Asymmetric Attestation Protocol for Constrained IoT Devices

TU Delft

Asymmetric Attestation Protocol for Constrained IoT Devices

By

Nikolaos Skartsilas

In partial fulfilment of the requirements for the degree of:

Master of Science
in Embedded Systems

at the Delft University of Technology,
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS),
to be defended publicly on December 17, 2025, at 14:00 PM.

Supervisor:	Dr.ir. M. Taouil MSc, TU Delft
Thesis committee:	Dr.ir. M. Taouil MSc, TU Delft
	Prof.dr. K.G. (Koen) Langendoen, TU Delft

Preface

This thesis was conducted in collaboration with **Intrinsic ID** and addresses security mechanisms for resource-constrained embedded systems. The work combines protocol design, security analysis, and a prototype implementation to evaluate the feasibility of a DICE-rooted, certificate-based asymmetric attestation approach on microcontroller-class devices.

Due to confidentiality constraints associated with the industrial development environment at Intrinsic ID, the complete prototype source code and certain implementation-specific artifacts cannot be included in this report. To support scientific evaluation, the thesis documents the design through explicit assumptions and requirements, protocol flows, pseudocode-level descriptions, and quantitative measurements on representative hardware.

Contents

Abstract

The evolution of computing systems, particularly in the Internet of Things (IoT), has emphasized openness to support innovation, but this same openness introduces critical security challenges. Modern cyber-attacks are increasingly sophisticated and persistent, exposing the limitations of traditional software-only defenses. IoT devices, often deployed in hostile environments and subject to stringent constraints in power, memory, and cost, lack the robust security mechanisms required for trust and resilience, especially as the demand for remote software updates grows.

To address these challenges, this thesis proposes a scalable and cost-effective security architecture that supports hardware-rooted identity, remote attestation, and secure device updates for resource-constrained embedded devices. The design is grounded in modest hardware assumptions compatible with commercial IoT platforms. A statistically unique, device-specific secret anchors the root of trust, enabling verifiable software identity throughout the device lifecycle. Building on the Device Identifier Composition Engine (DICE) standard, an asymmetric attestation protocol is developed specifically for constrained environments.

The architecture is validated through a prototype implementation on an STM32 microcontroller, demonstrating secure remote attestation via server communication. Performance measurements, including clock cycles and memory utilization, alongside a structured security analysis offer insight into the feasibility and resilience of the proposed solution. This work contributes to the advancement of DICE-based architectures by providing a practical and secure framework for verifying software execution in trusted IoT devices.

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude to my academic supervisors and committee members, Dr.ir. Mottaqiallah Taouil and Prof.dr. Koen Langendoen, for their guidance, feedback, and support throughout this thesis.

This work was conducted in collaboration with Intrinsic ID. I would particularly like to thank my company supervisor, Georgios Selimis, for his mentorship, technical input, and continuous support during the implementation and evaluation phases. I also thank my colleagues at Intrinsic ID for the constructive discussions and assistance.

Finally, I would like to thank my family, my friends, and Ifigeneia for their encouragement and support throughout my studies.

1 Introduction

This chapter concerns the motivation around this thesis, presents a summary of research studies that constitute the state of the art, describes the main contributions of the work done and provides an outline for the rest of the thesis. First, Section 1.1 refers to the importance of the rising sector of the Internet of Things (IoT), highlighting the need for effective security strategies against sophisticated attacks and threats. This section also describes the security challenges derived from such threats towards establishing a robust holistic security solution for end-to-end IoT nodes. Section 1.2 investigates past and present security solutions spanning many disciplines from software to hardware, focusing on attestation schemes. Section 1.3 describes the main contributions of this study, while Section 1.4 concludes the chapter with a short description of the other chapters of this thesis.

1.1 Motivation

The proliferation of open, autonomous embedded systems connected via the Internet or other networks has led to the emergence of the Internet of Things (IoT). The global deployment of IoT devices now numbers in billions. According to Gartner, 5.8 billion enterprise and automotive IoT endpoints were projected to be in use by the end of 2020 [1]. Forecasts suggest this growth will continue, with IDC anticipating that IoT devices will generate 79.4 zettabytes of data annually by 2025 [2]. McKinsey Digital further estimates that IoT could contribute up to $11 trillion annually to the global economy by 2025, potentially boosting corporate profits by 21% as early as 2022 [3]. These projections have incentivized major firms to invest in innovative IoT technologies that enhance business processes and improve operational efficiency. In particular, Industrial IoT (IIoT) has gained traction by promoting automation and leveraging big data analytics to reduce costs and enhance customer insights.

The IoT infrastructure is characterized by an openness that facilitates innovation but also presents substantial security challenges. This openness makes IoT systems attractive targets for cybercriminals, who exploit systemic vulnerabilities [4]. As shown in Figure 1.1, the expected surge in connected devices will only increase the attack surface. Ensuring the reliable and secure operation of IoT systems has thus become an increasingly complex and challenging task. Common wireless communication technologies, such as Bluetooth and Wi-Fi, possess known vulnerabilities that can be exploited by adversaries [6]. Further complicating matters are the resource constraints and physical inaccessibility of many IoT deployments, which hinder the implementation of robust security protections [7].

An expanded attack surface increases the risk of system compromise, yet many organizations fail to prioritize security due to limited expertise or budget constraints [8]. As a result, device manufacturers must prioritize security as a design imperative, not an afterthought. The heterogeneity of IoT ecosystems further complicates security: the threats concerning industrial sensors differ considerably from those affecting smart appliances or consumer wearables [9]. Privacy violations, intellectual property theft, impersonation, and device cloning are among the diverse threats IoT devices face. Notably, IoT systems can be compromised even without network connectivity. Security attacks span multiple domains, ranging from software-level exploits to direct hardware manipulation targeting the device's physical components. In 2015, security researchers successfully demonstrated remote hijacking of vehicle systems, manipulating functions ranging from braking to infotainment [10]. Financial and operational damage from such incidents is substantial for major companies, some of which report recovery costs exceeding $500,000 per breach [11]. Healthcare institutions have also been targeted,

with ransomware attacks encrypting critical patient data and demanding payment for its release [12]. High-profile hardware vulnerabilities like Meltdown and Spectre further underscore the risks inherent in modern computing architectures [13].

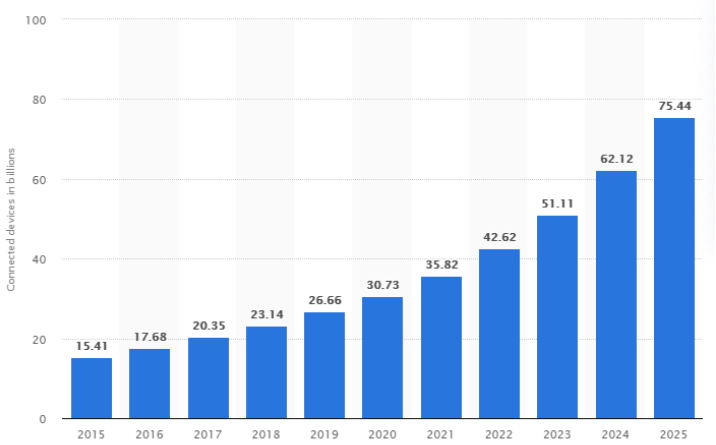

Figure 1.1 Connected IoT Devices Forecast by 2025 [5]

As connectivity among IoT devices grows, the risk of malware-based attacks increases significantly. A single infected device can spread malware to others in the network, triggering a chain reaction that may compromise sensitive system information. The absence of strong security mechanisms at the device level can make such infections especially damaging. Although previous research has explored the types of malware threats IoT systems face, relatively little attention has been given to developing effective countermeasures for infected or compromised IoT devices.

Preserving privacy and ensuring security in IoT systems entails a range of technical and procedural challenges. First, trust must be established across the entire lifecycle of each device, from development and manufacturing to deployment and decommissioning [14]. Equally important is the protection of secrets within embedded devices. Secure key storage and tamper-resistant provisioning mechanisms are essential. Although modern cryptographic algorithms can address many system-level vulnerabilities, long-term security remains elusive, particularly in the case of side-channel attacks and speculative execution flaws. Firmware updates represent another vector of vulnerability, exposing IoT endpoints to attack if not secured properly. Physical tampering and the lack of standardized defenses further complicate secure system design. Standardization efforts, while necessary, progress slowly and often lag behind emerging threats. To reduce liability and improve resilience, security considerations must be integrated into the design phase of IoT products. The next section surveys existing security solutions that form the state of the art in protecting embedded IoT systems.

1.2 State of the Art

The challenges outlined in the previous section have motivated researchers to explore security techniques aimed at preventing malicious adversaries from compromising critical operations in IoT systems. This section reviews existing security solutions for protecting IoT endpoint devices, with an emphasis on attestation techniques.

Memory protection is a common hardware-based countermeasure, involving the use of Memory Protection Units (MPUs) [15] and Memory Management Units (MMUs) [16] to prevent unauthorized access to sensitive memory regions by unprivileged software components. Early solutions, such as the segmentation mechanism introduced in Intel's 80286 architecture, linked memory segments to privilege levels, thereby enforcing access restrictions. More recently, Execution-Aware

Memory Protection (EA-MPU) [17] has been proposed to associate code segments with specific data regions, allowing for finer-grained isolation of software modules within a trusted runtime environment. Remote attestation is a foundational technique for verifying the internal state of an untrusted device (the *prover*) by a trusted entity (the *verifier*) [18]. It enables use cases such as secure firmware updates, patch validation, and system integrity checks across embedded platforms. Attestation protocols generally fall into three categories: (1) software-based, (2) hardware-based, and (3) hybrid schemes.

Software-based attestation techniques leverage platform constraints to detect unauthorized modifications. One notable example is Pioneer, proposed by Seshadri et al. in 2005 [19], which implements a software-based attestation technique by computing memory checksums using a verifier-specified algorithm. The design introduces intentional timing overhead to detect unauthorized modifications, as deviations in execution time can reveal compromise of the attestation code or underlying memory state. Similar methods have been adapted for various embedded platforms [20–23]. Despite their conceptual appeal, software-only attestation schemes remain vulnerable to attacks and typically depend on restrictive assumptions about the adversary's capabilities. These include the requirement of exclusive one-to-one communication between the verifier and the prover, which prohibits third-party attestation and reduces protocol flexibility. Moreover, the lack of persistent secret storage on the device necessitates such constraints. To address these shortcomings, some proposals introduce minimal hardware extensions for securely storing cryptographic secrets [24], [25]. However, even lightweight trust anchors often demand exclusive access to system resources, making it infeasible to support multiple trusted execution environments concurrently. As a result, software-based attestation schemes are generally unsuitable for deployment in realistic, multi-context IoT scenarios.

Hardware-based attestation schemes gained early traction with mechanisms such as Secure Boot [26], introduced by Arbaugh et al. in 1997. Secure Boot verifies the integrity of system components at startup using a trusted bootloader that hashes memory content and compares it with a signed reference hash stored in ROM. Another widely adopted mechanism is the Trusted Platform Module (TPM) [27], which extends this principle using Platform Configuration Registers (PCRs) for securely storing integrity measurements. TPM-enabled systems establish trust through early boot processes, supported by the BIOS. Several experimental TPM-based approaches have been studied [28–30]. Commercial alternatives, such as ARM TrustZone [31], implement secure execution environments via a set of privileged registers and isolated on-chip memory. However, these static root-of-trust models do not adequately protect against runtime attacks such as Return-Oriented Programming (ROP) [32]. Addressing these threats requires dynamic root-of-trust architectures that provide runtime integrity verification [33]. These approaches do not provide dynamic root of trust, while the cost of the TPM module is prohibitive for low-end embedded devices.

Dynamic Root of Trust (DRT) architectures extend TPM specifications and has been implemented by vendors such as Intel and AMD [34, 35]. These systems dynamically isolate memory regions and use CPU instructions to reset PCRs and measure memory contents during execution. While TrustZone is primarily based on secure boot, it has also been repurposed to support DRT functionality [36, 37]. McCune et al. introduced *Flicker* [38], a system architecture that utilizes Intel TXT and AMD SVM technologies to establish a dynamic root of trust on commodity computing platforms. Flicker ensures the secure execution of a minimal, isolated code segment, referred to as a Piece of Application Logic (PAL), even when critical system components such as the BIOS, operating system, or DMA subsystems are compromised by an adversary. Building on this foundation, *TrustVisor* was later proposed by McCune et al. [39] as an enhancement that integrates a minimal hypervisor to reduce performance overhead while maintaining strong isolation guarantees for PALs. Despite their robustness, these architectures rely heavily on platform-specific hardware features and impose

significant resource demands, making them impractical for constrained IoT environments. Other research efforts have explored trust establishment in remote systems using similar principles [40], [41].

To overcome the limitations of pure software and heavyweight hardware schemes, hybrid attestation approaches have been proposed. The Software-Protected Module (SPM), introduced by Strackx et al. [42], represents an early hardware-supported process isolation mechanism built atop a static root of trust. It achieves software compartmentalization by loading and measuring protected application logic into designated secure memory regions, referred to as “vaults.” While SPM offers strong isolation guarantees, it is primarily designed for high-end platforms equipped with Memory Management Units (MMUs) or Memory Protection Units (MPUs), limiting its feasibility on low-cost embedded systems. To overcome these constraints, Sancus [43] extends the SPM paradigm for low-end microcontrollers, achieving secure module isolation without relying on trusted software stacks. It supports remote attestation and inter-module message authentication via specialized hardware instructions. Similarly, SMART, proposed by El Defrawy et al. [44], introduces a minimalist hardware-software co-design for dynamic root-of-trust establishment on devices lacking MMUs or TPMs. SMART requires modest hardware modifications to the system's microcontroller to achieve attestation and memory integrity verification. TrustLite [45] builds upon this concept by integrating an Execution-Aware Memory Protection Unit (EA-MPU), enabling fine-grained isolation of lightweight software modules, referred to as “trustlets.” The platform also supports secure boot, ensuring the authenticity and confidentiality of trustlet code. More recent research has proposed further refinements of these lightweight architectures to support remote attestation and secure execution in severely resource-constrained embedded environments [46-48].

England et al. [49] demonstrated that even a minimal feature, such as a hardware-locked secret accessible only by boot ROM, can provide sufficient guarantees for boot-time attestation. Building on this insight, the Trusted Computing Group introduced the Device Identifier Composition Engine (DICE) [50], a lightweight attestation framework tailored for resource-constrained IoT devices. Its feasibility has been validated in multiple studies [51, 52].

1.3 Main Contributions

Building on the review of attestation methods in the previous section, this thesis identifies a critical gap in existing research: the absence of a unified, lightweight, and hardware-constrained security architecture capable of establishing trust across the full IoT device lifecycle. Although prior work addresses specific aspects, such as resilient transmission protocols or cryptographic primitives, few approaches integrate these into a cohesive end-to-end solution that begins at the silicon level and extends to cloud-based service infrastructure. This continuity of trust is essential, particularly for IoT environments characterized by limited computational resources, minimal silicon capabilities, and cost-sensitive deployment constraints.

This study proposes an end-to-end attestation architecture that maps robust security mechanisms onto commercially available hardware, enabling trust establishment from device provisioning to remote authentication. A key requirement addressed is the development of a device-specific authentication mechanism, grounded in a dedicated hardware-derived value, that enables secure interaction with remote cloud infrastructures. By combining minimal hardware trust anchors with proven cryptographic methods, the proposed design facilitates the creation of a cryptographically strong device identity and supports the secure delivery and verification of firmware updates.

The primary contributions of this thesis are as follows:

1. **Design of an Asymmetric Attestation Protocol**
   A lightweight attestation protocol tailored for resource-constrained IoT nodes is proposed. It combines secure boot, proven cryptographic primitives, and the Device Identifier Composition Engine (DICE) standard to create an adaptable architecture. At its core, the protocol derives a cryptographic identity from a statistically unique hardware root of trust. It also incorporates an authentication mechanism for device-to-cloud communication via TLS and supports secure, over-the-air firmware updates with integrated data protection.

2. **Security Analysis and Threat Modeling**
   A qualitative security evaluation is conducted based on the STRIDE threat modeling methodology. The analysis identifies potential threats across system components and communication interfaces, characterizing attacker capabilities, risk impact, and attack scalability. Based on this assessment, targeted countermeasures are formulated to address each identified vulnerability.

3. **Prototype Implementation on Embedded Hardware**
   The proposed attestation protocol is implemented on a commercial STM32 Nucleo-74LG board, which features a 32-bit ARM Cortex-M4 microcontroller. This prototype serves as a proof of concept, demonstrating the feasibility of deploying the security architecture on low-power embedded platforms.

4. **Quantitative Evaluation and Performance Metrics**
   The final implementation is evaluated in terms of runtime performance and memory footprint. Metrics such as clock cycle overhead and memory utilization are measured to assess the practicality of deploying the protocol on real-world resource-constrained devices.

1.4 Thesis Outline

The remainder of this thesis is structured as follows:

- **Chapter 2** presents essential background on security in edge-to-edge IoT systems. It introduces core principles and countermeasures against security threats, encompassing cryptographic algorithms, resilient communication protocols, hardware security features, and attestation techniques. Special attention is given to lightweight and silicon-efficient architectures, with a focus on the Device Identifier Composition Engine (DICE) as a representative model for cryptographically agile trust anchors.

- **Chapter 3** introduces the design of a security protocol for asymmetric remote attestation tailored to resource-constrained IoT devices. It begins with a V-Model-inspired design methodology, followed by a structured analysis of the system's functional, non-functional, and security requirements. The chapter details the protocol's layered architecture, including key derivation and operational flows across the device lifecycle. It concludes with a threat modeling assessment based on the STRIDE framework to evaluate the design's resilience against adversarial threats.

- **Chapter 4** describes the implementation and evaluation of a prototype that realizes the proposed protocol on a commercial STM32 microcontroller. Implementation details, design decisions, and performance metrics, such as memory usage and execution overhead, are presented to assess the feasibility of the architecture.

- **Chapter 5** summarizes the key findings and contributions of this work. It also discusses identified limitations and suggests potential directions for future research.

2 Background

This chapter provides foundational background on security principles, techniques, and technologies relevant to embedded systems. Section 2.1 introduces core concepts in embedded systems security, distinguishing between information security, which encompasses confidentiality, integrity, and authentication, and device security, which pertains to the protection of platforms responsible for processing sensitive data. Section 2.2 explores the role of cryptographic techniques in building secure systems, emphasizing the evolution and application of modern encryption schemes. Section 2.3 reviews authentication mechanisms and trust models underpinning Public Key Infrastructure (PKI), while Section 2.4 examines the Transport Layer Security (TLS) protocol as a cornerstone for secure communication between clients and servers. Finally, Section 2.5 presents the Device Identifier Composition Engine (DICE), a lightweight, cryptographically agile attestation framework designed for resource-constrained embedded devices.

2.1 Embedded Systems Security

An embedded system is a computing system designed for implementing dedicated functions within a larger system, electrical or mechanical [53]. It is a combination of computer hardware, software and optionally mechanical parts, referring to any computing system other than general purpose or mainframe computers [54]. Embedded systems run real-time operating systems (RTOS) that are aiming to control device-dedicated applications with real-time computing constraints. The low manufacturing cost of embedded systems makes them highly beneficial in a variety of intelligent and industrial application domains such as automotive electronics, factory automation, smart homes, transportation, commerce and finance, healthcare and many others [55]. Embedded systems are considered either closed or open depending on which level they communicate and share information with one another [56]. Within a closed system, devices interact only with devices that are part of that system, often through protocols and standards designed exclusively for the needs of the closed system. For example, modern cars are equipped with smart sensing and control systems that communicate and exchange information among these systems. However, the communicated data is not shared with other systems or devices outside of the car, for example the car manufacturer. Embedded systems connected with other devices or systems through a network are considered open embedded systems. Open embedded systems exchange data with other devices and systems using open communication standards and protocols for purposes such as information sharing.

The increasing number of embedded devices connected to the Internet formulates the Internet of Things (IoT). There does not exist only one definition as to what comprises the “Internet of Things”. According to [57], the IoT refers to a set of embedded devices or “things” that are embedded with software, sensors, and network and are capable of communicating data with one another. The “things” can use their communication protocols, although some sort of Internet connection may be necessary at some point. The term “Internet” does not necessarily refer to communication via Internet protocols. Apart from connecting embedded devices to the Internet, the IoT allows these devices to collect and exchange data. Data communication provides the IoT endpoint devices with flexibility minimizing the need for physical connectivity and manual intervention. The exchanged data, for example, the raw measurements of a humidity sensor are not of great value and do not produce any useful knowledge when examined individually. However, the manipulation of these data via filtering and contextualizing processes provides in-depth knowledge about the system, its users, the environment, and its objectives. Adding context to the exchanged raw information, patterns can be created containing information about a particular activity of the device. Categorization and processing of these data also

provide information regarding the repetition of certain processes. Finally, the organization of data provides valuable information regarding the relationships formulated between different pieces of information.

The growing deployment of interconnected systems renders necessary the increased protection of computer systems, networks and communication data, against sophisticated attacks from malicious cybercriminals, ensuring the security and reliability of the information systems. Interconnected embedded systems face important security challenges related to their resource constrained nature. Their limited processing capabilities, the battery-driven power, the wireless network connectivity, and the remote control of their software, render these devices vulnerable against attackers. These attacks may compromise important security properties resulting in loss of control of embedded systems. Therefore, the security of embedded systems is considered end-to-end, starting from the physical endpoint devices that receive and transmit data, to hubs and gateways layer that aggregate the edge devices to the large network, to cloud-based systems that store and analyze the data provided by the edge nodes. This section focuses on two important types of embedded systems security: (1) the information security, and (2) the device or platform security.

2.1.1 Information Security

The practice of protecting information and the systems that use, store, or transmit that information is defined as cybersecurity [58]. Cybersecurity is also known as information security and concerns the mitigation of security risks by preventing unauthorized access, use, or modification of information data [59]. According to Pipkin in the “Information security: protecting the global enterprise”, information security is the process of protecting the intellectual property of a company or organization [60]. The primary focus of information security is balancing the security risks by providing information assurances and by defining a set of security goals [61]. These goals are the result of a security analysis called risk management process and include a variety of security attributes and guidelines that span multiple disciplines based on the security model in use. Risk management involves the identification of information assets and potential threats, a risk evaluation of the impact of the identifying threats, a valid mitigation plan to address these risks, and the selection and implementation of appropriate security techniques [62].

One of the first security models in information security is the “Guidelines for the Security Systems and Networks” that was proposed by the Organization for Economic Cooperation and Development (OECD) in 1992. According to this model, there are nine security principles to consider, including awareness, responsibility, response, ethics, democracy, risk management, security design and implementation, security management and reassessment [63]. In 1998, Donn Parker in his MSc thesis with the title “The Parkerian Hexad” proposed the six elements of information [64], while in 2004, the National Institute of Standards and Technology (NIST) proposed 33 fundamental principles in their study “Engineering Principles for Information Technology Security” [65]. Another suggested security model is the information security management maturity standard O-ISM3, published by the Open Group [66]. According to this model, a set of management policies is proposed after the definition of the required security targets. Information security focuses on the balanced protection of the confidentiality, integrity, and availability of data, three of the most important attributes of information [67]. Together they formulate the C.I.A. triangle, a well-known and widely accepted security standard for computer security (Figure 2.1). The C.I.A. triangle is also referred to as the A.I.C triad to avoid confusion with the intelligence agency in the USA.

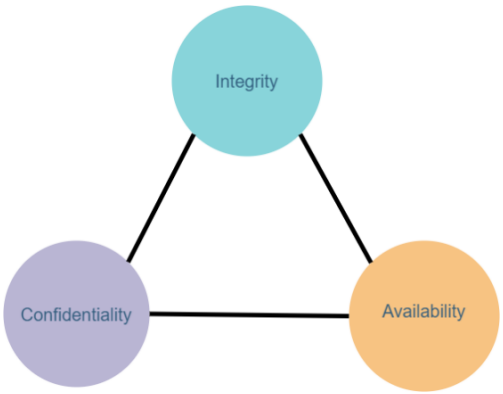

Figure 2.1 The C.I.A triad of Information Security

Confidentiality

Confidentiality is a security principle referring to the concealment of valuable information or resources in sensitive areas such as governance and industries [68]. Government services often apply classification systems that restrict the access to sensitive content to classified personnel [69]. The appearance of such systems is traced back in the mid-nineteenth century when the British Government published the "*Official Secret Art*", a document concerning diplomatic espionage and information disclosure [70]. Similar principles apply to the industry sector where companies protect their intellectual property by preventing competitors from gaining access to design information regarding their products. Confidentiality also refers to the fact that sometimes revealing the information is more important than the information itself. For example, the reveal that a company secretly monitors its employees is more important than the findings of such an action. Another important aspect of confidentiality is hiding critical systems and resources that an entity may not wish to be used without proper authorization. Examples of compromised confidentiality are password theft and email phishing. Protection of sensitive information concerns the definition of strict access policies that arrange the data into categories, according to the type of personnel that has access to it and their sensitivity level. Common confidentiality policies include biometric verification, two-factor authentication and traditional Unix-file access control lists. However, mechanisms that enforce confidentiality in a system should be trusted that they supply the correct verification data.

Integrity

Integrity constitutes a basic security element of information security that refers to the trustworthiness and accuracy of data resources and is responsible for protecting sensitive data against unauthorized alterations [71]. Integrity concerns not only the integrity of the data but the integrity of its source, a method often referred to as authentication. The reliability of the source of the information plays a vital role for a system to gain credibility and trust. For example, the credibility of the source of any news in a newspaper or magazine determines if the news is fake or not. Integrity mechanisms can be classified into two categories, detection mechanisms and prevention mechanisms. Integrity detection mechanisms report whether an information is trusted, or an integrity violation exists. Detection mechanisms use analysis tools to detect possible violations and report any corruption that may occur. Prevention mechanisms deny any unauthorized attempt to alter sensible data, or any attempt to alter data in unauthorized ways. Modifications or deletions from authorized users should be performed with caution in order to avoid unintentional or malicious alterations. For example, a company's authorized employee tries to embezzle money instead of moving it into a trusted account.

Availability

Availability refers to the last component of the triad and concerns the ability of data to be present and available when needed [72]. Systems with high availability aim to remain available at all times and at all costs, preventing failures and disruptions. This requires the design of a statistical model that analyzes the patterns of use and the existence of mechanisms able to act on these patterns. Ensuring availability involves protection against denial-of-service-attacks. In these kinds of attacks, a malicious attacker may deliberately deny access to services or information by making the system unavailable. Denial-of-service attacks have proven to be quite challenging to detect. The analyst should be able to determine if any unusual change in patterns of use is due to normal malfunctions or deliberate manipulation of the system resources by the adversary.

Although the CIA triad provides notable security policies for the information technology, it is debatable whether it can keep up with the latest technological developments and distinguish security from privacy [73]. The CIA triad gives the impression of a holistic security solution that is the answer to most of the security problems. However, it tends to ignore other equally important factors and therefore it should be considered only as a part of a broader security approach.

2.1.2 Device Security

Designing a trustworthy security approach for interconnected embedded devices goes beyond satisfying the basic principles of information security, such as confidentiality, integrity, availability, and non-repudiation, and extends to the device itself. Device security involves a range of security solutions that protect device resources and sensitive data from physical tampering, network attacks, and unauthorized access by malicious adversaries. These solutions vary from software protection techniques targeting remotely accessible devices to physical protection mechanisms that secure platform sensitive information. The rapid proliferation and deployment of embedded devices renders the design of device security solutions quite challenging. This section presents an overview of effective device security techniques and methods in a layered fashion, where each layer corresponds to the protection of a specific security feature. Figure 2.2 illustrates three device security layers: (1) secure storage of key device security components, (2) memory protection and secure boot mechanisms known as platform security, and (3) remote attestation authentication schemes that ensure code integrity when connecting to a remote cloud provider.

