6

Secure Data Handling using Physical Unclonable Functions

The IoV network is rapidly evolving, connecting a wide range of devices, from basic sensors to advanced actuators, forming a large-scale vehicular communication system[188]. The widespread deployment of IoV devices presents significant challenges, especially concerning resource constraints. These devices often operate with limited processing power, memory, and energy supply, making it challenging to ensure robust security[218]. The high connectivity of these devices increases the risk of unauthorized access and data breaches, posing severe threats to network security and user privacy[219]. To address these risks, cryptographic mechanisms, such as encryption, access control, and authentication, are essential. However, traditional software-based cryptographic methods[220, 221, 197, 222] such as AES, DES, RSA, and ECC may not be suitable for IoV devices due to their computational and power demands. Therefore, the need for efficient cryptographic solutions that are compatible with the limited resources of IoV devices is more critical[223]. To overcome these challenges, there has been a shift towards hardware cryptography as a feasible option for the security of IoV networks[224]. Hardware-based cryptographic solutions[225, 226, 227, 111] have the potential to alleviate the extensive computational workload associated with software cryptography. These techniques can enhance efficiency and security by embedding cryptographic operations in specialized hardware such as ASIC and FPGA[228]. Utilizing ASICs and FPGAs can enhance and expedite cryptographic processes, making AES an ideal alternative for IoV environments due to its less complex algorithm and lower resource demands. By incorporating AES into these hardware platforms, it is possible to achieve strong encryption capabilities without overburdening the already scarce resources of IoV devices. As a result, AES encryption is a leading solution in hardware cryptography that offers a practical and effective security option in the varied and resource-constrained environment of IoV. Furthermore, integrating AES encryption within AEAD frameworks provides a balanced approach to security and efficiency by combining encryption and authentication in a single step. However, nonce management in AEAD using AES is a critical factor that presents several challenges. Ensuring a unique nonce for each encryption operation can be challenging in systems with frequent transactions or when multiple devices need to stay synchronized. Nonce reuse can severely undermine security, leading to potential exposure of sensitive data. Additionally, generating and securely storing nonce adds complexity and resource demands, particularly in lightweight IoV devices[112].

In this chapter, the properties of PUF are integrated with AEAD algorithm. The proposed algorithm, PUF-AEAD, uses AES-CTR[229] and HMAC[230] with a 96-bit IV and 32-byte Associated Data (AD) generated using a PUF device. By leveraging the inherent randomness and uniqueness from their manufacturing process, PUFs generate nonce that are both unique and unpredictable, enhancing security against potential attacks. Furthermore, PUFs provide a secure way to store cryptographic keys. They generate keys

tied to the unique physical characteristics of the device, making them nearly impossible to clone or extract. This ensures that even if a device is compromised, the keys remain protected. PUFs also allow for on-demand key generation, reducing the need to store keys long-term and lowering the risk of key theft. This integration enhances security while maintaining efficiency and low power consumption, which is crucial for IoV devices. For hardware implementation, a 22AJ1097 micro-controller board Raspberry Pi Pico (RPP) is used as a PUF device to generate the IV, AD, authentication, and encryption keys. This PUF device uses Micro-Controller Unit (MCU) temperature and Light Dependent Resistor (LDR) scattering for entropy generation. This entropy provides the randomness in the PUF device. Another device, the ESP32-D0WD module, is dedicated to executing the proposed AEAD algorithm. The proposed PUF-based AEAD algorithm is tested on an Microprocessor Unit (MPU)-6050 3-axis Accelerometer and Gyro sensor, capturing 3-axis gyroscope and 3-axis accelerometer data for encryption. The key contribution of this research can be broken down into two parts:

1. **Modification of AEAD with PUF function:** A significant contribution of PUF-AEAD is the integration of PUF into the AEAD framework. This modification transforms the key and nonce generation process within AES-128 encryption and HMAC authentication, introducing a nonce-reuse-resistant mechanism.

2. **Implementation, performance, and security analysis:** The practical implementation of PUF-AEAD is executed on RPP 22AJ1097 and ESP32-DoWD micro-controllers, coupled with an MPU6050 gyroscope sensor module. This chapter presents a thorough analysis of the performance and security aspects of PUF-AEAD. The results demonstrate significant improvements in terms of encryption time, used RAM, current consumption, energy consumption, and overall security, validating the effectiveness of PUF-AEAD in real-world IoV applications.

6.1 PROPOSED SCHEME: PUF-AEAD

Traditional software-based encryption methods often face challenges in resource-constrained devices, mainly due to their limited processing power and memory. These challenges include high computational demands and increased power usage, which can make the devices less efficient and more vulnerable to attacks. PUF-AEAD integrates the unique characteristics of the PUF with the hardware implementation of the AEAD algorithm, where the AES-CTR encryption technique is used with HMAC authentication. The algorithm uses a PUF device to generate 16-byte nonce, such as IV and 32-byte AD, encryption and authentication keys used in AES-CTR encryption and HMAC authentication, respectively. This method enhances security while keeping the process efficient and low in power consumption, making it a better fit for IoV devices.

