An Efficient Secure Boot Mechanism Leveraging DICE as a Use Case

Utku Budak*, Malek Safieh†, Yigit Arda Ozen*, Fabrizio De Santis†, Georg Sigl*
*Chair of Security in Information Technology, Technical University of Munich, Germany
{utku.budak, yigit.ozen, sigl}@tum.de
†Siemens AG, Foundational Technologies, Munich, Germany
{malek.safieh, fabrizio.desantis}@siemens.com

***Abstract*—Secure boot ensures that only verified code is executed at boot time. It typically relies on asymmetric cryptography, which may pose boot time challenges for time-critical devices. We, therefore, propose an efficient secure boot (ESB) mechanism that extends the asymmetric cryptography-based approach with symmetric cryptography to reduce boot time. To demonstrate the practicality, an extended Device Identifier Composition Engine (DICE) architecture is leveraged as a use case. The evaluation results on an ARM-based MCU show that the proposed mechanism reduces boot time for regular boots while introducing a slightly higher overhead only during the initial boot phase.**

***Index Terms*—Secure Boot, DICE, Trusted Computing**

I. Introduction

The security of embedded devices must be addressed starting from the boot phase by detecting any malicious code [1]. Secure boot [2] ensures that the device boots only with verified code; otherwise, the boot process is terminated. However, it introduces a boot time overhead due to the extra verification steps [3]. In particular, secure boot implementations [4], [5] commonly rely on asymmetric cryptography, despite its relatively high computational cost [6]. This arises from the challenges of symmetric cryptography, such as key management [7], as the server must securely manage individual keys for each device, and the devices require secure storage for these keys. As a result, implementing secure boot becomes particularly challenging for devices with strict boot time requirements.

Another typical boot time security mechanism is the Device Identifier Composition Engine (DICE) [8], [9], designed for resource-constrained devices that cannot afford a Trusted Platform Module (TPM) [10] to measure boot time code integrity. It requires only an immutable memory, such as read-only memory (ROM), to store the DICE engine and a mechanism to latch the Unique Device Secret (UDS). The output of the DICE architecture, known as the Compound Device Identifier (CDI), can then be used as attestation evidence in a remote attestation. In [11], the DICE architecture is also used to provide single-layer secure boot, while [12] extends it to support multi-layer secure boot by both measuring and verifying the code.

II. General System Design

In this paper, we address the aforementioned challenges by introducing an efficient secure boot (ESB) mechanism that exploits advantages of both asymmetric and symmetric cryptography-based mechanisms, using the extended DICE architecture proposed in [12] as a representative use case.

A. Conceptual Overview

The high-level workflow is illustrated in Fig. 1, where the booted code is first symmetrically verified. At the initial boot, during the first start-up before deployment or after an update, symmetric verification fails due to the absence of a valid reference symmetric integrity check value (ICV). Hence, the code is verified through asymmetric digital signature. Upon successful verification, the ICV computed during symmetric verification is attached to the code as a reference ICV; otherwise, the boot process is terminated. In subsequent regular boots, the code can be directly verified using symmetric cryptography, eliminating the need for asymmetric operations and thereby reducing boot time. The proposed mechanism incurs additional overhead, such as ICV calculation and storage, only during the initial boot.

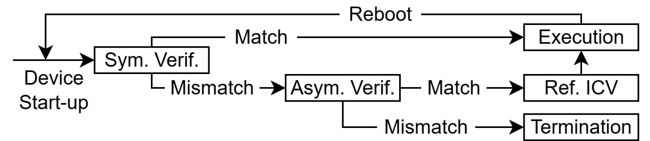

Fig. 1. Conceptual Overview of the Efficient Secure Boot (ESB) Mechanism.

B. Leveraging DICE as a Use Case

To show feasibility, the proposed mechanism is realized using DICE as a use case, as illustrated in Fig. 2, consisting of five main parts: (i) hash computation, (ii) standard DICE flow [9], (iii) DICE-based secure boot [12], (iv) secure boot control unit proposed in this paper, and (v) asymmetric cryptography-based secure boot. In the proposed design, the Root of Trust (RoT) is stored in ROM, whereas the remaining layers, along with signatures σ_0, σ_1, and reference ICVs are stored in flash memory. The UDS and the public key, pk_0, are stored in eFuses, whereas pk_1 is stored in flash with the bootloader.

The workflow of the ESB mechanism can be divided into two phases: 1) Initial Boot and 2) Regular Boot.