Secure Storage

Secure storage is an important device security mechanism that stores confidential data and cryptographic secret keys, namely the Root of Trust (RoT). The RoT is the basis of the device's unique identity, and it is inherently trusted every time the device is booted or reset [74]. At startup, the generation and provisioning of an immutable root secret must be ensured in order to establish reliable device authentication and secure communication. The generated root secret is typically encrypted and stored in non-volatile memory units, such as One Time Programmable (OTP) memory. One of the main benefits of OTP memory units is that they are programmable in a secure environment during device manufacturing. However, the provisioning of the root key during manufacturing lurks the risk of potential secret leakage to third parties. Physical Unclonable Functions (PUFs) is a flexible key provisioning solution aiming to circumvent this limitation. PUFs are electronic design components that exploit the unique silicon properties of individual integrated circuits (ICs) to create a statistically unique identity bound to the device [75]. The trusted generation and provisioning of the root keys ensures that the key material is unique and specific to the device, non-cloneable, non-modifiable by malicious adversaries, and unknown to non-authenticated third parties.

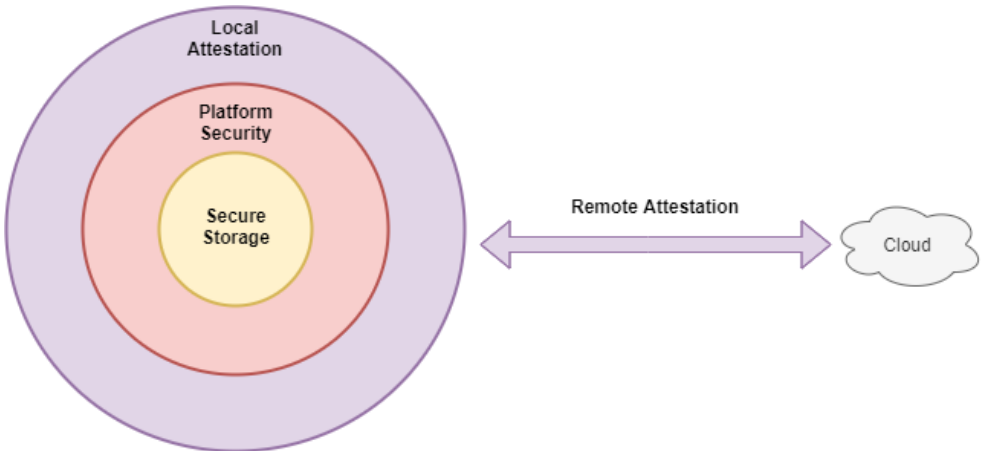

Figure 2.2 Device Security Techniques

Platform Security

The second layer of device security involves the enforcement of access control mechanisms and secure booting processes to enhance trustworthiness within the device. The first prevents unauthorized third parties from accessing sensitive device resources, while the latter ensures the integrity and authenticity of device firmware. It is important that access control mechanisms are implemented at all levels, both user and system. On a user level, authentication is ensured using simple methods, such as PIN entry, or even biometric identifiers, such as fingerprints or facial patterns [76]. On a system level, it is essential to mitigate the risk of unauthorized usage from malicious adversaries by providing protected access to System on Chip (SoC) busses and interfaces. The Memory Protection Unit (MPU) is an embedded system component that prevents access to confidential data stored in memory, by setting access permissions and attributes to specified memory regions [77]. Secure boot is the part of platform security responsible for validating the authenticity and integrity of the code running on the device [78]. A secure boot process involves trusted bootloaders, strong encryption schemes, and secure storage units, which ensure that only authorized firmware is executed when the device powers up. A typical example of code authentication is the storage of firmware in flash memory, executed by a trusted bootloader during boot.

Attestation

The last layer of device security concerns a process that verifies the identity of the internal state of an embedded device, namely attestation [79]. Architectures equipped with attestation capabilities provide adequate security guarantees of the attested software state of the device to a trusted third party, called verifier. Attestation can be classified as local and remote attestation. The first refers to an intra-platform mechanism, where application modules running on the same platform verify and authenticate their respective software images with one another. An example of local attestation is the Intel SGX Report mechanism that provides confidentiality, code integrity, and strong protection within the device [80]. Remote attestation involves an external trusted verifier entity for verifying the integrity of the code running on the device. Attestation is typically performed by calculating a code measurement of the attested software module and by sending it to the trusted third entity during manufacturing. The verifier may request the module code measurement at any time verifying its validity. A number of remote and local attestation solutions have been proposed by academia including software-based schemes, hardware-based protocols, and hybrid techniques. Attestation schemes are useful tools for a variety of services related to embedded devices such as over-the-air (OTA) software updates and patches, memory reset, and malware replacement.

2.2 Cryptography

This section provides the reader with useful background information regarding the terms and concepts behind basic cryptographic methods and techniques. Also, a detailed description of various cryptographic algorithms is given along with examples of their applications in numerous areas of information security. Cryptographic systems provide solutions towards mitigating the security threats and attacks present in embedded systems.

Cryptography can be described as the method of concealing information during communication, in the presence of malicious entities called adversaries [81]. The prefix "*crypt-*" of the term derives from the Greek word "*kryptos*" and means hidden or secret, while the suffix "*-graphy*" originates from the word "*graphein*", which stands for writing or script in Greek [82]. Cryptography is closely associated with encryption, a technique of transforming ordinary information, called plaintext, into an unintelligible form called ciphertext [83]. Another term for encryption, though less common, is encipherment, with decipherment or decryption standing for the opposite procedure. Both encryption and decryption rely upon a cipher, a procedure of well-defined steps known as algorithm and a piece of information, called key. The key is a secret value known only to the parties that communicate, the absence of which makes it impossible to decrypt ciphertext into plaintext and vice-versa (Figure 2.3). The communicating parties share the secret key with one another, a cryptographic method called key exchange or key establishment [84]. Key exchange together with a finite set of plaintexts, cyphertexts, keys, encryption and decryption algorithms constitute a cryptosystem. Cryptosystems can be classified into two kinds, symmetric and asymmetric. Symmetric systems use the same key to encrypt and decrypt the transmitted message. Asymmetric systems, on the other hand, use a public key to encrypt a message and a private secret key to decrypt it. In the crypto field, the common nomenclature for two communicating parties is usually "*Bob*" and "*Alice*", while a malicious third party is referred to as "*Malory*", an eavesdropper as "*Eve*" and a trusted party as "*Trend*".

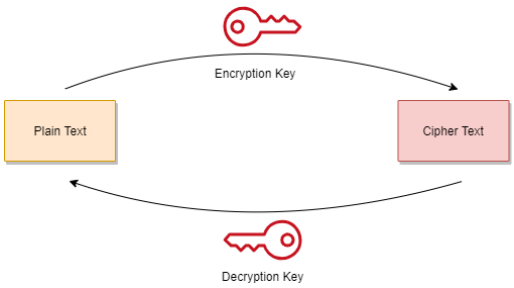

Figure 2.3 Cryptographic Encryption/Decryption

Cryptography is also related to the terms of cryptology and cryptanalysis. Cryptanalysis refers to the science of analyzing a cryptographic scheme aiming to expose possible vulnerabilities. It is performed either by searching for loopholes in the mathematical basis of the cipher or by finding logical flaws in its design [85]. A famous example of cryptanalysis is the breaking of the Enigma machine, the encryption device developed and used for encrypting military operations during the World War II [86]. Cryptology refers to a broader term that encompasses both cryptography and cryptanalysis, including designing of ciphers, key exchange techniques and cryptanalysis tools [87]. Applications of cryptography span multiple disciplines from electronic commerce and wireless payments to military communications and digital crypto currencies. Although cryptography has proven a powerful weapon against malicious adversaries, it is also being used as a tool for espionage, forcing several governments to constrain its use.

The attempt to establish message confidentiality in communications dates back to the ancient world, where political diplomats and military emissaries understood the necessity of a mechanism capable of concealing valuable information from the enemy. Sensitive messages were transported in safe boxes and were protected against tampering by armed envoys. The first proof of the use of cryptography is an encrypted text curved in stone in the Old Kingdom of Egypt at 1900 B.C. [88]. However, the true nature of that message remains a mystery to researchers. Stenography was a well-known method of information concealment, used by the Ancient Greeks and Persians of the fifth century B.C. and involved the use of microdots, invisible ink and merge of words with images [89]. Other classical cryptographic methods are transportation ciphers and substitution ciphers. Transportation ciphers refer to rearrangement of the order of the letters in a message, while substitution ciphers allow the replacement of groups of letters with other letters. The most famous substitution cipher is the Caesar cipher, where each letter in the plain text is shifted three positions further down the alphabet [90]. Most of these ciphers became vulnerable to cryptanalysis after the discovery of frequency analysis by the Arab mathematician Al-Kindi in the 9th century as they were based on the premise that the adversary has no knowledge of the cipher itself [91]. It was until the 19th century when Auguste Kerckhoffs claimed that a cryptographic scheme should remain secret even in the unwanted case of leak to the adversary, formulating the Kerckhoffs's principle. Kerckhoffs's principle was restated by the founder of information theory, Claude Shannon, formulating the fundamental theory of theoretical cryptography, the Shannon's Maxim [92].

By the end of the twentieth century, rapid advancements in telecommunications, electronics and computing systems led to revolution in the communications sector. Computing systems have become smaller, more powerful and cheaper and are able to communicate and exchange information via interconnected networks. Modern Cryptography constitutes a vital aspect of secure communications and is related to main principles of information security such as confidentiality, data integrity, system availability, authentication and non-repudiation. Modern cryptographic systems are based on mathematical algorithms and computer science principles rendering the breaking of such systems computationally exhausting and impractical, as it would require unlimited computing power and resources. The biggest challenge of these algorithms is the growing concerns regarding the processing power of future quantum computing systems and their ability to break many of the current cryptography encryption standards. Quantum computing systems make use of the quantum bits capable of representing both zero and one and performing two calculations at once, hence being computationally more powerful [93]. The cryptographic algorithms of modern cryptography are categorized into three types, based on the number of keys used for encryption and decryption, namely secret key of symmetric cryptography, public-key or asymmetric cryptography and hash functions (Figure 2.4).

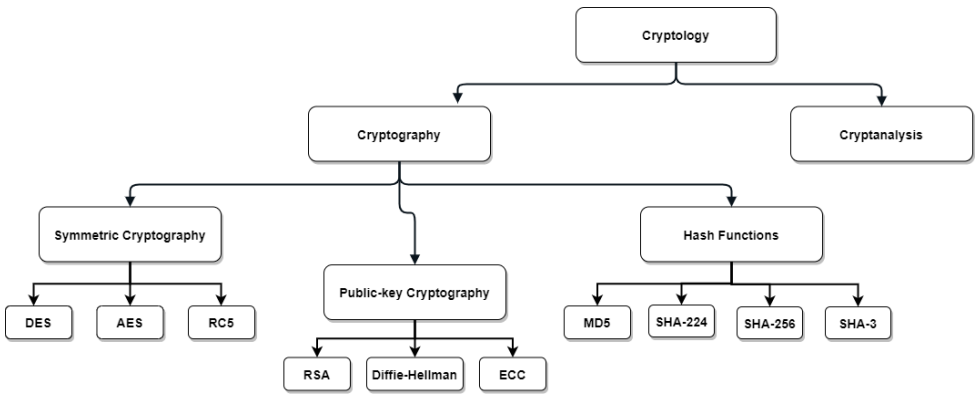

Figure 2.4 Classification of Cryptographic Algorithms

2.2.1 Symmetric Cryptography

Symmetric or secret key cryptography refers to information concealing methods where both communication entities share a single key for both encryption and decryption providing privacy and confidentiality (Figure 2.5). The secure key distribution between the two communicating entities is known as the key distribution problem, a limitation of symmetric cryptography [94]. Cryptographic schemes that follow this type of cryptography are classified into two categories, namely stream ciphers and block ciphers. Stream ciphers encrypt a single bit of plaintext at a time, using a relatively long stream of pseudorandom bits as key material. This pseudorandom generator needs to remain unpredictable so that the derived key gets to change constantly. Although stream ciphers do not propagate transmission errors, their periodic nature causes a repetition of the keystream, a vulnerability known to potential attackers [95]. A well-known application stream cipher is One-Time Pad, an ideal cipher that performs pure random key generation achieving maximum secrecy [96]. In contrast to stream ciphers, block ciphers are encryption schemes using a fixed size block of data for encryption at a time. The size of plaintext block is the same as the ciphertext block and varies from 64 bits to 128 and 256 bits. In cases of shorter size of plaintext bits, padding schemes are used [97]. A well-known block cipher is the Feistel cipher, where the encryption and decryption stages are similar causing a considerable reduction in size of code.

Figure 2.5 Symmetric cryptography

The majority of symmetric algorithms used today for secure communications are designed by cryptographers and mathematicians aiming to render the decryption of ciphertext infeasible without the possession of the appropriate encryption key. One of the most well-known and well-studied symmetric algorithms is the Data Encryption Standard (DES), designed by the National Bureau of Standards (NBS) in 1977 for governmental purposes [98]. The DES algorithm is a complex block cipher with key size of 56 bits and block size of 64 bits, designed for fast hardware implementations. In 1998 the NSA managed to break the DES algorithm using brute-force attack, exposing a mathematical backdoor in its design. This fact motivated the academic community to propose a DES variant called triple-DES that made the DES algorithm more robust [99]. However, the security implications of DES led the NSA to introduce a new cryptosystem called Advanced Encryption Standard (AES). The AES describes a 128-bit block cipher that features a key with 128, 192 and 256 bits and became a standard in December 2001 [100]. Other important symmetric ciphers that are considered secure are ChaCha20 and RC5 [101]. Although the sufficient key size for a symmetric algorithm to be considered as secure is between 112 and 128 bits, advances in quantum cryptography come as a threat to the level of security these algorithms possess.

2.2.2 Public Key Cryptography

Public-key cryptography, also referred to as asymmetric key cryptography, is a revolutionary cryptographic scheme proposed by Martin Hellman and Whitfield Diffie in 1976 aiming to mitigate the shortcomings of symmetric key cryptography regarding key distribution. The basic principle of asymmetric cryptography concerns the generation of a cryptographically related key pair consisting of a freely distributed part called public key and a secret part called private key, where it is computationally infeasible to derive one from the other [102]. While one key is used for plaintext encryption, the other is used for decrypting the ciphertext, regardless of which key is applied first (Figure 2.6). Public-key cryptography also concerns a user authentication mechanism that proves the identity of the message

sender. In essence, one communication entity ("*Alice*") encrypts information using its private key, while the other entity ("*Bob*") is capable of decrypting it, using the freely shared public key of *Alice*. In the described scenario, the sending entity cannot deny the sending of the message satisfying the non-repudiation security principle. Diffie and Hellman proven the feasibility of asymmetric cryptography by introducing the Diffie-Hellman key exchange protocol, where the two communicating parties should first agree upon a shared encryption key [103].

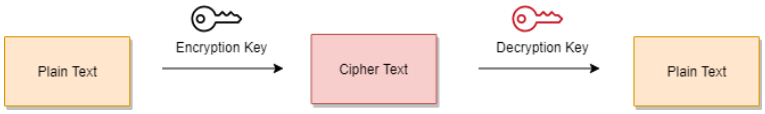

Figure 2.6 Public Key cryptography

The first successful attempt of an asymmetric scheme is the RSA algorithm that took its name from the initials of its creators, namely Ronald Rivest, Adi Shamir and Leonard Adleman, three notable MIT mathematicians [104]. RSA spans a wide range of uses from key exchange and digital signatures to encryption of small data blocks. The derived key pair is the product of two prime numbers with length of 100 or more bits each, yielding in a very large number n. Although it is difficult for an attacker to determine the prime factors of n, the advancements of modern computing systems to factorıze large numbers, are becoming a potential threat to the algorithm. In 2005, a test aiming to factor a 200-digit number was performed and took almost 50 years in computation time, while in 2009 Kleinjung claimed that factoring a 1024-bit RSA number would require almost one thousand years. Apart from the RSA, Diffie and Hellman proposed their own asymmetric cryptography suitable for key exchange. Another approach to public key cryptography is the Elliptic-curve Cryptography (ECC) which is based upon elliptic curves over finite fields. ECC uses smaller keys in comparison with RSA is suitable for resource constrained devices such as endpoint IoT devices [105]. ECC algorithms are suitable for key agreement and digital signatures and may be used in combination with symmetric encryption schemes. An example of such a scheme is the Elliptic-curve Diffie-Hellman (ECDH) key agreement protocol proposed by Diffie and Hellman [106]. Finally, the Digital Signature Algorithm (DSA) refers to digital signature capabilities with X.509 standard to be the most used format of this algorithm [107].

An important element of public key cryptography is a mathematical scheme for validating the authenticity of digital documents and messages, called digital signature. Digital signatures provide strong authentication and message integrity capabilities, enhancing the trust between two communication entities. Digital signatures are part of cryptographic methods and techniques applied in various applications such as online financial transactions, internet banking and software distribution [108]. A user authentication example using digital signatures is illustrated in Figure 2.7, where two communication parties validate one another. The sender entity "*Bob*" sends a message to the verifier entity "*Alice*", together with a digital signature that proves the authenticity of his identity. "*Bob*" feeds the message data to a signature algorithm and signs it with his private key, called encryption key. The produced digital signature is appended to the message and then both are sent to the receiver entity. The message receiver "*Alice*" verifies the authenticity of "*Bob*" by decrypting the received signature with the freely distributed public key of "*Bob*". This key is called verification key and is adamant towards the signature decryption. In case the resulting message value is different, "*Alice*" may correctly assume that the received message has been forged by a malicious adversary. On the contrary, if the signature gets verified by "*Alice*", that means that non-repudiation is provided as only "*Bob*" could be the sender of the message.

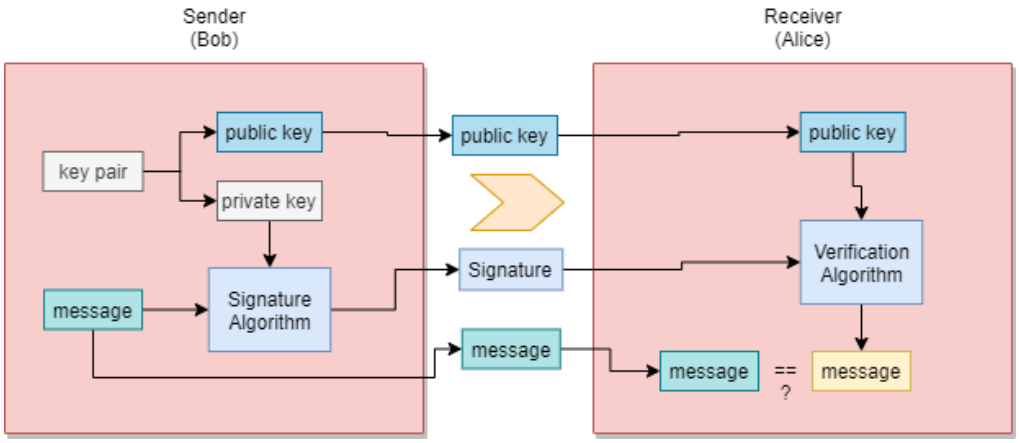

Figure 2.7 User authentication scheme

2.2.3 Hash Functions and Message Authentication Codes

Hash functions constitute the third type of modern cryptographic algorithms known also as one-way functions and message digests. Hash functions are capable of encrypting plaintext data irreversibly without the use of encryption keys by computing a fixed-length hash value of the plaintext. Any change in the message contexts will effectively result in the calculation of a completely different hash value, ensuring data integrity [109]. Cryptographic hash functions add a layer of security by verifying the authenticity of received data from an unknown source. Hash functions span a wide range of applications from digital fingerprints on sensitive file data to deployments in operating systems for password encryption purposes. A well-studied family of hash algorithms is the Message Digest (MD) algorithms that produce a 128-bit hash value regardless of the message length. MD4 is a well-known example of the MD family developed by Rivest for fast processing in software that is now broken [110]. After the appearance of vulnerabilities in the MD4 algorithm, an improved cryptographic scheme was introduced by Hans Dobbertin in 1996, under the name MD5 in his work "*Cryptanalysis of MD5 Compress*" [111]. The US national Security Agency (NSA) developed another important hash function family, the Secure Hash Algorithm standard (SHA) [112]. SHA-1 is the first deployed algorithm of the SHA family that produces a 160-bit hash value for data encryption. SHA-2 family was proposed by the NSA as an improvement to the SHA-1 family after reports of attacks against it. SHA-2 comprises of five algorithms namely SHA-1 plus, SHA-224, SHA-256, SHA-384 and SHA-512. Each one of the aforementioned algorithms is able to produce hash values of length 224, 256,384 and 512 respectively. However, vulnerabilities in the SHA-2 family led NSA to introduce a third family that is called SHA-3 in 2012 [113].

A similar integrity measurement scheme to the hash functions is the Message Authentication Codes (MACs). A MAC refers to a symmetric cryptographic technique capable of providing message authentication. Similar to hash, a MAC function encrypts an arbitrary long input of data into a fixed output using a key. A MAC consists of three algorithms, a key generator algorithm responsible for selection of a suitable key, a signing algorithm that returns a tag of the selected key and the message and a verifying algorithm that accepts or rejects the message [114]. In the example of Figure 2.8, the sending entity "*Bob*" creates a tag of the message that he wants to communicate with the receiving party "*Alice*", using a key and a signing mac algorithm. The message together with the generated tag are send to "*Alice*", who tries to generate her tag of the received message using the same key. The data integrity of the message is verified by an algorithm which compares the two tags and validates any possible tampering or alteration of the message. A special type of message authentication code is the hash-based MAC called HMAC. The HMAC uses a cryptographic hash function and a secret key for verifying the integrity of the data and authenticating the message.

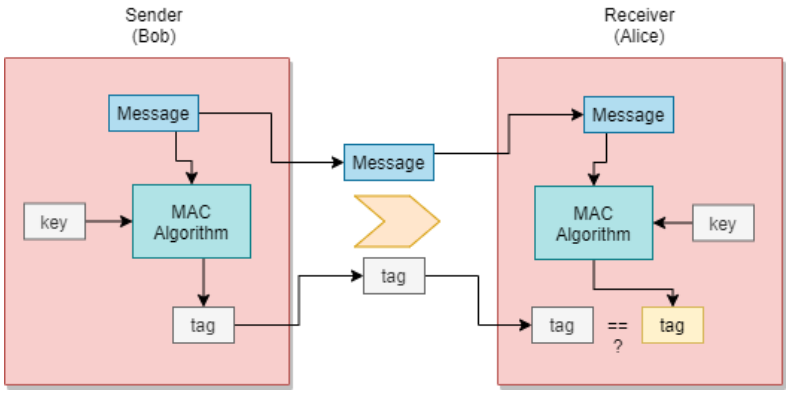

Figure 2.8 Message Authentication Code (MAC) usage

2.3 Public Key Infrastructure

Today an increasing number of companies are using the Internet as a platform to conduct their business transactions, including retail sales, marketing, and business-to-business operations. Although e-commerce has become an important tool for modern companies, it poses significant security and integrity issues. Online transactions differ from the traditional face-to-face business model and therefore require the development of robust security mechanisms. Symmetric cryptography is an ideal solution for data encryption, ensuring the confidentiality and privacy of the transaction data. Public key cryptography offers user authentication and non-repudiation, allowing companies and customers to validate their respective identities. Finally, hash functions and MACs provide data integrity, enhancing confidence between customers and enterprises. However, proper implementation of security requires more than sound cryptography. It involves the establishment of trust among the communication entities. Public key infrastructure (PKI) is a framework based on public key cryptography, that provides trustworthy digital communications over a network [115]. In cryptography, the PKI refers to the binding of public keys with digital identities of entities such as people or companies. PKI is an essential component of the overall system security strategy and a foundation element of network security.

2.3.1 PKI Components

This section demonstrates the key components of PKI facilitating trusted communications with confidentiality, integrity and non-repudiation among the communication parties. PKI involves public key certificates, certificate authorities, registration authorities, registration and insurance certificate policies, a centralized certificate management system, certificate chains, and hardware security modules.

Digital Certificates

In the dynamic environment of e-commerce, online transactions are performed among digital representations of physical entities such as people or organizations. These digital entities need to prove the authenticity of their identity while communicating with one another over a network. In PKI, digital identities are represented by digital documents called public key certificates or identity certificates. Public key certificates certify the binding of a digital identity to its public key, allowing another entity to validate the identity of the certificate owner. Digital certificates secure confidential information with encrypting methods and sign public keys with digital signatures, ensuring tamper protection. Typically, a certificate contains the public key of an entity, identifying information regarding the certificate owner, the name of the certificate issuer and the expiration date of the certificate. The certificate owner is called subject entity and usually differs from the issuer. However, there are certificates called self-issued certificates, where the two fields are identical. Additional information included in a digital certificate are

the types of cryptographic algorithms used, policies that describe how the certificate may be used, a serial number of the certificate, and a digital signature of the issuing entity of the certificate. Perhaps the most common used format for digital certificates is described by the IETF X.509 standard in the RFC 2459 document [116]. However, there is no single definition of digital certificates which allow vendors and users to generate their own version of digital certificates. Public key certificates are widely used to authenticate HTTPS-based websites, where web browsers validate the authenticity of a HTTPS web server, ensuring that there are no eavesdroppers during the client-server communication. However, digital certificates are vulnerable against treats and attacks from malicious adversaries. For example, a web browser may not notify a user in case of changes in the certificate provider or expiration date.

Certificate Authorities

The foundation element of PKI is a trusted third party called certificate authority (CA), responsible for issuing, signing, and storing digital certificates [117]. Examples of certificate authorities are a company that issues certificates to its employees or an internet service issuing certificates to its users. An end-user requesting a digital certificate from a certificate authority creates a certificate signing request (CSR), which contains the public key of the end-user and useful identity information such as the name of the certificate owner. The CSR is digitally signed with the private key of the certificate owner and is sent to a certificate authority called registration authority (RA). Registration authorities are responsible for accepting or rejecting requests for digital certificates and authenticating the subject entities of these requests. The authentication process involves the decryption of the end-user signature, incorporated in the CSR, using the public key of the end-user. Subsequently, an RA performs due diligence tests on the end-entity, examining the subject name, validity date and other important identity information. Although RAs constitute an important PKI element, they are not entitled to sign or issue digital certificates. After validating the end-user authenticity, a certificate authority signs the certificate request with its secret signing key, issuing a digital certificate. The issued certificates should be unique for constantly valid maintaining the trustworthiness provided by the CA. The validation mechanism of issued certificates is performed by a PKI entity called validation authority (VA). Common validation techniques involve the revocation of compromised or lost keys and the protection of public and private keys. Revocation information regarding invalid certificates is provided via the Online Certificate Status Protocol (OCSP) and Certificate Revocation Lists (CRLs). Prominent certificate authorities currently available in the market include DigiCert and Sectigo [118]. Common problems with certificate authorities involve purchases of cheap, low quality certificates from end-entities, dropping the quality of the certificate chain. Another CA vulnerability is unexpected changes to the built-in list of root-certificates provided to all web browsers by certified organizations. Such changes might be performed by inexperienced or even malicious developers resulting in erroneous certificates.