6.1.1 Security Enhancement Using PUFs

Traditional cryptographic systems often rely on the secure storage of long-term secret keys in non-volatile memory (NVM) such as EEPROM or flash. However, this approach introduces multiple vulnerabilities, including key extraction through memory dumping, device cloning, and side-channel analysis. In contrast, the proposed protocol leverages Physical Unclonable Functions (PUFs) to eliminate static key storage by enabling dynamic and device-specific key generation.
A PUF is a hardware-based primitive that exploits uncontrollable manufacturing variations to produce a

unique and repeatable response to a given challenge. Each device thus behaves like a hardware fingerprint. In this work, a cryptographic key is generated dynamically using a PUF and a fuzzy extractor equipped with Error Correction Codes (ECC). This ensures that even if minor noise affects the PUF output, the derived cryptographic key remains consistent and stable.
Let `Challenge` denote the input to the PUF and the response be computed as:

$$R_{\text{PUF}} = PUF(\texttt{Challenge}) \tag{6.1}$$

Since R_{PUF} may vary due to environmental conditions or intrinsic noise, a helper string W_{PUF} is used in conjunction with a reproducibility function $\text{Rep}(\cdot)$ from the fuzzy extractor framework:

$$K_{\text{PUF}} = \text{Rep}(R_{\text{PUF}}, W_{\text{PUF}}) \tag{6.2}$$

The helper data W_{PUF} is public and non-sensitive, and does not leak meaningful information about the response or key. The reconstructed key K_{PUF} is used for encryption and message authentication purposes but is never stored in the system's memory.
To ensure forward secrecy, the encryption nonce is also derived from PUF responses in a session-specific manner. The nonce for each session is computed as:

$$Nonce_{\text{PUF}} = H(R_{\text{PUF}} \oplus \text{Session-ID}) \tag{6.3}$$

where $H(\cdot)$ is a secure hash function and `Session-ID` uniquely identifies each communication session. This guarantees that no two sessions reuse the same nonce, mitigating replay and key-collision risks.
While PUFs provide enhanced security by avoiding key storage, they are not immune to certain attacks. Three major threat categories and their corresponding mitigation strategies are outlined below:

- **Modeling Attacks:** An adversary could attempt to learn the behavior of a strong PUF by collecting multiple Challenge-Response Pairs (CRPs) and training a machine-learning model. To counteract this, the protocol applies dynamic challenge obfuscation. The effective challenge for each session is randomized using a session-specific nonce:

  $$C_{\text{rand}} = H(\texttt{Challenge} \oplus N_{\text{session}}) \tag{6.4}$$

  This prevents the reuse of identical challenge-response behavior across sessions, rendering predictive modeling infeasible.

- **Environmental Variations and Aging:** PUF responses are sensitive to temperature, voltage, and aging-related factors. To ensure stability, the system incorporates Bose–Chaudhuri–Hocquenghem (BCH) codes for error correction. For instance, BCH(63,51,2) corrects up to 2-bit errors in the response:

  $$R_{\text{corrected}} = D(R_{\text{PUF}}) \tag{6.5}$$

  where $D(\cdot)$ is the decoding function. Aging compensation techniques also ensure the response threshold adjusts over time.

- **Side-Channel and Tampering Attacks:** Side-channel attacks such as Differential Power Analysis (DPA) or Timing Analysis can leak response patterns. The protocol includes power masking and

execution-time obfuscation to mitigate this:

$$P_{\text{observed}}(t) = P(t) + N(t) \tag{6.6}$$

where $N(t)$ is randomized noise added to the power signature. If an adversary physically tampers with the circuit (e.g., using laser fault injection), the PUF's analog properties shift unpredictably, leading to incorrect responses:

$$R_{\text{tampered}} \neq R_{\text{PUF}} \tag{6.7}$$

This built-in sensitivity to physical conditions is leveraged for intrusion detection and authentication denial.

6.1.2 Modified AEAD algorithm using PUF

PUF-AEAD is designed to address the limitations of traditional AEAD algorithms, particularly in the context of IoV security. The requirement for a unique nonce in AES-CTR for each encryption under the same key presents a significant challenge, especially in systems with frequent or high-volume data transmissions. Nonce management, if not handled correctly, can lead to security vulnerabilities. For example, consider a nonce reuse scenario, where $\mathbb{A}$'s goal is to access plain texts $PlainText_1$ and $PlainText_2$ from $CipherText_1$ and $CipherText_2$, encrypted under the same encryption key and same nonce.

