.MX 93 case study. The thesis does not fulfill the aim by delivering a complete production framework or a universally valid secure boot recipe. Instead, it fulfills the aim by connecting general secure boot principles to concrete implementation decisions, validation evidence, and workflow trade-offs.

The research questions support different parts of this aim. RQ1 establishes the key-management and provisioning principles needed for secure boot. RQ2 compares how selected secure boot solutions differ in trust-anchor design, lifecycle handling, tooling, documentation, and maintainability. RQ3 applies these concerns in the NXP i.MX 93 AHAB case study. RQ4 shows how Yocto, RAUC, dm-verity, update metadata, and artifact traceability affect practical security and maintainability. RQ5 then identifies which parts of the findings can be transferred as design principles and workflow guidance for similar embedded Linux systems.

The aim is therefore fulfilled as a structured analysis and method-oriented case study. The work provides a practical baseline for reasoning about secure boot integration in embedded Linux, while remaining explicit about its boundaries: the practical implementation was performed on one platform and one existing baseline project, the device was not transitioned to OEM Closed, and a complete production key-management and recovery process was outside scope.

9.2 Answers to the Research Questions

This section answers the research questions stated in Section 1.3 by combining the theoretical analysis, the comparative evaluation, the practical case study, and the discussion. The answers should be read together with the scope in Section 1.4 and the aim interpretation in Section 9.1.

RQ1: Which key management and provisioning principles are most important for secure boot in Linux-based embedded systems?

The most important principles are a clearly defined trust anchor, separation of signing authority from target-side verification material, controlled provisioning, and lifecycle-aware handling of key material. Chapters 2 and 3 showed that secure boot depends not only on cryptographic verification, but also on how trust anchors, keys, certificates, hashes, and provisioning steps are managed over time. The case study made this concrete in Sections 6.6 and 6.7 by separating AHAB development signing material, RAUC signing material, CST-generated SRK material, programmed SRKH fuse material, dm-verity metadata, and target-side verification material. The practical conclusion is that private signing keys should remain host-side, SRKH programming should be treated as a controlled provisioning boundary, and update-time metadata must remain synchronized with the artifacts it validates.

RQ2: How do selected secure boot implementations differ with respect to cryptographic design, tooling, openness, usability, and long-term maintainability?

The selected solutions differ primarily in how they anchor trust, expose provisioning workflows, support lifecycle transitions, and integrate with development and update workflows. Chapter 5 showed that UEFI Secure Boot is strongest as an open and policy-oriented firmware trust model, especially where updateable trust databases are important. NXP AHAB is the most directly applicable design for the selected i.MX 93 case study because it provides a hardware-rooted signed-image model and platform-specific provisioning path. TI AM64x secure boot is strong where explicit lifecycle transition and key-revision handling are central. Qualcomm secure boot is most relevant within Qualcomm-based Linux or UEFI-oriented products. Rockchip is treated more cautiously in this thesis, mainly as a contrast case for documentation visibility and engineering usability. The answer to RQ2 is therefore not that one secure boot model is universally best. Instead, the qualitative synthesis in Tables 4 and 5 shows that design suitability depends on engineering purpose, platform ecosystem, lifecycle needs, tooling availability, and documentation depth.

RQ3: How can secure boot be practically integrated on an NXP i.MX 93 using AHAB in a way that supports structured development and deployment workflows?

The case study shows that AHAB can be integrated as a structured signing, provisioning, and validation workflow around the existing Atlas boot flow. Section 6.6 describes how CST-generated SRK and SRKH material was generated, how the `flash_singleboot` artifact used by UUU was signed with matching CST material, how the SRKH fuse words were programmed and verified by U-Boot readback, and how ELE events were inspected in OEM Open. Section 7.2.6 and Table 14 show that a mismatched boot artifact produced an ELE key-hash mismatch indication, while the correctly signed `flash_singleboot` artifact produced no remaining ELE authentication events under the tested conditions. As scoped earlier, this validates the SRKH-programmed OEM Open workflow, not OEM Closed enforcement.

RQ4: How do supporting build, update, and release workflows, such as Yocto, RAUC, and artifact-traceability routines, affect the practical security and maintainability of such a system?

The case study shows that supporting tools and workflows are part of the practical security model rather than neutral implementation details. Yocto controls how the root filesystem, boot artifacts, WIC layout, initramfs, and dm-verity images are generated. RAUC controls how updates are delivered, verified, and installed into inactive slots. Sections 6.5 and 7.2.4 show that RAUC had to be adapted to use raw slots and a separate `/data/rauc` state directory, while Section 7.2.5 shows that update-time dm-verity metadata had to be synchronized per slot. Section 6.8 further shows that artifact traceability is necessary because valid components from different builds can fail if combined incorrectly.

RQ5: Which aspects of the studied solutions can be transferred into practical design principles and workflow guidance for similar embedded Linux systems?

The findings are best interpreted as transferable design principles and workflow guidance rather than as a fully general method validated across platforms. The most transferable principles are the separation of private signing material from target-side verification material, clear trust-anchor definition, controlled provisioning records, writable-state separation, root filesystem immutability, update-aware integrity metadata, and release artifact traceability. These principles are not tied to one command-line tool or one SoC.

Some findings are conditionally transferable. The RAUC and dm-verity integration shows a practical pattern for A/B updates with verified rootfs slots, but the exact implementation depends on the bootloader, update framework, initramfs design, and metadata storage mechanism used in a specific project. Similarly, Yocto-based artifact generation and WIC layout changes are useful for similar Yocto projects, but should not be treated as directly portable to all embedded Linux build systems. The least transferable findings are the AHAB-specific details: ELE event handling, SRK tables, SRKH fuse programming, UUU `flash_singleboot` behavior, and OEM Open/OEM Closed lifecycle semantics. These results are valuable because they show how a hardware-rooted secure boot workflow must be validated in practice, but their exact form is specific to the NXP i.MX 93 platform and the tested flashing workflow.

9.3 Main Contributions

The first contribution of this thesis is a structured analysis of secure boot, key management, and provisioning in Linux-based embedded systems. The theoretical chapters connect standards, research, vendor documentation, and workflow-oriented sources into a common framing of trust establishment, key lifecycle, provisioning, update security, and maintainability. This contribution is important because the problem is not limited to boot-time signature verification. It also includes how trust material is generated, stored, provisioned, updated, and connected to build and update workflows over the lifetime of a deployed device.

