POLITECNICO DI TORINO
Repository ISTITUZIONALE

An Experimental Approach for Establishing TLS Trusted Channels with Device Identification in IoT Environments

Original
An Experimental Approach for Establishing TLS Trusted Channels with Device Identification in IoT Environments / Berbecaru, D., Sisinni, S., Virga, F., Lioy, A.. - 4198:(2026). (ITASEC & SERICS 2026 Joint National Conference on Cybersecurity 2026 Cagliari (ITA) 9-13 February 2026).

Availability:
This version is available at: 11583/3006577 since: 2026-01-15T17:13:46Z

Publisher:
CEUR-WS

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

13 September 2026

An Experimental Approach for Establishing TLS Trusted Channels with Device Identification in IoT Environments

Diana Gratiela Berbecaru[1,*,†], Silvia Sisinni[1,†], Francesco Virga[1,†] and Antonio Lioy[1,†]

[1]Politecnico di Torino, Dipartimento di Automatica e Informatica (DAUIN), Torino 10129, Italy

Abstract

Internet of Things technologies continue to expand rapidly, and one of the most critical challenges is reliable device identification and authentication within the IoT networks. Although Trusted Platform Modules (TPMs) are commonly used to establish a hardware root of trust, their integration into IoT and embedded devices is often unfeasible due to limitations in physical space, power consumption, and computational capabilities. This work proposes an alternative approach for secure device identification that does not rely on dedicated TPM hardware. Instead, it leverages DICE (Device Identifier Composition Engine), a framework proposed to deliver cryptographically strong device identity used along with lightweight cryptographic mechanisms and software-based trust anchors. As root of trust, the system integrates the MARS (Measurement and Attestation RootS) framework, a TCG architectural framework specifically designed for constrained devices.

We develop a prototype and an experimental implementation that simulates a secure TLS-enabled MQTT (Message Queuing Telemetry Transport) communication channel between an IoT device and its broker. In this use case, device identification is performed through the establishment of a TLS-PSK (Pre-Shared Key) channel, where the pre-shared identity key serves both as an attestation and identification key, enabling thus an implicit attestation process during the secure session setup. Through experimental tests, we measure the device boot time, the key derivation time, and the time required to complete the TLS handshake, changing different parameters. Our solution demonstrates that small and resource-limited IoT devices can achieve secure device identity, authentication and attestation within a reasonable time.

Keywords

Measurement and Attestation RootS (MARS), Device Identifier Composition Engine (DICE), Trusted Communication Channels, Secure Channel Establishment, Root of Trust, IoT Security, Remote Attestation, Implicit Attestation, TLS-PSK, Symmetric Cryptography, Trusted Boot, Measured Boot, MQTT Security, Lightweight Authentication, Pre-Shared Key Authentication, HMAC-based Key Derivation Function

CEUR-WS.org/Vol-4198/paper20.pdf

1. Introduction

The Internet of Things (IoT) has evolved from a conceptual paradigm into a widespread technology ecosystem linking billions of devices globally. Recent surveys depict the IoT as a vast network of interconnected sensors, actuators, and embedded devices that communicate and collaborate autonomously to achieve shared objectives [1, 2, 3]. These systems are increasingly used in many contexts, including smart cities, industrial automation, healthcare, precision agriculture, and logistics [3, 4].

Architecturally, IoT deployments have expanded beyond the traditional three-layer model (perception, network, and application) toward more distributed paradigms that include cloud, edge, and fog computing [1]. This change tackles the requirements for localized analytics near data sources, bandwidth optimization, and low-latency processing. Applications that are responsive and scalable are made possible by such hybrid architectures, but they also add new challenges to data governance and device management. The convergence of enabling technologies such as artificial intelligence, machine learning, and 5G/6G networks is a defining feature of the IoT landscape. These integrations enable intelligent data-driven decisions, predictive maintenance, and adaptive system behavior. As connected devices

Joint National Conference on Cybersecurity (ITASEC & SERICS 2026), February 09-13, 2026, Cagliari, IT

*Corresponding author.

†These authors contributed equally.

diana.berbecaru@polito.it (D. G. Berbecaru); silvia.sisinni@polito.it (S. Sisinni); francesco.virga@studenti.polito.it (F. Virga); antonio.lioy@polito.it (A. Lioy)

0000-0003-1930-9473 (D. G. Berbecaru); 0000-0003-1870-6303 (S. Sisinni); 0000-0002-5669-9338 (A. Lioy)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-04-23

proliferate, challenges in interoperability, standardization, energy efficiency, and security remain active areas of research [4, 5, 6, 7].

The Trusted Computing Group (TCG) has established an architecture for a root of trust in trusted computing for small devices to address the absence of a TPM in such devices. The architecture known as DICE (Device Identifier Composition Engine) [8] is designed as a minimal silicon capability framework, enhanced by firmware, to deliver a cryptographically strong device identity, measurement-based boot layering, and support for endorsement and attestation. DICE utilizes a Unique Device Secret (UDS) embedded in the hardware. The UDS may be generated externally and installed during manufacture and/or generated internally during device provisioning Upon each boot of the device, a new secret known as the Compound Device Identifier (CDI) is derived in each subsequent layer from the secret of the previous layer and a measurement of the current layer's starting code. This provides each layer's boot with a unique secret, ensuring that compromising one layer does not easily affect the others [9] [10]. Additionally, DICE supports an attestation architecture that enables a device to generate cryptographic claims regarding layer measurements, which can be verified using manufacturer-endorsed credentials [11]. The TCG has also introduced Measurement and Attestation Roots (MARS), a hardware-level architecture and API specification for implementing the capabilities of a root of trust in embedded/IoT systems. MARS extends trusted computing features to environments where a TPM is impractical. It provides a lightweight hardware foundation for identity, secure measurement storage, and attestation so that they can be implemented into systems on chip while ensuring isolation from the main processor [12].

We propose a method to establish TLS-PSK attestation channels between devices that prove their identity using a DICE-generated secret, leveraging the UDS for strong device identity. We explore how the DICE framework aids attestation, focusing on key management and attestation processes outlined in TCG's DICE specification. TCG defines an attestation scheme based on a DICE identity, which underpins this attestation system and will be discussed in more detail later. The goal is to allow devices to assert identity and measurement claims without explicitly provisioning unique identity keys for each layer. The identity is derived via DICE secrets and cryptographic operations, starting from the CDI of the layer from which the claim should be verified. This means that with DICE, there is no requirement for externally provided identity keys; instead, they are created dynamically on the device in a controlled manner. By using the keys derived through DICE and MARS, we experiment with the establishment of a TLS channel using TLS-PSK between a broker and a sensor for a MQTT communication channel. We evaluate the times required for the boot time, derivation of the key exploited in the TLS-PSK channel, and the time spent for the creation of the TLS channel itself.

The paper is organized as follows: Section 2 presents the DICE architecture and the secret key derivation out from the UDS. Section 3 details the MARS architecture and the key hierarchy, while Section 4 explains the implicit attestation based on device identity and the TLS-PSK protocol. Section 5 details the designed MARS-enabled TLS trusted channels used for securing the MQTT communication, Section 6 provides insights on the implementation and the experiments performed, while Section 7 presents the related work. Finally, Section 8 concludes the paper.