$$CipherText_1 = PlainText_1 \oplus Enc_{key}(nonce).$$

$$CipherText_2 = PlainText_2 \oplus Enc_{key}(nonce).$$

Now, adversary has $CipherText_1$ and $CipherText_2$ and computes $CipherText_1 \oplus CipherText_2$ which simplifies $PlainText_1 \oplus PlainText_2$. Now, the adversary can use knowledge about the format of $PlainText_1$ or $PlainText_2$ to deduce information about the other. If $PlainText_1$ is known or partially known like predictable patterns or common headers, the adversary can recover $PlainText_2$. Furthermore, managing two separate keys (one for AES-CTR and one for HMAC) adds complexity to key management, increasing the risk of misconfiguration or key mishandling in practical implementations. The PUF-AEAD algorithm is designed to address the limitations of nonce reuse and key management. It introduces an innovative approach by incorporating PUFs for nonce and dynamic key generation. This method not only fortifies the encryption process by introducing non-predictability and uniqueness in nonce and key generation but also addresses the static key vulnerability inherent in traditional AEAD algorithms. The dynamic nature of PUF-generated secret parameters (nonce and encryption/authentication keys) means that the encryption keys are constantly changing, making it difficult for attackers to exploit them. By modifying the state arrays in AES-CTR with values derived from the PUF key, the proposed method adds an extra layer of randomness and security to the encryption rounds. For authentication, the method employs HMAC, but with a twist. Instead of using a static key, HMAC uses a key generated from the PUF or a derivative of the main encryption key. This ensures that the authentication process is as dynamic and secure as the encryption process, strengthening the overall security framework against various attacks. The proposed algorithm is divided into three steps: 1) Nonce generation, 2) Keys generation, and 3) Authenticated encryption. The overall process of PUF-based authenticated encryption, including nonce and key generation, is illustrated in Figure 6.1.

Nonce generation: For the AES-CTR encryption, a nonce and encryption key are needed. The nonce is generated by first producing a response as the IV by providing a challenge to the PUF device, represented as $IV = \phi(\text{Challenge}_1)$. To enhance security, the current timestamp is retrieved using $\tau = \text{get_current_timestamp}()$.

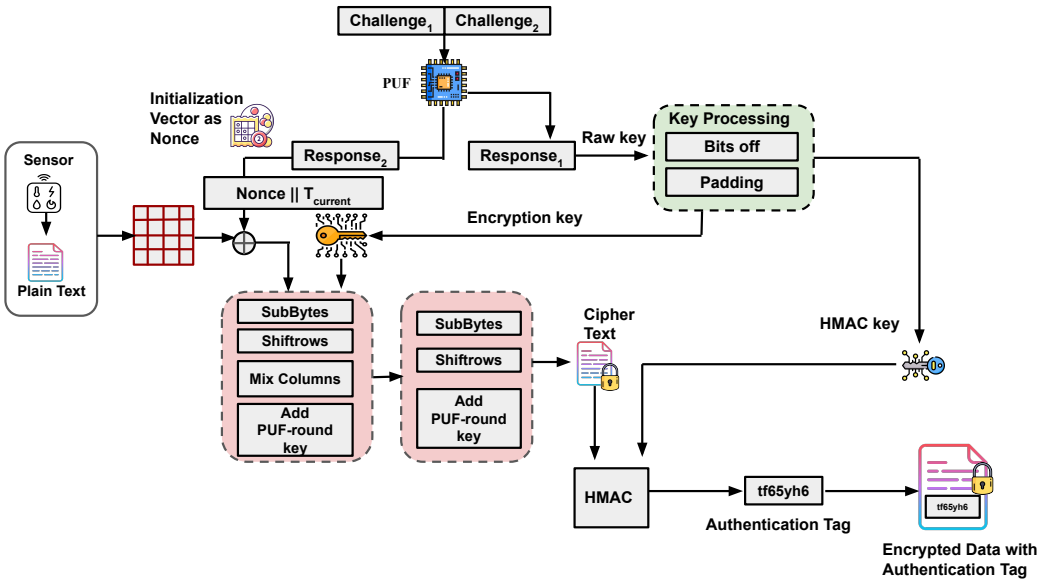

Figure 6.1: Processing of PUF-AEAD: Nonce and key generation, authentication tag generation

This timestamp is then concatenated with the generated IV to compute the nonce for the encryption process, formulated as Nonce $= (\text{IV} \| \tau)$.

Encryption/Authentication Key: The algorithm generates the raw key using the PUF device and derives the encryption and authentication keys using an error correction mechanism and hashing the raw key. First, a response is received as a raw key by passing the challenge to the PUF device, which is used to generate the encryption and authentication keys, represented as Raw_key $= \phi(\text{Challenge}_2)$. The encryption key is generated by hashing the raw key along with the randomly generated nonce α, formulated as Enc_key $= H(\text{Raw_key} \| \alpha)$. Similarly, an authentication key is generated by hashing the raw key along with the randomly generated nonce β, represented as HMAC_key $= H(\text{Raw_key} \| \beta)$.

Authenticated Encryption using AES-CTR and HMAC: The authenticated encryption process involves two primary operations: encryption using AES-CTR and authentication using HMAC. Here's a detailed description of each step:

- **Encryption Process:** The nonce is encrypted using AES-CTR mode where encryption key Enc_{key} and nonce $Nonce$ are derived using PUF device, current timestamp, and randomly generated number α.

$$\begin{aligned} CipherText &= aes_encrypt(PlainText, Enc_{key}, Nonce), \\ &= PlainText \oplus Enc_{key}(Nonce). \end{aligned} \tag{6.8}$$

  The plain text is then XORed with the encrypted nonce to produce the cipher text.