The second contribution is the comparative analysis of selected secure boot implementations in Chapter 5. The comparison provides a structured view of how different platforms and models handle cryptographic design, provisioning, key storage, lifecycle support, tooling, openness, and usability. This analysis makes the later case study easier to interpret because it shows that the NXP i.MX 93/AHAB workflow is one example of a broader class of hardware-rooted, vendor-specific secure boot models. The comparison also shows why maintainability and workflow clarity must be considered alongside cryptographic strength.

The third contribution is the practical case study on the NXP i.MX 93 and the company-provided Atlas Yocto baseline. The implementation added read-only root filesystem behavior, separated writable state under `/data` and `/config`, integrated dm-verity for A/B rootfs slots, adapted RAUC to install raw dm-verity images, synchronized slot-specific dm-verity metadata, and validated AHAB after SRKH fuse programming in OEM Open. The practical results in Section 7.2 show that both RAUC update directions succeeded while maintaining `/dev/mapper/rootfs` as the mounted root source, and that AHAB/ELE detected a mismatched boot artifact after SRKH programming while reporting no remaining events for the correctly signed `flash_singleboot` artifact under the tested conditions.

The fourth contribution is a workflow-oriented description of how the implemented mechanisms can be kept understandable and maintainable. Sections 6.7 and 6.8 separate private signing material, target-side verification material, dm-verity metadata, update state, and provisioning records, while Chapter 8 discusses the trade-offs, transferability, limitations, and societal implications of this design. This contribution is not a complete production release system or a universally validated secure boot method. It is a method-oriented case study and a set of transferable design principles that connect general secure boot concepts to concrete embedded Linux engineering decisions.

10. Future Work

The thesis validates a structured secure boot and integrity-verification workflow on one NXP i.MX 93 platform and one existing Yocto-based baseline. Future work should therefore focus on four directions: completing hardware-enforced AHAB validation, expanding negative and recovery testing, extending the protection model beyond read-only root filesystem integrity, and validating the workflow on additional platforms or industrial projects.

10.1 OEM Closed and Hardware-Enforced AHAB Validation

The most immediate continuation of this work is to complete the transition from SRKH-programmed OEM Open validation to OEM Closed hardware enforcement. In this thesis, AHAB signing was implemented, CST-based SRK and SRKH material was generated, SRKH fuses were programmed and verified by readback, and both matching and mismatching boot artifacts were inspected through ELE event output in OEM Open. Future work should transition a controlled test device to OEM Closed and then perform closed-state negative validation by attempting to boot unsigned, incorrectly signed, or wrong-key boot images.

This would clarify the boundary between the current OEM Open evidence and final production enforcement. In OEM Open, a mismatched image can produce an authentication event while the board still reaches later boot stages. In OEM Closed, the expected result is stronger: the hardware-rooted boot chain should reject an invalid early boot artifact. Such testing should be performed only on controlled test hardware with documented recovery expectations, because lifecycle closure can be irreversible or difficult to recover from.

10.2 Negative Update, Recovery, and Tamper Testing

Future work should also expand validation beyond the positive boot, update, and metadata-synchronization paths tested in this thesis. The case study showed that the normal RAUC A/B update flow can be preserved while using dm-verity-protected root filesystem slots, but it did not exhaustively test failure and recovery behavior. A production-oriented test campaign should include interrupted updates, power loss during slot installation, corrupted RAUC bundles, invalid bundle signatures, metadata mismatch, missing or stale `VERITY_A_*` / `VERITY_B_*` variables, corrupted U-Boot environment state, unresolved PARTUUID references, and long-duration update cycling.

The dm-verity part should also be tested with explicit tampering scenarios. Examples include modifying a rootfs block after installation, pairing a valid rootfs image with incorrect `dm-verity.env` metadata, or changing slot metadata so that the initramfs attempts to verify a rootfs image against the wrong root hash. These tests should confirm that the system fails safely by refusing to mount a mismatched root filesystem or by entering a defined recovery path. The present thesis should not report such tests as passed unless they are actually performed; they belong in future work until supported by validation evidence.

10.3 dm-integrity, dm-crypt, and Device-Local Secret Storage

The practical implementation focused on dm-verity because the main goal was integrity verification for a read-only root filesystem. Future work should evaluate mechanisms for protecting writable state and confidential data, because dm-verity is not designed to provide confidentiality or general writable-block integrity. This is especially relevant for `/data` and `/config`, which are separated from the immutable root filesystem but may still contain operational state, configuration, logs, or device-local credentials.

dm-integrity should be evaluated where writable block-level integrity is required. It is conceptually different from dm-verity because it targets writable integrity use cases rather than read-only verified images [27]. dm-crypt should be evaluated where confidentiality of stored data is part of the threat model, for example when physical access to storage media is realistic [26]. A complete design may also require a hardware-backed key source or protected key storage, such as a secure element, TPM, TEE-backed storage, EdgeLock-supported services, or another mechanism appropriate to the final product threat model.

These mechanisms should not be treated as replacements for secure boot. Instead, they would extend the protection model beyond the immutable root filesystem. Secure boot and dm-verity can help ensure that the system starts from authorized software and mounts an unmodified rootfs, while dm-crypt, dm-integrity, and hardware-backed key storage address different parts of the problem: confidentiality, writable-state integrity, and protection of device-local secrets.

10.4 Production Key Management and Release Automation

A further major continuation is to harden the operational workflow around keys, signing, provisioning, and release artifacts. The implementation separates private signing material from target-side verification material, but it does not define a complete production key-management process. Future work should specify how AHAB signing keys, RAUC bundle signing keys, SRK material, recovery keys, and device-local credentials are generated, stored, backed up, rotated, revoked, and retired.

The release workflow should also be automated further. A production-oriented pipeline should connect Yocto image generation, AHAB signing, RAUC bundle creation, dm-verity metadata generation, checksums, provisioning records, and validation evidence in one controlled process. Such a process should reduce manual handoff steps and make it harder to combine valid artifacts from different builds incorrectly. It should also produce release records that make it possible to audit which signed boot artifact, SRK/SRKH material, WIC image, RAUC bundle, rootfs image, and dm-verity metadata belong together.