2. DICE

DICE layering architecture involves dividing the boot sequence into multiple layers, regarded as untrusted until measured. DICE derives keys at each layer of the boot process by combining a device's immutable secret with measurements of the code being loaded, producing a new CDI and associated cryptographic keys at every stage [13]. This ensures that each layer of firmware or software has a unique identity tied to both the hardware and its measured state. DICE derives keys at each layer by recursively hashing the previous CDI with the measurement of the next firmware stage. The primary objective is to guarantee that if any layer is compromised, the secrets for the subsequent layers vary and cannot be predicted by an attacker, even with control over a prior layer [8].

DICE secret derivation. DICE security relies on a Unique Device Secret (UDS) specific to each device. This hardware-protected secret, which is not directly accessible by the firmware, is crucial for starting the first composition step. When the First Mutable Code (FMC) begins, it is measured by the DICE logic (e.g., via a cryptographic hash function) to produce a representation M_1. The FMC is the first updatable firmware executed after ROM. Its measurement, combined with the device's UDS, produces the initial CDI and keys. This ensures that every subsequent layer of the boot process inherits trust from both the immutable hardware root and the measured FMC. Then, a one-way composition function $f(\text{UDS}, M_1 \parallel \text{optional hardware state})$ is used to generate the first CDI, namely CDI_1. Variants of this procedure can incorporate additional hardware state or configuration bits, depending on the device and implementation [9].

$$\text{CDI}_1 = f(\text{UDS}, M_1 \parallel \text{optional hardware state})$$

Once the CDI_1 is generated the underlying hardware must enforce some security measures that will be explored later, to make the UDS unable to be read again. Then CDI_1 is passed to the layer 1 where CDI_2 is derived this way:

$$\text{CDI}_2 = f(\text{CDI}_1, M_2)$$

and so on for each layer. In general, when layer $i+1$ receives CDI_i, it performs a measurement that yields M_{i+1}, and then proceeds to compute:

$$\text{CDI}_{i+1} = f(\text{CDI}_i, M_{i+1})$$

Thus, a CDI at each layer is securely derived from its predecessor CDI (at previous layer) and the measurement data. Since each CDI is specific to the measured code and hardware context, any tampering with the firmware alters its measurement and, consequently, the derived secret, breaking the continuity of expected secrets [14]. DICE allows re-keying for a layer during a firmware update or when malicious code is detected. A new layer secret cannot be reused from a previous version, especially if it may have been compromised [15].

DICE mandates specific constraints and dedicated hardware protection to ensure the security of layering and secret derivation. Key hardware requirements and constraints include:

- Reading access must be disabled in hardware after the UDS generates the first CDI, preventing the firmware from reading it again.
- The logic for composition must be inherently trusted and cannot be altered by firmware or malicious code. If updates exist, they should not be incorporated into the CDI.
- Measurement of FMC: measuring the FMC is a mandatory step in the first composition, so that any changes can be captured.
- Optional integration of hardware state bits indicates that when the composition involves fixed hardware configurations or state bits, different hardware variants yield unique CDIs, even when utilizing the same firmware. This differentiation can be advantageous for maintenance purposes.
- Isolation between layers: each layer must keep its seed or secret received from other layers confidential, which is a fundamental principle of the DICE framework.
- No need to store the UDS or any CDI in non-volatile storage: the layering approach allows secrets to be derived at runtime, eliminating the need for multiple secrets or non-volatile memory.

The DICE layering model creates a trust chain starting with the UDS and each derived CDI during every boot stage. Since each CDI relies on the previous one and the measurement of the next layer, any change in software, even a single bit, modifies the CDI. This establishes a cryptographic chain of trust from hardware through all software layers. CDIs can generate keys for encryption, signing, or attestation. They provide secure identity and confirm that the device started with the intended software stack. The security features of this layered architecture can be expanded to create an attestation and identification framework, which we use in the proposed solution.

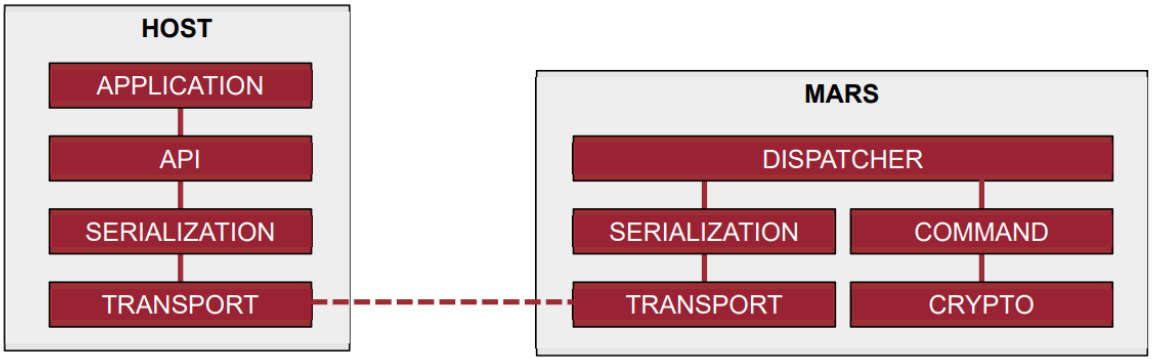

Figure 1: MARS serialized architecture(source:[18]).

3. MARS

The growth of small devices like embedded systems or IoT has created a demand for security features that consider their limited capabilities, making a root of trust essential for implementation [16]. The state-of-the-art for roots of trust is the TPM 2.0, a widely accepted standard integrated into a chip. Despite being compact and not serving as a cryptographic accelerator, some systems cannot support a TPM. Thus, the TCG developed MARS, a hardware architecture and API specification designed to implement root of trust capabilities in embedded and IoT systems. MARS aims to extend trusted computing features to environments where a TPM is impractical, by offering a lightweight hardware foundation for identity, secure measurement storage, and attestation, while maintaining isolation from the main processor [12].

Architecture. MARS specifies a minimal and modular architecture optimized for small platforms and defines a pair of RoT that collaborate to provide measurement and attestation functions. These roots operate as isolated logic domains implemented as either hardware state machines, lightweight cryptographic cores, or protected microcontrollers embedded in a larger System-on-Chip (SoC), minimizing area and power usage. Each MARS root must have access to a hardware-protected storage area that contains the immutable data, such as the root key, and to provide cryptographic and measurement services to the host system. This designed separation between a trusted and an untrusted environment for a physically or logically isolated execution for the measurement and attestation services prevents interferences and eventual malicious compromises from the host environment [12]. The MARS library specification refines this model by introducing a modular set of logical functions implemented by a standardized programming interface [17]. The serialization interface specification defines how these library functions can be transported through commands through various physical interfaces such as: SPI, UART or even an internet transport protocol, enabling interoperability across both SoC and discrete implementations [18], as shown in Figure 1.

The key aspects of a MARS oT include the following: a clear separation between trusted and untrusted logic domains; protected memory for the secure storage of measurements, credentials, and reports; and well-defined interfaces to interact with the host system while preserving isolation.