Chain of Trust

Certificate authorities enhance the validation capabilities of their authentication mechanism by establishing trust relationships with other CAs. A well-known method to achieve that is the use of certificate chain or chain of trust. Chain of trust refers to a list of certificates that authenticate one another in a hierarchical manner [119]. Starting from an end-entity, certificates are issued and signed by certificates higher in the certificate hierarchy, using the secret key corresponding to the next certificate in the chain. The authentication process involves the verification of the signature of the target certificate using the corresponding public key of the next certificate, until the last certificate in the chain is reached. The last certificate in the chain is a self-signed certificate which is inherently trusted by users. This certificate is called root certificate and is the trust anchor of the chain of trust. In case the

root certificate gets compromised by a malicious adversary, then all certificates issued by the root CA will be affected, resulting in re-issuing of new certificates, the validity of which will be questionable. Therefore, it is important for root CAs to use intermediate authorities to sign and issue end-entity certificates. A useful schema representing a chain of trust between three certificates is illustrated in Figure 2.9.

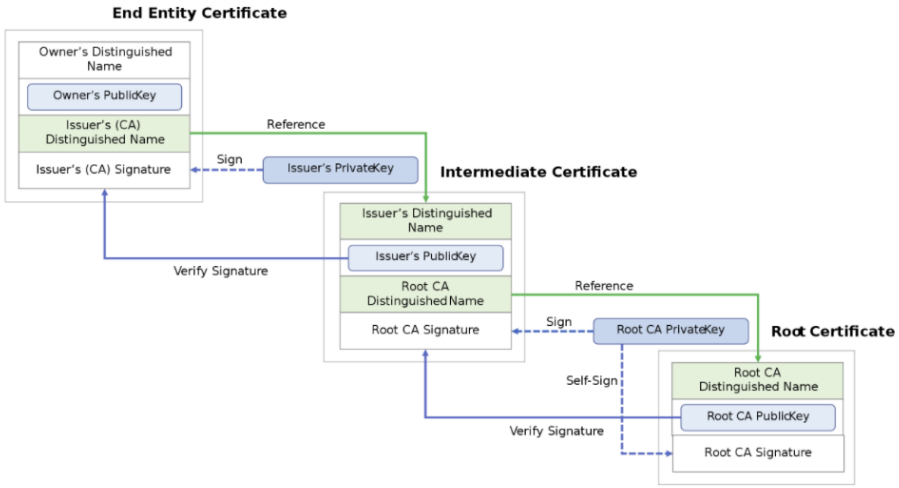

Figure 2.9 Certificate chain of Trust

Hardware Security Module

The trustworthiness of a Certificate Authority (CA) critically depends on the secure generation, storage, and usage of its private keys, operations typically performed within a **Hardware Security Module (HSM)**. An HSM is a tamper-resistant cryptographic processor that generates, stores, and executes cryptographic functions, such as digital signing and key management, entirely within its secure hardware perimeter, ensuring that private keys never leave the protected environment [120]. These devices are often certified under standards like FIPS 140-2/3 and are widely used in environments requiring high assurance, including internet banking, Public Key Infrastructure (PKI) applications, and cryptocurrency systems.

2.3.2 Trust Models

The public key infrastructure involves security policies and methods equipped with authentication capabilities, in order to establish trust within a system. The trust relationships developed in the PKI framework are a result of a collection of rules known as trust models. In this section an overview of three well-known examples of trust models is presented.

Hierarchical Model

The most common implementation of the PKI is the hierarchical trust model. It refers to a tree model consisting of a number of certificate authorities and end-entities, arranged in a hierarchical manner [121]. The root node of the tree is called root CA, while the inner nodes are referred to as intermediate or subordinate CAs and the leaves as end-entities. The root CA is the PKI trust anchor and usually issues a self-signed certificate containing the public key used to verify certificates issued by the root CA. in the hierarchical model, trust is established via a certification path. This path consists of a sequence of certificates that authenticate certificates next in the hierarchy based on information provided from certificates higher in the tree. An example user authentication is described in Figure 2.10.

In this PKI framework, "*Alice*", "*Bob*" and "*Carl*" are certified end-entities from different CAs. The end-entities "*Alice*" and "*Bob*" trust each other as they share a common issuing CA. Although "*Alice*" and "*Carl*" have different issuing certificate entities, they share a common root CA. In order for "*Alice*" to establish trust relationship with "*Carl*", she needs to verify the certification path of "*Carl*". The certification path consists of a self-signed certificate of the root CA, a signed certificate of the intermediate CA2 by the root CA, and the end-entity certificate of "*Carl*" signed by CA2. Since "*Alice*" inherently trusts the root CA, she subsequently trusts the intermediate CA2 and the end-entity "*Carl*".

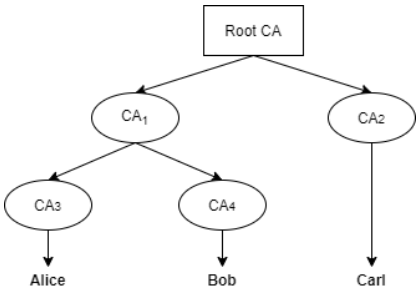

Figure 2.10 Hierarchical trust model

A tiered hierarchy with multiple CAs offers a high level of control mitigating the risk of trust violation. However, the number of intermediate CAs increases the administrative effort required to maintain the hierarchy and thus increases the risk. This can be very practical in the Internet environment where multiple users are connected to a web server. If the trustworthiness of the server gets compromised, all users are notified at once and refused access. However, this can be very impractical in e-commerce transactions where communication is on a one-on-one basis. On the other hand, a flat hierarchy using a single CA as trust anchor is more flexible and requires less administrative effort. The drawback of this hierarchy type is corruption of the entire certificate chain, in case of a failure in the root CA.

Peer-to-peer Model

The most basic PKI trust model is the direct or peer-to-peer model. In this model, there is no intermediate trusted third party and end-entities establish trust with one another on a direct manner [122]. Each end-entity relies on a local CA to issue its certificate, creating a local trust domain. In the local domain, the end-entity can verify the validity of the local CA signature, using the public key of that CA. The direct connection between the end-entity and the local CA ensures the secure provisional of the CA's public key to the end-entity. An example of a direct communication within a trust domain is the installation of software updates form a Linux server located on the Internet to a local machine. The server public key in usually provisioned either in the Linux distribution or in a CD or DVD, ensuring protection against theft from a malicious adversary. This model offers flexibility, making the extension of trust domain quite convenient. However, the increasing number of local CAs makes the manageability of the system quite challenging. Therefore, the peer-to-peer trust model is not an ideal candidate for e-commerce or web browsing. A variation of the peer-to-peer trust model is the cross-certification model. This model allows the CAs of two different trust domains to authenticate the public key to one another, creating a bidirectional trust.

Web of Trust

In cryptography, the term web of trust refers to decentralized systems where end-entities authenticate one another without using trusted third parties. Contrary to the hierarchical PKI model, no third party ensures the integrity and authenticity of an end-entity. End-entities are allowed to specify the validity of their certificate by indicating the number of trusted signatures placed on that certificate. The web of

trust works better with small groups of end-entities who have preexisting trust relationships. However, it does not scale well with big numbers of end nodes being unsuitable for e-commerce transactions [123]. The web of trust is applicable to a variety of well-known systems such as the PGP, GnuPG and other OpenPGP-related systems. Pretty Good Privacy (PGP) is a private e-mail scheme based on public key methods. A communication entity maintains a list of all trusted public keys of the entities that the entity trusts, formulating a web of trustworthy users. For example, the user "*Bob*" may obtain the public key of user "*Alice*" via an e-mail or a server containing the public key of "*Alice*". "*Bob*" is not aware of the validity of the stored key and therefore assumes that it is valid. Trust among users is established on a one-to-one basis. For example, the fact that user "*Carol*" claims that he has a copy of "*Alice's*" key in his database, does not necessarily mean that "*Bob*" should trust that key, even though he trusts "*Alice*". An important characteristic of this model is that trust establishment is not transitional and there is no chain of trust among the users of the web.

2.4 Transport layer Security

Secure communication is essential for internet-based applications to ensure data integrity, confidentiality, and mutual authentication between endpoints. The **Transport Layer Security (TLS)** protocol is the predominant solution at the transport layer, designed to prevent eavesdropping, tampering, and unauthorized data access [124]. TLS facilitates the establishment of an encrypted channel between a client and a server, enabling secure data exchange after connection negotiation. Originally developed by Netscape as the **Secure Sockets Layer (SSL)** in the mid-1990s, version 1.0 was never publicly released due to significant vulnerabilities, prompting the release of SSL 2.0 in 1995 and a complete redesign in SSL 3.0 by 1996. The Internet Engineering Task Force (IETF) later standardized the protocol under the TLS name, starting with TLS 1.0 in 1999 and evolving through versions 1.1 and 1.2.

The most recent iteration, TLS 1.3, was published as RFC 8446 in August 2018, introducing several key improvements: streamlined handshake with one round-trip, removal of obsolete and insecure cipher suites, mandatory forward secrecy, and enhanced privacy through encrypted handshake metadata [125]. Today, TLS is widely deployed across protocols such as HTTPS, SMTP, FTPS, VPNs, and VoIP, forming a cornerstone of. Clients and servers may support multiple TLS versions, negotiating the highest mutually supported version during handshake. While TLS 1.3 is now the default in modern browsers and servers, earlier versions remain in use for compatibility, though many deprecated versions carry known security risks.

The **TLS protocol** comprises two core components: (1) the **Handshake Protocol**, responsible for negotiating cryptographic parameters, mutual authentication, and key material exchange; and (2) the **Record Protocol**, which secures application data using keys and algorithms established during the handshake. The handshake proceeds through several defined steps to establish a secure session between client and server [126].

1. The client initiates the handshake with a **ClientHello** message, specifying supported cipher suites (for key exchange, authentication, encryption, and integrity), TLS versions, session identifiers, compression methods, and a 32-bit random nonce used in subsequent key derivation.
2. The server responds with a **ServerHello** message, selecting the protocol version, cipher suite, and compression method from the client's list and providing its own random nonce.
3. The negotiation concludes with the server sending a **ServerHelloDone** message, indicating readiness to proceed with the key establishment and authentication exchange.

Handshake Continuation and Key Exchange

Upon selecting the cipher suite, the server proves its identity to the client using the agreed authentication mechanism. Typically, this involves the server sending an **X.509 certificate**; client authentication is optional and initiated only if required by the server. When requested, the client similarly provides its certificate. Next, the handshake enters the **key exchange phase**, where both parties collaboratively derive the **"master secret key"**, which subsequently generates session encryption keys. The process typically proceeds as follows:

1. The client generates a **"pre-master secret"** using the negotiated key exchange algorithm.
2. This pre-master secret is encrypted with the server's public key (from its certificate) and sent to the server.
3. The server decrypts the message using its private key.
4. Both client and server then derive the **master secret key** from the pre-master secret and the nonces (random values) exchanged during the initial ClientHello and ServerHello steps.
5. Session keys are derived from the master secret; the client notifies readiness via the **ChangeCipherSpec** message and then sends a **"ClientFinished"** message.
6. The server echoes with its own **ChangeCipherSpec** and **"ServerFinished"** messages, confirming handshake completion.

Upon receipt of "ServerFinished", both client and server begin encrypted and authenticated data exchange. This protocol flow is summarized in Figure 2.11.

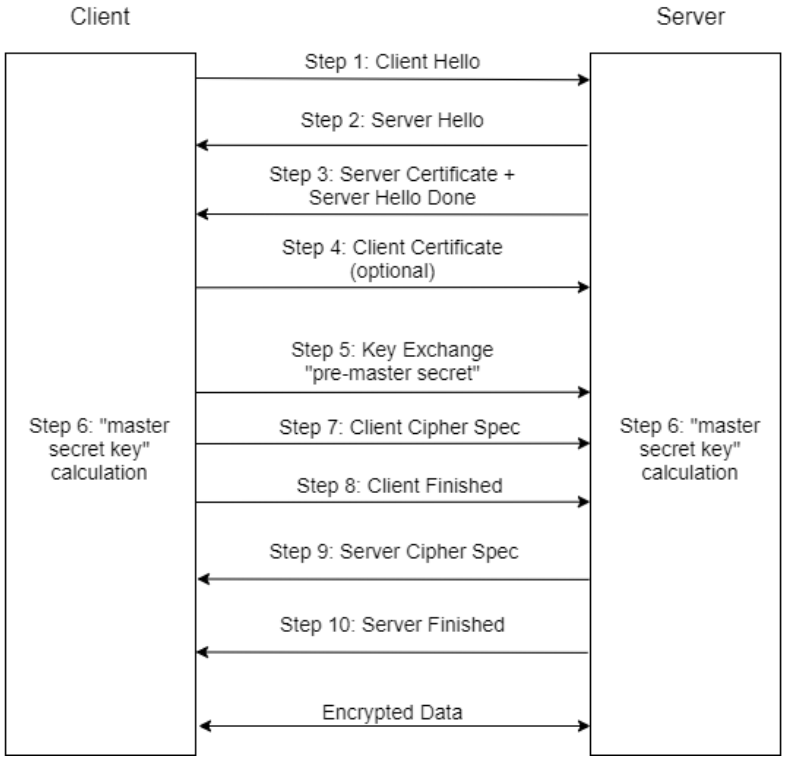

Figure 2.11 TLS Handshake protocol

TLS Record Protocol

The TLS **Record Protocol** ensures secure transmission of application data by providing confidentiality, integrity, and authenticity, all based on the cryptographic parameters negotiated during the handshake [127]. Each message is divided into discrete units known as *records*, which undergo the following processing steps:

1. **Fragmentation**: Messages are segmented into appropriately sized records.
2. **Compression** (optional): Data is compressed if negotiated.

3. **MAC Application**: A Message Authentication Code (MAC) is computed and appended to each record to ensure integrity.
4. **Encryption**: The record (including the MAC) is encrypted using the session key.
5. **Transmission**: The encrypted records are transmitted over the secure channel.

On the receiving end, the entity reverses these steps: decrypting each record, verifying the MAC to confirm integrity, decompressing if needed, and reassembling the data for delivery to higher-layer protocols. The Record Protocol operates in conjunction with upper-layer mechanisms such as ChangeCipherSpec and alert protocols to manage security state transitions and error handling.

TLS Vulnerabilities and Known Attacks

Although TLS is the de facto standard for transport-layer security, multiple vulnerabilities remain in older versions and implementations. Notably, the **POODLE** ("Padding Oracle On Downgraded Legacy Encryption") attack, disclosed in October 2014, exploits fallback mechanisms to compel clients and servers to negotiate SSL 3.0 connections. Because SSL 3.0 does not validate CBC-mode padding correctly, attackers can perform decryption via a man-in-the-middle approach [128]. Another significant vulnerability is **Heartbleed** (CVE-2014-0160), which emerged in April 2014 in OpenSSL's heartbeat extension. By sending malformed heartbeat requests with inflated payload lengths, attackers could trick servers into leaking up to 64 KB of sensitive memory, potentially exposing private keys, passwords, and confidential session data [129]. These cases underscore the persistent risk borne by deprecated protocol versions and flawed implementations, reinforcing the importance of rigorous version checks, secure library updates, and deprecation of obsolete features.

2.5 Device Identifier Composition Engine

The Device identifier Composition Engine (**DICE**) is a security standard developed by the Trusted Computing Group (TCG) within the DICE Architectures Work Group, in order to address the need for increased security in the IoT [130]. The TCG spans a wide range of security standards and resilient technologies aiming to provide critical security and privacy capabilities to the embedded world. The most notable among these technologies is the Trusted Platform Module (TPM) that provides a hardware root of trust for secure boot. However, the TPM is a heavyweight solution for systems with constraints related to cost, size and energy such as the end-to-end IoT devices [131]. Therefore, DICE is proposed as a lightweight and robust security solution that establishes trust within deployed IoT systems. The simple silicon requirements of DICE [132] together with software techniques form a foundation for important security capabilities such as attestation, strong device identity, verified firmware updates and secure device recovery. The ability of DICE to integrate into existing hardware infrastructure and being compatible with existing security standards makes it adaptable to the majority of systems and components at almost zero cost. A major benefit of DICE is that it is based on sound security principles, developed and tested by industry experts, such as hash functions and integrity measurements.

The key concept behind DICE is that with only a unique device-specific secret and a one-way cryptographic function rooted in hardware, a device is capable of verifying its state. Firmware, divided into layers, is running on top of the DICE engine and creates keys for multiple purposes enhancing the foundational trust services within the device. Those trust services include the formulation of a cryptographically strong device identity based on a robust hardware root-of-trust. The device is able to authenticate its identity and the identity of its software when connecting to cloud provider services, an operation known as attestation. Attestation ensures that only authenticated code is running on the device attesting this way to the trustworthiness of the device itself. Furthermore, DICE protects data by preventing access to old software versions. Other trust services offered by DICE include secure data

storage (sealing), data integrity and safe deployment and verification of software updates, a frequent source of malware attacks.

An important feature that distinguishes DICE from other software-only solutions is its approach towards the device boot process. In a DICE architecture, the boot process breaks up into layers that communicate with one another by passing secret values that are unique not only to the device itself, but to each layer respectively. The first layer of the booting process is immutable, meaning that is not subjected to alterations. This layer contains a unique, device-dedicated initial secret value that is the root of trust for the identity of the targeted system. The immutable DICE code is stored in an NV-memory and cannot be altered. For this purpose, an access preventing mechanism is preferably implemented. The generation of secrets is performed by one-way functions starting from the initial secret value. The secret of the previous layer together with configuration data or code measurement from the next layer in the hierarchy, are cryptographically mixed in a way that is infeasible to derive one secret from another. The code measurement of a layer refers typically to a cryptographic hash of the code or data of the layer. The derived secret is provided to the next layer in the boot chain after all remnants of its creation are erased. Secret confidentiality is of paramount importance and thus secrets should remain strictly confidential within their respective layer. The secret derivation process continues during startup, resulting in a measurement chain that is rooted in the device's identity and is based on measured code. The boot model of DICE is illustrated in the Figure 2.12.

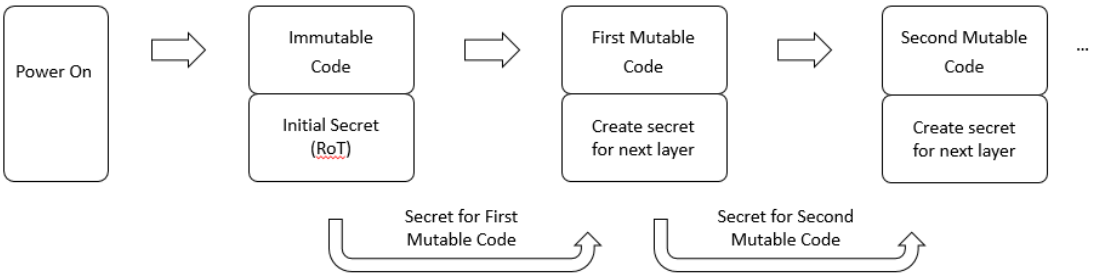

Figure 2.12 DICE boot model

The secrets that are used for establishing device identity or data sealing may leak if the code used for their manipulation gets compromised by an adversary attack. The process of secure re-keying such compromised devices may be a difficult task. ARM's TrustZone establishes a Trusted Execution Environment (TEE) where only trusted applications run [133]. The secrets are stored in one-time programmable memories with limited access from run-time software. This way the risk for secrets to get compromised is considerably reduced. However, most TEEs contain thousands of lines of code retaining the high risk of compromise.

Resilient cyber-attacking systems create malformed programs aiming to steal the device identity. Therefore, all data structures processed in early boot should be simple in order to minimize the chances of exploitable bugs. File systems should not be used in early boot code due to their complexity. Also, vendors should deploy technologies to guard against hardware attacks such as glitching. DICE standard tackles these problems by providing techniques that minimize the amount of code that gets access to the device secret. The uniqueness of device secrets and keys in each layer of the DICE architecture ensures that if new layer code or configuration is modified, the derived secret changes for the corresponding layer. Changing of a secret in one-layer results in changing of all the following secrets in the chain change as well. If a vulnerability exists and a secret is disclosed, patching the code automatically creates a new secret, effectively re-keying the device. In other words, when malware is present, the device is automatically re-keyed, and secrets are protected. An example of layer code change is illustrated in Figure 2.13.

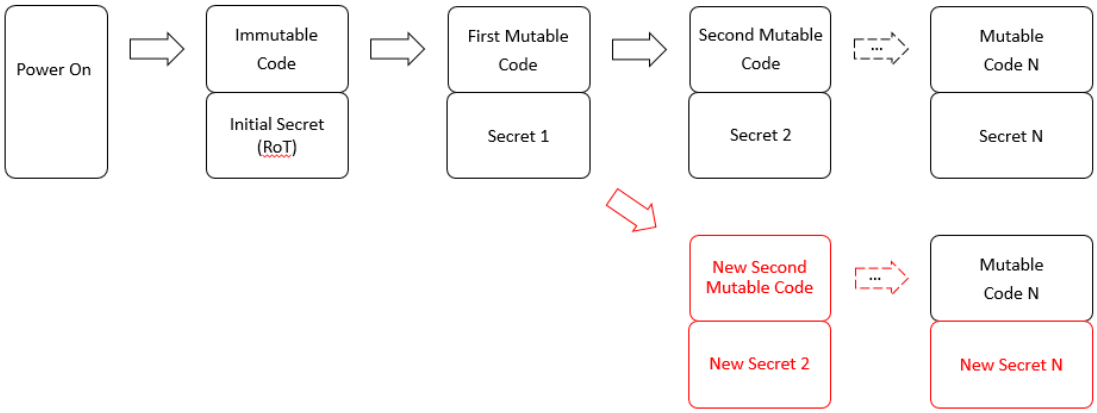

Figure 2.13 Layer code change

The DICE engine is the first code that is executed unconditionally by the device after it is powered or reset. The DICE boot code for simple devices is read-only and preferably one time programmable during manufacture. An updatable implementation is possible for more complex systems and requires a vendor-certified secure update mechanism. Ideally DICE is implemented by silicon vendors in ROM firmware. In DICE specification the device secret is an uncorrelated and statistically unique identity value called Unique Device Secret (UDS). It is generated intrinsically in a trusted environment every time the device powers on or resets. The access to UDS should be limited by a hardware access preventing mechanism incorporated in DICE hardware. This mechanism should disable read-access privileges to UDS before transferring control to the firmware. A system reset should ensure that malware that arrives later in boot or at runtime cannot have access to the UDS. Also, storing the UDS in a NV-memory or Read-only memory (ROM) further strengthens confidentiality of the device secret, ensuring that the UDS is only available to DICE at boot time.

Although such a mechanism can protect access to the UDS, boot code can still be compromised if an adversary makes a copy of it in RAM. Using a one-way cryptographic function to transform the UDS, mitigates the problem to a later state. One-way function or message digest is a function designed in such way that even a slight change in the input string should cause the hash value to change drastically. Even if one bit is flipped in the input string, at least half of the bits in the hash value will flip as a result. In this way, if the derived cryptographic identity value gets compromised, the original UDS is still secure. In addition to the lockout mechanism, DICE needs to ensure that no data is left in the registers or cache memory that might assist an attacker extracting the UDS.

The cryptographic identity value derived from the UDS is called Compound Device Identifier (CDI) and is provided to the early boot code of the device. The CDI is a combination of the UDS and a cryptographic representation of the early boot code running on the device, called First Mutable code. It can optionally include hardware state measurements and configuration data that affect the execution of the First Mutable code. A simple one-way function for the derivation of CDI is a secure hash algorithm used to hash the concatenation of the two values. Another approach is the use of an HMAC for higher level of protection.

$$CDI = H\big(UDS \parallel H(First\ Mutable\ code)\big)$$

The general process of the DICE Engine is illustrated in the Figure 2.14.

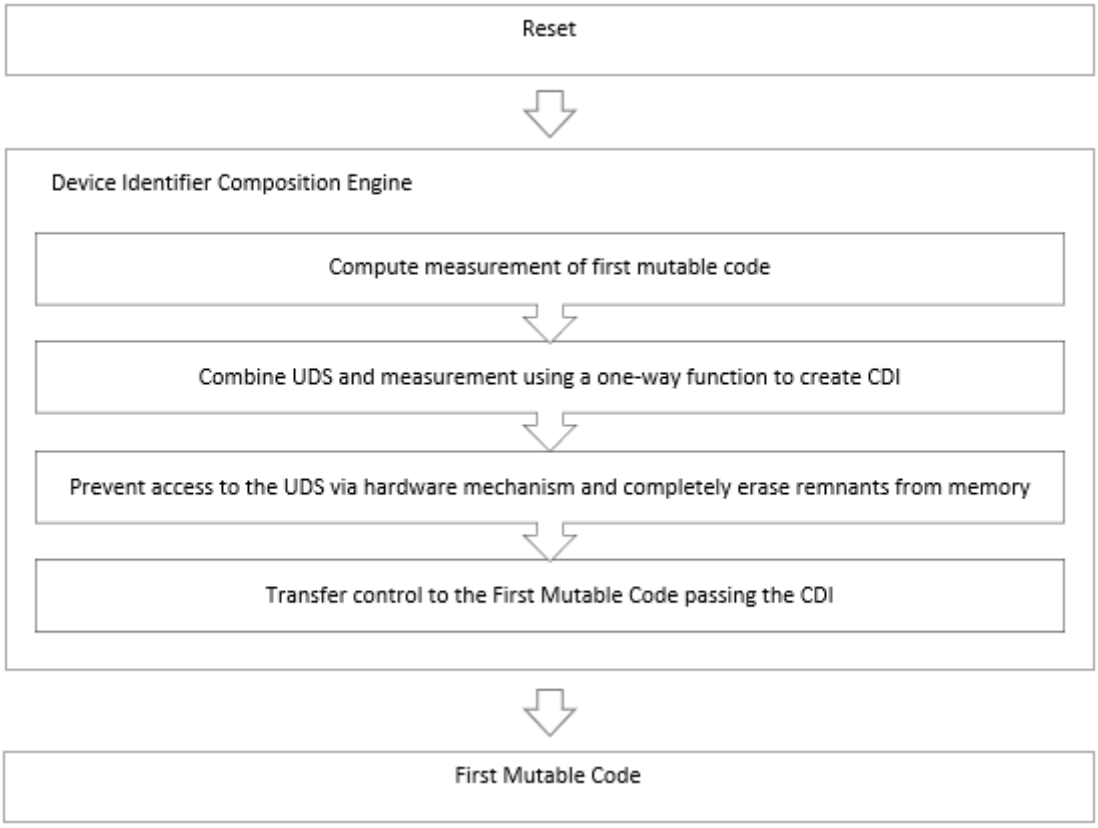

Figure 2.14 DICE Engine

Based on the CDI, keys in the form of secrets are generated for attestation and other purposes. These keys are bound to both UDS and the firmware. A change in either of them would result in a different CDI effectively re-keying the device. For example, if the First Mutable code is replaced by malware (Figure 2.16), the attacking program gets a different hash and thus obtains a different CDI key than the authorized program. In case the application accidentally discloses the CDI, the device must be re-keyed. Flashing a patched firmware will result in creating a new CDI value restoring trust within the device. The CDI value is provided to the First Mutable Code and can be stored in memory or registers. The First Mutable code should be able to read the CDI and then use it to create keys for attestation and other purposes. However, disclosure of the CDI to a latter executing code may compromise the First Mutable code. Therefore, the CDI value should be hidden or deleted maintaining it confidential within the First Mutable code.