- **Authentication Process:** An authentication data string $HMAC_{data}$ is created by concatenating the cipher text, nonce, and the additional AD. This string includes all the data that needs to be authenticated to ensure it hasn't been tampered with during transmission.

$$HMAC_{data} = H((HMAC_{key} \oplus ipad)||CipherText||Nonce||AD). \tag{6.9}$$

The HMAC function is then applied to $HMAC_{data}$ using a separate HMAC key $HMAC_{key}$. HMAC involves hashing the combined input data $HMAC_{data}$ with $HMAC_{key}$ to produce an authentication tag (Tag_{auth}).

$$Tag_{auth} = HMAC(HMAC_{key}, HMAC_{data}, AD). \tag{6.10}$$

This tag is used to verify the integrity and authenticity of the message during decryption.

Algorithm 18 is divided into separate procedures for clarity. It starts by generating a unique nonce using a PUF response and a current timestamp, ensuring nonce uniqueness. Separate encryption and authentication keys are derived from another PUF response, followed by an AES-CTR encryption process that uses the nonce and encryption key. The HMAC procedure, using the authentication key, secures the ciphertext integrity and authenticity. The decryption process includes HMAC verification for authentication, and upon success, the ciphertext is decrypted using AES-CTR. This algorithm effectively integrates PUFs for enhanced security in key and nonce generation, ensuring robust encryption and authentication in cryptographic operations.

6.1.3 Proposed Cryptographic Hardware Accelerator

In the effort to enhance the security of IoV applications, a secure sensor data acquisition system is developed using an ESP32-D0WD microcontroller, RPP, and MPU6050 sensor. The objective of this system is to ensure the confidentiality and integrity of the sensor data by utilizing AES-CTR encryption and HMAC before transmission or storage. The system comprises one ESP32-D0WD and one RPP microcontroller, with the RPP used for PUF devices and the ESP32-D0WD responsible for reading data from the MPU6050 sensor and encrypting it using PUF-AEAD (AES-CTR and HMAC) algorithm. The MPU6050 sensor measures

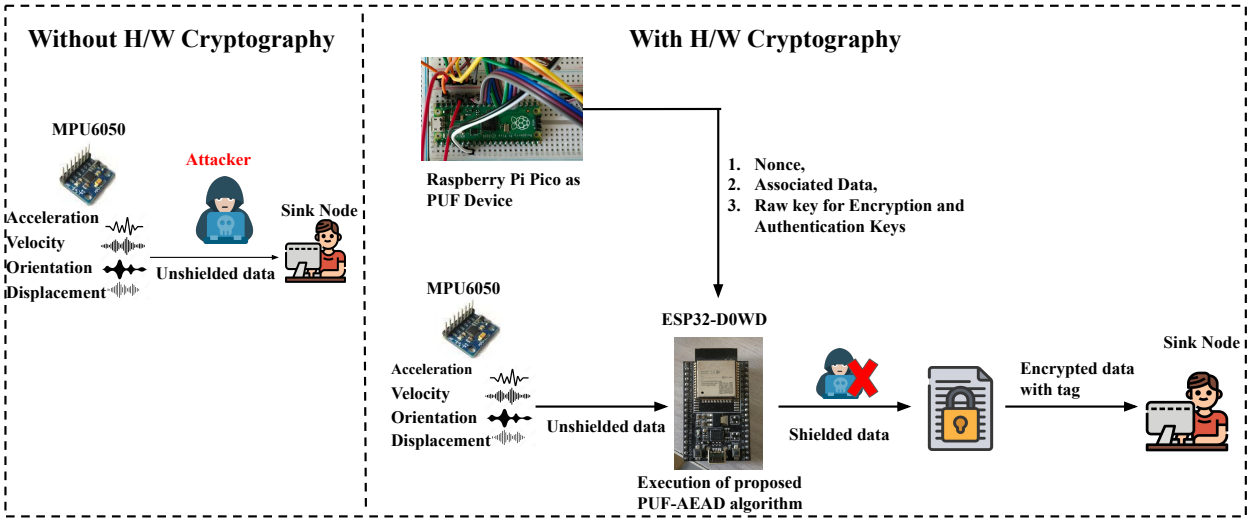

Figure 6.2: Graphical representation of securing sensor data using PUF-based hardware encryption using RPP as PUF device and ESP32-D0WD as hardware accelerator

acceleration, velocity, orientation, and displacement, providing a comprehensive dataset for various IoV applications. The hardware implementation procedure is described in Algorithm 19. The implementation begins by incorporating the required libraries, including *Adafruit_MPU6050.h*, *Adafruit_Sensor.h*, *Wire.h*, and *AESLib.h*. The 128-bit AES-CTR encryption key and a 96-bit nonce (IV) are derived using the RPP

Algorithm 18 Proposed PUF-based authenticated encryption for resource-constrained devices

Input: $Challenge_1$, $Challenge_2$, $PlainText$
Output: $CipherText$, Tag_{auth}