10.5 Cross-Platform and Industrial Baseline Validation

Finally, future work should evaluate the transferability of the proposed workflow on additional platforms, baselines, or customer-like projects. The thesis identifies transferable design principles, but the practical implementation was performed on one NXP i.MX 93 platform and one existing Yocto-based baseline. Testing the same structured approach on another SoC, another boot chain, another update framework, or another industrial baseline would make it possible to

distinguish more clearly between NXP-specific findings and generally useful embedded Linux security guidance.

This future work should include both technical validation and usability evaluation. Technical validation would examine whether the same artifact-boundary reasoning applies when the trust anchor, provisioning mechanism, update metadata, or bootloader integration differs. Usability evaluation would examine whether engineers can apply the baseline consistently, understand the required records, avoid unsafe key-handling practices, and explain to customers which parts of the framework are generic and which parts remain project-specific.

References

- [1] A. Regenscheid, “Platform firmware resiliency guidelines,” National Institute of Standards and Technology, Gaithersburg, MD, 800-193, 2018. doi: 10.6028/NIST.SP.800-193.
- [2] E. Barker, “Recommendation for key management: Part 1 – general,” National Institute of Standards and Technology, Gaithersburg, MD, 800-57 Part 1 Rev. 5, 2020. doi: 10.6028/NIST.SP.800-57pt1r5.
- [3] E. Barker, “A framework for designing cryptographic key management systems,” National Institute of Standards and Technology, Gaithersburg, MD, 800-130, 2013. doi: 10.6028/NIST.SP.800-130.
- [4] G. Jank, S. Schmidt, and M. Koschuch, “Secure key management in embedded systems: A first proposal,” in *Proceedings of the 6th international conference on internet of things, big data and security (IoTBDS)*, 2021, pp. 81–91, doi: 10.5220/0010475600810091.
- [5] S. Joshi, K. Crowther, and J. Robinson, “Tradeoffs in key rotation strategies for industrial internet of things devices and firmware,” *Applied Sciences*, vol. 14, no. 21, p. 9942, 2024, doi: 10.3390/app14219942.
- [6] Yocto Project Contributors, “Yocto project concepts.” 2026, Accessed: May 21, 2026. [Online]. Available: <https://docs.yoctoproject.org/overview-manual/concepts.html>.
- [7] RAUC Contributors, “RAUC basics.” 2026, Accessed: May 21, 2026. [Online]. Available: <https://rauc.readthedocs.io/en/latest/basic.html>.
- [8] C. Profentzas, M. Günes, Y. Nikolakopoulos, O. Landsiedel, and M. Almgren, “Performance of secure boot in embedded systems,” in *Proceedings of the 15th annual international conference on distributed computing in sensor systems (DCOSS)*, 2019, pp. 198–204, doi: 10.1109/DCOSS.2019.00054.
- [9] D. Papp, M. Zsombor, and L. Buttyán, “TEE-based protection of cryptographic keys on embedded IoT devices,” *Annales Mathematicae et Informaticae*, vol. 53, pp. 245–256, 2021, doi: 10.33039/ami.2021.02.002.
- [10] R. Wang and Y. Yan, “A survey of secure boot schemes for embedded devices,” in *2022 24th international conference on advanced communication technology (ICACT)*, 2022, pp. 224–227, doi: 10.23919/ICACT53585.2022.9728840.
- [11] Arm Limited, “Arm Platform Security Architecture: Trusted boot and firmware update,” Arm Limited, DEN 0072, Version 1.1 Beta 0, 2020. Accessed: May 21, 2026. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/PSA/DEN0072-PSA_TBFU_1.1-BETA0.pdf.
- [12] B. Moran, H. Tschofenig, D. Brown, and M. Meriac, “A firmware update architecture for internet of things,” RFC Editor, RFC 9019, 2021. doi: 10.17487/RFC9019.
- [13] NXP Semiconductors, “i.MX 93 applications processor family.” 2026, Accessed: May 21, 2026. [Online]. Available: <https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-9-processors/i-mx-93-applications-processor-family-arm-cortex-a55-ml-acceleration-power-efficient-mpu:i.MX93>.

- [14] NXP Semiconductors, “i.MX linux user’s guide,” NXP Semiconductors, UG10163, Rev. LF6.18.2_1.0.0, 2026. Accessed: May 21, 2026. [Online]. Available: <https://www.nxp.com/docs/en/user-guide/UG10163.pdf>.
- [15] NXP Semiconductors, “i.MX 93 migration guide from i.MX 8M nano,” NXP Semiconductors, AN14022, Rev. 1, 2023. Accessed: May 21, 2026. [Online]. Available: <https://www.nxp.com/docs/en/application-note/AN14022.pdf>.
- [16] NXP Semiconductors, “Secure provisioning SDK (SPSDK).” 2026, Accessed: May 21, 2026. [Online]. Available: <https://www.nxp.com/design/design-center/software/development-software/secure-provisioning-sdk-spsdk:SPSDK>.
- [17] SLSA Community, “SLSA specification, version 1.2.” 2026, Accessed: May 21, 2026. [Online]. Available: <https://slsa.dev/spec/v1.2/>.
- [18] SLSA Community, “SLSA provenance.” 2026, Accessed: May 21, 2026. [Online]. Available: <https://slsa.dev/spec/v1.2/provenance>.
- [19] RAUC Project, “Using RAUC: hooks.” 2026, Accessed: May 21, 2026. [Online]. Available: <https://rauc.readthedocs.io/en/latest/using.html#hooks>.
- [20] The Linux Kernel Documentation Project, “ext4 general information.” 2026, Accessed: June 10, 2026. [Online]. Available: <https://docs.kernel.org/admin-guide/ext4.html>.
- [21] Linux man-pages project, “fstab(5) linux manual page.” 2026, Accessed: June 10, 2026. [Online]. Available: <https://man7.org/linux/man-pages/man5/fstab.5.html>.
- [22] Linux man-pages project, “findmnt(8) linux manual page.” 2026, Accessed: June 10, 2026. [Online]. Available: <https://man7.org/linux/man-pages/man8/findmnt.8.html>.
- [23] systemd, “systemd-tmpfiles.” 2026, Accessed: June 10, 2026. [Online]. Available: <https://www.freedesktop.org/software/systemd/man/systemd-tmpfiles-setup.service.html>.
- [24] The Linux Kernel Documentation Project, “Device mapper.” 2026, Accessed: June 10, 2026. [Online]. Available: <https://docs.kernel.org/admin-guide/device-mapper/index.html>.
- [25] The Linux Kernel Documentation Project, “dm-verity.” 2026, Accessed: May 21, 2026. [Online]. Available: <https://docs.kernel.org/admin-guide/device-mapper/verity.html>.
- [26] The Linux Kernel Documentation Project, “dm-crypt.” 2026, Accessed: June 10, 2026. [Online]. Available: <https://docs.kernel.org/admin-guide/device-mapper/dm-crypt.html>.
- [27] The Linux Kernel Documentation Project, “dm-integrity.” 2026, Accessed: June 10, 2026. [Online]. Available: <https://docs.kernel.org/admin-guide/device-mapper/dm-integrity.html>.
- [28] UEFI Forum, “Unified Extensible Firmware Interface specification, version 2.11,” UEFI Forum, 2024. Accessed: May 21, 2026. [Online]. Available: https://uefi.org/sites/default/files/resources/UEFI_Spec_Final_2.11.pdf.