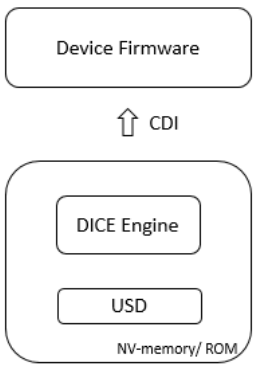

Figure 2.15 DICE Engine layer and firmware

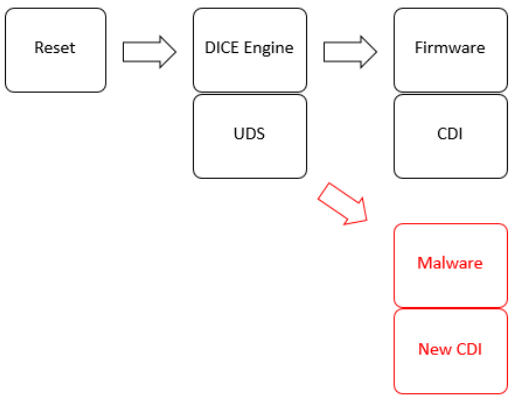

Figure 2.16 Malware attack scenario

3 System Architecture and Protocol Design

This chapter presents the design of a security protocol that enables asymmetric remote attestation for resource-constrained IoT devices. The proposed architecture builds upon the Device Identifier Composition Engine (DICE) standard, which provides cryptographically verifiable device identity and firmware integrity using a hardware-based trust anchor. Section 3.1 introduces the design methodology, motivating the choice of a V-Model to organize development and validation activities. Section 3.2 defines the system's functional, non-functional, and security requirements, establishing the objectives and constraints guiding the protocol's development. Section 3.3 describes the protocol architecture in detail, including its layered components, key derivation mechanisms, and operational flows across the device lifecycle. Finally, Section 3.3.3 provides a structured security analysis of the protocol using the STRIDE threat modeling framework, evaluating how the design mitigates common attack vectors across the defined trust boundaries. Together, these sections form the basis for the implementation and evaluation presented in the following chapter.

3.1 Design Approach

System design in software engineering is the disciplined process of shaping an IT artifact from a set of interconnected components [134]. Offerman et al. classify such artifacts as systems, methods, algorithms, requirements and metrics [135]. Aligned with the design-science research (DSR) methodology of Peffers et al. [136], this thesis adopts a **Systems Development Life Cycle** (**SDLC**) approach [137], which structures development into sequential stages that transforms an abstract problem into a verified and maintainable implementation.

Among various SDLC models, the **waterfall model**, originally proposed by Royce [138] and later adopted by the U.S. Department of Defense [139], divides development into six cascading phases (Figure 3.1). Each phase begins only after the deliverables of its predecessor phase have been completed and reviewed, providing clarity of scope, explicit milestones, and predictable cost control. Although the waterfall approach suits tightly scoped projects with stable requirements, security-critical systems demand an even stronger emphasis on early and systematic verification and validation.

The **V-Model**, an extension of the waterfall paradigm, integrates verification and validation activities in every design phase [140]. Its left branch contains development activities (requirements analysis, architecture design, module design, and here security requirement analysis), while the right branch mirrors them with corresponding test activities (unit, integration, and system validation). The vertical axis denotes the abstraction level, whereas the horizontal axis denotes project time. Verification confirms that artefacts comply with specifications, whereas validation ensures that the final system satisfies stakeholder needs [141], [142]. Widely used instantiations include the German V-Modell XT used by the federal government [143], the U.S. Department of Transportation standard [144], and the general testing V-Model adopted in commercial software engineering.

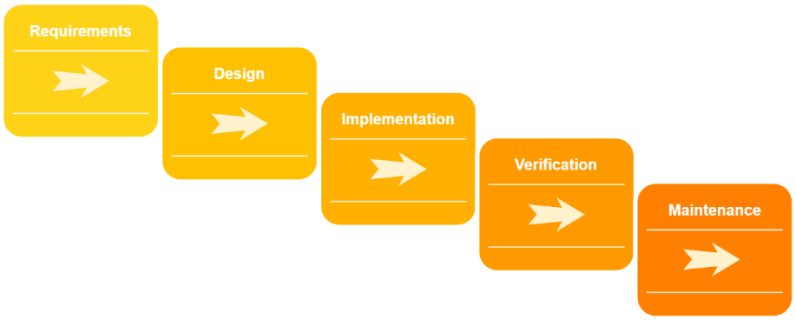

Figure 3.1 Waterfall Model for Sequential Software Development

Design approach

This study adopts a general testing V-Model approach (Figure 3.2) because it

1. offers straightforward deliverables at each stage
2. associates clear validation criteria with every design decision
3. embeds review processes that facilitate early fault detection, essential for IoT hardware where rework is costly
4. remains lightweight for projects with clearly defined goals

Its main limitations are i) reduced flexibility with late-changing requirements and ii) suitability for small-to-medium-scale projects. However, these limitations are acceptable within the context of a security mechanism targeted at constrained IoT devices. The selected V-Model offers a disciplined yet pragmatic framework that aligns with the iterative nature of security engineering: every design decision is paired with an explicit verification activity, ensuring that the resulting architecture meets both functional and non-functional.

In the proposed model (Figure 3.2):

- **Requirement-analysis** defines the system's functional and non-functional requirements as well as the security goals that drive architecture design.
- **Architecture design** produces a high-level overview of the protocol, identifies candidate cryptographic primitives, and assesses feasibility for resource-constrained platforms (high-level design).
- **Module design** refines the architecture into discrete components with well-defined interfaces, enabling isolated implementation and unit testing (low-level design).
- **Security requirements analysis** conducts a list of potential threats and attacks after careful investigation of the system's main components and interfaces. Several attributes that span multiple disciplines are examined for the accessibility risk and the potential reward to the attacker. These attributes vary from the cost of the attacking equipment to the scalability of the performed attack. A counter-measurement list is conducted to protect the system against malicious attackers.
- **Code implementation** realizes all system components on actual hardware.
- **Integration and implementation testing** verify end-to-end functionality and feasibility of the proposed security architecture. is performed during the prototype implementation (demonstrator) to
- Performance validation measures execution time (clock cycles) and memory footprint (code size) to validate that the prototype satisfies the original set constraints.

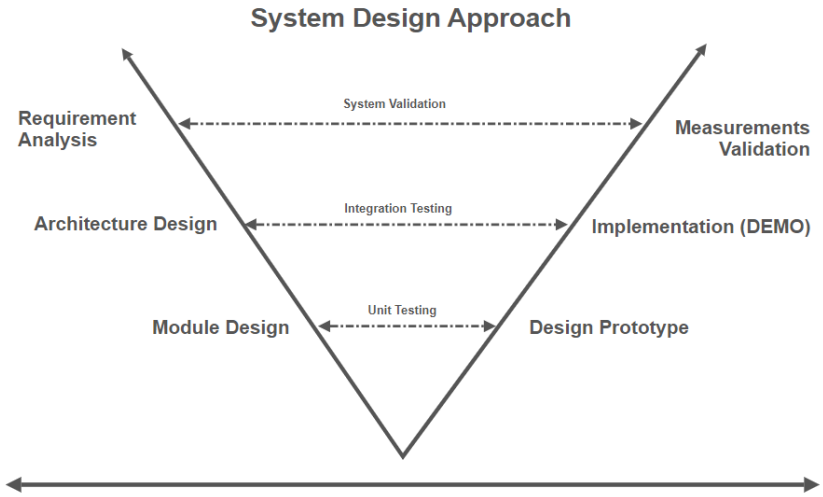

Figure 3.2 General Testing V-Model for Security-Critical IoT Systems

Table 3.1 maps the design phases adopted in this work to the main contributions (MC) introduced in Section 1.3:

Table 3.1 Mapping of Design Phases to Main Contributions

Design phase	Main contribution
Requirement analysis	MC1 – Asymmetric attestation protocol design
Architecture design	MC1 – Asymmetric attestation protocol design
Module design	MC1 – Asymmetric attestation protocol design
Security requirements analysis	MC3 – Qualitative security analysis
Prototype implementation	MC3 – Demonstration prototype
Prototype evaluation	MC4 – Qualitative performance analysis

3.2 Design Objectives and Requirements

This section establishes the security objectives and system constraints that shape the protocol architecture presented in this section 3.3.1. The target deployment scenario involves resource-constrained IoT devices, characterized by limitations in size, power, and computational capability. While these constraints reduce design flexibility, they pose the compelling challenge to deliver robust security guarantees on minimal hardware.

Section 3.2.1 outlines the high-level design objectives that the protocol must fulfil, including secure identification, authentication, remote attestation, and software update capabilities. Section 3.2.2 analyzes the different types of requirements involved in achieving these objectives, distinguishing between functional, non-functional, and security requirements, while defining assumptions that limit the design.

3.2.1 Design Objectives

The overarching goal of the design is to embed a hardware-rooted trust mechanism into commercial IoT nodes. Achieving this objective requires equipping constrained IoT devices with robust security capabilities based on cutting-edge cryptographic principles and protocols. To this end, the architecture is designed to:

- Generate a device-unique asymmetric key pair at boot to establish a unique digital identity.
- Ensure firmware integrity and confidentiality by preventing the execution of unauthorized or outdated software.
- Support mutual authentication with cloud infrastructure through a remote attestation mechanism.
- Offer a lightweight solution with minimal silicon footprint, enabling secure deployment and verification of software updates.

3.2.2 Requirement Analysis

Once the design goals have been established, a requirement analysis is conducted to identify the technical solutions the system must satisfy to meet its objectives.

General Requirements and Classifications

According to Brennan [145], a *requirement* is a premise necessary for a design to achieve its objective or satisfy a functional need. Requirements must conform to formally defined standards or specifications, and play a central role in the verification process, serving as reference points for evaluating the correctness and completeness of system testing. In both system and software engineering, requirements are typically formulated prior to implementation and ensure that the final product is robust, maintainable, and free of critical flaws. Early-stage requirement analysis contributes directly to eliminating exploitable bugs and aligning developer efforts with end-user expectations and constraints. As Hay [146] notes, requirements define the foundational functions and properties needed by a system, and must be documented in a clear, consistent, and unambiguous manner. However, this process can be lengthy and complex, especially in large-scale designs where the number of system requirements can grow rapidly [147].

The IEEE Standard Glossary of Software Engineering Terminology [148] classifies system requirements into two main categories:

1. **Functional Requirements**. Define *what* a system should do, describing the relationship between inputs and expected outputs [149].

2. **Non-Functional Requirements**. Define *how* a system should behave, imposing quality constraints such as performance, reliability, flexibility, and security [150].

Functional requirements are typically considered mandatory (“*must-do”*), whereas non-functional requirements, though equally important, are often classified as conditional or qualitative constraints (“*shall-do”*) [151].

These categories form the foundation for specifying the system- and security-level requirements in the remainder of this section. They directly inform the design of the proposed attestation protocol, where both functional and non-functional properties are critical.

System Requirements

The proposed architecture relies on a set of well-defined requirements that are modest and applicable to the majority of targeted IoT platforms. Together with a small set of basic assumptions, these requirements define the core principles that guide the system’s design and implementation. In the context of software engineering, an *assumption* is typically defined as “a thing accepted as true or certain to happen, without proof” [152] or “a fact or statement taken for granted” [153]. While assumptions are often necessary for practical reasons, they can introduce potential risks. If an assumption proves false in a given deployment, it may lead to software faults, misconfigurations, or even security vulnerabilities. For this reason, the number of assumptions made in this design are intentionally kept to a minimum. Fewer assumptions increase the breadth of verification and testing scenarios that can be applied to the system and reduce the likelihood of errors.

The proposed design is based on a set of six concrete requirements (R1–R6) defined to balance security strength with practical deployment constraints (Table 3.2). These requirements are deliberately minimal and portable to commercially available IoT platforms. Together with a set of limited assumptions, they serve as the foundation for protocol design.

Table 3.2 System Requirements

ID	Requirement
R1	The device must have enough power to derive asymmetric key-pairs and generate digital signatures.
R2	The private key must remain confidential. The public key may be freely disseminated.
R3	The device must be able to establish a connection and communication with a network infrastructure.
R4	The device should use the Transport Layer Security (TLS) to communicate with a cloud server provider.
R5	The device shall use client certificates for mutual authentication (including certificate chaining).
R6	The design should require minimal silicon overhead.

Security Requirements

Within this study, requirements are further examined from a security perspective, where protection against misuse, and unauthorized access is paramount. Security requirements can originate from multiple points in the design process, and they represent the explicit security goals of the application. To be effective, security requirements must be clear, measurable, consistent, and formally verified. A particularly useful methodology is architecture risk analysis [154], which helps identify vulnerabilities that may be exploitable in deployed software.

The Open Security Architecture (OSA) framework [155] classifies IT security requirements into four types:

- **Secure Functional Requirements**. State *what shall not happen* and are embedded within traditional functional requirements.
- **Functional Security Requirements**. Define the specific system behavior that enforces security. Examples include authentication mechanisms, access control, and data integrity enforcement.
- **Non-Functional Security Requirements**. Describe the quality attributes the system must meet to remain secure (e.g. encryption strength, fault tolerance).
- **Secure Development Requirements**. Ensure that development practices and tooling reduce vulnerability exposure.

Capability Mapping

This thesis defines **four core security capabilities** summarized in Table 3.3:

Table 3.3 Security Capability Mapping and Corresponding Requirements

Capability	Security Objectives	Functional Requirements	Non-Functional Requirements	Linked Requirement(s)
Device Identity	Cryptographically verifiable, device-unique identity	On-device key pair derivation	Confidential private key storage	R1, R2, R6

Capability	Security Objectives	Functional Requirements	Non-Functional Requirements	Linked Requirement(s)
Device Authentication	Mutual verification between device and server	TLS with client certificates	Resistance to impersonation	R4, R5
Remote Attestation	Firmware integrity and freshness	Certificate-based reporting	Tamper/replay protection	R1, R2, R4, R5
Secure Software Update	Authenticated and authorized firmware updates	Update enforcement	Rollback protection, minimal silicon overhead	R1, R3, R6

Each of these capabilities is supported by a subset of the system requirements and collectively they fulfill the architecture's security objectives. Some of the previously defined requirements are strict, while others offer more flexibility depending on the deployment context. For example, a modest increase in silicon area may significantly improve performance or strengthen security guarantees—highlighting the need for careful trade-offs. As a result, the design must strike a balance between competing objectives such as resource constraints, security, and operational efficiency.

With the system objectives and constraints now defined, the next section introduces a DICE-based attestation protocol. It outlines the protocol's structure, involved entities, and adversarial assumptions, showing how the identified requirements are translated into a layered boot and attestation framework suitable for resource-constrained IoT devices.

3.3 Protocol Design

This section presents a DICE-based protocol tailored to the constraints and operational requirements of resource-constrained IoT devices. The protocol is designed to achieve four foundational security capabilities: **device identity**, **device authentication**, **remote attestation**, and **secure software updates**. These capabilities are essential for establishing trust in distributed IoT ecosystems, particularly in scenarios where devices are physically exposed, and secure interaction with cloud infrastructures is critical.

The design builds upon two key industry specifications: the **Hardware Requirements for a Device Identifier Composition Engine (DICE)**, developed by the *Trusted Computing Group (TCG)* [130], and the **Device Identity with DICE and RIoT** architecture, proposed by *Microsoft Corporation* [49]. DICE introduces a hardware-rooted mechanism for deriving cryptographically strong identities that are tightly bound to both the device's hardware and the integrity of its software state. This facilitates not only unique device identification, but also effective tamper detection.

Compared to traditional hardware security modules such as the Trusted Platform Module (TPM), DICE offers a light-weight and more scalable alternative. TPMs, while powerful, often demand significant silicon area and power resources, making them impractical for constrained environments such as embedded sensors, microcontroller-based devices, or low-cost IoT platforms. In contrast, DICE is explicitly designed to operate with minimal hardware footprints, relying on immutable boot code and simple cryptographic operations that fit well within the capabilities of modern microcontrollers.

Building on this foundation, the proposed protocol incorporates **asymmetric cryptography**, **public key infrastructure (PKI) digital certificates**, and a **lightweight attestation flow** compatible with **TLS-based cloud infrastructures**. By integrating these elements, the protocol ensures that devices can:

- Establish cryptographically verifiable identity.
- Authenticate securely to external services.
- Attest their current software configuration.
- Securely receive, verify, and install firmware updates.

These objectives are achieved through a layered protocol flow described in the sections that follow.

Scope

The protocol is defined within a clearly bounded scope to ensure its feasibility for constrained platforms, while still offering enough generality for practical deployment, verification, and extension. These boundaries are shaped by a modest set of assumptions, which reduce complexity without compromising the protocol's security or its relevance to real-world use cases.

Assumptions

- **DICE Support**: Devices implement the DICE specification, including the generation of a Compound Device Identifier (CDI) during the boot process.
- **Computational Capabilities**: Devices are capable of generating asymmetric key pairs, signing operations, and basic certificate handling.
- **Network Capability**: Devices are equipped with network interfaces that allow them to establish secure connections to remote servers. This is a prerequisite for remote attestation and software update delivery.
- **Manufacturer-Cloud Separation**: The cloud service provider does not require prior knowledge of the device internals. Any critical provisioning, such as key enrollment or metadata registration, is assumed to occur during manufacturing in a secure, trusted environment.
- **Hardware Capabilities**: The proposed architecture can be implemented by a variety of IoT nodes that feature a micro-processor with basic hardware-based security capabilities.

3.3.1 System Architecture

DICE-based architecture provides four foundational security capabilities—device identity, authentication, remote attestation, and secure software updates—by segmenting the device's boot process into discrete, layered stages (Figure 3.3 a). Each layer generates a unique secret using a one-way key derivation function (KDF), combining the prior layer's secret with either configuration data or a cryptographic hash of the subsequent layer. These secrets are cryptographically isolated: deriving one from another is computationally infeasible. Each secret is securely erased after use, ensuring confidentiality throughout the boot sequence and forming a cryptographic measurement chain rooted in device identity.

As illustrated in Figure 3.3 (b), architecture separates the immutable DICE Engine from the layered mutable firmware. The DICE Engine executes immediately upon power-on and resides in ROM or equivalent non-writable memory to enforce immutability. It derives a Compound Device Identifier (CDI) from a unique device secret (UDS) and a measurement of the first mutable firmware layer. The UDS serves as the hardware root of trust while the CDI becomes the first cryptographic anchor handed off to subsequent firmware layers.

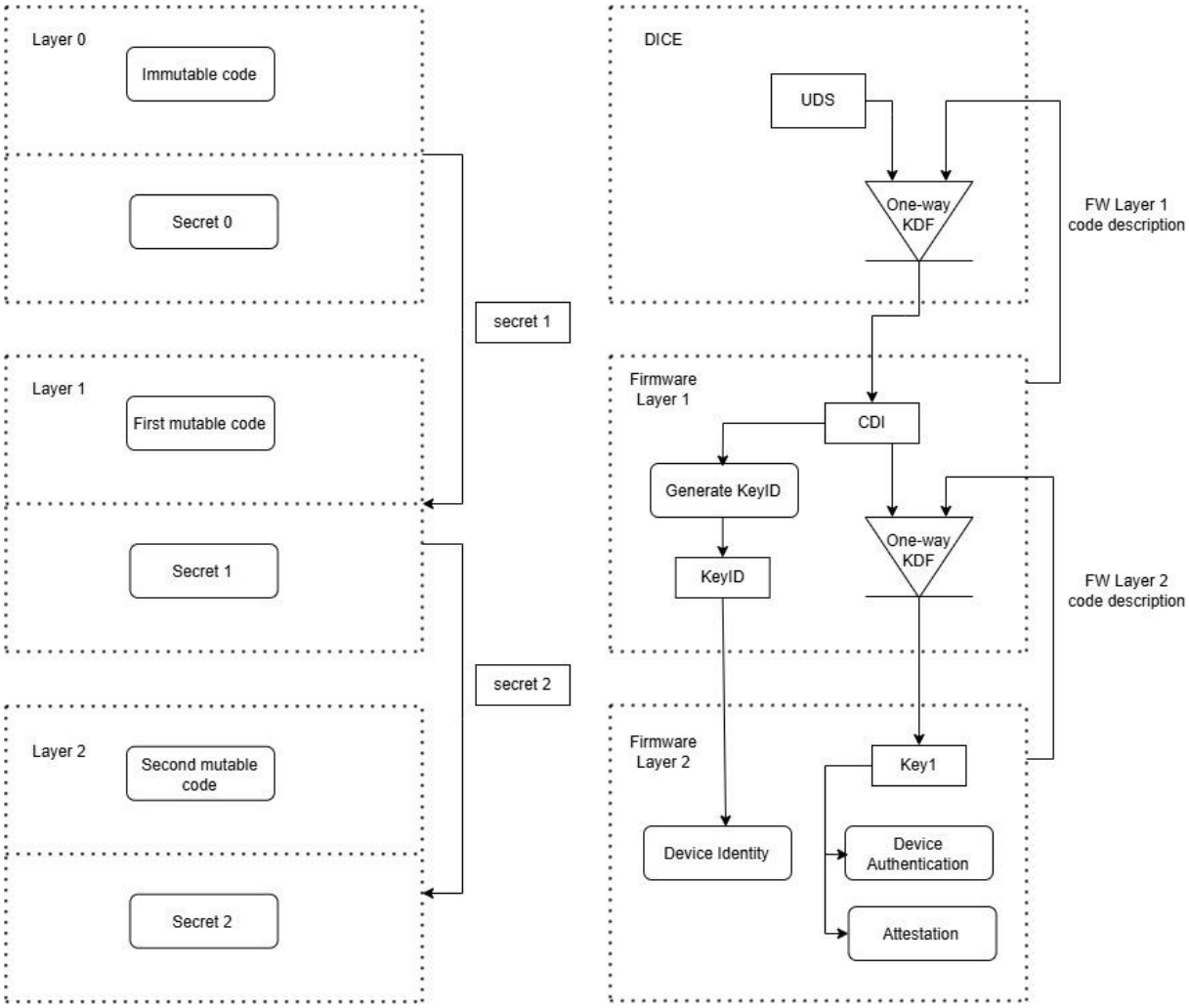

Figure 3.3 Layered Boot Process in DICE-Based Architecture
(a) Secret derivation and isolation across layers (b) DICE Engine and firmware layer segmentation

The layered firmware uses CDI to derive keys for identification, authentication, and attestation. This design ensures that device identity persists through firmware updates. The first firmware layer derives a persistent device identity key (Key_{ID}) directly from the CDI and a second key (Key_{Layer1}) using the CDI and a descriptor of the next layer. These are passed to the second firmware layer, which connects to a cloud (remote) server, authenticates the device, verifies the firmware integrity, and enables secure software updates.

The system architecture consists of three major components:

1. **DICE Engine (First Major Component)**
2. **Firmware Layer 1 (Second Major Component)**
3. **Firmware Layer 2 (Third Major Component)**

Each major component comprises smaller, well-defined system units. The next sections describe these components in detail.

DICE Engine (First Major Component)

The DICE Engine is the root of the trust chain and adheres to the "**Hardware Requirements for a Device Identifier Composition Engine**" specification [130]. It is the first code executed after a system

reset, operating from ROM or equivalent non-volatile memory to ensure immutability. Its role is to derive a unique CDI using a one-way function from the UDS and a deterministic cryptographic digest (e.g., SHA-256) of the firmware layer 1 code.

The UDS serves as a hardware-bound identity anchor, while the firmware measurement captures the initial execution state. Any modification to either input yields a new CDI, thus invalidating any previous trust anchors. For example, firmware modification by malware will trigger CDI regeneration, invalidating derived keys. While valid patches restore trust, compromise of the UDS results in a permanent loss of device identity.

To ensure trustworthiness, the DICE Engine must meet these requirements:

- **One-time Programmable UDS**: At least 256 bits, statistically unique, and securely stored in non-volatile memory or ROM.
- **Exclusive Access**: The UDS must be readable only by DICE; no external access is permitted.
- **Debug Interface Lockdown**: Debug ports must be disabled or configured to prevent UDS access during CDI generation.
- **Immutable Boot Code**: Must execute from read-only or one-time programmable memory.
- **Memory**: All sensitive data, including UDS and CDI remnants, must be cleared from memory and registers post-execution.

Although newer drafts describe “updatable” DICE models, this design assumes a strictly immutable DICE implementation. The UDS may be generated intrinsically during boot or securely provisioned during manufacturing (see Section 3.3.2 for provisioning options). Figure 3.4 illustrates the internal structure of this component.

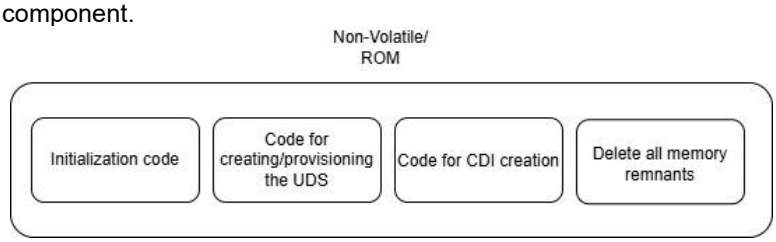

Figure 3.4 Internal Components of the DICE Engine

This architecture adopts a two-layer firmware model for simplicity, though it can be extended. This approach is consistent with principles from the “**RIoT: A Foundation for Trust in the Internet of Things**” specification [49].

Firmware Layer 1 (Second Major Component)

Firmware $Layer_1$ continues the chain of trust as the first mutable code executed after the DICE Engine. Its design and function are shown in Figure 3.5. This layer must remain small, simple, and auditable to reduce attack surfaces. Updates to this layer trigger regeneration of the CDI and all derived keys. As such, it should remain static and updated only under secure, manufacturer-controlled conditions.

Using the CDI, Firmware $Layer_1$ deterministically derives an asymmetric key pair (Key_{ID}) during manufacture in a secure environment. Device key is used to achieve device identity. The public portion is certified by the manufacturer in a PKI certificate and linked to a trusted root authority. The private key remains securely contained within this layer and is never exposed externally. Due to limited computing resources in IoT devices, ECC-based key derivation is favored over RSA for its smaller keys and computational efficiency. Chapter 4 details the selection of ECC curves for this implementation.

This layer also derives a second asymmetric key pair, Key_{Layer1}, from the CDI and a measurement or descriptor of the next firmware layer. Key_{Layer1} is signed by the private portion of Key_{ID}, producing a $Layer_1$ certificate. The private key and certificate are securely passed to Firmware $Layer_2$, supporting attestation without compromising the device identity.

To prevent misuse, the firmware descriptor for $Layer_2$—including version, size, and cryptographic hash—is embedded in the Layer 1 certificate as a Firmware Identity (FWID). This value may be encrypted using authenticated symmetric encryption for confidentiality. External verifiers can then validate firmware integrity without exposing sensitive implementation details. Table 3.4 illustrates a possible structure for the layer and device certificates Including the FWID extension).