Procedure NONCEGENERATION($Challenge_1$) // `Generate a nonce using a PUF and the current timestamp`
$IV \leftarrow PUF(Challenge_1)$
$\tau \leftarrow get_current_timestamp()$
$Nonce \leftarrow (IV||\tau)$
return $Nonce$
end Procedure
Procedure KEYGENERATION($Challenge_2$) // `Generate encryption and HMAC keys using a PUF`
$Raw_{key} \leftarrow PUF(Challenge_2)$
$Enc_{key} \leftarrow H(Raw_{key}||\alpha)$
$HMAC_{key} \leftarrow H(Raw_{key}||\beta)$
return $Enc_{key}, HMAC_{key}$
end Procedure
Procedure AES-CTR_ENCRYPT($PlainText, Enc_{key}, Nonce$) // `Encrypt plaintext using AES-CTR mode and a PUF-modified state`
$state \leftarrow$ InitializeState($Nonce$)
$state \leftarrow$ AddRoundKey($state, Enc_{key}, 0$)
for $round \leftarrow 1,9$ **do**
    $state \leftarrow$ SubBytes($state$)
    $state \leftarrow$ ShiftRows($state$)
    $state \leftarrow$ MixColumns($state$)
    $state \leftarrow$ PUF_ModifyState($state, Enc_{key}$)
    $state \leftarrow$ AddRoundKey($state, Enc_{key}, round$)
end for
$state \leftarrow$ SubBytes($state$)
$state \leftarrow$ ShiftRows($state$)
$state \leftarrow$ AddRoundKey($state, Enc_{key}, 10$)
$CipherText \leftarrow PlainText \oplus state$
return $CipherText$
end Procedure
Procedure HMAC($HMAC_{key}, CipherText$, AD) // `Generate an authentication tag using HMAC`
$opad \leftarrow Inpad$
$ipad \leftarrow Outpad$
$HMAC_{data} \leftarrow H((HMAC_{key} \oplus ipad)||CipherText||Nonce||AD)$
$Tag_{auth} \leftarrow H((HMAC_{key} \oplus opad)||HMAC_{data})$
return Tag_{auth}
end Procedure
Procedure MODIFIED_ENCRYPT(M) // `Encrypt and authenticate the plaintext message`
$(Enc_{key}, HMAC_{key}) \leftarrow$ KEYGENERATION()
$Nonce \leftarrow$ NONCEGENERATION()
$C \leftarrow$ AES-CTR_Encrypt($PlainText, Enc_{key}, Nonce$)
$Tag_{auth} \leftarrow$ HMAC($HMAC_{key}, C, AD$)
$\tau_{send} \leftarrow get_current_timestamp()$
return $(C, Tag_{auth}, \tau_{send})$
end Procedure
Procedure MODIFIED_DECRYPT(C, Tag_{auth}) // `Decrypt and authenticate the ciphertext message`
$(Enc_{key}, HMAC_{key}) \leftarrow$ KEYGENERATION()
if HMAC($HMAC_{key}, C$) $\neq Tag_{auth}$ **then**
    return "Authentication Failed"
end if
$PlainText \leftarrow$ AES-CTR_Decrypt(C, Enc_{key})
return $PlainText$
end Procedure

PUF device. In the *setup()* function, the MPU6050 sensor is initialized with a specified accelerometer range, gyro range, and filter bandwidth. The sensor data is read and processed within the *getSensorDataString()* function, which combines acceleration, velocity, orientation, and displacement measurements into a formatted string. To encrypt the sensor data, the *encryptSensorData()* function is implemented. This function accepts a character array containing the sensor data and an IV as input parameters. The sensor data string is encrypted using the AES-CTR encryption algorithm provided by the AESLib library, and the resulting ciphertext is returned as a String object. In the main loop of the program, sensor data is continuously acquired using the *getSensorDataString()* function, converted to a character array, and encrypted using the *encryptSensorData()* function. The encrypted sensor data is then printed to the serial monitor, showcasing the secure data acquisition and encryption process. By employing the ESP32-D0WD microcontroller and AES-CTR encryption, a secure sensor data acquisition system for IoV applications is successfully developed. This approach ensures the confidentiality and integrity of the sensor data, paving the way for more robust and secure IoV deployments. The graphical representation of the importance of the proposed method is demonstrated in Figure 6.2, highlighting the impact of hardware-level encryption on sensor data can be observed.

Algorithm 19 Proposed resource-constraints data acquisition and encryption algorithm

Input: $Sensordata(a,g,temp), iv, Enc_{key}$
Output: Encrypted string as: *encrypted*

Procedure SETUP // `Initialize serial communication and sensor settings`
Serial.begin(115200)
Initialize *mpu*
$mpu.setAccelerometerRange(MPU6050_RANGE_8_G)$
$mpu.setGyroRange(MPU6050_RANGE_500_DEG)$
$mpu.setFilterBandwidth(MPU6050_BAND_5_HZ)$
Display sensor settings on serial monitor
end Procedure

Procedure GETSENSORDATASTRING // `Acquire sensor data and concatenate into a single string`
$a,g,temp \leftarrow$ sensor events from *mpu*
$dataString \leftarrow$ Concatenate$(a,g,temp)$
return *dataString*
end Procedure