A key design element of MARS is the separation between trusted execution, where the MARS logic resides, and the host domain. The host should invoke MARS services through the APIs or a serialized command interface, but it cannot access internal registers, cryptographic keys, and measurement logs. The serialization interface formalizes the isolation of one execution logic from another by defining a communication protocol that separates the trusted operations from the host control flows. MARS commands are encoded as structured payloads, transported over a link (a physical link or a transport protocol, depending on the implementation) and then parsed and executed within the RoT domain. This separation allows MARS to exist either as a standalone co-processor or as a hardware logic block

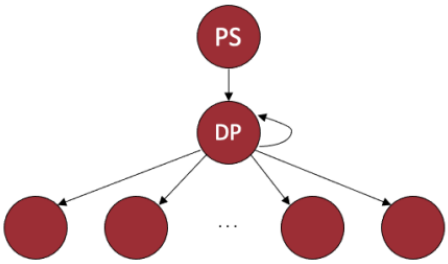

Figure 2: MARS simplistic key hierarchy (source:[17]).

within a larger chip that maintains the same logical interfaces.

The MARS framework is composed of several functional subsystems that cooperate to deliver the functionalities of a RoT:

- Device identity and root secrets: each instance that implements MARS functionalities is provisioned with a root identity, typically injected during the manufacturing process. This root serves as the base seed for key derivation and attestation. Its access is limited to few processes and it is placed on an isolated part of the memory.
- Measurement Engine: performs cryptographic hashing operations on the designed memory areas that needs to be attested. These functions form the basis for establishing the current operational state of the device that will be reported to external verifiers.
- Secure storage: the MARS RoT maintains a protected measurement log similar to the PCRs of a TPM, but optimized for size and performance. Multiple PCRs can be configured on a MARS platform. Additionally, another type of protected storage is designed, called Trusted Sensor Register, which records sensor data that will be used in a remote attestation flow.
- Attestation and reporting engine: the attestation subsystem generates cryptographic reports that combine measurement and configuration data. These reports are signed with an attestation key derived from the root identity of the device. The serialization interface defines how these reports are formatted and transported.

MARS creates a modular separation between function and implementation by defining both library-level APIs and transport-level interfaces. This allows manufacturers to integrate MARS logic into hardware, emulate it in firmware, or externalize it as a TPM chip, while adhering to specifications. This modularity sets MARS apart from previous trusted computing primitives, as it represents a flexible architectural framework rather than a singular chip or function. It supports TCG's broader attestation ecosystem, enabling even the smallest devices to perform remote attestation [12][18].

Key hierarchy. MARS's cryptographic foundation allows devices to securely perform measurement, attestation, and identity binding, even under strict hardware constraints. The library specifications outline the procedures for consistently executing these functions across various implementations. [17]. MARS RoTs begin with a root key known as *Primary Seed* (PS), which serves as a permanent anchor for all derived keys and cryptographic functions, as illustrated in Figure 2. The PS is typically stored in non-volatile secure memory or generated from unique hardware sources. From the PS, MARS creates a subordinate key called *Derived Parent* (DP), from which all other keys for identity, attestation, sealing, or binding are derived. The derivation process ensures forward secrecy and measurement binding, so that changes in firmware also lead to changes in derived keys, thus maintaining integrity continuity.

Cryptographic capabilities. MARS supports a minimal yet complete set of cryptographic primitives, which includes: secure hashing (e.g., SHA-256 or lightweight alternatives), key derivation for generating subordinate or session keys, digital signatures or MAC-based mechanisms for attestation, lightweight sealing or binding of data to verified states. In contrast to TPMs, which offer a broad range of asymmetric

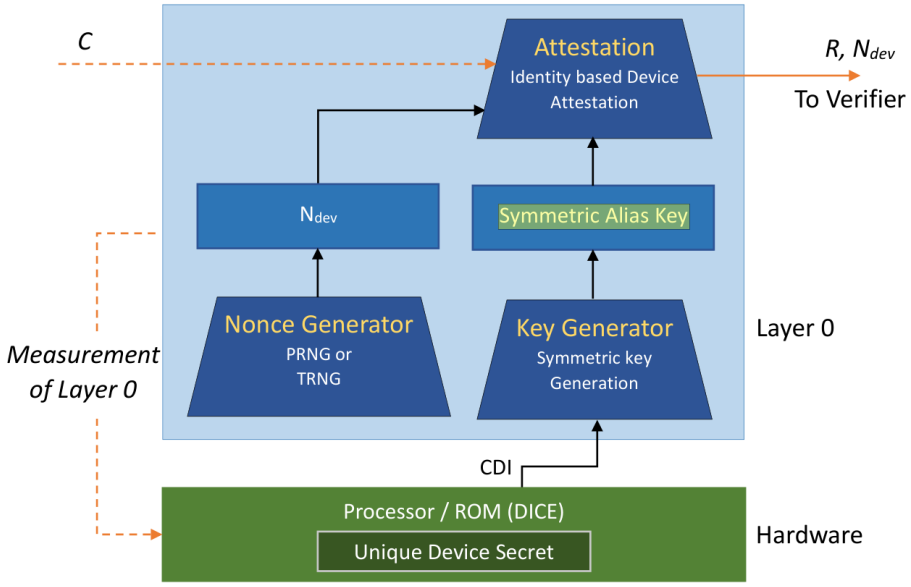

Figure 3: DICE architecture with symmetric cryptography (source:[11]).

and symmetric cryptographic services, MARS is limited to providing only the essential functionality required for measurement and attestation.

4. Designed solution

We designed and implemented a solution to provide hardware-based identity and attestation for devices with limited computational resources that cannot support a TPM root of trust and asymmetric cryptography. MARS serves as the RoT for these devices, featuring security protocols for attestation and identification.

4.1. Implicit attestation based on device identity

We explain briefly first how to achieve implicit attestation for a device using a key linked to its identity according to the DICE architecture with asymmetric cryptography.

In DICE, the First Mutable Code (FMC) refers to the initial code executed upon device startup, establishing the device's identity. Upon execution of the FMC, a key pair is produced, known as the *Identity key pair*, which represents the device's cryptographic identity and its FMC. This key pair can be derived during manufacturing before device configuration and deployment, where the public portion is accessible. It is up to the manufacturer to assign a public key certificate to this key pair. According to the DICE specifications, the private part of the Identity key pair must not be used outside the FMC. Additionally, the FMC is mandated to erase any plaintext data related to the CDI value and Identity private key from memory and registers [11]. The Identity key pair could theoretically be used for attestation; however, its restricted use outside the FMC complicates this, and the exposure of the Identity private key should be minimized. Consequently, in the DICE layer 0, another key pair known as the *Alias key pair* is derived from the CDI and the firmware image. This key pair will be used for attestation, while the Identity key pair certifies it, establishing a chain of trust without revealing its private component outside layer 0. If the Alias key pair is certified, it demonstrates that the device executed the manufacturer's FMC.