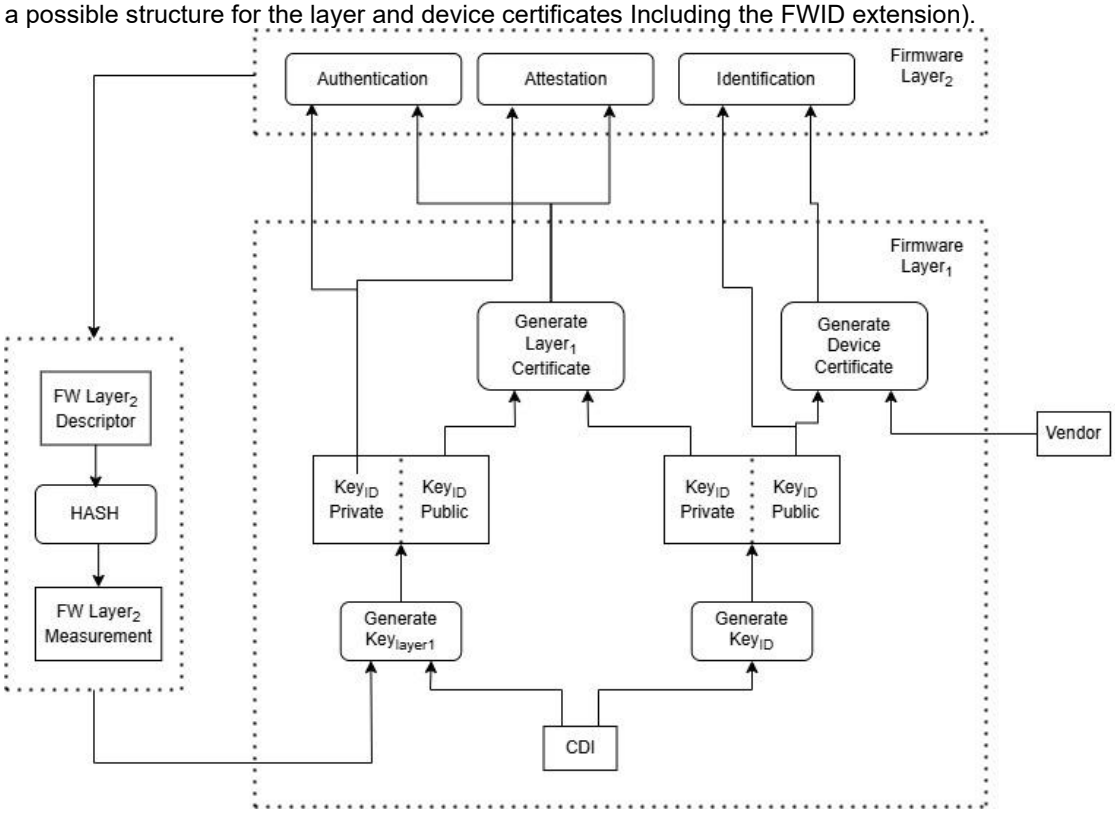

Figure 3.5 Structure of Firmware Layer 1

Table 3.4 Structure of Device and Layer Certificates with FWID Extension

Field	Description
Version	Certificate format
Serial Number	Unique identifier from the issuing Certificate Authority
Signature Algorithm	Algorithm used for signing (e.g., ECDSA)
Issuer	Distinguished Name (DN) of the Certificate Authority
Validity Period	Timestamps defining certificate lifetime
Subject	DN of the certificate subject (e.g., device or vendor)
Subject Public Key Info	Public key and algorithm (e.g., ECC)
Extensions	FWID and optional fields such as usage constraints
Signature	Digital signature from the issuer over the certificate contents

Firmware Layer 2 (Third Major Component)

Firmware $Layer_2$ is the third major component that acts as the application layer, interfacing with external systems. After taking control from $Layer_1$, it is responsible for:

1. Authenticating the device

2. Attesting firmware integrity
3. Establishing secure communication with a cloud service

Network connectivity is assumed as a design requirement. Depending on the platform, secure communication can be implemented using TLS (for TCP-based transport), DTLS (for UDP-based environments), or OSCORE (for application-layer object security in CoAP-based systems). For constrained environments, DTLS with raw public keys or CoAP secured by OSCORE is preferred. Protocol selection is explored in Section 3.3.2 and implemented in Chapter 4.
Firmware Layer 2 performs authentication and attestation using:

1. The device certificate (public portion of $\mathbf{Key_{ID}}$).
2. The KeyLayer1 key pair.
3. The Layer1 certificate (signed by $\mathbf{Key_{ID}}$).

Once connected to a cloud server, the device presents a certificate chain from $Layer_1$ to the trusted manufacturer root. The $Layer_1$ certificate contains the FWID of $Layer_2$, which the cloud server uses to verify firmware integrity. A mismatch (e.g., caused by malware) invalidates the chain and may prompt a secure update. Crucially, the device identity (Key_{ID}) remains unchanged. This architecture supports cloud-agnostic deployments. While the cloud service and manufacturer may be distinct, all necessary metadata (device certificates, FWIDs) are assumed to be securely distributed by the manufacturer to trusted cloud platforms, supporting multi-cloud and vendor-agnostic trust infrastructures.

3.3.2 Protocol Functional Flows

Having established the layered architectural components of the DICE-based protocol, this section describes the functional behavior of the system across its lifecycle. It distinguishes between a **manufacturing phase**, where device identities are securely provisioned and certified in a controlled environment, and a **deployment phase**, which governs the device's behavior once deployed in the field. These functional flows examine how each core security capability (device identity, device authentication, remote attestation and secure software updates) is achieved in each stage.

3.3.2.1 Manufacturing Phase

Initial enrollment (also known as device provisioning) is a crucial step during IoT device manufacturing, where the device is prepared to securely identify itself and communicate with cloud services or other trusted systems. This process establishes the device's identity, trust anchor, and optionally, its cryptographic credentials [156]. This section focuses on the tasks performed during the initial provisioning of the device in a secure and trusted manner, while considering the foundation security capabilities and design requirements set in section 3.2.
The scope is defined by a limited set of assumptions that maintain generality for practical deployments. These assumptions include:

- **Trusted Environment**: The manufacturing environment is secure and under the control of a trusted entity.
- **Network Capability**: Access to a network is not required.
- **UDS Provisioning**: The Unique Device Secret (UDS) is securely injected into the device.

The key functions of the manufacturing phase include:

1. Unique device identity establishment.
2. Cryptographic key generation.
3. Secure storage of private keys.
4. Initial device certificate enrollment.
5. Secure boot and attestation configuration.

Figure 3.6 shows the manufacturing phase of the protocol, highlighting key interactions between the device and the manufacturer.

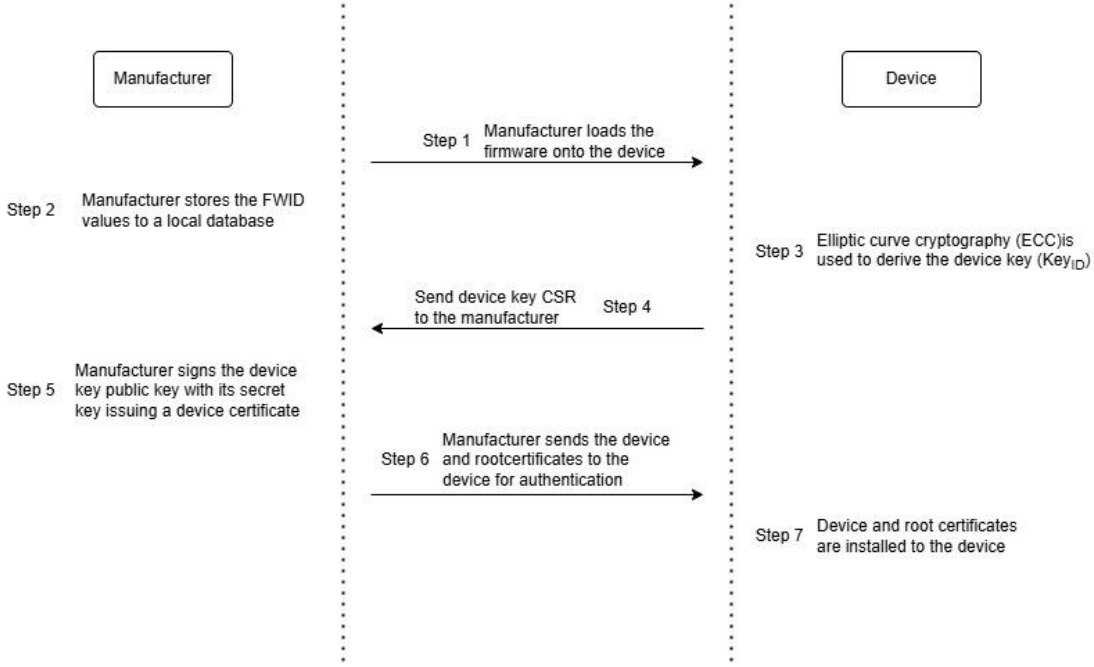

Figure 3.6 Protocol Manufacturing Phase Flow

Functionality Flow

The process begins with the manufacturer loading the firmware onto the device in a secure environment. A trusted environment ensures the confidentiality, integrity, and authenticity of the provisioned data, while minimizing exposure to untrusted entities, including personnel and software. The UDS is injected into the device and stored in read-only memory (ROM). Upon boot, the DICE Engine executes first and uses the UDS and a cryptographic hash (SHA-256) of the first firmware layer to compute the CDI. A recommended mechanism for CDI derivation is a HMAC-based key derivation function (HKDF) using SHA-256, which ensures that the UDS cannot be derived from the CDI

The device then uses elliptic curve cryptography (ECC) to generate its asymmetric key pair (**Key_{ID}**) securely within a microcontroller unit (MCU) or secure element. The private portion of this key remains confined to the device. To certify its identity, the device generates a certificate signing request (CSR), which includes the public key and is signed using the corresponding private key. This CSR is transmitted securely to the manufacturer's certificate authority (CA) over an authenticated and encrypted channel.

The manufacturer's CA verifies the CSR and signs it, creating the device certificate. This certificate binds the device's public key to its identity and includes the CA's digital signature. The signed certificate is then securely installed on the device in flash or non-volatile memory. To reduce risk, debugging interfaces such as JTAG and UART should be either disabled or access-controlled to prevent unauthorized modification after provisioning.

The manufacturer also installs its own CA certificate (root or intermediate) on the device to support authentication once the device is in the field. Finally, the firmware identities (FWIDs) of the device's software layers are extracted and stored in a local database or backend cloud. This process implicitly initializes secure boot, as the measurement of the first firmware layer is cryptographically bound to the CDI. The recorded FWID then serves as a reference for future attestation, enabling remote verifiers to detect firmware tampering or unauthorized updates.

At the end of the manufacturing phase, the device undergoes final mechanical assembly. Firmware, device identity credentials, and UDS are written to protected storage. The device holds a signed certificate that authenticates its public key, and the manufacturer's CA retains the corresponding certificate chain for use in later authentication and attestation processes. A final integrity check confirms the correctness of the installed software and configuration, after which the device is flagged in the manufacturer's backend system as eligible for deployment. Manufacturing functional flow fulfills the first design requirement (see **Section 3.2**) by enabling the derivation of an asymmetric device key pair (**Key_{ID}**). It also satisfies the second requirement, ensuring that the private portion of the key remains protected through secure manufacturing practices.

3.3.2.2 Deployment Phase

Following secure provisioning during the manufacturing phase, the deployment of an IoT device marks its transition from a controlled environment to its operational state in the field [26]. Deployment refers to the physical installation of the device at its intended location, initialization with network-specific parameters, and integration into a live infrastructure. This phase not only activates communication interfaces but also initiates secure connectivity with external services, particularly cloud infrastructure.

During deployment, the device must authenticate its identity using the cryptographic credentials established during manufacturing and validate its software integrity through secure boot and remote attestation. Proper deployment ensures that the device integrates into a larger system architecture while preserving the security guarantees established during provisioning, particularly with respect to identity, data confidentiality, and resistance to tampering.

Transport Layer Security (TLS) is the protocol selected for secure communication and authentication in this design. Its widespread adoption, standardized interoperability, and proven security guarantees make TLS particularly suitable for embedded systems with constrained resources [157]. The use of TLS in this context satisfies system requirement R4, as defined in Section 3.2.

The remainder of this section analyzes the functional steps involved in the deployment process, as illustrated in Figure 3.7. These steps form the operational flow of the device as it transitions into the production environment:

1. Network configuration
2. TLS handshake with a cloud server
3. Device authentication
4. Remote Attestation
5. End of TLS Handshake

Step 1 - Network configuration

Upon initial power-up, the device configures its network stack, thereby satisfying system requirement R3 (see Section 3.2). It activates the appropriate communication interface and retrieves network parameters using DHCP and DNS or relies on static configuration if required. Parameters include IP addresses, subnet masks, default gateways, and port numbers.

Step 2 - TLS handshake with the cloud server

To establish a secure and authenticated communication channel, the IoT device initiates a TLS handshake by sending a ClientHello message to the cloud server. This message includes the supported TLS version (e.g., 1.2 or 1.3), a list of preferred cipher suites, and relevant extensions such as Server Name Indication (SNI), enabling virtual server selection. A client-generated nonce is also included to contribute to session key derivation.

The server responds with a ServerHello, confirming the negotiated parameters: selected TLS version, cipher suite, and a server nonce. Depending on the TLS version and configuration, session-specific parameters such as session IDs or resumption tickets may also be included.

Following the negotiation, both parties exchange ephemeral key shares and compute a shared secret using the Elliptic Curve Diffie-Hellman (ECDH) key exchange. This enables the derivation of symmetric session keys without directly transmitting them, thereby ensuring confidentiality and forward secrecy.

Step 3 - Device authentication

Once the TLS channel is established, mutual authentication proceeds using X.509 certificates. The server first sends its certificate to the device, which validates it against a trusted root certificate installed during manufacturing. This step ensures the device communicates with a trusted and authorized infrastructure.

The device then authenticates itself to the server. It computes a digital signature over the entire handshake transcript using SHA-256 and signs it with its private key, derived from the final stage of its DICE-based key hierarchy. This signature cryptographically binds the device's identity to the handshake process.

Next, the device transmits its certificate chain, which includes its own certificate and any intermediate certificates linking it to a trusted manufacturer root. The server validates the chain by verifying digital signatures, confirming the chain terminates at a trusted authority, and checking certificate validity periods. If all checks succeed, the device is authenticated, and a mutually trusted session is established.

Step 4 – Remote attestation

After authentication, the cloud server attests to the integrity of the device's software by verifying firmware identities embedded in the certificate chain exchanged during the TLS handshake. In the DICE-based architecture (see Section 3.3.1), each certificate corresponds to a firmware layer and includes a structured extension field containing its firmware identifier (FWID).

The server extracts these FWIDs and compares them against a trusted database of known-good reference hashes registered during manufacturing. A match confirms that the corresponding firmware layer remains unaltered and authorized. If all FWIDs align with the expected values, the device's software stack is validated as genuine, completing the attestation process and establishing a high-assurance root of trust for secure operation.

Step 5 - End of TLS handshake

Following successful authentication and attestation, the TLS handshake proceeds to its final stage: confirming session integrity and transitioning to secure communication. The device initiates this phase by sending a ChangeCipherSpec message, indicating that subsequent traffic will be encrypted using the negotiated symmetric session keys. It then transmits a Finished message containing a hash-based Message Authentication Code (MAC) over the entire handshake transcript, binding all prior messages, certificates, and parameters to a single cryptographic context.

The server validates the MAC by computing its own version from the handshake transcript and comparing it to the device's. If verified, the server responds with its own ChangeCipherSpec and Finished messages, completing the mutual authentication and key confirmation. The device performs a final MAC verification, after which the session transitions to encrypted and authenticated communication. At this point, application data may be exchanged securely between the device and the cloud server.

3.3.3 Protocol Security Analysis

With the protocol's functional flows now established, it is essential to assess its resilience against potential security threats. This section presents a qualitative security analysis of the proposed design, guided by established threat modeling practices. The analysis evaluates the protocol's capacity to maintain device trustworthiness by identifying relevant threats, outlining key assets, and mapping them to the system and security requirements defined in Section 3.2. Special attention is given to the protocol's ability to support trusted device identity, remote attestation, and secure update mechanisms within the constraints of IoT environments. In doing so, this section provides a structured foundation for understanding the protocol's robustness throughout its lifecycle.

Threat Model

This section presents a structured security analysis of the proposed asymmetric attestation protocol for constrained IoT devices. To ensure methodological rigor and alignment with best practices in security engineering, the analysis adopts the STRIDE threat modeling framework developed by Microsoft [158]. STRIDE evaluates six threat categories, **Spoofing**, **Tampering**, **Repudiation**, **Information Disclosure**, **Denial of Service**, and **Elevation of Privilege,** each mapped to relevant system assets, trust boundaries, and security requirements identified in Section 3.2.

System Scope and Assumptions

The attestation protocol operates in a potentially adversarial environment, where attackers may intercept communication, access devices physically, or attempt firmware manipulation. The protocol assumes:

- Secure on-device derivation and protection of asymmetric keys (R1, R2).
- Network availability with TLS-based mutual authentication (R3, R4, R5).
- Hardware support for minimal isolation mechanisms, such as secure boot and protected memory regions (R6).

Adversarial Capabilities

The attacker is assumed to be resourceful and persistent, with the following capabilities:

- **Network-level control**: interception, modification, or replay of messages.
- **Physical access**: attempts to extract secrets or modify firmware.
- **Software-level exploitation**: injection of malicious firmware or use of vulnerable code paths.
- **Side-channel attacks**: extraction of secrets via power analysis, fault injection, or timing-based techniques.

The attacker is assumed unable to break standard cryptographic primitives (e.g., ECC, SHA-256) or compromise immutable hardware roots of trust (e.g., ROM-based bootloaders or one-time programmable secrets).

Trust Boundaries

The protocol defines three primary trust boundaries:

- **DICE Layer 0** (ROM): generates the Compound Device Identifier (CDI).
- **Secure Bootloader**: verified, integrity-protected code that anchors mutable trust.
- **TLS Stack and crypto libraries**: perform mutual authentication and remote attestation.

All other software, including application and update code, is untrusted until verified through attestation. To systematically evaluate the protocol's resilience against common attack vectors, Table 3.5 organizes the identified threats using the STRIDE model, linking each threat category to relevant system components and the mitigation measures implemented through design choices and requirements.

Table 3.5 Threat matrix using STRIDE

Threat	Description	Relevant Components	Mitigation Strategy
Spoofing	Attacker impersonates a legitimate device or server	Device credentials, identity keys, TLS	Unique key derivation from UDS (R1, R2), mutual authentication via TLS client certificates (R4, R5)
Tampering	Modification of firmware or messages in transit	Firmware, attestation tokens, TLS sessions	Secure boot (R1), signed firmware updates, TLS encryption and integrity protection (R4)
Repudiation	Entities deny having performed actions	Logs, attestation results	Digital signatures on reports (R1, R2), authenticated logging (optional)
Information Disclosure	Unauthorized access to confidential information	Private keys, firmware, TLS session data	Confidential storage of keys (R2, R6), use of secure memory, TLS with strong cipher suites, debug interfaces disabled
Denial of Service	Disruption of device operation	Network stack, update logic	Input validation, timeout mechanisms, TLS rate limiting (optional), minimal attack surface
Elevation of Privilege	Unauthorized escalation of access rights	Bootloader, firmware update routines	Memory isolation (R6), debug lockout, layered trust via secure boot, separation of responsibilities across firmware components

Each STRIDE category corresponds to a specific class of system vulnerability. By systematically addressing each through architectural and cryptographic controls, the proposed design provides strong protections against both remote and physical attackers. The countermeasures derive directly from the system requirements defined in Section 3.2.2, ensuring consistency between the threat model and the protocol's trust anchors.

Asset Analysis

To maintain a trustworthy system state throughout the device lifecycle, the protocol design carefully identifies critical assets involved in identity derivation, remote attestation, and secure software updates. Each asset plays a distinct role within the security architecture and requires tailored protection strategies to mitigate potential threats.

The table below summarizes the key assets, their function within the protocol, and the associated storage or protection mechanisms.

Table 3.6 Asset protection table

Asset	Description / Role	Storage / Protection Mechanism
UDS Entropy	Source of device-specific entropy used for unique key derivation	Volatile memory accessed during boot; cleared post-boot to prevent leakage
Private Key(s)	Device-unique asymmetric signing keys	Write-protected memory region or secure hardware vault; never exposed externally

Asset	Description / Role	Storage / Protection Mechanism
Public Key(s)	Used in TLS mutual authentication and certificate chaining	Stored in non-volatile memory (NVM), protected by memory access restrictions
Bootloader & FW Layer 1	Verifies integrity of firmware and performs initial measurements	Write-protected flash; subject to secure boot verification
Certificates	Assert device identity and software integrity to remote verifiers	Issued and anchored during provisioning; validated during TLS handshake
Firmware Layer 2	Executes main application logic; subject to integrity checks	Validated at boot using cryptographic measurements; updates require signature

The asset management strategy follows a defense-in-depth principle, beginning with identity derivation from hardware-resident secrets during the initial boot stage. Through DICE-based layering, each successive stage propagates trust via cryptographic bindings, verified certificates, and authenticated execution. Protecting the generation, storage, and validation of these assets is central to ensuring the confidentiality, integrity, and authenticity of the device across its operational lifecycle.

Assumptions and Security Requirements

The security guarantees of the proposed protocol rest on a set of well-defined assumptions and a minimal, yet essential, set of system and security requirements. These foundations are critical for interpreting the threat landscape defined in the previous subsection and for ensuring that the architectural protections described throughout Chapter 3 remain valid under real-world deployment conditions.

Design Assumptions

As established in Section 3.2.2, the protocol makes a small number of explicit assumptions regarding the operational environment and hardware platform. These assumptions are intentionally minimized to enhance portability and reduce reliance on external conditions:

- **A1 – Cryptographic Trust Primitives**: The underlying cryptographic algorithms (ECC, SHA-256) are assumed secure and not vulnerable to feasible mathematical or implementation-based attacks.
- **A2 – Hardware Root of Trust**: The immutable boot ROM is trusted and cannot be modified or bypassed by an attacker.
- **A3 – Secure Key Storage**: The platform provides sufficient memory protection to prevent unauthorized access to confidential key material.
- **A4 – Secure Communication Stack**: The TLS stack used for device-server communication is assumed to be correctly implemented and configured with strong cipher suites.
- **A5 – Debug Interfaces Disabled**: Post-provisioning, all hardware debug interfaces (e.g., JTAG, SWD) are disabled or locked to prevent physical exploitation.

These assumptions correspond to realistic capabilities of commercially available IoT microcontrollers and reflect common industry practices in secure embedded systems design.

Mapping Requirements to Security Capabilities

The security goals outlined in the protocol design, device identity, device authentication, remote attestation, and secure software update, each impose distinct protection requirements. Table 3.7 in Section 3.2 maps these capabilities to system requirements R1-R6. This section further clarifies how

those requirements, together with the assumptions above, contribute to mitigating the threats identified in the STRIDE-based model.

Table 3.7 Mapping of security capabilities to system requirements

Security Capability	Required Protections	Supported By
Device Identity	Secure generation and storage of asymmetric keys; uniqueness and non-reproducibility	A1, A2, A3; Requirements R1, R2, R6
Device Authentication	Mutual TLS authentication with certificate validation and chain-of-trust anchoring	A4; Requirements R4, R5
Remote Attestation	Integrity-protected firmware measurements and authenticated attestation reports	A1, A2, A3, A4; Requirements R1, R2, R4, R5
Secure Updates	Verification of firmware signatures, rollback protection, and controlled access to update interfaces	A2, A5; Requirements R1, R3, R6

Each security capability is thus grounded in both a set of implementation safeguards and protocol-level design principles, ensuring layered defense against a wide range of threat vectors.

Security Design Rationale

By explicitly enumerating both assumptions and requirements, this design promotes clarity in the system's trust boundaries and facilitates rigorous validation. Moreover, it aligns with best practices in security engineering by:

- Minimizing the Trusted Computing Base (TCB) to only essential components such as the bootloader, cryptographic primitives, and certificate store.
- Constraining assumptions to industry-standard conditions that are auditable and testable.
- Ensuring coverage of all STRIDE threat categories through direct architectural mitigations.

This structured security baseline allows the protocol to maintain consistent trust guarantees across the device lifecycle, from provisioning to field operation, even under adversarial conditions.

This security analysis demonstrates that the proposed asymmetric attestation protocol incorporates a robust set of architectural protections, grounded in formal threat modeling and aligned with best practices for embedded systems. By leveraging a minimal yet sufficient Trusted Computing Base, enforcing layered cryptographic validation, and maintaining clear security boundaries, the protocol mitigates a wide spectrum of realistic threat scenarios. The use of the STRIDE framework has enabled systematic reasoning about adversarial behavior and validation of defense mechanisms at each critical stage of the boot and update process. Ultimately, the design achieves a balance between security assurance and resource efficiency, fulfilling the protocol's objectives under constrained IoT deployment conditions.

4 Implementation and Results

This chapter presents the implementation and evaluation of the asymmetric attestation protocol introduced in Chapter 3. The main objective is to demonstrate the feasibility of realizing the protocol on a constrained embedded platform while preserving its core security goals. To this end, a structured implementation strategy was adopted, consisting of a software/hardware co-design on an STM32-based embedded system. Section 4.1 introduces the system architecture, mapping the protocol's layered trust model to the physical and logical structure of an embedded microcontroller. It details the design rationale for each subsystem, including the ECC and symmetric cryptographic units, memory layout, and power and debug interfaces. Section 4.2 describes the development and testing of two complementary prototypes: a software-only proof of concept and a hardware-backed demonstration on an STM32 Nucleo board. This section walks through the implementation of manufacturing and enrollment phases, integration of cryptographic components, and setup of a demonstrator environment for live validation. Section 4.3 reports the empirical evaluation results, quantifying cycle counts, memory usage, and protocol execution timings using hardware instrumentation. It also reflects on measurement challenges, design trade-offs, and potential improvements for future iterations. Together, these implementation efforts validate the practical feasibility of the attestation protocol under real-world constraints and lay the groundwork for further system integration and deployment. The outcomes also support the broader evaluation and future considerations discussed in Chapter 5.

4.1 System Architecture and Design Exploration

To demonstrate the practical feasibility of the attestation protocol introduced in Section 3.3, its abstract design must be realized through a concrete implementation on an embedded platform. This translation from theoretical protocol behavior to operational system behavior allows for validation of the protocol's core security guarantees under realistic platform constraints. The chosen implementation targets a resource-constrained IoT device, representative of edge environments where memory, computational power, and energy efficiency are highly limited. To support this mapping, a well-structured system architecture is needed—one that clearly organizes both hardware and software elements involved in the attestation flow and aligns with the trust and layering principles defined in the protocol design.

The architectural design presented in this chapter is guided by the functional and security requirements established in Section 3.2.1. To satisfy these objectives, the system must deliver four essential capabilities: (1) derivation of a unique device identity anchored in a hardware-protected secret, (2) authentication with a remote server through PKI-based credentials over a secure TLS channel, (3) attestation of the integrity of the installed firmware, and (4) validation and application of software updates in a manner that preserves the authenticity and integrity of the device. The following sections elaborate on the system architecture developed to support these capabilities and examine the design choices made across hardware platforms, cryptographic libraries, and communication interfaces

4.1.1 High-Level Architecture

Implementing the asymmetric attestation protocol introduced in Section 3.3 on a physical IoT device necessitates a secure and modular system architecture. This section introduces the fundamental components required to realize the protocol and explains how these elements interact within the constraints of a resource-limited embedded environment. The architecture builds upon the Device Identifier Composition Engine (DICE) framework, and each system component is tailored to meet

specific requirements for secure identity derivation, cryptographic processing, and controlled internal and external communication.

The proposed system architecture instantiates the layered DICE-based protocol architecture by mapping each of its conceptual stages directly onto the embedded memory layout of the target device. Execution begins with the immutable DICE layer, implemented in ROM, which is triggered immediately upon system reset. This initial layer has exclusive access to the Unique Device Secret (UDS) and generates the Compound Device Identifier (CDI) by applying a cryptographic one-way function to both the UDS and a measurement of the first mutable firmware. The first mutable layer, located in Flash memory, uses the resulting CDI to derive the device's asymmetric identity key pair. It also generates a layer-specific key used for authentication and attestation. A second mutable layer, loaded into SRAM, hosts runtime services such as network communication, remote attestation, and secure firmware updates. This tiered memory organization enforces a one-way trust chain, whereby each successive stage inherits and builds upon the cryptographic trust established by its predecessor. The system architecture, illustrated in **Figure 4.1**, is composed of the following functional components:

ECC Scalar Multiplication Unit

Elliptic Curve Cryptography (ECC) plays a foundational role in the proposed attestation protocol, particularly in establishing a cryptographically verifiable device identity. During the manufacturing and enrollment phases, the device executes scalar multiplication to derive asymmetric key pairs, one forming the core identity of the device, and others serving layer-specific purposes such as attestation and authentication. Given the computational intensity and security sensitivity of scalar multiplication, careful consideration is given to how this operation is implemented. Whether executed in software or supported by hardware acceleration, the performance and correctness of this unit are critical. Section 4.1.2 provides further details on the selected ECC curve and implementation choices. Notably, this unit supports key system requirements such as R1 (identity rooted in hardware) and R2 (cryptographic flexibility within constrained environments).