Procedure ENCRYPTSENSORDATA(msg, iv) // `Encrypt the sensor data using AES-CTR mode`
$msgLen \leftarrow \|msg\|$
$encrypted \leftarrow Null, \quad \|encrypted\| = 2 \cdot msgLen$
$encrypted \leftarrow$ AES-CTR_Encrypt(msg, Enc_{key}, iv)
return *encrypted*
end Procedure

Procedure LOOP // `Main loop to continuously acquire and encrypt sensor data`
$sensorData \leftarrow getSensorDataString()$
$dataCharArr \leftarrow$ toCharArray$(sensorData)$
$encryptedSensorData \leftarrow encryptSensorData(dataCharArr, aes_iv)$
Serial.println$(encryptedSensorData)$
delay(5000)
end Procedure

6.2 SECURITY ANALYSIS

In this section, we have provided a theoretical security proof against the considered design goals including nonce-reuse attacks, replay and man-in-the-middle attacks, data integrity, authentication, key management, physical attacks, and device theft. The proposed method is also compared with the existing state-of-the-art schemes[160, 161], and a detailed comparison is shown in Table 6.1.

Nonce reuse resistant: The proposed modified algorithm provides nonce reuse resistance because each nonce is the unique combination of the PUF-generated IV and the current timestamp. The following step of the proposed algorithm guarantees to reduce the risk of nonce reuse.

$$IV = PUF(Challenge_1).$$

$$\tau = get_current_timestamp().$$

$$Nonce = (IV||\tau).$$

Table 6.1: Security comparison of PUF-AEAD with state-of-the-art methods based on different primitives

Primitives	**[160]**	**[161]**	**PUF-AEAD**
Nonce-reuse resistance	○	○	●
Replay attack	●	●	●
MITM attack	●	○	●
Data integrity and authentication	◐	◐	●
Key management	○	○	●
Physical attack/ device theft	○	○	●

Replay and man-in-the-middle attack: The dynamic nature of the encryption key and nonce for each transaction makes it extremely challenging for an attacker to decrypt or forge messages. To avoid a replay attack, we use the current timestamp τ_{send} that ensures the freshness of the received encrypted data. The following equations are used to avoid the replay and MITM attacks.

$$C = \text{AES-CTR_Encrypt}(PlainText, Enc_{key}, Nonce).$$

$$Tag_{auth} = \text{HMAC}(HMAC_{key}, C, AD).$$

$$\tau_{send} = get_current_timestamp().$$

The encrypted data is now collected in the form of $(C, Tag_{auth}, \tau_{send})$.

Secures data integrity and authenticity: PUF-AEAD uses HMAC to ensure integrity, authenticity, and confidentiality. It ensures that the cipher text itself has not been altered and is from the legitimate source.

$$opad \leftarrow Outpad, ipad \leftarrow Inpad.$$

$$HMAC_{data} = H((HMAC_{key} \oplus ipad)||CipherText||Nonce||AD).$$

$$Tag_{auth} = H((HMAC_{key} \oplus opad)||HMAC_{data}).$$

Key management: PUF-generated raw keys, when hashed with different nonces, produce unique encryption and HMAC keys, enhancing security against key leakage and unauthorized duplication.

$$Raw_{key} = PUF(Challenge_2).$$

$$Enc_{key} = H(Raw_{key}||\alpha).$$
$$HMAC_{key} = H(Raw_{key}||\beta).$$

Physical attack and device theft: PUFs rely on the inherent physical characteristics of the device, making direct copying or replication infeasible. Even if a device is stolen, compromised keys used on that device won't affect the communication of other devices due to their unique random PUF outputs. This randomness can be quantified in terms of entropy, a concept from information theory. The entropy of a random variable X representing a PUF response can be mathematically expressed as:

$$Entropy(X) = -\Sigma_{i=1}^{n} P(x_i) log_b P(x_i), \tag{6.11}$$

Here, $P(x)$ is the probability of each possible output value x. Higher entropy signifies greater randomness and unpredictability, making the PUF output more difficult to replicate or predict in a physical attack.

PUF-Specific Threat Mitigation and Advanced Security Guarantees: Beyond the fundamental benefits of PUFs in eliminating static key storage, the proposed protocol incorporates several advanced security measures to address threats specific to PUF-based cryptographic systems.

- **Modeling Attack Resistance:** To prevent machine learning-based modeling attacks—where an adversary collects CRPs and attempts to predict future PUF behavior—the protocol applies a dynamic challenge obfuscation mechanism. Each PUF challenge is randomized using a session-specific nonce:

  $$C_{\text{rand}} = H(\texttt{Challenge} \oplus N_{\text{session}}) \tag{6.12}$$

  This prevents attackers from building generalized models across sessions, rendering CRP-based training ineffective.