The risk of exposing the Alias private key is lower because it can be changed via a firmware update, whereas the Identity key pair generation is linked to the UDS, making changes much more challenging, if not impossible, in some situations. However, the Alias key pair derivation should rely on both the CDI and firmware image for attestation and identification. Additionally, the FMC remains unchanged

except when establishing a new device identity. If the firmware is updatable, it allows for easier bug fixes and refreshes of the Alias key pair through updates. Both the Identity and Alias key pairs are tied to an X.509 certificate and play roles in the attestation process. The Alias key pair and its certificate should have a validity shorter than those of the Identity key pair.

However, in many resource-constrained devices, asymmetric cryptography and public-key certificates can be impractical as they may slow down the attestation process and significantly degrade performance. Therefore, symmetric cryptography is better suited for these cases. Figure 3 illustrates the schema exploiting a single identity key, the *Symmetric Alias key*, derived from the CDI in the layer 0. The verifier needs to have the Symmetric Alias key. Moreover, the verifier must be trusted not to impersonate the device.

In our prototyped solution, the device consists of two DICE layers. The key derived in the second layer will serve as a long-term identity key in TLS-PSK. We also add salt to the CDI when deriving this key to increase entropy and use a specific MARS function for key derivation. The first layer, namely the boot layer, handles preliminary operations and the 1st CDI derivation, while the software layer manages the second CDI derivation and handles data specific for the application protocol used, such as fetching sensor data and managing broker connections. The key generation/derivation is performed as shown below:

- 1st CDI derivation: (1) the boot layer is measured, M_{boot}; (2) extend PCR_1 with M_{boot}; (3) a specific MARS function named `MARS_Derive()` is called to produce:
  $\texttt{CDI}_1 = \texttt{MARS_Derive}(\texttt{HASH}(M_{\text{boot}}), \texttt{DP}, \texttt{label}, \texttt{context})$
- 2nd CDI derivation: (1) the software layer is measured, M_{sw}; (2) extend PCR_1 with M_{sw}; (3) the implemented HKDF is called to produce:
  $\texttt{CDI}_2 = \texttt{HKDF}(\texttt{HASH}(\texttt{CDI}_1 | M_{\text{sw}}), \texttt{salt}, \texttt{context})$

As MARS is the root of trust, trusting its implementation is essential. The `MARS_Derive()` function produces a value associated with the boot layer measurement by utilizing the DP and, indirectly, the PS. This process differs from other RoT operation flows because it uses context information. Here, layer 0 represents the MARS RoT, while layer 1 represents the boot layer. The software further derives the 2nd CDI by using CDI_1 as key material input and an HKDF (HMAC-based Key Derivation Function), which remains compliant with the specification because hardware KDFs are not required for non-root layers.

4.2. TLS-PSK based attestation protocol

After establishing an architecture for the IoT attestation system, a protocol ensuring necessary security features in untrusted environments is needed. A straightforward challenge-response protocol using symmetric cryptography and HMAC can be summarized in six steps:

1. the verifier sends a challenge C to the device.
2. the device generates a nonce N_{dev}.
3. the device responds to the challenge by using the *Symmetric alias key*:
   MAC($C|N_{dev}$, Symmetric alias key).
4. the device sends the response R to the verifier.
5. the verifier has a copy of the symmetric key and computes the expected response R'.
6. the verifier compares R and R', if they match then the attestation is successful.

This basic protocol can facilitate attestation using symmetric keys or can be layered over existing and custom protocols, provided that security level requirements and potential attacks are duly considered. For example, using a nonce mitigates replay attacks but does not prevent Denial of Service (DoS) attacks. Thus, a timer that drops attestation requests after a successful attestation should be implemented on the device to mitigate such attacks. However, in an MQTT network, TLS is the most effective option, as it enables security using symmetric cryptography. TLS is used in its variant of TLS-PSK described in RFC-4279 [19] as an alternative to classic TLS in environments with limited CPU power or where setting a PSK is more convenient than using digital certificates.

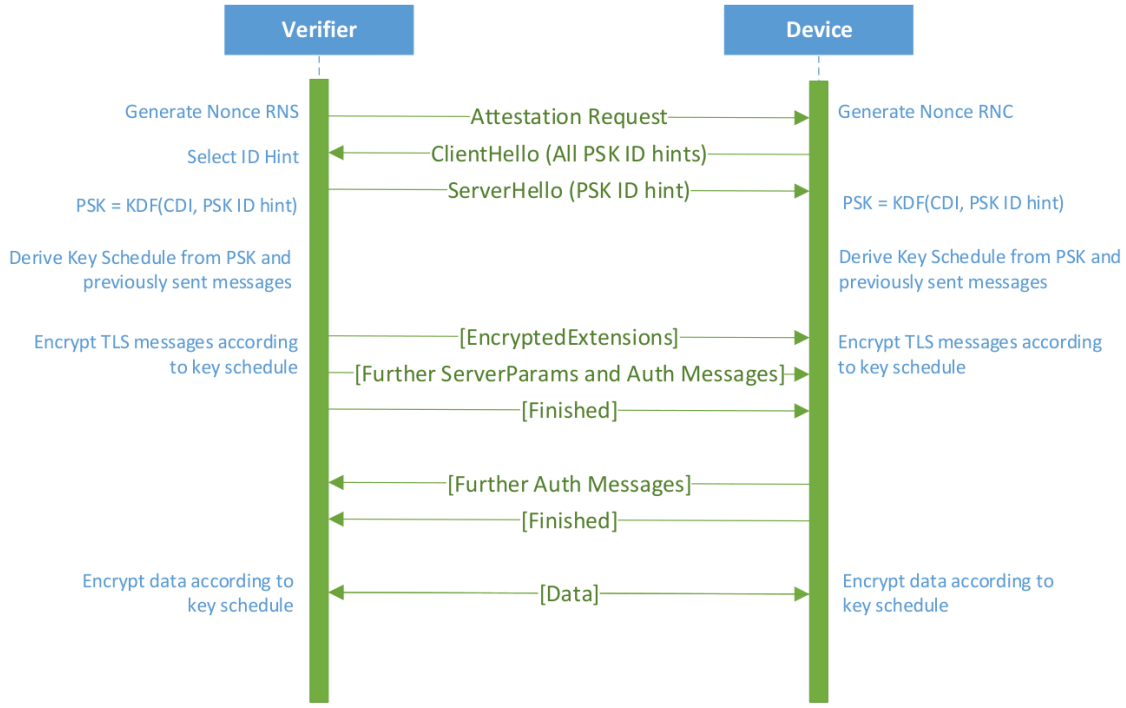

Figure 4: TLS-PSK handshake using a PSK derived from CDI (source:[20]).

With TLS-PSK, a pre-shared key is provisioned to both client and server, shown as *Device* and *Verifier* in Figure 4. When the server starts an attestation request, a PSK ID hint is exchanged between the two parties during the exchange of ClientHello and ServerHello messages, allowing the PSK to be either derived (required for clients in a DICE-based architecture) or retrieved for deriving the premaster secret. Once the premaster secret is set, the protocol proceeds like classic TLS, but using only symmetric algorithms.

As stated, the PSK is derived from the second CDI with the same key derivation mechanism exploited for the Symmetric Alias key (shown in Figure 3). This means that a successful TLS handshake is an implicit attestation of the device identity and its firmware integrity.