- [29] Texas Instruments, “Secure boot — processor SDK AM64X documentation.” 2025, Accessed: May 21, 2026. [Online]. Available: https://software-dl.ti.com/processor-sdk-linux/esd/AM64X/11_01_05_03/exports/docs/linux/Foundational_Components_Secure_Boot.html.
- [30] Rockchip, “Boot option — rockchip open source document.” 2019, Accessed: May 21, 2026. [Online]. Available: https://opensource.rock-chips.com/wiki_Boot_option.
- [31] S. Aggarwal, “Key management standards and best practices for embedded systems.” 2025, Accessed: May 21, 2026. [Online]. Available: <https://www.hcltech.com/white-papers/key-management-standards-and-best-practices-for-embedded-systems>.
- [32] prpl Foundation, “An introduction to secure boot,” prpl Foundation, 2023. Accessed: May 21, 2026. [Online]. Available: <https://prplfoundation.org/wp-content/uploads/2023/08/prpl-Secure-Boot-Whitepaper-v1.0.pdf>.
- [33] Qualcomm Technologies, Inc., “Secure boot — qualcomm linux security guide.” 2025, Accessed: May 21, 2026. [Online]. Available: <https://docs.qualcomm.com/bundle/publicresource/topics/80-70018-11/secure-boot.html>.
- [34] NXP Semiconductors, “Processor-specific workflows — secure provisioning tool: i.MX 9x device workflow.” 2025, Accessed: May 21, 2026. [Online]. Available: https://docs.mcuxpresso.nxp.com/secure/latest/06_processor_specific_workflow.html.
- [35] Texas Instruments, “Key writer — TISCI user guide.” 2026, Accessed: May 21, 2026. [Online]. Available: https://software-dl.ti.com/tisci/esd/latest/6_topic_user_guides/key_writer.html.
- [36] Texas Instruments, “AM64x MCU+ SDK: Runtime key revision update example.” 2025, Accessed: May 21, 2026. [Online]. Available: https://software-dl.ti.com/mcu-plus-sdk/esd/AM64X/11_00_00_15/exports/docs/api_guide_am64x/EXAMPLES_RUNTIME_KEYREV.html.
- [37] Qualcomm Technologies, Inc., “Configure and secure boot with systemd-boot and UKI.” 2025, Accessed: May 21, 2026. [Online]. Available: https://docs.qualcomm.com/bundle/publicresource/topics/80-70022-27/configure_and_secure_boot_with_systemd_boot_and_uki.html.
- [38] Qualcomm Technologies, Inc., “Security tools — qualcomm linux security guide.” 2025, Accessed: May 21, 2026. [Online]. Available: <https://docs.qualcomm.com/bundle/publicresource/topics/80-70018-11/tools.html>.
- [39] Rockchip Electronics, “RK3288 datasheet, version 2.2,” Rockchip Electronics, 2017. Accessed: May 21, 2026. [Online]. Available: https://opensource.rock-chips.com/images/4/4e/Rockchip_RK3288_Datasheet_V2.2-20170301.pdf.
- [40] Rockchip Electronics, “RK1808 datasheet, version 1.2,” Rockchip Electronics, 2019. Accessed: May 21, 2026. [Online]. Available: https://opensource.rock-chips.com/images/4/43/Rockchip_RK1808_Datasheet_V1.2_20190527.pdf.
- [41] Texas Instruments, “Using extended OTP — TISCI user guide.” 2026, Accessed: May 21, 2026. [Online]. Available: https://software-dl.ti.com/tisci/esd/latest/6_topic_user_guides/extended_otp.html.

- [42] Texas Instruments, “OTP KEYWRITER — platform development kit (PDK).” 2020, Accessed: May 21, 2026. [Online]. Available: https://software-dl.ti.com/jacinto7/esd/processor-sdk-rtos-jacinto7/07_03_00_07/exports/docs/pdk_jacinto_07_03_00_29/docs/userguide/jacinto/modules/keywriter.html.
- [43] NXP Semiconductors, “AN14601: Ease CRA compliance with i.MX 93.” 2025, Accessed: May 21, 2026. [Online]. Available: https://docs.nxp.com/bundle/AN14601/page/topics/requirement_2_cybersecurity_requirements.html.
- [44] NXP Semiconductors, “RM00284: EdgeLock Enclave hardware security module API,” NXP Semiconductors, 2026. Accessed: May 21, 2026. [Online]. Available: <https://www.nxp.com/docs/en/reference-manual/RM00284.pdf>.
- [45] Texas Instruments, “AM64x MCU+ SDK: Security related tools.” 2026, Accessed: May 21, 2026. [Online]. Available: https://software-dl.ti.com/mcu-plus-sdk/esd/AM64X/latest/exports/docs/api_guide_am64x/TOOLS_SECURITY.html.
- [46] TianoCore, “What is TianoCore?” 2026, Accessed: May 21, 2026. [Online]. Available: <https://www.tianocore.org/>.
- [47] Texas Instruments, “TISCI user guide.” 2026, Accessed: May 21, 2026. [Online]. Available: <https://software-dl.ti.com/tisci/esd/latest/index.html>.
- [48] RAUC Contributors, “Integration — RAUC documentation.” 2026, Accessed: May 21, 2026. [Online]. Available: <https://rauc.readthedocs.io/en/latest/integration.html>.
- [49] The U-Boot Project, “U-Boot FIT signature verification.” 2026, Accessed: May 21, 2026. [Online]. Available: <https://docs.u-boot.org/en/latest/usage/fit/signature.html>.