Symmetric Cryptographic Unit

Symmetric cryptographic functions play a supporting role in the protocol’s security architecture, particularly in the efficient handling of encryption and key derivation tasks. The Advanced Encryption Standard (AES) is utilized to ensure confidentiality and data integrity where needed. Moreover, AES may serve within a Key Derivation Function (KDF) to produce layer-specific keys in a resource-efficient manner, aligning with the design constraints of embedded platforms.

One-Way Function Unit

The one-way function is responsible for deriving the Compound Device Identifier (CDI) from two inputs: the Unique Device Secret (UDS), which serves as the hardware-based root of trust, and a cryptographic measurement of the first mutable firmware layer. This function must ensure strong pre-image resistance, making it computationally infeasible to reconstruct the UDS from the resulting CDI. This cryptographic irreversibility enforces a strict isolation between immutable hardware secrets and software-derived identities, in accordance with the DICE hardware requirements outlined in Section 3.3.1. The derivation process is embedded in the ROM-based root layer, ensuring that the integrity and confidentiality of the root-of-trust are preserved from the earliest stage of system execution. More details about the selection of a proper one-way function are provided in Section 4.1.2.

Non-Volatile Memory (NVM) Unit

The root layer of the DICE architecture, along with its early boot logic, is stored in non-volatile memory to ensure that system initialization begins from a stable and tamper-resistant baseline. ROM or Flash memory is used to persist the logic responsible for accessing the UDS and deriving the CDI. By anchoring these routines in immutable storage, the design establishes a reliable hardware-based root of trust that underpins the device's security lifecycle.

Test and Debug Unit

Test and debug interfaces are essential during system development, enabling low-level inspection, validation, and troubleshooting of embedded software and hardware behavior. However, in the context of secure deployments, especially within the DICE framework, these interfaces pose serious risks if left active. During the execution of the DICE root layer, all debugging features, such as JTAG, SWD, and serial wire interfaces, must be strictly disabled to prevent any possibility of exposing the UDS or leaking side-channel information that could aid in reconstructing the CDI. This requirement extends to the execution of the first firmware layer, where cryptographic keys derived from the CDI are initialized. To uphold the integrity of the trust chain, the system must enforce permanent deactivation or rigorous access control of debug ports in production environments.

Power Management Unit

Energy efficiency is a critical consideration in embedded and IoT systems, where long-term deployment often depends on constrained power sources. The system must incorporate features such as low-power operation modes, clock gating, and peripheral shutdown to minimize unnecessary energy consumption during inactive periods. These mechanisms help ensure that the introduction of cryptographic protocols and security monitoring does not compromise the device's operational lifetime or violate its energy constraints.

Control Unit

The control unit embodies the logical core responsible for orchestrating the execution of the attestation protocol and managing both internal coordination and external communication. Conceptually, it encompasses all control functionality required to implement the protocol, including device manufacturing, identity enrollment, integrity attestation, and secure firmware updates. While conceptually abstracted here, this functionality is typically realized on the MCU hardware in a concrete implementation. When implemented on a constrained embedded platform, the control unit must operate within strict limits on processing power, memory, and energy consumption. Despite these constraints, it must uphold the security and performance objectives defined in Section 3.2.1.

Optional: Register File Analog (RFA) and Register File Digital (RFD)

In certain configurations, auxiliary subsystems such as Register File Analog (RFA) and Register File Digital (RFD) may be integrated to streamline peripheral coordination and manage internal state transitions. While not essential to the attestation protocol itself, these units can contribute to deterministic execution behavior, particularly in systems operating under tight timing or resource constraints. Their inclusion depends on the intended application profile and may provide improved control over low-level system dynamics.

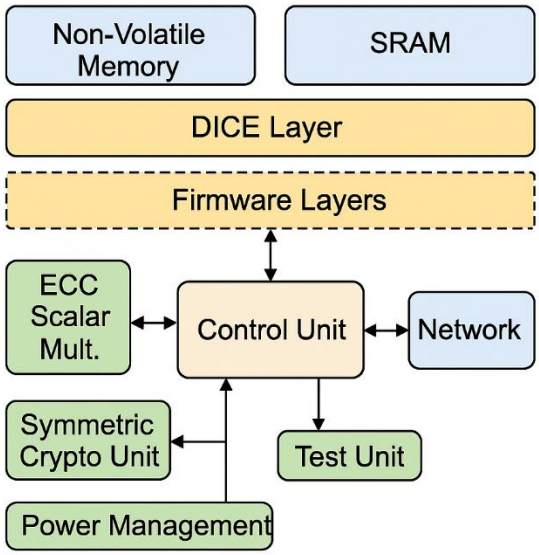

Figure 4.1 High-level system architecture

The architecture presented here represents a deliberate integration of the protocol's security goals with the practical constraints of embedded IoT systems. As illustrated in Figure 4.1, the design organizes the essential functional components, such as cryptographic engines, memory regions, and communication interfaces, into a cohesive framework that supports layered trust establishment. The figure reinforces the hierarchical structure discussed above and provides a visual summary of how the attestation protocol is distributed across boot stages. Building on this foundation, the next section examines available implementation paths and the rationale behind key design decisions in both hardware and software domains.

4.1.2 Design Space Exploration

Developing a secure embedded system, such as the architecture introduced in Section 4.1.1, requires careful navigation through a wide array of design choices. From hardware platforms and cryptographic primitives (e.g., ECC or AES accelerators) to memory hierarchies and communication protocols, each decision entails trade-offs in computational efficiency, security assurance, and development complexity. The challenge lies in selecting a coherent set of subsystems that not only meets the functional and security requirements set forth in Section 3.2.1 but also ensures practical feasibility on a resource-constrained IoT platform. Achieving this balance is vital to ensure that the final implementation remains both resilient and efficient within the embedded system's constraints.

A structured design space exploration (DSE) was conducted to identify viable combinations of hardware platforms, cryptographic libraries, and communication interfaces compatible with the system architecture presented in Figure 4.1. In this context, DSE refers to a systematic evaluation of implementation alternatives, constrained by factors such as memory footprint, power efficiency, cryptographic support, integration complexity, and development overhead [159]. This method is well-suited for secure embedded systems, which demand both correctness and cost-effectiveness. The goal of this exploration was to identify a prototype setup that both fits the hardware constraints and remains consistent with the protocol design from Chapter 3.

The design space exploration considered three implementation strategies: software-only designs, hardware-dedicated architectures, and software/hardware co-designs. A software-based implementation provides flexibility for rapid prototyping and streamlined debugging. For instance, the attestation protocol can be emulated as a user-space application on a general-purpose processor

(GPP), such as a desktop-class CPU. This abstraction reduces integration effort by isolating implementation from hardware dependencies. However, such implementations do not accurately reflect the constraints of embedded IoT platforms, particularly in memory availability, energy efficiency, and execution latency. In addition, critical platform-level security properties, such as physical isolation of secrets and immutable root-of-trust enforcement, cannot be reliably modeled in a purely software environment.

A hardware-only implementation, such as one based on a custom ASIC or an FPGA architecture, was ultimately deemed impractical for the scope of this thesis. Although such solutions can offer excellent performance, physical isolation, and tight integration of cryptographic functions, they present significant barriers in terms of design flexibility, development time, and tooling complexity. In particular, implementing the DICE protocol entirely in hardware would require low-level control over key derivation and attestation flows, tasks that are better suited for iterative software development in early-stage prototypes. While commercial embedded platforms with built-in DICE support do exist, they are often tightly coupled to proprietary toolchains or vendor-specific implementations. As such, a hardware-only path is considered more appropriate for mature industrial deployments, where long-term maintainability and certification justify the engineering overhead.

A software/hardware co-design presents a more practical model for implementing security protocols in embedded environments. In this approach, computationally intensive operations, such as elliptic curve scalar multiplication and symmetric encryption, are delegated to dedicated hardware modules, while system control logic, attestation routines, and communication flows are managed in software. This architecture achieves a balance between efficiency and adaptability: hardware acceleration improves performance and reduces energy consumption, while software remains flexible for future updates or protocol refinements. Several modern microcontroller platforms support this design paradigm by integrating cryptographic engines, secure key storage, and low-power processing features within a compact and cost-effective footprint.

Given these factors, this thesis adopts a software/hardware co-design, deploying it on a commercially available microcontroller. This approach provides a practical balance between efficiency, adaptability, and system realism. It enables end-to-end validation of asymmetric attestation introduced in Chapter 3 under realistic deployment conditions, while remaining compatible with the resource limitations and deployment scenarios typical of embedded IoT systems. Following this design choice, the next task was to evaluate practical implementation options for each system component.

Implementing the architecture depicted in Figure 4.1 necessitates a careful selection of hardware and software components that meet the system requirements established in Chapter 3. Each functional block, whether focused on cryptographic operations, secure data retention, or communication, must perform reliably within the stringent constraints of low-power, low-memory embedded environments. To this end, a comparative evaluation of candidate microcontrollers, libraries, and protocols was carried out, with each option weighed against specific trade-offs in computational efficiency, memory usage, platform support, and compliance with the protocol's design principles.

The following subsections detail the selected design solutions for each system component, providing rationale for their inclusion and assessing their suitability for the embedded attestation prototype.

Hardware Platform Selection.

A range of embedded platforms was evaluated for their suitability as hosts for the prototype, including Arduino-based boards (e.g., Uno, Mega), ESP32 variants, Nordic Semiconductor's nRF52 series, Microchip's SAMD and SAM L11 secure MCUs, NXP's i.MX RT series, and STMicroelectronics STM32 Nucleo line. The evaluation criteria spanned computational performance, availability of hardware

cryptographic modules, secure storage features, power efficiency, and ecosystem maturity. After comparative analysis, the STM32L4 Nucleo board was selected for its strong balance between processing capability and low-power operation. It offers hardware-accelerated AES, a built-in true random number generator (TRNG), and a comprehensive set of low-power modes, features aligned with protocol objectives. Furthermore, its integration with ST's development ecosystem simplifies firmware development, debugging, and deployment in constrained environments [160].

ECC Scalar Multiplication Unit.

Elliptic curve scalar multiplication serves as the foundation for all asymmetric key derivation operations in the attestation protocol, including both the device identity key (Key_{ID}) and firmware-specific layer keys (Key_{Layer_n}). Several cryptographic libraries were considered for this task, namely micro-ecc, wolfSSL, and mbedTLS, each offering different trade-offs in terms of footprint, portability, and standards compliance. The mbedTLS library was selected due to its modular architecture, comprehensive standards support, and seamless integration with STM32 Cortex-M platforms. Unlike micro-ecc, which is minimal but lacks certain protocol-level primitives, or wolfSSL, which introduces additional licensing and integration overhead, mbedTLS provides a balanced cryptographic stack tailored for embedded applications [161] – [163].

For elliptic curve operations, the SECP256R1 curve (also referred to as NIST P-256) was chosen. This decision reflects its widespread use in TLS-based infrastructures and its established role in standardized public-key authentication schemes. While alternatives such as Curve25519 offer performance benefits and stronger resistance to certain side-channel attacks, SECP256R1 aligns more closely with certification pathways and platform compatibility requirements. To ensure correct ECC key generation, a deterministic HMAC-based random number generator is used, enabling reproducibility in test scenarios while maintaining cryptographic integrity. On the STM32L4 platform, ECC scalar multiplication is further optimized by leveraging hardware-assisted floating-point and DSP instructions, enabling secure and efficient key generation within the computational constraints of the embedded system.

One-Way Function Unit.

The One-Way Function Unit is responsible for deriving the CDI from the hardware-protected UDS, a critical step in establishing the device's root of trust. The transformation must guarantee strong pre-image resistance, ensuring that knowledge of the CDI does not compromise the confidentiality of the UDS. Several cryptographic constructions are suitable for this purpose, particularly Message Authentication Codes (MACs), which offer robust key protection and are well-suited to iterative processing of structured input data [164]. Among MAC-based options, HMAC constructions are especially attractive due to their proven resistance against collision and pre-image attacks, even when the underlying hash function exhibits moderate cryptographic weaknesses [165]. Alternative schemes, such as MACs built on block ciphers (e.g. CBC-MAC) can offer computational advantages in resource-constrained environments, particularly when paired with hardware-accelerated AES support [166].

For this prototype, an HMAC based on SHA-256 was selected to realize the one-way transformation. This design balances security strength, implementation simplicity, and compatibility with widely adopted cryptographic frameworks. The STM32L4 platform's support for hardware AES could potentially support CBC-MAC constructions in future versions, but HMAC-SHA256 was prioritized due to its broader tooling support and cryptographic maturity. The implementation leverages the mbedTLS cryptographic library, which offers a lightweight and modular interface for embedded systems and allows integration of the one-way function as part of the DICE initialization flow.

Symmetric Cryptographic Unit.
The Symmetric Cryptographic Unit is responsible for deriving the layer-specific keys required in the attestation protocol. Specifically, it employs a HMAC-based Key Derivation Function (HKDF) to expand the entropy provided by the Compound Device Identifier (CDI) into cryptographic key material. The key material is subsequently used by the ECC unit to generate asymmetric key pairs unique to each firmware layer (Key_{LayerN}). While the underlying HMAC primitive is also used in the One-Way Function Unit for CDI derivation, its role here is distinct and aligned with symmetric key expansion. The HKDF implementation is provided by the mbedTLS library, which supports compact and reliable integration on Cortex-M microcontrollers. This setup ensures that key derivation remains efficient, modular, and secure under the resource constraints of the STM32L4 platform.

Non-Volatile Memory Unit.
The DICE root layer, comprising early boot logic, UDS access routines, and CDI derivation functions, is stored in the STM32L476's internal Flash memory (1 MB). This non-volatile storage provides the immutability required to satisfy the DICE model's trust anchor requirements, ensuring that execution begins from a consistent and tamper-resistant state. On the Nucleo-L476RG board, this Flash is mapped to a fixed memory region with read-only protection options available through hardware fuses and option bytes. These features reinforce the assumption of an immutable root of trust and allow for secure storage of the boot-critical firmware segment.

Test and Debug Unit.
During development, the Nucleo board's integrated ST-Link/V2-1 interface was used for real-time debugging, code stepping, and logging over UART. GPIOs were also configured for simple logic probing and external instrumentation. While such interfaces are essential for validation and early-stage testing, they represent significant security liabilities in a production context. If left active, debug pathways may expose sensitive memory contents or allow bypassing of attestation checks. To mitigate these risks, STM32 devices support disabling debug access through option byte configuration and Read-Out Protection (RDP) levels. In a deployment scenario, these features must be explicitly enabled to ensure the integrity of the trust establishment process.

Power Management Unit.
Power efficiency is a critical consideration for resource-constrained IoT platforms, particularly when executing cryptographic routines that demand sustained processing. The STM32L476 microcontroller supports several low-power operational modes, **Sleep**, **Stop**, and **Standby,** each offering progressive reductions in power consumption while preserving essential system functionality. These modes are configured via dedicated power control registers and allow the system to dynamically scale energy usage based on activity state. **Stop mode** was selected during idle phases of attestation execution to balance energy savings with acceptable wake-up latency. Integrated voltage regulators and a low-power real-time clock (RTC) enable timed wake-ups and state retention across suspend intervals. Power was supplied via USB or external 3.3 V/5 V sources, with current consumption profiled during runtime to assess the feasibility of cryptographic operations under realistic energy constraints. These features contribute to meeting the energy-efficiency goals outlined in Section 3.2.1.

Control Unit.
The Cortex-M4 microcontroller acted as the central orchestrator of protocol execution, coordinating interactions among system components and managing communication with external entities such as the manufacturer and verifier. Its rich peripheral set—including UART, I²C, SPI, and USB Full-

Speed—facilitated versatile data exchange between internal cryptographic units and external interfaces. Depending on deployment requirements, the CAN interface could also be leveraged to enable fault-tolerant communication in noisy or distributed environments. While abstracted here as a system control module, this functionality was concretely realized through the STM32 platform's MCU, which operated under strict power and memory constraints in accordance with the system goals defined in Section 3.2.1.

4.2 Prototype Development

Building on the architectural framework and component-level design choices outlined in Section 4.1, this section presents the development of two complementary implementations of the attestation protocol introduced in Chapter 3. The previous section identified the key hardware and software subsystems necessary to realize the DICE-based architecture on a constrained embedded platform. It further explored architectural trade-offs and justified the adoption of a software/hardware co-design. The present section transitions from architectural planning to practical realization, describing how the proposed design was instantiated in both a host-based simulation environment and a hardware-based embedded prototype.

The first implementation is a software-only proof of concept developed on a general-purpose processor. This initial stage aims to validate the core security functionality of the protocol, particularly the DICE-based identity derivation and layered key hierarchy, without the constraints of embedded hardware. It provides early validation of the protocol logic, facilitates debugging and testing, and enables rapid exploration of cryptographic operations such as key generation and attestation token creation. The second implementation builds upon this foundation by deploying the protocol on an STM32-based microcontroller system, as defined in the system architecture of Figure 4.1. This software/hardware co-design prototype demonstrates the protocol's practical feasibility under resource-constrained conditions. In the final part of this section, measurements are presented to evaluate the prototype's area and performance characteristics, thereby providing empirical evidence to support the robustness and deploy ability of the design.

4.2.1 Proof of Concept Implementation

To bridge the gap between the protocol design outlined in Chapter 3 and a practical embedded deployment, an early-stage proof-of-concept implementation was developed. This intermediate step serves multiple purposes. First, it provides a safe and controlled environment for validating the functional flows described in Section 3.3.2, including key derivation, attestation message structure, and verification logic. Second, it allows protocol behavior to be tested independently of hardware-specific limitations, enabling rapid iteration and debugging of security-critical components. A software-only setup thus facilitates timely feedback on architectural decisions, ensures that cryptographic primitives and their composition behave as intended, and confirms the interoperability of protocol elements within a realistic host environment. This early validation step is critical for reducing integration risk in later hardware deployment stages, while also substantiating the protocol's feasibility from an engineering and academic standpoint.

To implement and test the proposed attestation protocol in a controlled environment, a software-only prototype was developed using Python 3.x and auxiliary Bash scripts. The primary development was conducted in the PyCharm integrated development environment (IDE), which provided structured debugging, version control integration, and support for interactive testing. At the cryptographic core of the prototype lies a proprietary software library provided by Intrinsic ID (version *iidtvg-17.2.0*). This library implements a suite of security primitives, including random number

generation, elliptic curve key derivation, HMAC construction, key derivation functions (KDFs), and standard hash functions. Due to confidentiality agreements, the internal implementation details and source code of the library are not disclosed in this report. In addition to the proprietary library, OpenSSL was used as an auxiliary tool for generating certificate signing requests (CSRs), verifying digital signatures, and handling X.509 public-key certificates. This modular stack allows for rapid prototyping and early validation of the cryptographic flows defined in Chapter 3 without dependency on embedded hardware limitations.

Due to confidentiality constraints, the source code of the proof-of-concept implementation cannot be disclosed. Nevertheless, the functionality of the prototype is documented through system-level descriptions. This abstraction enables a clear understanding of the implementation logic while preserving the proprietary nature of the underlying cryptographic library. The proof-of-concept simulates a DICE-based architecture composed of three layers: an immutable DICE root, a first firmware layer, and a second firmware layer, reflecting the protocol architecture defined in Section 3.3.1. As described in Chapter 3, this layered architecture delivers four foundational security capabilities: (1) device identity, (2) device authentication, (3) remote attestation, and (4) secure software updates. The goal of this implementation is to verify the feasibility of realizing these capabilities within a software-only testbench.

The prototype focuses on emulating the manufacturing and enrollment phases, wherein a unique device identity is generated and validated according to the protocol directives. In particular, the flow includes key steps such as generation of a Unique Device Secret (UDS), derivation of the Compound Device Identifier (CDI), generation of asymmetric key-pairs, and creation of a device certificate signing request (CSR). These steps are implemented in alignment with the functional flows introduced in Section 3.3.2 and provide early evidence that the cryptographic and architectural principles of the protocol are practical and interoperable.

Manufacturing Phase

To establish the foundational trust anchors required by the DICE-based architecture, the manufacturing phase initiates the creation of the manufacturer's asymmetric key pair and its corresponding self-signed root certificate. This operation is a prerequisite for enabling secure device enrollment and authentication within the broader Public Key Infrastructure (PKI) of the system. As the source code of the prototype is confidential, the implementation is described through structured explanations. The manufacturer employs OpenSSL, an open-source cryptographic toolkit widely used for TLS and X.509 operations, to perform these tasks. Specifically, an elliptic curve (EC) private key is generated using the *prime256v1* (SECP256R1) curve, a standard supported by most TLS implementations. This key is then used to create a certificate signing request (CSR), which is self-signed to yield the root certificate. The root certificate serves as the ultimate trust anchor for subsequent device certificate chains. This procedure, while simple, represents a critical step in provisioning a trusted enrollment infrastructure, ensuring that all subsequent device identities can be verified by a recognized and cryptographically bound authority.

Enrollment Phase

DICE Layer

The Enrollment Phase begins with the execution of the DICE root layer, which is responsible for deriving a cryptographic anchor for the device identity. This is achieved through a series of deterministic operations that simulate the behavior of a real DICE engine, emulated here in a software-only environment.

The first step involves computing a cryptographic measurement of the first mutable firmware region, referred to as *Layer 1*. In the prototype, this is modeled as a 5,000-byte block of pseudo-random

data generated using a deterministic seed, ensuring repeatability across simulations. This emulates the behavior of a firmware hashing operation typically performed by a DICE hardware component. Next, a **Unique Device Secret (UDS)** is generated. This UDS simulates a device-embedded root secret, typically provisioned during manufacturing and permanently fused into hardware. In this proof-of-concept, a 32-byte UDS is generated using a seeded random number generator to emulate deterministic behavior for testing.

The core of the DICE process lies in the derivation of the **Compound Device Identifier (CDI)**. This is implemented via an HMAC operation, where the UDS acts as the cryptographic key and the Layer 1 code measurement as the message. SHA-256 is used as the digest function, in line with DICE recommendations for cryptographic strength. The result, a 32-byte CDI, acts as a root secret for all subsequent identity derivation steps. In accordance with DICE principles, the UDS is not reused beyond this point and access to it is programmatically restricted.

This software emulation of the DICE layer successfully demonstrates that CDI derivation logic can be implemented in a modular and testable manner, offering a reliable cryptographic foundation for downstream identity and attestation flows.

First Firmware Layer – Device Identity

Following successful CDI derivation by the DICE root layer, control transitions to the first mutable firmware region, **Layer 1,** which is responsible for establishing the device identity. This stage demonstrates how a unique asymmetric key pair can be deterministically derived from the CDI, and how that identity is formally enrolled into the system via a manufacturer-signed certificate.

In the implementation, the **CDI** produced by the DICE layer is used directly as the private key input to an elliptic curve key generation routine. Specifically, the CDI serves as the d parameter in the instantiation of an EC key object based on the SECP256R1 curve (also known as prime256v1). This operation yields a public/private key pair deterministically bound to the UDS and the first firmware measurement, in accordance with the DICE key derivation model.

To enable authentication and establish trust within the system's public key infrastructure (PKI), the device generates a **Certificate Signing Request (CSR)** for its newly derived public key. This CSR is formatted using OpenSSL tooling and contains identifying information as well as the public key material. The request is then submitted to the manufacturer, who acts as the root Certificate Authority (CA). The manufacturer signs the CSR using its private EC key (created during the Manufacturing Phase), thereby issuing a valid X.509 device certificate. This certificate serves as a formal endorsement of the device identity and enables external verifiers to validate its authenticity during attestation.

This portion of the Enrollment Phase confirms that identity derivation, CSR creation, and PKI-based enrollment can be successfully implemented in a modular, interoperable fashion using standard cryptographic tools and deterministic protocol inputs.

First Firmware Layer – Layer Identity

Once the device identity has been established, the protocol proceeds to derive a second asymmetric key pair associated with the **Layer 2 firmware**. This additional key serves to reinforce the layered trust model central to DICE-based architectures, where each firmware layer is cryptographically bound to both its predecessor and the underlying device identity.

As with the previous layer, the process begins by generating a **measurement** of the Layer 2 code. In the proof-of-concept, this measurement is modeled as a random byte sequence and is computed using a deterministic random seed to emulate a stable hash of the firmware contents. The private component of the layer key (**Key_{layer1}**) is then deterministically derived by applying an HMAC-based construction to the previously computed CDI and the Layer 2 code measurement. The resulting

256-bit output acts as the private key input to an ECC key object using the SECP256R1 curve. A corresponding public key is derived, and the complete key pair forms the cryptographic basis for Layer 2.

To enable this new key to be integrated into the device's certificate chain, the device generates a **Certificate Signing Request (CSR)** for the Layer Key using OpenSSL tooling. Unlike the device certificate, which is signed by the manufacturer, the Layer Key's CSR is signed by the device private key, thus creating a cryptographic delegation within the device's internal certificate hierarchy. This demonstrates both **key isolation** between layers and secure delegation of trust, as described in the layered attestation protocol of Section 3.3.

Through these operations, the proof-of-concept successfully validates a key architectural requirement of the protocol: the ability to derive and authenticate firmware-layer-specific identities in a chained, verifiable, and deterministic manner, without exposing underlying secrets or requiring external input beyond initial enrollment.

Discussion

Despite the successful demonstration of identity derivation and cryptographic key flows in a software-only environment, the proof-of-concept implementation lacks real-world constraints such as memory limitations, peripheral interactions, and runtime performance bottlenecks. As such, it does not assess the practical viability of the protocol under deployment conditions. Moreover, critical features such as secure boot, interrupt-handling, and resistance to physical tampering require hardware-level integration. These limitations motivate the development of a microcontroller-based prototype, which is presented in the following subsection to further validate the protocol's deployability and efficiency.

4.2.2 Embedded Implementation

To complement the software-only proof of concept presented earlier, a hardware-based prototype was developed using an STM32 Nucleo development board. This implementation instantiates the attestation protocol in a resource-constrained embedded environment, validating the feasibility of key derivation, certificate handling, and attestation flows under realistic deployment conditions. This section articulates the development and evaluation of a proof-of-concept prototype that validates the four core security objectives of the proposed protocol, device identity, device authentication, remote attestation, and secure software updates. The prototype builds upon the high-level system architecture presented in Figure 4.1 and represents a software/hardware co-design tailored to embedded constraints. The implementation includes a practical setup involving an STM32-based microcontroller, a layered firmware structure, and a supporting software stack for communication and cryptographic operations. The demonstrator scenario emulates realistic deployment conditions, culminating in a live execution of the attestation protocol and validation against a remote verifier. The interaction of system entities, certificate provisioning workflow, and TLS-based verification are summarized in Figure 4.2.

Test Platform

The embedded prototype is based on the STM32L476 Nucleo board, a Cortex-M4 development platform selected for its relevance to industrial IoT applications and its support for low-power operation, hardware security primitives, and flexible peripheral interfaces. The microcontroller features 1 MB of Flash memory and 128 KB of SRAM, allowing for layered firmware deployment and secure storage of cryptographic materials. Its boot sequence is configurable via hardware boot pins, enabling the separation of a DICE root layer, first firmware (Layer 1), and second firmware (Layer 2) as defined in the protocol architecture.

The STM32Cube HAL was used to manage low-level hardware interactions, while the cryptographic operations were implemented using the mbedTLS library. Notably, hardware RNG capabilities were

leveraged to support key generation and secure seed material, consistent with protocol requirements for entropy sourcing. The software stack was structured into a minimal bootloader, two sequential firmware images, and a secure communication module interfacing with external Python and Bash scripts. These scripts facilitated certificate signing, UART communication, and TLS message parsing on the host side.