4.3. Key provisioning

In closed environments with mostly manual configuration, using a PSK is simpler than using an X.509 certificate. Here, key provisioning occurs during each device's manual configuration phase. The verifier must have the CDI or the key to generate the Alias key. Because a leaked key enables fake attestations, the key must be provisioned in a secure environment before device deployment. There are three options for the verifier:

1. the verifier holds the pre-shared key. If the device's FMC is updated, the device must be reconfigured and the key extracted again. However, the risk of disclosure is reduced as the key can be recomputed without altering the device's identity.
2. the verifier holds the UDS. If the FMC is updated, the verifier can compute a new CDI and derive another key. However, having the UDS stored externally poses privacy risks, especially if the device is used in various environments. If the verifier is compromised and the UDS is accessed, the device's security is jeopardized. Therefore, retaining a copy of the UDS is inadvisable in any scenario, including for the manufacturer.
3. the verifier holds the CDI. This case resembles the previous one but carries less risk. If the CDI is compromised or leaked, the device identity should be refreshed, and a new CDI computed and provided to the verifier; however, the device remains secure for use afterward.

Regardless of the choice, device-related secrets in the verifier should be handled with care and stored in a restricted area. For instance, it is recommended to protect them with a hardware security module like a TPM. We assume in our use case that the broker runs on a platform with TPM 2.0. The keys in the TPM could be used to protect IoT devices' CDIs, so that they are bound to the broker's state. This operation is done during broker configuration, which should take place in a secure environment.

5. Implementation use case: TLS-PSK secure channel for MQTT

MQTT (Message Queuing Telemetry Transport) is a lightweight application-layer messaging protocol designed for constrained IoT environments, optimized for low-bandwidth, high-latency, and energy-limited embedded devices [21]. Its publish–subscribe architecture decouples data producers from consumers through a centralized broker, enabling scalable communication patterns while minimizing client-side state, making it attractive for sensor networks and cloud-connected IoT deployments [22].

MQTT and TLS. Due to its minimal header overhead and simple control structure, MQTT is widely used in battery-powered systems and metered networks where efficiency is paramount [23]. Although MQTT itself does not provide encryption or authentication, it is typically deployed over TLS, leveraging the protocol's strong AKE properties and the widespread deployment maturity to ensure confidentiality and integrity for IoT telemetry streams. This aligns with broader observations on secure channel deployment and interoperability discussed in the transport-security literature [24, 25, 26]. For UDP-based or low-overhead environments, MQTT-SN adapts MQTT for datagram networks and can use DTLS for security, remaining compatible with highly constrained IoT devices [27]. Emerging research has also explored running MQTT over QUIC to leverage QUIC's built-in transport security, multiplexing, and metadata protection. [28]. However, these integrations remain experimental, and they are not yet reflected in the mainstream IoT standards [29]. MQTT also serves as a suitable substrate for remote attestation workflows, where attestation evidence and verifier challenges can be transmitted using the publish–subscribe model, relying on the underlying secure channel protocol to enforce authenticity, integrity, and confidentiality of the attestation exchange. Consequently, MQTT remains a foundational application protocol in IoT architectures, complementing secure-channel mechanisms such as TLS 1.3, DTLS 1.3, and QUIC and enabling efficient, scalable communication across heterogeneous networks. This protocol was used to create a test environment that would reflect a real world scenario, although without the scalability to mimic a real IoT network.

MQTT security. The MQTT 3.1 specification provides only guidelines with respect to security and recommends where possible to follow the industry standards [30]. Since MQTT only handles message transmission, security is the responsibility of solution providers, who must implement measures tailored to each case's specific security needs. Most MQTT solutions are built for hostile environments, which introduces various potential threats, such as: (a) devices could be compromised; (b) data at rest in clients and servers might be accessible; (c) protocol behaviors could have vulnerabilities or be susceptible to timing attacks; (d) DoS attacks; (e) the traffic could be sniffed, altered, or rerouted.

Another important detail is that by design the broker is a single point of failure since it manages all the traffic, and it is critical to protect it. There are many security concerns to consider when deploying a solution for an MQTT network, because a specific implementation may need to achieve some or all of the following (security) requirements, including authentication mechanisms for client and servers (R1), authorization mechanisms for accessing server resources (R2), integrity for MQTT control packets (R3), and privacy for MQTT control packets (R4).

MQTT supports basic authentication of clients by the server with the Username and Password fields in the CONNECT packet. However, username and password authentication alone is often insufficient for achieving high security because credentials may be transmitted in clear without a secure protocol on top of MQTT, and passwords are vulnerable to brute-force or dictionary attacks. MQTT does not have a built-in encryption system, so it relies on an external secure protocol for message integrity and confidentiality, such as TLS. MQTT brokers often implement Access Control Lists(ACLs) as authorization mechanism for clients. They define which topics a client can publish or subscribe to and the QoS levels

allowed. For finer authorization controls, some brokers includes external policy engines, have optional extensions for advanced authorization policies, or use role-based access authorization models.

6. Implementation details

This section offers further technical specifics on the implementation and the software employed. The attestation protocol utilizing TLS-PSK is the one outlined in Section 4.2. However, due to technology limitations and the fact that most brokers currently available do not support TLS 1.3 with pre-shared keys, TLS 1.2 was used. Nevertheless, this choice does not undermine the validity of the implemented security features.

Currently, MARS is in the experimental phase, resulting in limited device utilization and deployments. For testing purposes, the TCG has provided an emulator that adheres to the MARS specifications [31]. The emulator, available in C and Python, enables testing of the RoT via two command-line programs. The emulator is still in development, and some components are not yet implemented. The testing application we created uses the C version of the emulator since the MQTT interfaces for device-broker communication are built in C. This application employs Mosquitto, a lightweight, open-source MQTT server that supports MQTT standards 3.1, 3.1.1, and 5.0, making it ideal for both low-power embedded devices and large servers. It includes the C-based client library (libmosquitto) that provides a comprehensive API for MQTT client operations (connect, publish, subscribe, loop) and supports command-line tools *mosquitto_pub* and *mosquitto_sub*. Additionally, Mosquitto supports a plugin system. For instance, the ”Dynamic Security” plugin allows for role-based authentication and access control, which configurable at runtime through a specific MQTT topic interface, thereby enhancing the broker's functionality without altering the core code [32].

6.1. Application and experimental testbed details

The application is composed of two modules: 1) the broker module that uses a custom configuration and a plugin for Eclipse Mosquitto to implement the broker key management; 2) the MARS device module that emulates a temperature sensor that publish and subscribe on the same topic through Docker containers deployed with limited resources to match the performance of a small device. In the experiments, we used a laptop with an Intel Core I5 - 2.7 GHz CPU and 16 GB of memory.

Broker module. The broker acts as a central communication and authentication hub for all nodes. It is a server-side component that enables client interaction via a publish/subscribe protocol like MQTT. Clients publish data to specific topics instead of messaging each other directly. The broker receives these publications and forwards them to the relevant subscribers. At startup, the broker initializes internal data structures, loads configuration files, and prepares the environment for client sessions. This startup process may take time depending on the retained data and configuration complexity, but it happens only once during startup. Additionally, MQTT brokers provide configuration files or management interfaces for administrators to manage options like authentication methods, Quality of Service (QoS) levels, and persistence behaviors. Test brokers or sandbox instances are used for development and experimentation without affecting the production system. Platforms like Eclipse Mosquitto provide public test brokers for developers to use for free, but production-grade deployments typically require a dedicated instance and internal management of access credentials.