Appendix

Appendix A: WIC Storage Layout

Shows the partitioning for the AHAB-signed boot image, rootfs A, rootfs B, data, and config, and how the `.ext4.verity` image is copied into the root filesystem slots.

```
part u-boot --source rawcopy --sourceparams="file=${DEPLOY_DIR_IMAGE}/imx-boot.tagged" --no-table --align 32

# boot partition - 100M for bootloader and potential initramfs image
part --source bootimg-partition --fixed-size 100M --fstype=vfat --label boot --active --align 8192

# root A partition - dm-verity image, copied byte-for-byte
part --source rawcopy --fixed-size 1000M \
--sourceparams="file=${IMGDEPLOYDIR}/${DM_VERITY_IMAGE}-${MACHINE}${IMAGE_NAME_SUFFIX}.${DM_VERITY_IMAGE_TYPE}.verity" \
--align 8192

# root B partition - dm-verity image, copied byte-for-byte
part --source rawcopy --fixed-size 1000M \
--sourceparams="file=${IMGDEPLOYDIR}/${DM_VERITY_IMAGE}-${MACHINE}${IMAGE_NAME_SUFFIX}.${DM_VERITY_IMAGE_TYPE}.verity" \
--align 8192

# data partition - 500M for persistent data
part --fixed-size 500M --fstype=ext4 --label data --align 8192

# config partition
part --fixed-size 64M --fstype=ext4 --label config --align 8192 --use-uuid --fsoptions="ro,nodev,nosuid,noexec"

# Use DOS partition table
bootloader --ptable msdos
```

Appendix B: Image Recipe Configuration

The image-level Yocto configuration used to select the custom WIC layout, enable a read-only root filesystem, and install the dm-verity initramfs kernel as the boot image.

```
WKS_FILE = "devsec-ab-storage.wks.in"

IMAGE_FEATURES += "read-only-rootfs"

IMAGE_BOOT_FILES:remove = "Image"
IMAGE_BOOT_FILES:append = " Image-initramfs-${MACHINE}.bin;Image"
```

Appendix C: Filesystem Mount Setup

Shows how the `/config` mount point is added, how it is mounted read-only, and how the `/data/rauc` runtime directory is created.

```
do_install:append() {
    install -d ${D}/boot/vfat
    install -d ${D}/config
    install -d ${D}${sysconfdir}/tmpfiles.d

    # Config partition: read-only by default.
    sed -i '\|[[[:space:]]/config[[[:space:]]|d' ${D}${sysconfdir}/fstab
    echo 'LABEL=config /config ext4 ro,nodev,nosuid,noexec 0 0' >> ${D}${sysconfdir}/fstab

    # Runtime-created RAUC data dir on the writable /data partition
    printf '%s\n' 'd /data/rauc 0755 root root -' > ${D}${sysconfdir}/tmpfiles.d/devsec-rauc.conf
```

```
}
FILES:${PN} += " /boot /boot/vfat /config ${sysconfdir}/tmpfiles.d/devsec-rauc.conf"
```

Appendix D: RAUC Target Configuration Patch

Shows how the installed RAUC system configuration is patched to use /data/rauc, raw root filesystem slots, and no filesystem resizing.

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"

# Patch installed RAUC system.conf for read-only/dm-verity rootfs slots.
do_install:append() {
    conf="${D}${sysconfdir}/rauc/system.conf"

    if [ ! -f "$conf" ]; then
        bbfatal "RAUC system.conf not found at $conf"
    fi

    # Keep RAUC status/state on the writable data partition.
    sed -i '/^data-directory=/d' "$conf"
    sed -i '/^[system\]/a data-directory=/data/rauc' "$conf"

    # dm-verity slots must be written as raw images, not mounted/resized as ext4.
    sed -i 's/^type=ext4$/type=raw/' "$conf"
    sed -i '/^resize=true$/d' "$conf"
}
```

Appendix E: U-Boot A/B Boot Script

Shows the U-Boot boot script used for A/B slot selection, loading the kernel and device tree from the boot partition, resolving the selected root filesystem by PARTUUID, and passing slot-specific dm-verity metadata to the initramfs.

```
test -n "${BOOT_ORDER}" || setenv BOOT_ORDER "A B"
test -n "${BOOT_A_LEFT}" || setenv BOOT_A_LEFT 3
test -n "${BOOT_B_LEFT}" || setenv BOOT_B_LEFT 3

setenv bootargs
setenv default_bootargs console=${console} ${optargs} rootfstype=ext4 rootwait
setenv fdtaddr ${fdt_addr_r}

for BOOT_SLOT in "${BOOT_ORDER}"; do
    if test "x${bootargs}" != "x"; then
        # skip remaining slots
    elif test "x${BOOT_SLOT}" = "xA"; then
        if test ${BOOT_A_LEFT} -gt 0; then
            echo "Found valid slot A, ${BOOT_A_LEFT} attempts remaining"
            setexpr BOOT_A_LEFT ${BOOT_A_LEFT} - 1
            setenv bootpart ${devnum}:1
            part uuid mmc ${devnum}:2 rootuuid
            setenv verityargs
            if test -n "${VERITY_A_ROOT_HASH}" && test -n "${VERITY_A_DATA_SIZE}" && test -n \
                "${VERITY_A_DATA_BLOCK_SIZE}"; then
                setenv verityargs "verity_root_hash=${VERITY_A_ROOT_HASH} verity_data_size=${VERITY_A_DATA_SIZE} \
                    verity_data_block_size=${VERITY_A_DATA_BLOCK_SIZE}"
            fi
        fi
    fi
done
```