1) Initial Boot: begins with the RoT hashing the bootloader, denoted as (i) in Fig. 2, as follows:

$$H_{BL} = H(Bootloader). \quad (1)$$

Next, the RoT derives two additional keys from the UDS using a key derivation function (KDF) with domain separation [13] as follows:

$$UDS_{S/E} = KDF(UDS, ID_{S/E}). \quad (2)$$

978-3-9826741-1-7/DATE26/© 2026 EDAA

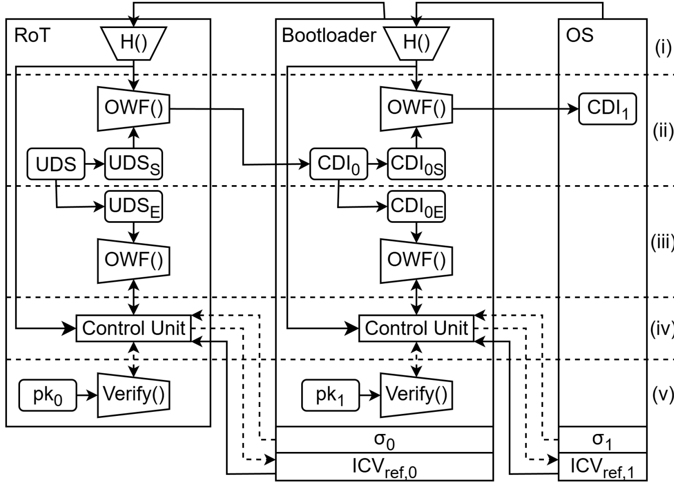

Fig. 2. Efficient Secure Boot (ESB) Mechanism Leveraging DICE as a Use Case. Solid arrows are executed during every boot, whereas dashed arrows are executed only during the initial boot.

These are the UDS_S used in the standard DICE flow, and the UDS_E used for DICE-based secure boot [12]. The RoT then computes the ICV of layer 0, denoted as ICV_0, using a one-way function (OWF), such as a hash-based message authentication code (HMAC), as follows:

$$ICV_0 = OWF(UDS_E, H_{BL}), \tag{3}$$

and compares it with the reference ICV, $ICV_{ref,0}$. During initial boot, the comparison fails, since the $ICV_{ref,0}$ is not present. Therefore, the RoT first verifies the signature of the bootloader, denoted as (v) in Fig. 2, as follows:

$$Verify(pk_0, H_{BL}, \sigma_0). \tag{4}$$

If verification fails, the RoT terminates the boot process. If it succeeds, the RoT stores the ICV_0 as $ICV_{ref,0}$ in flash together with the bootloader, and executes the bootloader. The bootloader verifies the operating system (OS) in the same way.

2) Regular Boot: starts with the RoT measuring bootloader, as in Eq. (1) and then calculating the ICV, ICV_0 as in Eq. (3). Next, the RoT verifies the ICV_0 against the $ICV_{ref,0}$ over the bootloader. If the verification succeeds, the RoT directly proceeds with executing the upcoming layer, i.e., the bootloader. Otherwise, if the verification fails, the RoT again verifies the signature, σ_0, of the bootloader and stores the ICV_0 as $ICV_{ref,0}$ in case of successful signature verification. The bootloader again follows the same steps for verifying the OS.

III. Implementation & Results

The proposed ESB mechanism is implemented on the LPCXpresso55S69 [14], featuring an LPC55S69 MCU running at 96 MHz. The selected MCU provides a SHA256 hash hardware (HW) accelerator, which allows assessing its impact on hashing and OWF computations. MCUboot [15] is used as the bootloader and extended with the DICE-based secure boot proposed in [12], leveraging OpenDICE [16]. Zephyr [17] is selected as the OS and built with MCUboot, supporting mbed TLS. The evaluation is limited to MCUboot booting the application (OS), as the RoT is often vendor-specific and provisioned at manufacturing time. In compliance with MCUboot [15], SHA-512 is used with Ed25519, while SHA-256 is used with RSA-2048/3072 to compute the application hash. The results of MCUboot booting a 50 KB application with standard asymmetric cryptography-based secure boot are presented in Table I.

TABLE I
Boot Time Results of Standard Secure Boot

Algorithm	Boot Time [ms]	
	SHA256/HW	SHA512
RSA-2048	182.6/107.8	-
RSA-3072	280.1/207.7	-
Ed25519	-/-	167.6

The results of MCUboot booting the same application with the ESB mechanism using HMAC-SHA256/512 as OWF are shown in Table II. Compared with the standard secure boot, the proposed mechanism adds overhead only during the initial boot, arising from the computation and storage of the ICV. However, an essential acceleration is achieved in regular boots, especially using the hash HW accelerator. This gain is achieved through accelerated hash and OWF computations and by eliminating asymmetric cryptography during regular boot.