Broker key management. Mosquitto's basic authentication uses a simple username–password scheme stored in a file with credentials in cleartext. The default protection of the pre-shared keys, if configured to use TLS-PSK, is also absent since it uses another file in cleartext where the identities and the keys are stored. This design is not well-suited to support secure authentication and attestation. Mosquitto allows customization of its authentication mechanisms via plugins that can be loaded in the configuration file, enabling security through a plugin. Assuming that the broker can mount a TPM, it is

possible to use the TPM keys to protect the (sensor device) keys. However, this aspect is considered out of scope of this work.

Docker container deployment. All experiments to evaluate the performance of the MARS device emulator were conducted in a dedicated Docker container to ensure a controlled and reproducible environment. The Docker-based setup served two primary objectives: (1) emulate the resource constraints typical of IoT devices by limiting CPU and memory availability; (2) isolate the emulator from the host system, ensuring deterministic behavior across different runs and hardware platforms.

The container image is based on Linux Ubuntu 22.04, which is the same operating system used for developing and compiling the emulator executables, thereby ensuring compatibility. The Dockerfile installs only the minimal set of required packages: `libmosquitto1`, `bash`, and `procps`, keeping the container lightweight and minimizing interference from unnecessary system services. The MARS emulator directory and start script were copied into the container's working directory, and execute permissions were granted to all emulator binaries during the build. This makes the container self-sufficient at runtime, requiring no external bindings aside from resource limits. A custom entrypoint script initializes the container by keeping it in an idle state instead of immediately running the emulator. This design decision allows interactive exploration and debugging. It lets users attach to the container, inspect files, and manually test emulator configurations without rebuilding or restarting the image. It also keeps timing measurements consistent by separating container startup from emulator execution.

Each running Docker container, whether a publisher or subscriber, includes a helper shell script that simplifies using the MARS emulator, which requires two executables and several parameters. The script enables the selection of the core cryptographic algorithm, the salt for PSK derivation, and the TLS ciphersuite for handshake measurements. Automating parameter selection ensures consistent execution patterns across tests and minimizes human error during data collection. Combined with CPU and memory limits (0.25 CPU, 8 MB RAM), this setup effectively simulates the computational constraints of a small embedded platform.

6.2. Tests description and results

Three main tests were run, each using a different core algorithm on the sensor with different key sizes:

Test 1. device boot time. This test includes the preliminary tests that MARS must perform, the self-measurement, and the derivation of the first CDI.

Test 2. pre-shared key (2nd CDI) derivation time.

Test 3. TLS-PSK handshake time.

All timing measurements were conducted using the standard C library `time.h`. Using the system's monotonic clock ensures that measured intervals remain unaffected by changes to system time, such as NTP adjustments or manual alterations, which could lead to inconsistencies or negative deltas in recorded values. Measurements were recorded by noting the timestamps before and after the target operation, calculating the elapsed time with nanosecond precision. This method provides a consistent way to evaluate performance in a limited Docker environment, guaranteeing that the reported values reflect the actual execution time of operations, not the impact of external system scheduling.

A quantitative evaluation of performance is beneficial for the measured boot procedure since the MARS RoT used in development was implemented solely in software. The sensor emulator consists of two DICE layers (as detailed in Section 4.1), where the measured boot sequence covers two layers until the second CDI is generated. The emulator was executed inside a Docker container with limited resources. We utilized 0.25% of CPU power, keeping computation time realistic, and 8 MB of RAM, which exceeds what's available on an ESP32-like device but is sufficient for hardware emulation.

The measurements show a resource-constrained IoT device, but a direct comparison with a hardware-backed MARS implementation isn't possible because it's unavailable, so only software-based timing results were collected. The boot layer begins by initializing the MARS APIs and gathering device configuration parameters such as the active core cryptographic algorithm, digest size, key and signature

lengths, and the chosen hash function. The first CDI is generated by hashing the boot section code and extending this measurement into the first PCR before invoking the key derivation API. The second CDI follows a similar two-step process, measuring the software layer in the first phase and stopping before deriving the PSK in the second phase.

Core cryptographic algorithm	128-bit key	256-bit key
Ascon	from 2.181 ms to 2.712 ms, average: 2.5005 ms	from 2.271 ms to 4.293 ms, average: 2.7031 ms
SHE AES128	from 2.181 ms to 2.968 ms, average: 2.524 ms	from 2.320 ms to 3.594 ms, average: 2.6155 ms
HMAC SHA256	from 2.196 ms to 3.078 ms, average: 2.494 ms	from 2.250 ms to 4.084 ms, average: 2.8767 ms
HMAC SHA3-256	from 2.230 ms to 2.709 ms, average: 2.4811 ms	from 2.163 ms to 3.676 ms, average: 2.580 ms

Table 1
Boot computation time for different MARS key sizes for each core algorithm.

Time results are heavily influenced by the emulator's software-only design and the restrictive container setup. Although the system works, the lack of hardware cryptographic support forces all hashing, measurement, and sequence processing to run sequentially in user space. Table 1 shows the time ranges and average values for each core cryptographic function implementation on the MARS emulator. The execution time for various ciphers ranges from about 2.18 ms to 4.29 ms, with average times between 2.48 ms and 2.87 ms, depending on the algorithm strength and key size. We note that there is no significant difference. All tested cryptographic configurations complete within a tight millisecond range with minimal variation. These results indicate that the MARS boot pipeline remains stable and performs consistently across multiple measurements, even in a resource-constrained environment. In hardware implementations, some operations can be more efficient but introduce security constraints. Therefore, although the findings show that a full DICE-based measured boot is feasible in a constrained environment, they do not represent a real hardware MARS device. In these devices, security-critical tasks such as PCR updates and CDI derivations rely on dedicated hardware components, resulting in notably different timing characteristics.

Besides boot time, we have also measured the time required for key derivation, namely for the second CDI, which acts as the PSK in the TLS-PSK channel with the broker. Table 2 summarizes the time results for the two key sizes. Both configurations show similar performance characteristics. This is because the main cost of HMAC with SHA-512 is independent of the final key length. The slightly lower average time for the 256-bit PSK is due to normal measurement variance, not to differences in the derivation algorithm. The HKDF used internally expands the key material to much larger lengths and then truncates it to the desired size. Overall, this result confirms that the computation of the PSK introduces only a marginal additional delay compared to the rest of the measured boot flow. Even under constrained CPU and memory conditions, the derivation remains stable and predictable, reinforcing the viability of executing a full DICE compliant bootstrap sequence in IoT devices.

128-bit PSK	256-bit PSK
from 2.635 ms to 3.318 ms, average: 3.0064 ms	from 2.691 ms to 3.003 ms, average: 2.8102 ms

Table 2
Pre-shared key derivation time for different key sizes.