Demonstration Procedure

The demonstration begins with the STM32 device powering up and executing its DICE-based initialization routine. Upon reset, the DICE root layer computes the Compound Device Identifier (CDI) from the Unique Device Secret (UDS) and a measurement of Layer 1. Using the CDI, the first firmware layer derives the device key-pair and generates a certificate signing request (CSR). This CSR is transmitted to a local Certificate Authority (CA) hosted on a connected PC via UART. The PC uses Python scripts and OpenSSL to sign the CSR and return the signed X.509 certificate to the device.

Once the device certificate is installed, Layer 2 generates its own key-pair based on a hash of its firmware and the CDI. A second CSR is issued and signed internally by the device using the Layer 1 private key, completing the internal certificate hierarchy. With both certificates in place, the device initiates a secure TLS connection to a cloud-based verifier. During handshake, the device presents its certificate chain and a static firmware identifier (FWID). The verifier performs chain validation, authenticates the device identity, and checks the FWID value against known-good references to complete remote attestation.

This real-time demonstration validates the full lifecycle of identity enrollment and remote verification. The TLS session was successfully established using mbedTLS, and FWID values were correctly verified by the server. The entire process is represented in **Figure 4.2**, which depicts the communication flow, cryptographic tasks, and exchanged values.

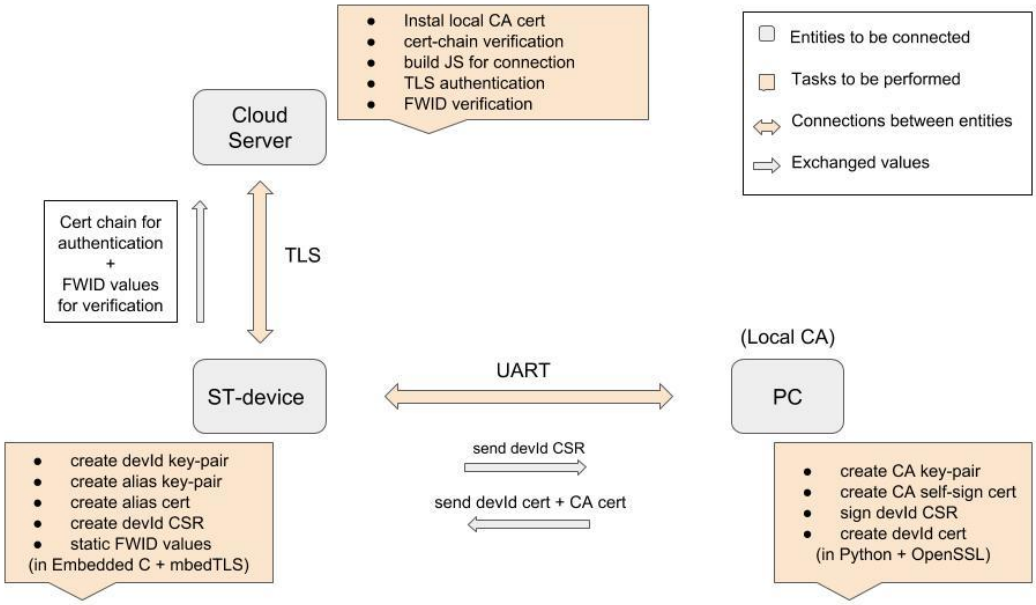

Figure 4.2 Demonstrator scenario

Adversarial Validation Scenario

To evaluate the protocol's resilience against malicious firmware manipulation, an adversarial test was conducted in which the firmware image for Layer 2 was intentionally modified. This tampering altered the measurement used in the CDI-based key derivation and led to a mismatch in the derived alias key. When the device attempted to generate a valid certificate for the altered Layer 2, the resulting signature verification failed during attestation.

The cloud verifier detected the inconsistency in the certificate chain and rejected the connection attempt, effectively preventing unauthorized access. This scenario confirms that the binding between firmware measurements and cryptographic identities—enforced through deterministic key derivation—acts as a reliable mechanism for detecting unauthorized changes. Although not exhaustive, this attack simulation reinforces the importance of measurement-based identity in preserving system integrity under adversarial conditions.

4.3 Empirical Evaluation and Discussion

To assess the feasibility of the attestation protocol under resource-constrained conditions, an empirical evaluation was conducted using the STM32L4 Nucleo-based prototype described in Section 4.2. The evaluation focuses on profiling the computational cost of the protocol's cryptographic building blocks in terms of CPU cycle counts and stack usage, using instrumentation around representative mbedTLS routines. These measurements offer insight into the practicality and efficiency of the layered protocol when implemented in a real embedded environment.

4.3.1 Measurement Limitations and Challenges

Obtaining accurate performance metrics in deeply embedded systems presents several challenges. The STM32L4 platform offers limited real-time debugging visibility, and some timing measurements are susceptible to jitter introduced by peripheral latency or interrupt-driven behavior. Moreover, the lack of dedicated cycle counters or tracing tools necessitated the insertion of instrumentation code to estimate execution time in CPU cycles.

To mitigate these limitations, the evaluation relied on internal timers and benchmarking instrumentation placed around key cryptographic routines. While this approach provides reasonable estimates of computational cost, it does not account for hardware noise, caching effects, or asynchronous events that may arise during execution. Additionally, measurements were conducted in a non-interfered, single-threaded environment, and communication delays over UART or TLS were assumed to be deterministic. Consequently, the results presented should be interpreted as lower-bound approximations rather than absolute timings.

4.3.2 Cycle Count and Stack Usage Analysis

The prototype's performance evaluation was structured around the critical building blocks of the protocol: hashing, HMAC computation, ECC key generation, and X.509 certificate creation. All operations were implemented using the mbedTLS cryptographic library.

- **SHA-256 Hashing of Firmware Layers**
  The SHA-256 measurements reflect the profiled call sequence around the mbedTLS SHA-256 routine (init/start/update/finish/free) used by the prototype. In this setup, the cost is dominated by the mbedtls_sha256_finish() routine (≈72 cycles), with minimal overhead from initialization and cleanup.

- **CDI Derivation via HMAC-SHA256**
  CDI derivation required ≈330 cycles. The most expensive component was mbedtls_md_hmac_finish() at 218 cycles, with smaller contributions from setup (20 cycles) and initialization (80 cycles). These results confirm that one-way key derivation is computationally lightweight.

- **ECC Key Generation (Device Identity)**
  The generation of the DeviceID key pair was by far the most intensive operation, taking approximately 352,400 cycles for mbedtls_ecp_gen_keypair(). Additional overhead for

random seeding, group initialization, and PEM formatting brings the total above 354,000 cycles.

- **ECC Key Generation (Layer Key)**
  Layer key derivation follows the same path and was measured at ≈353,500 cycles. The negligible difference reflects minor variability in the input entropy.

- **X.509 Certificate Creation (Alias Certificate)**
  The creation of the alias certificate was the most expensive individual operation. The total cost, including parsing, subject/issuer handling, and PEM encoding, exceeded **376,000 cycles**. Stack usage for this function peaked at **2616 bytes**, highlighting the resource burden of standard certificate handling on constrained devices.

Figure 4.3 below summarizes total cycle counts across key cryptographic operations:

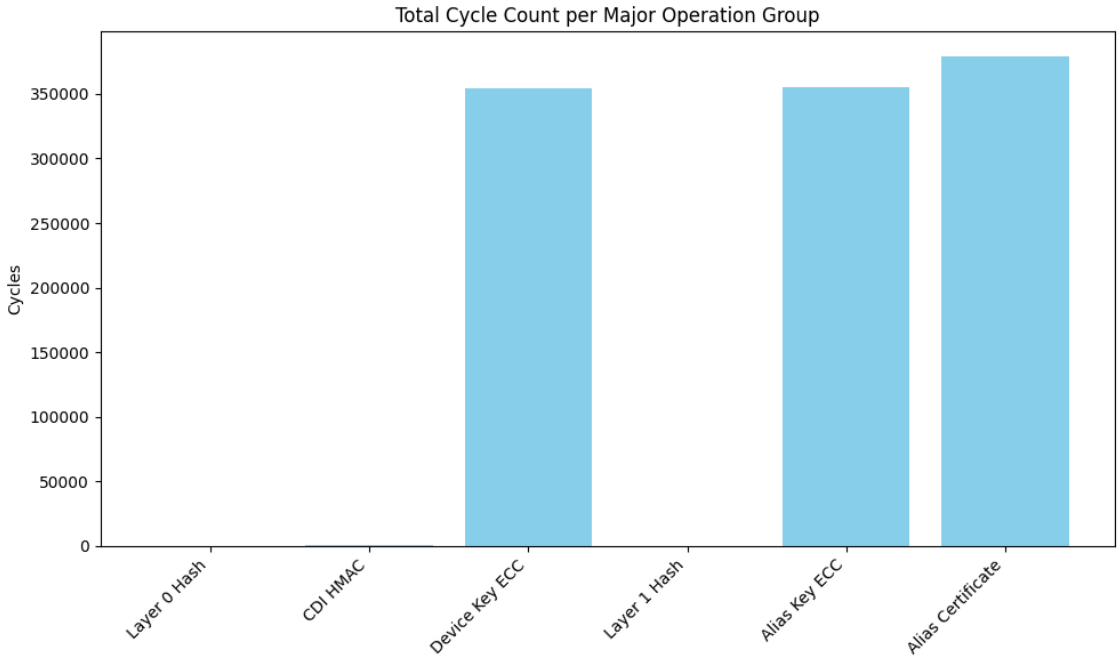

Figure 4.3 Total CPU cycle cost per cryptographic operation

The next figure 4.4 further decomposes ECC key generation steps for both DeviceID and Layer keys:

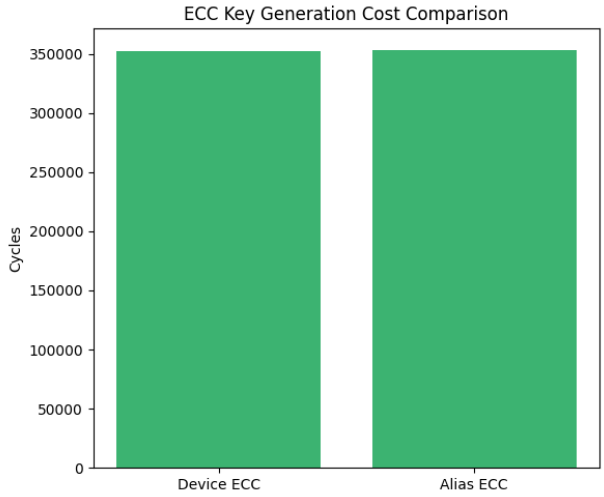

Figure 4.4 Internal breakdown of ECC key generation for DeviceID and Layer keys

Figure 4.5 shows stack usage of selected cryptographic operations, highlighting memory constraints:

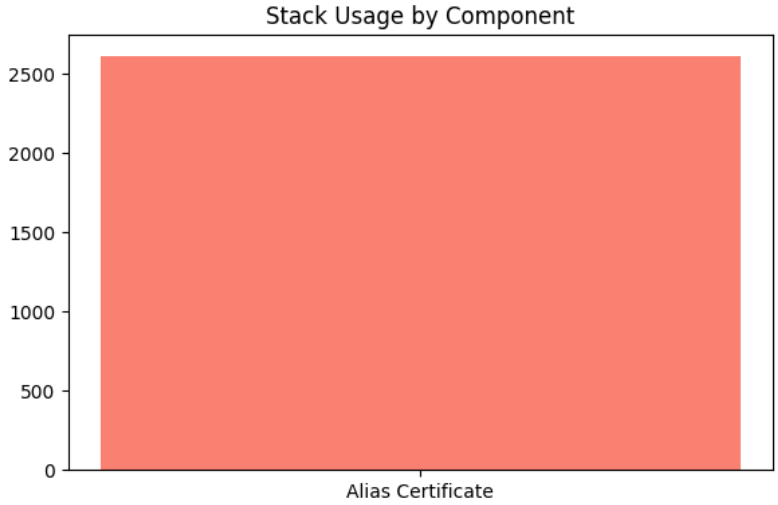

Figure 4.5 Stack memory usage for selected cryptographic operations

Figure 4.6 presents a detailed breakdown of the alias certificate creation process:

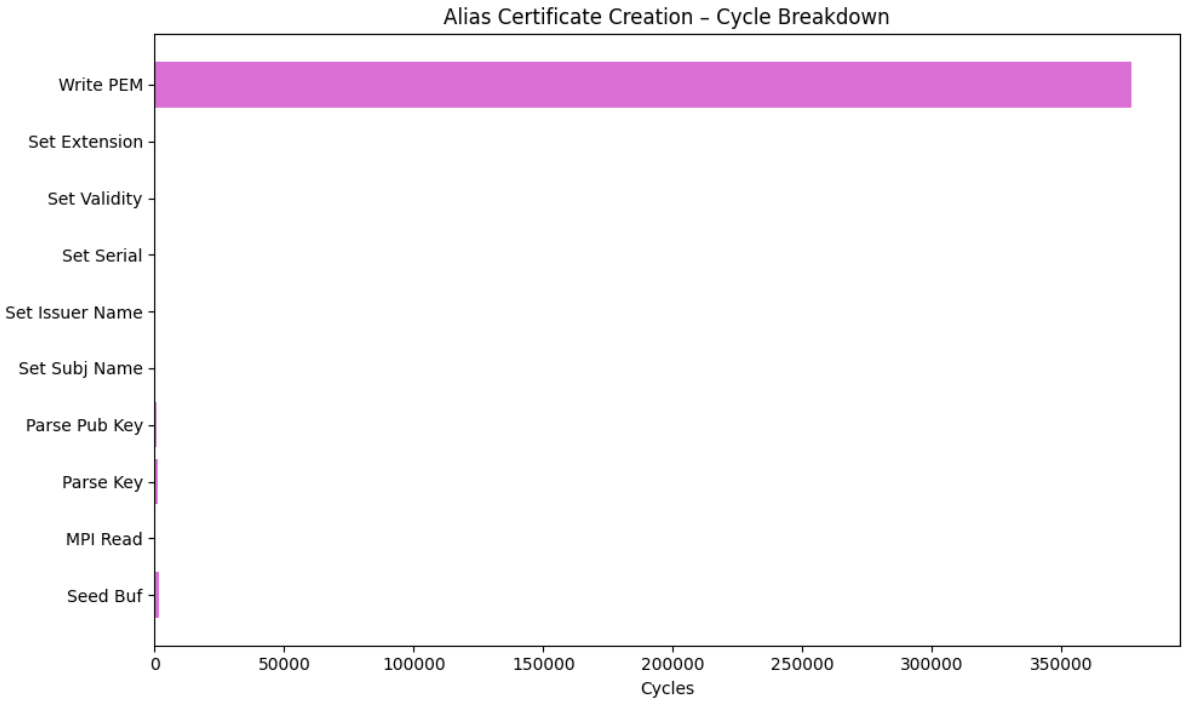

Figure 4.6 Cycle breakdown of alias certificate creation using mbedTLS routines

Figure 4.3.5 aggregates stack and memory usage for critical flows:

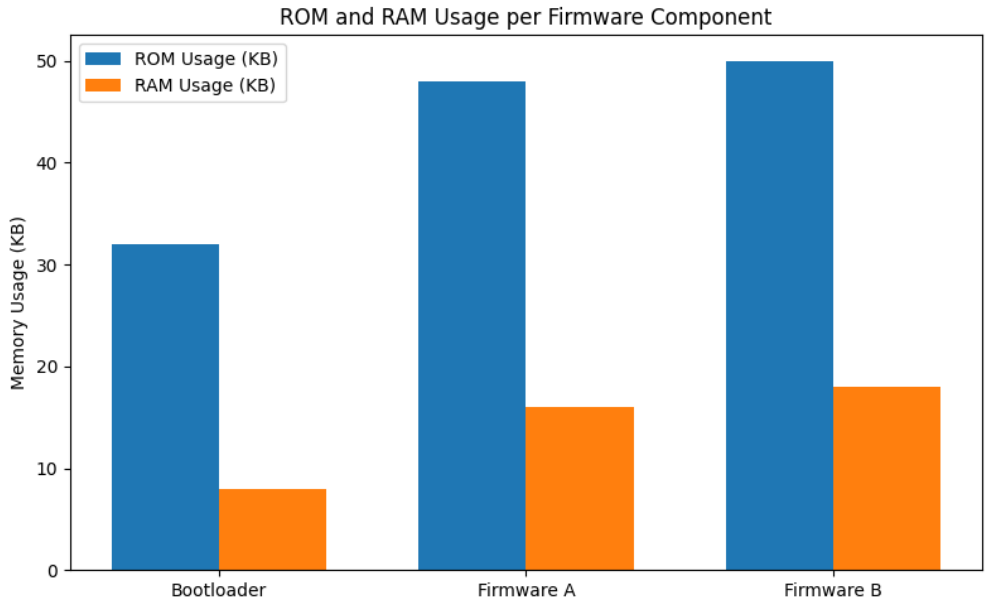

Figure 4.7 Summary of stack and code memory usage across components

4.3.3 Analysis and Discussion

These measurements confirm that while the DICE-based attestation protocol is viable on a microcontroller-class device, its feasibility hinges on judicious use of computational resources. Lightweight primitives such as hashing and HMAC can be executed efficiently, even in real-time contexts. However, public-key operations, particularly ECC key generation and certificate formatting, introduce substantial latency and memory overhead. These operations collectively consume hundreds of thousands of cycles and require over 2 KB of stack memory.

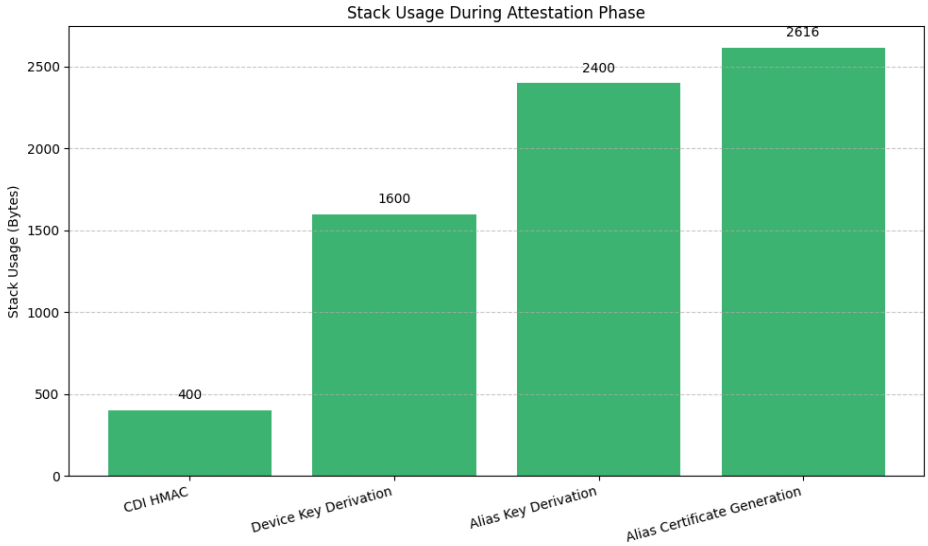

Figure 4.8 Stack usage across attestation stages (DICE-based prototype)

From a security perspective, the implementation fulfills all critical requirements established in Section 3.2.1. Unique device identities are derived from a hardware-bound secret (R1), attestation flows are cryptographically bound and verifiable (R3), and layered key isolation is successfully demonstrated

(R5). However, the reliance on host-side tooling (e.g., OpenSSL for certificate signing) introduces partial external dependencies that must be resolved in future fully self-contained implementations. Despite these constraints, the prototype demonstrates key engineering strengths:

- **Modularity:** The codebase is logically partitioned into identity derivation, attestation logic, and secure update routines.
- **Feasibility:** All essential flows, including device identity derivation and certificate generation, execute successfully within the STM32's processing envelope.
- **Robustness:** Deterministic behavior was achieved using seeded random number generation and reproducible cryptographic operations.

Some limitations remain. Debug visibility was constrained by the platform's peripheral limitations. Confidentiality restrictions also precluded direct sharing of the source code and prevented full transparency in performance instrumentation. These constraints affected both reproducibility and collaborative validation.

Future Work should prioritize tighter hardware integration, including dedicated ECC and X.509 accelerators, and transition away from host-assisted components such as Python scripts and manual OpenSSL commands. Additional steps may include dynamic attestation over the network, integration of secure boot, and real-time update negotiation.

5 Conclusions

This chapter concludes the thesis by summarizing the main outcomes of the work and reflecting on the implications of a DICE-rooted, certificate-based asymmetric attestation approach for resource-constrained IoT devices. Section 5.1 consolidates the key results across the background, protocol design, STRIDE-based security analysis, and the prototype implementation and measurements. Section 5.2 outlines focused directions for future work, including hardware-backed key protection, update hardening (anti-rollback and recovery), clearer evidence semantics and freshness, and considerations for fleet-scale deployment and deeper validation.

5.1 Thesis Summary

This thesis investigated the design and security of a lightweight asymmetric attestation protocol tailored for resource-constrained IoT devices. The primary objective was to enable verifiable device identity, firmware integrity, and secure software updates by leveraging a hardware-rooted trust anchor compliant with the Device Identifier Composition Engine (DICE) standard. By establishing this foundation, the work addressed the need for scalable and standards-aligned remote attestation mechanisms that remain feasible on distributed embedded systems.

Chapter 1 motivated the work by outlining the security challenges that arise when low-cost, network-connected embedded devices are deployed at scale. It reviewed representative threats and prior attestation techniques, and it positioned the thesis goal as the development of an end-to-end approach that can establish device trustworthiness across the device lifecycle.

Chapter 2 provided the background necessary to ground the design choices, including embedded security fundamentals, cryptographic primitives, public key infrastructure concepts, and the role of TLS in mutual authentication and secure transport. This background clarifies how certificate-based authentication and chain validation integrate naturally with the DICE-derived identity and attestation mechanisms.

Chapter 3 presented the core design of the proposed solution. Section 3.1 described the engineering approach, grounded in a V-Model, to structure development and validation activities. Section 3.2 defined the design objectives and derived six concrete system requirements (R1–R6), spanning device identity derivation, authenticated communication, attestation, and update-related constraints, while keeping hardware assumptions explicit and minimal. These requirements were organized using functional, non-functional, and security classifications, and refined through the Open Security Architecture (OSA) framework.

Based on these requirements, Section 3.3 introduced a layered protocol architecture built on DICE. The device lifecycle was captured through manufacturing and deployment phases, including provisioning and enrollment steps where certificates and firmware identities (FWIDs) are anchored into a verifiable chain of trust. The resulting functional flows describe how device identity is established from hardware-resident secrets, how attestation evidence is conveyed through the certificate chain, and how authenticated updates are supported. The protocol was then assessed using a STRIDE-based threat model, with an accompanying threat matrix, asset analysis, and explicit assumptions. Together, these elements link the identified threats to concrete mitigations and to the requirements defined in Section 3.2.

Chapter 4 translated the protocol design into an executable prototype and evaluation on a commercial microcontroller platform. It documented the selection of an STM32L4 Nucleo target, the

choice of cryptographic building blocks and libraries (notably mbedTLS for ECC, hashing, and X.509 handling), and the integration of the protocol's layered key derivation and certificate generation steps. A demonstrator validated the intended flows, and empirical measurements quantified feasibility under constrained conditions, reporting cycle counts and stack usage for SHA-256 hashing, HMAC-based CDI derivation, ECC key generation, and X.509 certificate creation. The results indicate that lightweight primitives incur modest cost, while public-key operations and certificate processing dominate runtime and memory overhead.

Overall, the thesis shows that a DICE-rooted, certificate-based asymmetric attestation protocol can be implemented on microcontroller-class devices when the trusted computing base is carefully constrained and the lifecycle flows are explicitly mapped to requirements. The resulting design provides a standards-aligned basis for device identity, attestation, and authenticated updates, and the prototype results offer practical evidence of feasibility and clear directions for engineering hardening in future work.

5.2 Future Work

The work in this thesis demonstrates that a DICE-rooted, certificate-based asymmetric attestation protocol can be realized on constrained microcontrollers with modest hardware assumptions. At the same time, the protocol and prototype were intentionally scoped to remain feasible within the thesis timeframe and the available tooling. The following directions outline next steps that would strengthen the architecture and improve its deployability without changing its core design principles.

Hardware-backed key protection.
The protocol assumes that long-term secrets are protected on-device and that debug access is disabled after provisioning. A practical extension is to make these assumptions explicit in the implementation by using hardware-enforced key isolation rather than relying primarily on software-based containment. This can be achieved by storing long-term secrets, such as UDS/CDI-derived private keys and intermediate key material, in protected storage with enforced access control. Suitable mechanisms include a secure element, a vendor key store, or microcontroller security features such as readout protection and privilege separation. In addition, debug interfaces (e.g., SWD/JTAG) should be permanently disabled or irreversibly locked on fielded devices.

Where supported, the root secret can be strengthened further by deriving the UDS intrinsically on the device. This reduces provisioning sensitivity and improves resistance to device cloning because the root secret is not injected but originates from device-specific physical properties. One concrete direction is to use an SRAM PUF-based approach, such as Intrinsic ID's solution, as the source for UDS generation. In this setting, the UDS is reconstructed from device-unique behavior at boot, aligning with the DICE expectation of a high-entropy, statistically unique secret. Removing explicit UDS injection also reduces supply-chain exposure and narrows the set of manufacturing steps that require strict protection.

Anti-rollback and update recovery.
While the protocol defines authenticated updates through signature and certificate validation, a deployable system must also prevent rollback to older, vulnerable firmware versions. This requires an explicit anti-rollback policy, for example by enforcing monotonically increasing firmware versions using a protected version counter or an equivalent mechanism. In addition, the update process should be resilient to failures. Interrupted or invalid updates must not leave the device unbootable. A practical

approach is to include a well-defined fallback image and to report update outcomes to the backend for monitoring and recovery.

Attestation evidence and validation.
The current approach represents the attested firmware state through firmware identities (FWIDs) conveyed via the certificate chain. A practical next step is to specify this evidence model more explicitly for both manufacturing and field operation by defining how FWIDs are encoded in certificates, what each FWID represents (for example, which firmware layer and measurement basis), and which verifier-side validation steps are mandatory. In addition, freshness and session binding should be enforced explicitly by incorporating verifier-provided challenges into the attestation exchange and ensuring that the presented evidence is cryptographically bound to the authenticated session context.

Deployment and scalability.
The thesis validates the protocol through a single-device flow. Future work should therefore evaluate the operational aspects that emerge when the same approach is deployed on a scale. Key aspects include validating enrollment and provisioning workflows under operational constraints, such as failure handling, re-enrollment, and clear authorization boundaries for associating devices with specific deployments. In addition, deployments in which devices authenticate to infrastructure not operated by the manufacturer require careful trust-domain separation, for example to support multi-tenant operation without weakening verification guarantees. Addressing these topics is primarily an engineering effort focused on defining lifecycle states, interfaces, and operational procedures, rather than modifying the cryptographic core of the protocol.

Validation and evaluation
While Chapter 3 provides a structured STRIDE-based analysis with explicit assumptions and adversary capabilities, future work should strengthen assurance through targeted empirical validation aligned with the identified attacker classes. This includes practical testing that exercises debug misuse attempts, firmware and update-path tampering, and leakage-oriented adversaries informed by timing, power, or fault behavior. In addition, the evaluation can be expanded beyond cycle counts and memory footprint by quantifying energy consumption and end-to-end latency during authentication and attestation. Finally, the security guarantees can be expressed more explicitly as properties tied to the stated assumptions, improving both confidence in the results and comparability with related designs.