- **Environmental Variation Resilience:** PUF responses are known to exhibit slight fluctuations under varying temperature, voltage, and aging conditions. To ensure reliable key regeneration, a fuzzy extractor combined with BCH codes is employed. Minor bit errors in the raw response are corrected through ECC mechanisms as follows:

  $$R_{\text{corrected}} = D(R_{\text{PUF}}) \tag{6.13}$$

  where $D(\cdot)$ is the BCH decoding function. In this design, BCH(63,51,2) allows up to two-bit error correction, ensuring robust key consistency even in fluctuating environments.

- **Side-Channel Attack Mitigation:** To protect against Side-Channel Attacks (SCAs) such as power analysis and timing-based leakage, the system incorporates countermeasures including power masking and constant-time execution. Noise is injected into power traces as:

  $$P_{\text{observed}}(t) = P(t) + N(t) \tag{6.14}$$

where $N(t)$ is a session-specific noise function. Additionally, operations are executed in constant clock cycles to eliminate timing-based distinguishability.

- **Tamper Detection:** In case of invasive attacks such as microprobing or fault injection, any physical alteration to the PUF circuitry modifies its analog characteristics. As a result, the tampered response deviates from the original:

$$R_{\text{tampered}} \neq R_{\text{PUF}} \tag{6.15}$$

This inherent sensitivity is used as a tamper-evidence mechanism to invalidate authentication attempts. Further, aging compensation mechanisms ensure that even long-term physical drift in silicon properties does not degrade key reliability.

6.3 EXPERIMENTAL SETUP AND PERFORMANCE ANALYSIS

In the experiment, RPP and ESP32-D0WD modules are used. The Table 6.2 contains the hardware specifications of the ESP32-D0WD micro-controller, and RPP module. The RPP module operates as a PUF device, utilizing MCU, temperature (T) and light scattering based on the LDR sensor for entropy generation. This PUF-generated unique number fulfills three crucial roles within the AEAD encryption process. Firstly, it serves as a nonce (IV) for AES-CTR encryption, providing a starting point for the encryption algorithm. Secondly, the PUF device generates a distinct random number used as the AD for AEAD, ensuring the integrity and association of supplementary information. Additionally, it generates another unique number that acts as the raw key for the generation of encryption and authentication keys, which enables authentication and integrity verification of the encrypted data. The ESP32-D0WD module is dedicated to

Table 6.2: Hardware specifications for ESP32-D0WD and Raspberry Pi Pico devices

Component	**ESP32-D0WD**	**Raspberry Pi Pico**
Micro-controller	ESP32-D0WD	RP2040
Manufacturer	Espressif systems	Raspberry Pi foundation
Processor	Xtensa LX6	Dual-core ARM Cortex-M0+
Clock speed	Up to 240 MHz	Up to 133 MHz
Flash memory	448 KB	2 MB Flash
RAM	520 KB	264 KB
GPIO pins	36	26
Analog pins	18	3
Wireless	Wi-Fi, Bluetooth	-
USB interface	Yes	Yes
Dimensions (mm)	Various	21.6 x 51mm
Price	$2 - $10	$4

encryption operations within the AEAD framework, employing AES-CTR mode and HMAC. This module serves as the encryption engine responsible for securing the data obtained from any sensors. We used an MPU6050 sensor to gather a 3-axis gyroscope and 3-axis accelerometer data for encryption. It's crucial to note that within the AEAD framework, the IV requires a 16-byte key, while the AD and encryption/ HMAC key necessitate a 32-byte key. To address this key size variation and to ensure seamless integration without parsing issues, the PUF device incorporates an encoding function using base64url. This encoding function serves dual purpose: it not only ensures compatibility and seamless integration but also assists in adjusting the key size to meet the required specifications for IV, AD, and encryption/ HMAC keys. This ensures

smooth key transitions and module compatibility, optimizing the encryption process. Additionally, serial communication operates between these two modules at 460800 bps. This high baud rate is chosen for stable and reliable communication, facilitating efficient data exchange and encryption between the ESP32-D0WD module and RPP.

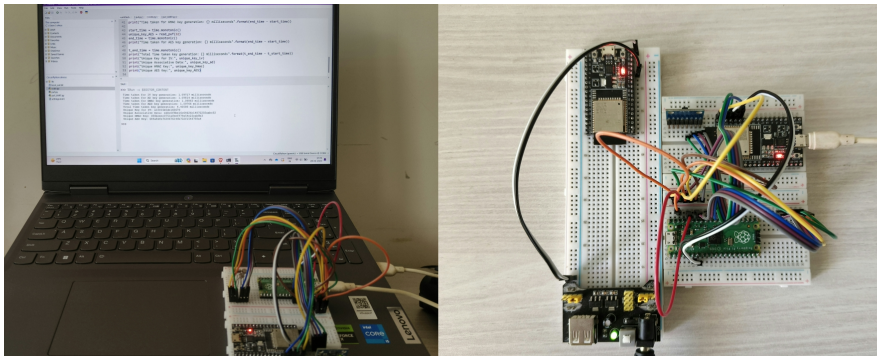

Figure 6.3: Experimental setup for proposed method using ESP32-D0WD micro-controller, RPP, MPU6050 sensor module and Linux-based OS having Arduino IDE

The experimental setup for the hardware implementation of PUF-AEAD is presented in Figure 6.3, which includes the required devices and a Linux-based operating system. The ESP32-D0WD, RPP, and MPU6050 are connected using a breadboard, providing a flexible and easy-to-use platform for establishing connections between the devices. The breadboard circuit includes connections for power, ground, and data transfer between the ESP32-D0WD and MPU6050. The ESP32-D0WD micro-controller is programmed using the Arduino IDE version 2.0.4 running on a laptop with an Intel Core i5 12450H processor, 16GB of RAM, and a 512GB SSD. The laptop operates on a Linux-based Zorin OS, providing a stable and reliable operating system environment for the experiment.