In addition to deriving the second CDI, further experiments were conducted to assess the TLS handshake performance utilizing the derived PSK for secure communication with implicit boot attestation. These tests demonstrate the viability of using TLS-PSK in resource-constrained environments and showed how various ciphersuites affect handshake latency. The PSK was integrated into the TLS context right after derivation, without session caching or resumption, which means that every TLS handshake required a complete key exchange. The chosen ciphersuites ensure perfect forward secrecy, offering both DHE-PSK and ECDHE-PSK options, as well as several symmetric algorithms and hash functions. Performance depended on the symmetric cipher and the computational cost of Diffie–Hellman operations, regardless of the use of elliptic curves. The timing results are reported in the Table 3. All three ciphersuites exhibited similar average handshake times. This indicates that, under resource constraints, the primary performance factor is the CPU-bound ephemeral key exchange, rather than the symmetric encryption algorithm or PSK size. The minor millisecond-scale variations among ciphersuites suggest that complex key-exchange mechanisms like ECDHE can still be effective for IoT devices when implemented in software. As with earlier measurements, these results exclude hardware-based secure processors, whose dedicated cryptographic accelerators would significantly improve performance.

Ciphersuite	**PSK size (bit)**	**Elapsed time**
DHE-PSK-AES128-GCM-SHA256	128 bit	from 5.198 ms to 6.588 ms, average: 5.6286
DHE-PSK-AES256-GCM-SHA384	256 bit	from 4.917 ms to 7.759 ms, average: 5.6819 ms
ECDHE-PSK-CHACHA20-POLY1305	256 bit	from 4.758 ms to 7.096 ms, average: 5.5989 ms

Table 3
Time to complete TLS handshake using 128-bit and 256-bit PSK keys with selected ciphersuites.

7. Related Work

IoT device attestation is increasingly recognized as a critical and prominent challenge for cybersecurity researchers, owing to the exponential growth of infrastructures that rely on IoT-based networks for their operation. To improve the reliability of target device attestation in highly dynamic and decentralized network environments, a swarm attestation approach can be exploited. For instance, by utilizing Physical Unclonable Functions (PUFs) as the root of trust, PISA [33] develops a lightweight swarm attestation and aggregation mechanism to identify suspicious target devices within a cluster. ESDRA [34] proposes a secure, distributed swarm attestation scheme for IoT networks that employs a many-to-one attestation method to mitigate single-point-of-failure risks.

Control-flow attestation is a specialized form of remote attestation, designed for small-scale or embedded systems, that aims to detect control-flow hijacking attacks during program execution. To enable trustless validation and ensure compatibility with low-end embedded systems, RAGE [35] employs a lightweight control-flow attestation mechanism that can reliably detect code reuse attacks.

Recognizing the critical need to verify the trustworthiness of IoT networks, MATCH-IN [10] proposed a novel approach to establishing mutual attestation among devices within heterogeneous IoT networks. MATCH-IN schema exploits the TCG DICE architecture. It enhances the security of unstructured IoT networks by allowing devices in the field to mutually attest their identities and configurations, without relying on a centralized verifier.

SEDA (Scalable Embedded Device Attestation) [36] was the first work to propose a swarm attestation scheme for efficiently attesting large numbers of IoT devices. SEDA operates under the assumptions

that all devices in the network satisfy the requirements for hybrid attestation, that the configuration of the IoT network remains static with respect to both the number of devices and their physical placement, and that each device is capable of communicating exclusively with its immediate neighbors.

Two works, SALAD [7] and PADS [37], also examine attestation in heterogeneous, highly dynamic IoT networks. They propose a different approach from prior solutions, using a gossip protocol and distributed consensus.

To counter software manipulation attacks on running processes, data, or configuration files in TPM-aware IoT devices, prior work has used the Keylime framework for remote attestation [38]. The authors showed that remote attestation can be completed within seconds, limiting the time these devices are exposed to attacks against their running software and hosted data. Keylime has been also exploited in [39] to achieve hardware-based continuous attestation for cloud native and edge deployments.

8. Conclusions

IoT environments are in continuous expansion, but the devices exploited in such environments present significant limitations, physical constraints, energy use, and computational demands. However, a key requirement for building security services in IoT contexts is identifying and authenticating devices. TPMs are commonly used for secure key storage and remote attestation, but are generally unsuitable for IoT devices. We present a solution for device identification and attestation for IoT devices that lack a TPM. Our solution uses the DICE architecture and MARS framework to derive secrets that are securely bound to device hardware. These secret keys are then used to establish a secure TLS-PSK trusted channel, providing implicit attestation of the device's identity and firmware integrity.

We built an application and an experimental prototype using an extended MARS emulator to support the MQTT use case. It simulates a sensor communicating with an MQTT broker, where the sensor proves its hardware-based identity to the broker using a TLS-PSK attestation protocol. Using a pre-shared identity key for sensor identification and attestation avoids complex asymmetric cryptography while still providing sufficient security. We conducted experimental tests to evaluate the performance in terms of computation and communication, focusing on the device boot process, key derivation time, and TLS handshake time across different cryptographic setups. The findings suggest that lightweight trust architectures effectively balance security and efficiency, enabling constrained devices to achieve secure authentication and attestation with minimal computational resources.

Acknowledgments

This work was supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] A. Dauda, O. Flauzac, F. Nolot, A Survey on IoT Application Architectures, Sensors 24 (2024) 5320. doi:10.3390/s24165320.

[2] A. Choudhary, Internet of Things: a comprehensive overview, architectures, applications, simulation tools, challenges and future directions, Discover Internet of Things 4 (2024). doi:10.1007/s43926-024-00084-3.

[3] F. G. Abdulkadhim, Z. Yi, C. Tang, M. Khalid, S. A. Waheeb, A Survey on the Applications of IoT: An Investigation into Existing Environments, Present Challenges and Future Opportunities,

TELKOMNIKA (Telecommunication, Computing, Electronics and Control) 18 (2024) 1221–1233. doi:10.12928/telkomnika.v18i3.15604.

[4] V. Bhardwaj, A. Anooja, L. S. Vermani, Sunita, B. K. Dhaliwal, Smart Cities and the IoT: An In-Depth Analysis of Global Research Trends and Future Directions, Discover Internet of Things 4 (2024). doi:10.1007/s43926-024-00076-3.

[5] M. S. Ahsan, A.-S. K. Pathan, A Comprehensive Survey on the Requirements, Applications, and Future Challenges for Access Control Models in IoT: The State of the Art, IoT 6 (2025) 9. doi:10.3390/iot6010009.

[6] A. Seshadri, A. Perrig, L. van Doorn, P. Khosla, Swatt: software-based attestation for embedded devices, in: IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004, 2004, pp. 272–282. doi:10.1109/SECPRI.2004.1301329.

[7] F. Kohnhäuser, N. Büscher, S. Katzenbeisser, Salad: Secure and lightweight attestation of highly dynamic and disruptive networks, in: Proceedings of the 2018 on Asia Conference on Computer and Communications Security, ASIACCS '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 329–342. URL: https://doi.org/10.1145/3196494.3196544. doi:10.1145/3196494.3196544.

[8] Trusted Computing Group, What is a Device Identifier Composition Engine, 2023. https://trustedcomputinggroup.org/what-is-a-device-identifier-composition-engine-dice.