```

        fi
        setenv bootargs "${default_bootargs} ro root=PARTUUID=${rootuuid} ${verityargs} rauc.slot=A"
    fi
    elif test "x${BOOT_SLOT}" = "xB"; then
        if test ${BOOT_B_LEFT} -gt 0; then
            echo "Found valid slot B, ${BOOT_B_LEFT} attempts remaining"
            setexpr BOOT_B_LEFT ${BOOT_B_LEFT} - 1
            setenv bootpart ${devnum}:1
            part uuid mmc ${devnum}:3 rootuuid
            setenv verityargs
            if test -n "${VERITY_B_ROOT_HASH}" && test -n "${VERITY_B_DATA_SIZE}" && test -n \
                "${VERITY_B_DATA_BLOCK_SIZE}"; then
                setenv verityargs "verity_root_hash=${VERITY_B_ROOT_HASH} verity_data_size=${VERITY_B_DATA_SIZE} \
                    verity_data_block_size=${VERITY_B_DATA_BLOCK_SIZE}"
            fi
            setenv bootargs "${default_bootargs} ro root=PARTUUID=${rootuuid} ${verityargs} rauc.slot=B"
        fi
    fi
done

if test -n "${bootargs}"; then
    saveenv
else
    echo "No valid slot found, resetting tries to 3"
    setenv BOOT_A_LEFT 3
    setenv BOOT_B_LEFT 3
    saveenv
    reset
fi

setenv fdtaddr 0x88000000
setenv devtype mmc

if test "${bootpart}" = "${devnum}:1"; then
    echo "Loading kernel from boot partition"
    fatload mmc ${bootpart} ${loadaddr} Image
    echo "Loading device tree from boot partition"
    fatload mmc ${bootpart} ${fdtaddr} ${fdtfile}
    echo "Starting kernel"
    booti ${loadaddr} - ${fdtaddr}
else
    echo "Loading kernel"
    ext4load mmc ${bootpart} ${loadaddr} /boot/zImage
    echo "Loading device tree"
    ext4load mmc ${bootpart} ${fdtaddr} /boot/${fdtfile}
    echo "Starting kernel"
    bootz ${loadaddr} - ${fdtaddr}
fi

```

Appendix F: RAUC Bundle Configuration for dm-verity Slots

Contains the RAUC bundle configuration used to package the dm-verity-protected root filesystem as a raw update payload and synchronize slot-specific dm-verity metadata after installation.

F.1 Original Atlas RAUC Bundle Recipe

This listing shows the pre-existing Atlas RAUC bundle recipe used as the baseline for comparison. It is included to make the later meta-devsec modifications explicit; the recipe itself is not presented as an original contribution of this thesis.

```
inherit bundle
```

```

IMAGE_RECIPE = "atlas-demo-image"
ROOTFS_IMAGE_FILE = "${IMAGE_RECIPE}-${MACHINE}.rootfs.ext4"

FILESETRAPATHS:prepend := "${THISDIR}/files:"
SRC_URI += " \
    file://hook.sh \
    file://atlas-demo.key.pem \
    file://atlas-demo.cert.pem \
"

RAUC_KEY_FILE ?= "${WORKDIR}/atlas-demo.key.pem"
RAUC_CERT_FILE ?= "${WORKDIR}/atlas-demo.cert.pem"

LICENSE="CLOSED"

RAUC_BUNDLE_COMPATIBLE ?= "ATLAS-DEMO"
RAUC_BUNDLE_VERSION ?= "2024-11-06"
RAUC_BUNDLE_HOOKS[file] ?= "hook.sh"

RAUC_BUNDLE_SLOTS = "rootfs"

RAUC_SLOT_rootfs = "${IMAGE_RECIPE}"
RAUC_SLOT_rootfs[fstype] ?= "ext4"
RAUC_SLOT_rootfs[file] ?= "${ROOTFS_IMAGE_FILE}"
RAUC_SLOT_rootfs[hooks] ?= "post-install"

RAUC_BUNDLE_FORMAT = "plain"

```

F.2 DevSec RAUC Bundle Append

Shows the meta-devsec override that selects the generated `.ext4.verity` root filesystem image, renames it to a generic `.img` payload inside the RAUC bundle, and includes the generated `dm-verity.env` metadata file in the bundle.

```

FILESETRAPATHS:prepend := "${THISDIR}/files:"

# Use the dm-verity-protected rootfs image as the source payload.
ROOTFS_IMAGE_FILE = "${IMAGE_RECIPE}-${MACHINE}.rootfs.ext4.verity"
RAUC_SLOT_rootfs[file] = "${ROOTFS_IMAGE_FILE}"

# Rename the payload inside the RAUC bundle so RAUC treats it as a generic raw image.
RAUC_SLOT_rootfs[rename] = "${IMAGE_RECIPE}-${MACHINE}.rootfs.img"

# Add the dm-verity metadata generated for the rootfs payload to the bundle.
VERITY_ENV_FILE ?= "${STAGING_VERITY_DIR}/${DM_VERITY_IMAGE}.${DM_VERITY_IMAGE_TYPE}.verity.env"

do_bundle:prepend() {
    verity_env="${VERITY_ENV_FILE}"

    if [ ! -f "$verity_env" ]; then
        verity_env="${TMPDIR}/work-shared/${MACHINE}/dm-verity/${DM_VERITY_IMAGE}.${DM_VERITY_IMAGE_TYPE}.verity.env"
    fi

    if [ ! -f "$verity_env" ]; then

```

```

        bbfatal "dm-verity metadata not found: $verity_env"
    fi

    install -m 0644 "$verity_env" "${BUNDLE_DIR}/dm-verity.env"
}

```

F.3 Raw Slot Post-Install Hook

Shows the RAUC post-install hook used to read `dm-verity.env` from the bundle and write slot-specific `VERITY_A_*` or `VERITY_B_*` metadata to the U-Boot environment without modifying the installed root filesystem image.