TABLE II
Boot Time Results of Efficient Secure Boot (ESB)

Algorithm	Initial Boot Time [ms]		Regular Boot Time [ms]	
	SHA256/HW	SHA512	SHA256/HW	SHA512
RSA-2048	184.4/113.2	-	89.0/15.2	-
RSA-3072	293.7/214.3	-	89.2/15.5	-
Ed25519	-	174.0	-	133.7

For an in-depth analysis, the boot time results of the proposed ESB mechanism for regular boot and the standard secure boot across different applications are shown in Fig. 3. The results indicate that the proposed mechanism provides an almost constant reduction in boot time compared with the standard secure boot, regardless of the application size. This improvement is a direct consequence of replacing asymmetric cryptography with symmetric cryptography. However, the boot time reduction is relative, i.e., the percentage of the achieved reduction decreases for larger applications. This is because the hash measurement of the larger applications dominates the boot time, which highlights the benefit of using a hash HW accelerator.

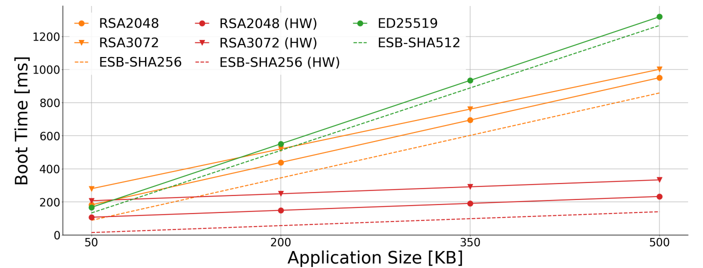

Fig. 3. Boot Time Results of Standard Secure Boot and Regular Boot Phase of Efficient Secure Boot (ESB) for Different Application Sizes.

References

[1] H. A. Noman and O. M. Abu-Sharkh, "Code injection attacks in wireless-based internet of things (iot): A comprehensive review and practical implementations," *Sensors*, vol. 23, no. 13, p. 6067, 2023.

[2] Microsoft, "Secure boot." https://learn.microsoft.com/. Accessed: August 2025.

[3] C. Profentzas, M. Günes, Y. Nikolakopoulos, O. Landsiedel, and M. Almgren, "Performance of secure boot in embedded systems," in *2019 15th International conference on distributed computing in sensor systems (DCOSS)*, pp. 198–204, IEEE, 2019.

[4] Texas Instruments, "Secure boot on embedded sitara processors." https://www.ti.com/lit/wp/spry305a/spry305a.pdf?ts=1736351132509, 2018.

[5] NXP Semiconductors, "Step by step process to prepare secure boot for lpc54s0xx - an13390." https://www.nxp.com/docs/en/application-note/AN13390.pdf, 2021.

[6] C. Paar and J. Pelzl, *Understanding cryptography*, vol. 1. Springer, 2010.

[7] E. Barker and Q. Dang, "Nist special publication 800-57 part 1, revision 4," *NIST, Tech. Rep*, vol. 16, p. 51, 2016.

[8] "Hardware requirements for device identifier composition engine level 00, revision 78," 2018.

[9] "Dice layering architecture version 1.0, revision 0.19," 2020.

[10] "Trusted platform module library specification, family 2.0, level 00, revision 01.59," 2019.

[11] U. Budak, F. De Santis, and G. Sigl, "A lightweight firmware resilience engine for iot devices leveraging minimal processor features," in *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*, pp. 486–491, IEEE, 2024.

[12] U. Budak, M. Safieh, F. De Santis, and G. Sigl, "A cyber-resilient dice architecture for resource-constrained devices," in *International Conference on Availability, Reliability and Security*, pp. 74–91, Springer, 2025.

[13] H. Krawczyk and P. Eronen, "Hmac-based extract-and-expand key derivation function (hkdf), 2010."

[14] "Lpcxpresso55s69 development board with lpc55s69 dual core arm cortex-m33."

[15] "Mcuboot: Secure boot for 32-bit microcontrollers." https://docs.mcuboot.com/. Accessed: September 2025.

[16] "Open profile for dice." https://pigweed.googlesource.com/open-dice/. Accessed: August 2025.

[17] "The zephyr project." https://www.zephyrproject.org/. Accessed: August 2025.