Bibliography

[1] Gartner. (2019). Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020. [Online] Available at: https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io [Accessed 27 Feb. 2020].

[2] The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025, According to a New IDC Forecast. (2019). Retrieved 27 February 2020, from https://www.idc.com/getdoc.jsp?containerId=prUS45213219.

[3] McKinsey & Company. (n.d.). Unlocking the potential of the Internet of Things. [online] Available at: https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world [Accessed 28 Feb. 2020].

[4] Langner, R. (2011). Stuxnet: Dissecting a cyberwarfare weapon. IEEE Security & Privacy, 9(3), 49-51.

[5] IoT: number of connected devices worldwide 2012-2025 | Statista. (2020). Retrieved 27 February 2020, from https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[6] Cope, P., Campbell, J., & Hayajneh, T. (2017, January). An investigation of Bluetooth security vulnerabilities. In 2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC) (pp. 1-7). IEEE.

[7] Serpanos, D. N., & Voyiatzis, A. G. (2013). Security challenges in embedded systems. ACM Transactions on embedded computing systems (TECS), 12(1s), 1-10.

[8] O'Donnell, L. (2018). IoT Security Concerns Peaking – With No End In Sight. Retrieved 9 March 2020, from https://threatpost.com/iot-security-concerns-peaking-with-no-end-in-sight/131308/.

[9] Román-Castro, R., López, J., & Gritzalis, S. (2018). Evolution and trends in iot security. Computer, 51(7), 16-25.

[10] Greenberg, A., Barrett, B., & Newman, L. (2015). Hackers Remotely Kill a Jeep on the Highway—With Me in It. Retrieved 9 March 2020, from https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[11] Kaspersky Lab. (2015). Damage Control: The Cost of Security Breaches IT Security Risks Special report Series. [online] Available at: https://media.kaspersky.com/pdf/it-risks-survey-report-cost-of-security-breaches.pdf [Accessed 7 Mar. 2020].

[12] Micro, T., 2018. SAMSAM Ransomware Hits US Hospital, Management Pays $55K Ransom - Security News - Trend Micro MY. [online] Trendmicro.com. Available at: <https://www.trendmicro.com/vinfo/my/security/news/cyber-attacks/samsam-ransomware-hits-us-hospital-management-pays-55k-ransom> [Accessed 10 March 2020].

[13] G. U. of Technology. Meltdown and Spectre - Vulnerabilities in modern computers leak passwords and sensitive data. Retrieved 10 March 2020, from https://meltdownattack.com

[14] D. N. Migwi and R. Romaniuk, “Trusted Computing in the Internet of Things: Securing the Edge through Hardware-Enforced Trust,” *IEEE Trans. Emerging Topics in Computing*, vol. XX, no. YY, pp. ZZ–ZZ, May 2025.

[15] B. Madabhushi, C. S. Mummidi, S. Kundu, and D. Holcomb, “Resurrection Attack: Defeating Xilinx MPU’s Memory Protection,” arXiv:2405.13933, May 2024.

[16] C. Heinz and A. Koch, “DD-MPU: Dynamic and Distributed Memory Protection Unit for Embedded System-on-Chips,” in *Proc. SAMOS ’23*, 2023.

[17] T. Hoang *et al.*, “TrustLite: A Security Architecture for Tiny Embedded Devices,” in *Proc. ESWEEK*, 2013, pp. 23–28.

[18] A. S. Banks, M. Kisiel, and P. Korsholm, “Remote attestation: A literature review,” arXiv preprint arXiv:2105.02466, May 2021.
[19] Seshadri, A., Luk, M., Shi, E., Perrig, A., Van Doorn, L., & Khosla, P. (2005, October). Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In Proceedings of the twentieth ACM symposium on Operating systems principles (pp. 1-16).
[20] Seshadri, A., Luk, M., & Perrig, A. (2008, June). SAKE: Software attestation for key establishment in sensor networks. In International Conference on Distributed Computing in Sensor Systems (pp. 372-385). Springer, Berlin, Heidelberg.
[21] Yang, Y., Wang, X., Zhu, S., & Cao, G. (2007, October). Distributed software-based attestation for node compromise detection in sensor networks. In 2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007) (pp. 219-230). IEEE.
[22] Li, Y., McCune, J. M., & Perrig, A. (2011, October). VIPER: verifying the integrity of PERipherals' firmware. In Proceedings of the 18th ACM conference on Computer and communications security (pp. 3-16).
[23] Kovah, X., Kallenberg, C., Weathers, C., Herzog, A., Albin, M., & Butterworth, J. (2012, May). New results for timing-based attestation. In 2012 IEEE Symposium on Security and Privacy (pp. 239-253). IEEE.
[24] Jakobsson, M., & Johansson, K. A. (2010). Retroactive Detection of Malware with Applications to Mobile Platforms. HotSec, 10, 1-13.
[25] Strackx, R., Piessens, F., & Preneel, B. (2010, September). Efficient isolation of trusted subsystems in embedded systems. In International Conference on Security and Privacy in Communication Systems (pp. 344-361). Springer, Berlin, Heidelberg.
[26] Arbaugh, W. A., Farber, D. J., & Smith, J. M. (1997, May). A secure and reliable bootstrap architecture. In Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097) (pp. 65-71). IEEE.
[27] Level, T. M. S. (2). Version 1.2, Revision 103 (Trusted Computing Group).
[28] Pearson, S., Mont, M. C., & Crane, S. (2005, May). Persistent and dynamic trust: analysis and the related impact of trusted platforms. In International Conference on Trust Management (pp. 355-363). Springer, Berlin, Heidelberg.
[29] Kil, C., Sezer, E. C., Azab, A. M., Ning, P., & Zhang, X. (2009, June). Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In 2009 IEEE/IFIP International Conference on Dependable Systems & Networks (pp. 115-124). IEEE.
[30] Sailer, R., Zhang, X., Jaeger, T., & Van Doorn, L. (2004, August). Design and Implementation of a TCG-based Integrity Measurement Architecture. In USENIX Security symposium (Vol. 13, No. 2004, pp. 223-238).
[31] ARM CORPORATION. Building a secure system using TrustZone technology. Publication number: PRD29- GENC-009492C.
[32] Shacham, H. (2007, October). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM conference on Computer and communications security (pp. 552-561).
[33] G. Klein *et al*., “seL4: Formal verification of an OS kernel,” in *Proc. 22nd ACM SIGOPS Symp. Operating Systems Principles (SOSP)*, Big Sky, MT, USA, Oct. 2009, pp. 207–220.
[34] INTEL CORPORATION. Intel Trusted Execution Technology (Intel TXT) – Software Development Guide, December 2009. Document Number: 315168-006.
[35] Virtualization, A. (2005). Secure virtual machine architecture reference manual. AMD Publication, 33047.

[36] Costan, V., Sarmenta, L. F., Van Dijk, M., & Devadas, S. (2008, September). The trusted execution module: Commodity general-purpose trusted computing. In International Conference on Smart Card Research and Advanced Applications (pp. 133-148). Springer, Berlin, Heidelberg.
[37] Kostiainen, K., Dmitrienko, A., Ekberg, J. E., Sadeghi, A. R., & Asokan, N. (2010, June). Key attestation from trusted execution environments. In International Conference on Trust and Trustworthy Computing (pp. 30-46). Springer, Berlin, Heidelberg.
[38] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., & Isozaki, H. (2007). An execution infrastructure for TCB minimization.
[39] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., & Perrig, A. (2010, May). TrustVisor: Efficient TCB reduction and attestation. In 2010 IEEE Symposium on Security and Privacy (pp. 143-158). IEEE.
[40] Nie, C. (2007). Dynamic root of trust in trusted computing. In TKK T1105290 Seminar on Network Security.
[41] Parno, B., McCune, J. M., & Perrig, A. (2010, May). Bootstrapping trust in commodity computers. In 2010 IEEE Symposium on Security and Privacy (pp. 414-429). IEEE.
[42] Strackx, R., Piessens, F., & Preneel, B. (2010, September). Efficient isolation of trusted subsystems in embedded systems. In International Conference on Security and Privacy in Communication Systems (pp. 344-361). Springer, Berlin, Heidelberg.
[43] J. Noorman *et al*., “Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base,” in *Proc. 22nd USENIX Security Symp. (USENIX Security ’13)*, Washington, D.C., USA, Aug. 2013, pp. 479–498.
[44] Eldefrawy, K., Tsudik, G., Francillon, A., & Perito, D. (2012, February). Smart: secure and minimal architecture for (establishing dynamic) root of trust. In Ndss (Vol. 12, pp. 1-15).
[45] Koeberl, P., Schulz, S., Sadeghi, A. R., & Varadharajan, V. (2014, April). TrustLite: A security architecture for tiny embedded devices. In Proceedings of the Ninth European Conference on Computer Systems (pp. 1-14).
[46] Eldefrawy, K., Rattanavipanon, N., & Tsudik, G. (2017, July). HYDRA: hybrid design for remote attestation (using a formally verified microkernel). In Proceedings of the 10th ACM Conference on Security and Privacy in wireless and Mobile Networks (pp. 99-110).
[47] Brasser, F., Rasmussen, K. B., Sadeghi, A. R., & Tsudik, G. (2016, June). Remote attestation for low-end embedded devices: the prover's perspective. In 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC) (pp. 1-6). IEEE.
[48] Schulz, S., Schaller, A., Kohnhäuser, F., & Katzenbeisser, S. (2017, September). Boot attestation: Secure remote reporting with off-the-shelf iot sensors. In European Symposium on Research in Computer Security (pp. 437-455). Springer, Cham.
[49] England, P., Marochko, A., Mattoon, D., Spiger, R., Thom, S., & Wooten, D. (2016). Riot-a foundation for trust in the internet of things. Microsoft Research.
[50] DICE | Trusted Computing Group. (2016). Retrieved 15 April 2020, from https://trustedcomputinggroup.org/work-groups/ dice-architectures/
[51] Jäger, L., Petri, R., & Fuchs, A. (2017, August). Rolling dice: Lightweight remote attestation for cots iot hardware. In Proceedings of the 12th International Conference on Availability, Reliability and Security (pp. 1-8).
[52] Hristozov, S., Heyszl, J., Wagner, S., & Sigl, G. (2018, February). Practical runtime attestation for tiny iot devices. In Proceedings of the 2018 Workshop on Decentralized IoT Security and Standards, San Diego, CA, USA (Vol. 18).

[53] Heath, S. (2002). Embedded systems design. Elsevier.

[54] Vahid, F., & Givargis, T. (1999). Embedded system design: A unified hardware/software approach. Department of Computer Science and Engineering University of California.
[55] Marwedel, P. (2017). Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things. Springer.
[56] Cai, L. Z., & Zuhairi, M. F. (2017, September). Security challenges for open embedded systems. In 2017 International Conference on Engineering Technology and Technopreneurship (ICE2T) (pp. 1-6). IEEE.
[57] Bahga, A., & Madisetti, V. (2014). Internet of Things: A hands-on approach. Vpt.
[58] Schatz, D., Bashroush, R., & Wall, J. (2017). Towards a more representative definition of cyber security. Journal of Digital Forensics, Security and Law, 12(2), 53-74.
[59] Whitman, M. E., & Mattord, H. J. (2011). Principles of information security. Cengage Learning.
[60] Pipkin, D. L. (2000). Information security: protecting the global enterprise. Prentice-Hall, Inc.
[61] Cherdantseva, Y., & Hilton, J. (2015). Information security and information assurance: discussion about the meaning, scope, and goals. In Standards and Standardization: Concepts, Methodologies, Tools, and Applications (pp. 1204-1235). IGI Global.
[62] Newsome, B. (2013). A practical introduction to security and risk management. SAGE Publications.
[63] Organization for Economic Cooperation and Development (OECD). (2002). Guidelines for the Security of Information Systems and Networks: Towards a Culture of Security.
[64] Pender-Bey, G. (2013). The Parkerian hexad: The CIA triad model expanded. Master of Science in Information Security at Lewis University, 1-31.
[65] Stoneburner, G., Hayden, C., & Feringa, A. (2001). Engineering principles for information technology security (a baseline for achieving security). Booz-Allen and Hamilton Inc Mclean VA.
[66] Open Group. (2017). Open Information Security Management Maturity Model (O-ISM3), Version 2.0.
[67] Fenrich, K. (2008). Securing your control system: the" CIA triad" is a widely used benchmark for evaluating information system security effectiveness. Power Engineering, 112(2), 44-49.
[68] Andress, J. (2014). The basics of information security: understanding the fundamentals of InfoSec in theory and practice. Syngress.
[69] Beckers, K., Heisel, M., & Hatebur, D. (2015). Pattern and Security Requirements. Pattern Secur. Requir. Eng. Establ. Secur. Stand, 1-474.
[70] Thomas, R. (2016). Espionage and Secrecy (Routledge Revivals): The Official Secrets Acts 1911-1989 of the United Kingdom. Routledge.
[71] Boritz, J. E. (2005). IS practitioners' views on core concepts of information integrity. International Journal of Accounting Information Systems, 6(4), 260-279.
[72] Loukas, G., & Öke, G. (2010). Protection against denial of service attacks: A survey. The Computer Journal, 53(7), 1020-1037.
[73] Samonas, S., & Coss, D. (2014). THE CIA STRIKES BACK: REDEFINING CONFIDENTIALITY, INTEGRITY AND AVAILABILITY IN SECURITY. Journal of Information System Security, 10(3).
[74] What is Root of Trust? | Thales. Retrieved from https://cpl.thalesgroup.com/faq/hardware-security-modules/what-root-trust
[75] Flexible Key Provisioning with SRAM PUF white paper landing page - Intrinsic ID | Home of PUF Technology. Retrieved from https://www.intrinsic-id.com/resources/white-papers/white-paper-flexible-key-provisioning-sram-puf/
[76] Biometric Access Control Systems: Everything You Should Know. Retrieved 16 October 2020, from https://keyo.co/biometric-news/biometric-access-control-systems-101-everything-you-should-know
[77] Bai, Y. (2016). ARM® Memory Protection Unit (MPU).

[78] Arbaugh, W. A., Farber, D. J., & Smith, J. M. (1997, May). A secure and reliable bootstrap architecture. In Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097) (pp. 65-71). IEEE.
[79] G. Coker *et al*., "Principles of remote attestation," *Int. J. Inf. Security*, vol. 10, no. 2, pp. 63–81, 2011.
[80] Costan, V., & Devadas, S. (2016). Intel SGX Explained. IACR Cryptol. ePrint Arch., 2016(86), 1-118.
[81] Smart, N. P. (2016). Cryptography made simple. Springer.
[82] Liddell, H. G., & Scott, R. (1897). A greek-english lexicon. New York: American Book Company.
[83] Kahn, D. (1996). The Codebreakers: The comprehensive history of secret communication from ancient times to the internet. Simon and Schuster.
[84] Dulaney, E., & Easttom, C. (2017). CompTIA Security+ Study Guide: Exam SY0-501. John Wiley & Sons.
[85] Goldreich, O. (2007). Foundations of cryptography: volume 1, basic tools. Cambridge university press.
[86] Hayler, W. B., & Sebag-Montefiore, H. (2001). Enigma: The Battle for the Code. Naval War College Review, 54(4), 26
[87] Cryptology. (2020). Retrieved 30 November 2020, from https://www.britannica.com/topic/cryptology
[88] Singh, S., 2000. The Code Book: The Science of Secrecy from Ancient Egypt To Quantum Cryptography. New York: Anchor Books.
[89] Petitcolas, F. A., Anderson, R. J., & Kuhn, M. G. (1999). Information hiding-a survey. Proceedings of the IEEE, 87(7), 1062-1078.
[90] Suetonius Tranquillus, G. and Edwards, C., 2008. Lives of The Caesars. Oxford [etc.]: Oxford University Press, p.28.
[91] Leaman, O. (Ed.). (2015). The Biographical Encyclopedia of Islamic Philosophy. Bloomsbury Publishing.
[92] Shannon, C. E. (1949). Communication theory of security. Bell System Technical Journal, 28, 656-715.
[93] Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A. (2018). Handbook of applied cryptography. CRC press.
[94] Diffie, W., & Hellman, M. (1976). New directions in cryptography. IEEE transactions on Information Theory, 22(6), 644-654.
[95] Rueppel, R. A. (2012). Analysis and design of stream ciphers. Springer Science & Business Media.
[96] Rubin, F. (1996). One-time pad cryptography. Cryptologia, 20(4), 359-364.
[97] Chakraborty, D., & Henríquez, F. R. (2009). Block cipher modes of operation from a hardware implementation perspective. In Cryptographic Engineering (pp. 321-363). Springer, Boston, MA.
[98] Coppersmith, D. (1994). The Data Encryption Standard (DES) and its strength against attacks. IBM journal of research and development, 38(3), 243-250.
[99] Karn, P., Metzger, P., & Simpson, W. (1995). The ESP triple DES transform. RFC1851.
[100] Rijmen, V., & Daemen, J. (2001). Advanced encryption standard. Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology, 19-22.
[101] Rivest, R. L. (1994, December). The RC5 encryption algorithm. In International Workshop on Fast Software Encryption (pp. 86-96). Springer, Berlin, Heidelberg.
[102] Diffie, W., & Hellman, M. E. (1976, June). Multiuser cryptographic techniques. In Proceedings of the June 7-10, 1976, national computer conference and exposition (pp. 109-112).

[103] Diffie, W., & Hellman, M. (1976). New directions in cryptography. IEEE transactions on Information Theory, 22(6), 644-654.
[104] Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), 120-126.
[105] Lopez, J., & Dahab, R. (2000). An overview of elliptic curve cryptography.
[106] Haakegaard, R., & Lang, J. (2015). The elliptic curve diffie-hellman (ecdh). Online at https://koclab. cs. ucsb. edu/teaching/ecc/project/2015Projects/Haakegaard+ Lang. pdf.
[107] Bruce, S. (1996). Applied cryptography. 2nd John Wiley and Sons, Inc.
[108] Katz, J. (2010). Digital signatures. Springer Science & Business Media.
[109] Rogaway, P., & Shrimpton, T. (2004, February). Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In International workshop on fast software encryption (pp. 371-388). Springer, Berlin, Heidelberg.
[110] Rivest, R. L. (1990, August). The MD4 message digest algorithm. In Conference on the Theory and Application of Cryptography (pp. 303-311). Springer, Berlin, Heidelberg.
[111] Rivest, R. (1992). RFC1321: The MD5 message-digest algorithm.
[112] Burrows, J. H. (1995). Secure hash standard. Department of Commerce Washington DC.
[113] Boutin, C. (2012). NIST selects winner of Secure Hash Algorithm (SHA-3) Competition. Press release., October 2.
[114] Bellare, M., Canetti, R., & Krawczyk, H. (1996, August). Keying hash functions for message authentication. In Annual international cryptology conference (pp. 1-15). Springer, Berlin, Heidelberg.
2.3 Public Key Infrastructure
[115] Buchmann, J. A., Karatsiolis, E., & Wiesmaier, A. (2013). Introduction to public key infrastructures. Springer Science & Business Media.
[116] Housley, R., Ford, W., Polk, W., & Solo, D. (1999). RFC2459: Internet X. 509 public key infrastructure certificate and CRL profile.
[117] What Are CA Certificates?: Public Key; Security Services. Retrieved 15 November 2020, from https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc778623(v=ws.10)?redirectedfrom=MSDN
[118] DigiCert FAQ | DigiCert. Retrieved 15 November 2020, from https://www.websecurity.digicert.com/en/us/digicert-and-symantec-faq
[119] Retrieved 12 November 2020, from http://open.nlnetlabs.nl/downloads/publications/CSI-report.pdf
[120] M. Sommerhalder, *Hardware Security Module*, in *Trends in Data Protection and Encryption Technologies*, Springer Nature Switzerland, 2023.
[121] Trust Model implementation by PKI. Retrieved 9 November 2020, from https://medium.com/@meghdadshamsaei/trust-model-implementation-by-pki-7cddcdb72513
[122] Retrieved 10 November 2020, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.362.7257&rep=rep1&type=pdf
[123] Tanwar, S., & Prema, K. V. Trust Models in Public Key Infrastructure.
[124] C. J. Mitchell, "Transport Layer Security (TLS)," in Encyclopedia of Cryptography and Security, 2nd ed., H. C. A. van Tilborg and S. Jajodia, Eds. Boston, MA: Springer, 2011, pp. 1330–1333. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5_520
[125] E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.3, RFC 8446, Aug. 2018. [Online]. Available: https://doi.org/10.17487/RFC8446
[126] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” RFC 5246, Aug. 2008.

[127] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” RFC 8446, Aug. 2018.
[128] CISA, “SSL 3.0 Protocol Vulnerability and POODLE Attack,” Cybersecurity Advisory, Oct. 17, 2014. [Online]. Available: (link elided)
[129] “Heartbleed Bug,” heartbleed.com, 2014. [Online]. Available: (link elided)
[130] Trustedcomputinggroup.org. 2005. Design, Implementation, And Usage Principles Version 2.0. [online] Available at: <https://www.trustedcomputinggroup.org/wp-content/uploads/Best_Practices_Principles_Document_V2_0.pdf> [Accessed 22 November 2020].
[131] Trusted Platform Module 2.0: A Brief Introduction | Trusted Computing Group. (2019). Retrieved 22 November 2020, from https://trustedcomputinggroup.org/resource/trusted-platform-module-2-0-a-brief-introduction/ []
[132] Ronald Aigner, Nicolai Kuntze, David Wooten, and Graeme Proudler. 2016. Trusted Platform Architecture - Hardware Requirements for a Device Identifier Composition Engine. Technical Report. Trusted Computing Group. https://trustedcomputinggroup.org/wp-content/uploads/Hardware-Requirements-for-Device-Identifier-Composition-Engine-r78_For-Publication.pdf.
[133] ARM Limited, *Building a Secure System using TrustZone Technology*, White Paper PRD29-GENC-009492C, 2009.
[134] Orlikowski, W. J., & Iacono, C. S. (2001). Research commentary: desperately seeking the “IT” in IT research. A call to theorizing the IT artifact. Information Systems Research, 12(2), 121–134.
[135] Offermann, P., Blom, S., Schönherr, M., & Bub, U. (2010). Artifact types in information systems design science – a literature review. In R. Winter, J. L. Zhao & S. Aier (Eds.), Global Perspectives on Design Science Research (Vol. 6105, pp. 77-92): Springer Berlin Heidelberg.
[136] Peffers, K., Tunanen, T., Rothenberger, M. A., & Chatterjee, S. (2008). A design science research methodology for information systems research. Journal of Management Information Systems, 24(3), 45-77.
[137] Ruparelia, N. B. (2010). Software development lifecycle models. ACM SIGSOFT Software Engineering Notes, 35(3), 8-13.
[138] Royce, Winston (1970). Managing the Development of Large Systems, Proceedings of IEEE WESCON, 26 August: 1-9.
[139] Military Standard Defense System Software Development
[140] Forsberg, K., Mooz, H. (1998). System Engineering for Faster, Cheaper, Better. Centre of Systems Management. Archived from the original on April 20, 2003.
[141] IEEE. IEEE Guide--Adoption of the Project Management Institute (PMI) Standard A Guide to the Project Management Body of Knowledge (PMBOK Guide) -Fourth Edition.
[142] Forsberg, K., Mooz, H. The Relationship of System Engineering to the Project Cycle, in Proceedings of the First Annual Symposium of National Council on System Engineering, October 1991: 57–65.
[143] German Directive 250, Software Development Standard for the German Federal Armed Forces, V-Model, Software Lifecycle Process Model, August 1992.
[144] Systems Engineering for Intelligent Transportation Systems. US Dept. of Transportation. p. 10.
[145] Brennan, K. (2009). A guide to the Business analysis body of knowledge (BABOK guide). Version 2.0. International Institute of business analysis.
[146] Hay, David C. (2003). Requirements Analysis: From Business Views to Architecture (1st ed.). Upper Saddle River, NJ: Prentice Hall.
[147] McConnell, Steve (1996). Rapid Development: Taming Wild Software Schedules (1st ed.). Redmond, WA: Microsoft Press. ISBN 1-55615-900-5.
[148] IEEE Standard Glossary of Software Engineering Terminology," in IEEE Std 610.12-1990 , vol., no., pp.1-84, 31 Dec. 1990.

[149] Fulton R, Vandermolen R (2017). "Chapter 4: Requirements - Writing Requirements". Airborne Electronic Hardware Design Assurance: A Practitioner's Guide to RTCA/DO-254. CRC Press. pp. 89–93.
[150] Chung, L., Nixon, B. A., Yu, E., & Mylopoulos, J. (2012). Non-functional requirements in software engineering (Vol. 5). Springer Science & Business Media.
[151] Loucopoulos, P., & Karakostas, V. (1995). System requirements engineering. McGraw-Hill, Inc..
[152] Stellman, A., & Greene, J. (2005). Applied software project management. " O'Reilly Media, Inc.".
[153] Opensecurityarchitecture.org. (2020). [online] Available at: https://www.opensecurityarchitecture.org/cms/definitions/it_security_requirements [Accessed 16 Feb. 2020].
[154] Ali, M. S., Babar, M. A., Chen, L., & Stol, K. J. (2010). A systematic review of comparative evidence of aspect-oriented programming. Information and software Technology, 52(9), 871-887.
[155] Chen, L., Babar, M. A., & Zhang, H. (2010, April). Towards an evidence-based understanding of electronic data sources. In 14th International Conference on Evaluation and Assessment in Software Engineering (EASE) (pp. 1-4).
[156] C. W. Schmitt, M. Kupreev, and S. Nürnberger, “Securing the IoT supply chain: On the effectiveness of secure elements and hardware-based roots of trust,” 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), Genoa, Italy, 2020, pp. 100–109.
[157] H. Tschofenig, T. Fossati, and M. Richardson, “TLS/DTLS 1.3 Profiles for the Internet of Things,” IETF Internet-Draft draft-ietf-uta-tls13-iot-profile-14, May 2025.
[158] Microsoft, “The STRIDE Threat Model,” MSDN, 2002. [Online]. Available: https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx
[159] G. Martin and G. Smith, “High-Level Design: A Survey of Current Practices,” in Proc. IEEE/ACM Int. Conf. Computer-Aided Design, Nov. 2009, pp. 303–310.
[160] STMicroelectronics, STM32L4x6 advanced ARM®-based 32-bit MCUs: ultra-low-power with FPU, DSP, and analog features, Reference Manual, RM0351, Rev 7, Mar. 2024. [Online]. Available: https://www.st.com/resource/en/reference_manual/dm00151940-stm32l4x6-advanced-armbased-32bit-mcus-stmicroelectronics.pdf
[161] "micro-ecc: A small ECDH and ECDSA library," [Online]. Available: https://github.com/kmackay/micro-ecc
[162] "wolfSSL Embedded SSL/TLS Library," [Online]. Available: https://www.wolfssl.com
[163] "mbedTLS Documentation," Arm Ltd., [Online]. Available: https://github.com/Mbed-TLS/mbedtls
[164] M. Bellare, R. Canetti, and H. Krawczyk, “Keying Hash Functions for Message Authentication,” in Advances in Cryptology — CRYPTO’ 96, vol. 1109, N. Koblitz, Ed. Berlin, Heidelberg: Springer, 1996, pp. 1–15.
[165] National Institute of Standards and Technology, The Keyed-Hash Message Authentication Code (HMAC), FIPS PUB 198-1, Jul. 2008. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.198-1.pdf
[166] M. Bellare, J. Kilian, and P. Rogaway, “The Security of the Cipher Block Chaining Message Authentication Code,” in Advances in Cryptology — CRYPTO’ 94, vol. 839, Y. Desmedt, Ed. Berlin, Heidelberg: Springer, 1994, pp. 341–358.