```

#!/bin/sh

die() {
    echo "$1" >&2
    exit 1
}

get_var() {
    eval "printf '%s' \"\${$1:-}\""
}

set_verity_var() {
    key="$1"
    value="$(get_var "$key")"

    [ -n "$value" ] || die "Missing dm-verity metadata value: $key"

    fw_setenv "VERITY_${slot}_${key}" "$value" || \
        die "Failed to write VERITY_${slot}_${key}"
}

case "$1" in
    slot-post-install)
        [ "${RAUC_SLOT_CLASS:-}" = "rootfs" ] || exit 0

        hook_dir="$(CDPATH= cd -- "$(dirname -- "$0")" && pwd -P)"
        env_file="${hook_dir}/dm-verity.env"

        [ -f "$env_file" ] || die "dm-verity.env not found in bundle"

        . "$env_file"

        slot="${RAUC_SLOT_BOOTNAME:-}"

        if [ -z "$slot" ]; then
            case "${RAUC_SLOT_NAME:-}" in
                rootfs.0) slot="A" ;;
                rootfs.1) slot="B" ;;
                *) die "Unable to map RAUC slot '${RAUC_SLOT_NAME:-unknown}' to boot slot" ;;
            esac
        fi

        case "$slot" in

```

```

        A|B) ;;
        *) die "Unsupported boot slot: $slot" ;;
    esac

    set_verity_var ROOT_HASH
    set_verity_var DATA_SIZE
    set_verity_var DATA_BLOCK_SIZE

    # Optional metadata. These are stored if present, but the current
    # initramfs path only requires the values above for appended-hash mode.
    for key in HASH_BLOCK_SIZE HASH_ALGORITHM SALT SEPARATE_HASH; do
        value="$(get_var "$key")"
        if [ -n "$value" ]; then
            fw_setenv "VERITY_${slot}_${key}" "$value" || \
                die "Failed to write VERITY_${slot}_${key}"
        fi
    done

    sync
    ;;
*)
    exit 1
    ;;
esac

exit 0

```

Appendix G: Initramfs dm-verity Metadata Override

Shows the initramfs image append that patches the dm-verity initramfs script so slot-specific metadata supplied on the kernel command line overrides the built-in `dm-verity.env` fallback.

```

ROOTFS_POSTPROCESS_COMMAND += "devsec_patch_dmverity_cmdline; "

devsec_patch_dmverity_cmdline() {
    script="${IMAGE_ROOTFS}/init.d/80-dmverity"

    if [ ! -f "$script" ]; then
        bbfatal "dm-verity initramfs script not found at $script"
    fi

    if grep -q "devsec: cmdline dm-verity metadata override" "$script"; then
        return
    fi

    override="${WORKDIR}/devsec-dmverity-cmdline-override.sh"

    printf '%s\n' \
' # devsec: cmdline dm-verity metadata override' \
' for arg in $(cat /proc/cmdline); do' \
'     case "$arg" in' \
'         verity_root_hash=*)' \
'             ROOT_HASH="$(printf "%s\n" "$arg" | sed "s/^verity_root_hash=//" )"' \
'             ;;' \
'         verity_data_size=*)' \
'             DATA_SIZE="$(printf "%s\n" "$arg" | sed "s/^verity_data_size=//" )"' \
'             ;;' \
'         verity_data_block_size=*)' \
'             DATA_BLOCK_SIZE="$(printf "%s\n" "$arg" | sed "s/^verity_data_block_size=//" )"' \
'             ;;' \
'     esac' \
' done' \
'

```

```

'         verity_separate_hash=*)' \
'         SEPARATE_HASH="$(printf "%s\n" "$arg" | sed "s/^verity_separate_hash=//" )" \
'         ;;' \
'     esac' \
' done' \
' > "$soverride"

sed -i "\|^[:space:]]*\.[[:space:]]*/usr/share/misc/dm-verity\.env[:space:]]*${r} $soverride" "$script"

if ! grep -q "devsec: cmdline dm-verity metadata override" "$script"; then
    bbfatal "Failed to patch dm-verity initramfs script"
fi
}

```

Appendix H: AHAB Signing and SRKH Programming Workflow

This appendix contains sanitized configuration and command excerpts for the AHAB signing and SRKH programming workflow. The workflow uses CST-generated AHAB key and certificate material, SRK table generation, SRKH fuse material generation, nxpele fuse programming, U-Boot readback, and ELE event inspection. Private keys, passphrase files, generated signed binaries, exact fuse-word values, and other sensitive provisioning records are not included in the report.

H.1 U-Boot AHAB Configuration

U-Boot recipe append and configuration fragment used to enable AHAB support in U-Boot.

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"
```

```
SRC_URI += "file://ahab.cfg"
```

```
CONFIG_AHAB_BOOT=y
```

H.2 CST-based AHAB Signing Configuration

The following sanitized SPSDK/imx-signer configuration was used with the CST-generated key and certificate material. The private key file name is shown to document the signing relationship, but private key contents, passphrases, generated signed binaries, and exact fuse values are not included.

```
family: mimx9352
```

```
revision: latest
```

```
srk_set: oem
```

```
used_srk_id: 0
```

```
signer: SRK1_sha384_secp384r1_v3_usr_key.pem
```

```
srk_table:
```

```
    flag_ca: false
```

```
    hash_algorithm: sha384
```

```
    srk_array:
```

- SRK1_sha384_secp384r1_v3_usr.crt.pem
- SRK2_sha384_secp384r1_v3_usr.crt.pem
- SRK3_sha384_secp384r1_v3_usr.crt.pem
- SRK4_sha384_secp384r1_v3_usr.crt.pem