[9] Trusted Computing Group, Hardware Requirements for a Device Identifier Composition Engine, Version 1.0, Revision 0.91, 2024. https://trustedcomputinggroup.org/resource/hardware-requirements-for-a-device-identifier-composition-engine/.

[10] S. Sisinni, D. G. Berbecaru, V. Donnini, A. Lioy, Match-in: Mutual attestation for trusted collaboration in heterogeneous iot networks, in: 2024 IEEE Symposium on Computers and Communications (ISCC), 2024, pp. 1–6. doi:10.1109/ISCC61673.2024.10733616.

[11] Trusted Computing Group, Implicit Identity Based Device Attestation, Reference, Version 1.0, Revision 0.93, 2018. https://trustedcomputinggroup.org/resource/implicit-identity-based-device-attestation/.

[12] Trusted Computing Group, Measurement and Attestation RootS (MARS) Work Group, 2021. https://trustedcomputinggroup.org/work-groups/mars/.

[13] Trusted Computing Group, TCG Announces DICE Architecture for Security and Privacy in IoT and Embedded Devices, 2017. https://trustedcomputinggroup.org/tcg-announces-dice-architecture-security-privacy-iot-embedded-devices.

[14] Trusted Computing Group, DICE Provides Trust Foundation and Security to IoT Devices, 2018. https://trustedcomputinggroup.org/dice-provides-trust-foundation-security-iot-embedded-devices.

[15] Trusted Computing Group, Accurately Attest the Integrity of Devices with DICE, 2023. https://trustedcomputinggroup.org/accurately-attest-the-integrity-of-devices-with-dice.

[16] Lawrence Liu, RoT: The Foundation of Security, eMemory Technology Inc., 2020. https://www.ememory.com.tw/en-en-US/Article/2020-05-18/RoT-The-Foundation-of-Security.

[17] Trusted Computing Group, Measurement and Attestation RootS (MARS) Library Specification, Specification, Version 1, Revision 14, 2023. https://trustedcomputinggroup.org/wp-content/uploads/TCG_MARS_Library_Spec_v1r14_pub.pdf.

[18] Trusted Computing Group, Measurement and Attestation RootS (MARS) Serialization Interface Specification, Specification, Version 1, Revision 0, 2024. https://trustedcomputinggroup.org/wp-content/uploads/Measurement-and-Attestation-RootS-Serialization-Interface-Specification-Version-1-Revision-0_pub-1.pdf.

[19] P. Eronen, H. Tschofenig, Pre-Shared Key Ciphersuites for Transport Layer Security (TLS), RFC-4279, 2005. doi:10.17487/RFC4279.

[20] Trusted Computing Group, Symmetric Identity Based Device Attestation, Specification, Version 1.0, Revision 0.95, 2020. https://trustedcomputinggroup.org/wp-content/uploads/TCG_DICE_SymIDAttest_v1_r0p95_pub-1.pdf.

[21] OASIS Message Queuing Telemetry Transport (MQTT) Technical Committee, Mqtt version 3.1.1, OASIS Standard, 2014. https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[22] U. Hunkeler, H. L. Truong, A. Stanford-Clark, MQTT-S – A Publish/Subscribe Protocol For Wireless Sensor Networks, in: 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08), Bangalore (India), 2008, pp. 791–798. doi:10.1109/COMSWA.2008.4554519.

[23] N. Naik, Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP, in: IEEE International Systems Engineering Symposium (ISSE), Vienna (Austria), 2017, pp. 1–7. doi:10.1109/SysEng.2017.8088251.

[24] T. Enghardt, T. Pauly, C. Perkins, K. Rose, C. Wood, A Survey of the Interaction between Security Protocols and Transport Services, RFC-8922, 2020. doi:10.17487/RFC8922.

[25] B. Dowling, M. Fischlin, F. Günther, D. Stebila, A Cryptographic Analysis of the TLS 1.3 Handshake Protocol, Journal of Cryptology 34 (2021). doi:10.1007/s00145-021-09384-1.

[26] P. Duplys, R. Schmitz, TLS Cryptography In-Depth, Sciendo, 2024. https://reference-global.com/book/9781804615966.

[27] E. Rescorla, H. Tschofenig, N. Modadugu, The Datagram Transport Layer Security (DTLS) Protocol Version 1.3, RFC-9147, 2022. doi:10.17487/RFC9147.

[28] J. Iyengar, M. Thomson, QUIC: A UDP-Based Multiplexed and Secure Transport, IETF Internet-Draft, 2021. https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport.

[29] F. Fernández, M. Zverev, P. Garrido, J. R. Juárez, J. Bilbao, R. Agüero, And QUIC meets IoT: performance assessment of MQTT over QUIC, in: 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), Thessaloniki (Greece), 2020, pp. 1–6. doi:10.1109/WiMob50308.2020.9253384.

[30] OASIS Open, MQTT Specifications, 2020. https://mqtt.org/mqtt-specification/.

[31] Trusted Computing Group, MARS – Measurement and Attestation RootS, Emulator Repository, 2023. https://github.com/TrustedComputingGroup/MARS/tree/main.

[32] Eclipse Foundation, Mosquitto Documentation, 2010. https://mosquitto.org/documentation/.

[33] S. S. Mehjabin, M. Younis, PISA: PUF-Based IoT Swarm Attestation Protocol, IEEE Internet of Things Journal 12 (2025) 36094–36111. doi:10.1109/JIOT.2025.3583269.

[34] B. Kuang, A. Fu, S. Yu, G. Yang, M. Su, Y. Zhang, ESDRA: An Efficient and Secure Distributed Remote Attestation Scheme for IoT Swarms, IEEE Internet of Things Journal 6 (2019) 8372–8383. doi:10.1109/JIOT.2019.2917223.

[35] M. Chilese, R. Mitev, M. Orenbach, R. Thorburn, A. Atamli, A.-R. Sadeghi, GNN-based Control-Flow Attestation for Embedded Devices , in: 2024 IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, Los Alamitos, CA, USA, 2024, pp. 3346–3364. URL: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00251. doi:10.1109/SP54263.2024.00251.

[36] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, C. Wachsmann, SEDA: Scalable Embedded Device Attestation, in: 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver (CO, USA), 2015, p. 964–975. URL: https://doi.org/10.1145/2810103.2813670.

[37] M. Ambrosin, M. Conti, R. Lazzaretti, M. Rabbani, S. Ranise, PADS: Practical Attestation for Highly Dynamic Swarm Topologies, in: International Workshop on Secure Internet of Things (SIoT), 2018, p. 18–27. doi:10.1109/siot.2018.00009.

[38] D. G. Berbecaru, S. Sisinni, Counteracting software integrity attacks on IoT devices with remote attestation: a prototype, in: 2022 26th International Conference on System Theory, Control and Computing (ICSTCC), 2022, pp. 380–385. doi:10.1109/ICSTCC55426.2022.9931765.

[39] A. N. B. Emran, R. Upadhyay, R. Paudyal, L. Donnan, D. Wijesekera, TPM-based continuous remote attestation and integrity verification for 5G VNFs on kubernetes, in: 2025 IEEE 7th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA), 2025, pp. 435–444. doi:10.1109/TPS-ISA67132.2025.00053.