H.3 CST-based SRK and Fuse Material Generation Script

The following script was used to generate the CST-style AHAB key package, SRK table, SRKH fuse material, and fuse programming command files. The script expects the AHAB password and serial number to be supplied through environment variables or an environment file. Actual password values, serial values, private keys, and generated fuse values are not included in the report.

```
#!/usr/bin/env bash
set -eu

# Path to the CST tarball
CSTPATH=${CSTPATH:-"${PWD}/cst-4.0.1.tgz"}
ENVFILE=${ENVFILE:-"${PWD}/ahab_values.env"}

# Sum (sha256) of the cst tarball, bump this on new version. For resiliency.
CSTSUM="60fffc243daa5e4e2ccfac8a9b74aec6d21446122a453fcb896dc52881d2a779d"

# Tarball with logs
TARBALL_NAME="ahab_keys_$(date -I).tar.gz"

# Tarball with minimal content for bitbake consumption
TARBALL_NAME_BB="ahab_keys_bitbake_$(date -I).tar.gz"

# Extract the top-level directory from the tarball
CSTROOT=$(tar -tf "${CSTPATH}" | head -n1 | cut -d/ -f1)

ROOT=$(pwd)
CSTDIR=${ROOT}/${CSTROOT}
echo "Using CST directory: ${CSTDIR}"

if printf '%s %s\n' "$CSTSUM" "$CSTPATH" | sha256sum -c - >/dev/null 2>&1; then
    echo "Checksums OK"
else
    echo "Checksum FAILED" >&2
    exit 1
fi

if [ -f "${ENVFILE}" ]; then
    echo "Sourcing environment data from: ${ENVFILE}"
    . "${ENVFILE}"
else
    echo "Not using env file. Continuing."
```

```

fi

: "${AHAB_PASSWORD:?AHAB_PASSWORD is undefined}"
: "${AHAB_SERIAL:?AHAB_SERIAL is undefined}"

if [ ! -d ${CSTDIR} ]; then
    tar -C . -xf "${CSTPATH}"
fi

generate() {
    cd "${CSTDIR}/keys"

    echo ${AHAB_SERIAL} >serial

    echo ${AHAB_PASSWORD} >key_pass.txt
    echo ${AHAB_PASSWORD} >>key_pass.txt

    ./ahab_pki_tree.sh \
        -existing-ca n \
        -kt ecc \
        -kl p384 \
        -da sha384 \
        -duration 2000 \
        -srk-ca n \
        2>&1 | tee ahab_pki_tree.log
}

srkgen() {
    cd "${CSTDIR}/crts"

    SRK1_CERT="SRK1_sha384_secp384r1_v3_usr.crt.pem"
    SRK2_CERT="SRK2_sha384_secp384r1_v3_usr.crt.pem"
    SRK3_CERT="SRK3_sha384_secp384r1_v3_usr.crt.pem"
    SRK4_CERT="SRK4_sha384_secp384r1_v3_usr.crt.pem"

    SRK_CERT_LIST="${SRK1_CERT},${SRK2_CERT},${SRK3_CERT},${SRK4_CERT}"

    ../linux64/bin/srktool \
        --ahab_ver \
        --digest sha256 \
        --sign_digest sha384 \
        --table SRK1234table.bin \
        --efuses SRK1234fuse.bin \
        --fuse_format 1 \
        --certs ${SRK_CERT_LIST} \
        2>&1 | tee srktool.log
}

```

```

gen_fuse_cmds() {
    cd "${ROOT}"
    HEX=$(od -An -t x4 "${CSTDIR}"/crts/SRK1234fuse.bin)

    rm -f "${CSTDIR}"/crts/*-fuses.{cmd,bcf}

    i=0
    for x in ${HEX}; do
        echo "fuse prog 16 $i 0x$x" >>"${CSTDIR}"/crts/uboot-fuses.cmd
        i=$((i + 1))
    done

    a=128
    for x in ${HEX}; do
        echo "write-fuse --index $a --data 0x$x" >>"${CSTDIR}"/crts/nxpele-fuses.bcf
        a=$((a + 1))
    done
}

package() {
    cd "${ROOT}"

    tar --owner=0 --group=0 --numeric-owner \
        -C "${CSTDIR}" -czf "${TARBALL_NAME}" \
        --transform "s|^|ahab_keys_$(date -I)/|" \
        crts keys
}

package_for_bitbake() {
    cd "${CSTDIR}"

    tar --owner=0 --group=0 --numeric-owner \
        -czf "${ROOT}/${TARBALL_NAME_BB}" \
        --transform "s|^|iMX9/|" \
        $(find crts -type f \( -name 'SRK*' -o -iname 'ca1*' \)) \
        $(find keys -type f \( -iname '*.pem' -o -iname '*.der' \)) \
        $(find keys -type f -name 'index*.txt') \
        $(find keys -type f -name 'serial' -o -name 'key_pass.txt') \
        $(find crts -type f -name '*.bcf' -o -name '*.cmd')
}

generate
srkgen
gen_fuse_cmds
package
package_for_bitbake

cd ${ROOT}

```

```
echo -e "#####"
echo -e "All good! Keys packaged"
echo -e "#####"
```

H.4 SRKH Fuse Programming and Readback Commands

The generated `nxpele-fuses.bcf` file was used to program the SRKH fuse words:

```
nxpele -f mimx9352 \
  -d uboot_serial \
  -p /dev/ttyACM0 \
  --uboot-prompt "u-boot=>" \
  batch ahab/dev/iMX9/crts/nxpele-fuses.bcf
```

The programmed values were then read back from U-Boot using bank 16, words 0 through 7:

```
fuse read 16 0
fuse read 16 1
fuse read 16 2
fuse read 16 3
fuse read 16 4
fuse read 16 5
fuse read 16 6
fuse read 16 7
```

The exact readback values are omitted from the report. The readback was compared against the generated CST fuse material and matched the expected values.

H.5 Manual AHAB Signing and ELE Validation Commands

The UUU flashing flow used the `flash_singleboot` boot artifact as the first argument. This artifact was signed for the final AHAB validation:

```
export SIG_TOOL_PATH="$HOME/tools/spsdk-venv/bin"
export SIG_DATA_PATH="$REPO/ahab/dev/iMX9"

./tools/nxp-imx-signer/src/imx_signer \
  -i "tmp/deploy/images/$MACHINE/imx-boot-imx93-11x11-lpddr4x-frdm-atlas-sd.bin-flash_singleboot" \
  -c "$REPO/ahab/dev/configs/spsdk_ahab_imx93_cst.yaml" \
  -d

# The signed flash_singleboot artifact was then used as the first argument to the UUU flashing command:
sudo ./uuu -b emmc_all \
  imx-boot-imx93-11x11-lpddr4x-frdm-atlas-sd.bin-flash_singleboot \
  atlas-demo-image-imx93-11x11-lpddr4x-frdm-atlas.rootfs.wic.zst

# After flashing and booting, ELE events were inspected through U-Boot serial access:
nxpele -f mimx9352 \
  -d uboot_serial \
  -p /dev/ttyACM0 \
  --uboot-prompt "u-boot=>" \
  get-events
```