The encryption process is performed multiple times with variable data sizes such as 16 bytes, 32 bytes, 64 bytes, and so on. Figure 6.4 provides a clear visual analysis of the encryption times across different rounds measured in microseconds (μs). As observed, the time taken for encryption increases with each round, indicating a direct relationship between the complexity of the operation and the time consumed. Initially, at round 0, the times are almost identical, with encryption taking just 97 μs. However, as the rounds progress, a gradual increase in time is noticeable, with the most significant leap observed in the final round (round 7), where encryption time reaches 3650 μs. Further, using the encryption and decryption time, the energy consumption and device throughput for each round can be calculated using the following formula:

$$E = P_{max} * T_{Enc}, \tag{6.16}$$

Here, E denotes energy, P_{max} indicates the maximum power, and T_{Enc} represents the encryption time. Power is the product of voltage (V) and current (I): $P = V \times I$. Now, data size D, encryption time T_{Enc}, memory usage M, and current consumption I_{Enc} are incorporated. First, the energy consumed during the encryption process is considered. It is assumed that the energy consumption is directly proportional to the data size and encryption time: $E_{Enc} \propto D \times T_{Enc}$. Now, the energy consumed by memory usage is considered. It is assumed that the energy consumption is directly proportional to the memory usage and the time required for encryption: $E_{Mem} \propto M \times T_{Enc}$. The total energy consumed can be considered as the sum of the energy consumed during encryption and the energy consumed by memory usage: $E_{total} = E_{Enc} + E_{Mem}$. Since both E_{Enc} and E_{Mem} are proportional to T_{Enc}, it can be factored out: $E_{total} \propto (D+M) \times T_{Enc}$. This can be related

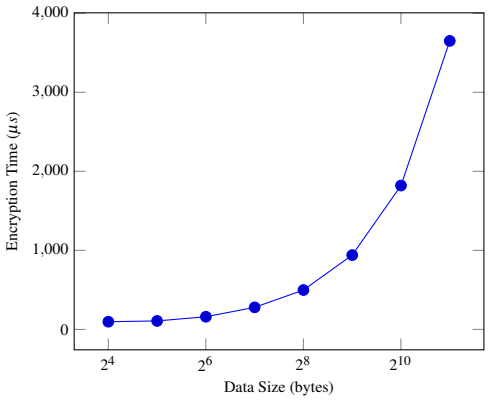

Figure 6.4: The impact of increased data size on encryption time (μs)

to the electrical energy equation by introducing the current consumption and voltage:

$$E_{total} = V \times I_{Enc} \times T_{Enc} \times (D + M). \tag{6.17}$$

Similarly, the throughput is measured as the amount of data processed in one second and is formulated as follows:

$$\text{Throughput} = \frac{D \times 8}{\left(\frac{T_{Enc}}{10^6}\right)}, \tag{6.18}$$

Here, D is the data size, and 10^6 is used to convert the encryption time from μs to seconds, since throughput is typically measured in bits per second. Table 6.3 provides detailed information about RAM utilization in

Table 6.3: Performance analysis of the proposed scheme in terms of RAM utilization, current consumption, and energy consumption

Rounds	**Data size (bits)**	**RAM (KB)**	**Current consumption (mA)**	**Energy consumption** (μj)
0	16	257	8	2.53
1	32	257	14	5.03
2	64	257	20	10.82
3	128	257	24	21.62
4	256	258	30	49.69
5	512	258	38	116.99
6	1024	259	42	252.80
7	2048	262	49	587.77

kilobytes (KB), current consumption in milliamperes (mA), and energy consumption in microjoules (μJ) for each round with different data sizes. The observations indicate that the encryption time for encrypting 2048 bytes uses only 262 KB of RAM and consumes 587.78 μJ of energy at a frequency of 240 MHz. An increase in computation power leads to higher energy consumption and requires more resources to achieve the same results. However, the proposed method is designed specifically for resource-constrained devices, emphasizing low energy consumption. Furthermore, the experimental analysis, visualized in Figure 6.5, showcases the execution times of key generation operations within the proposed algorithm using a PUF device. The figure combines four screenshots representing different executions. Each screenshot displays the unique IV, AD, HMAC key, and AES key generated, along with their corresponding execution times in milliseconds. Notably, the execution times demonstrate high consistency across all iterations. For instance, the IV generation consistently requires around 1.09 ms, highlighting the algorithm's reliability

and repeatability. The proposed method is compared with the existing PUF-based authentication scheme

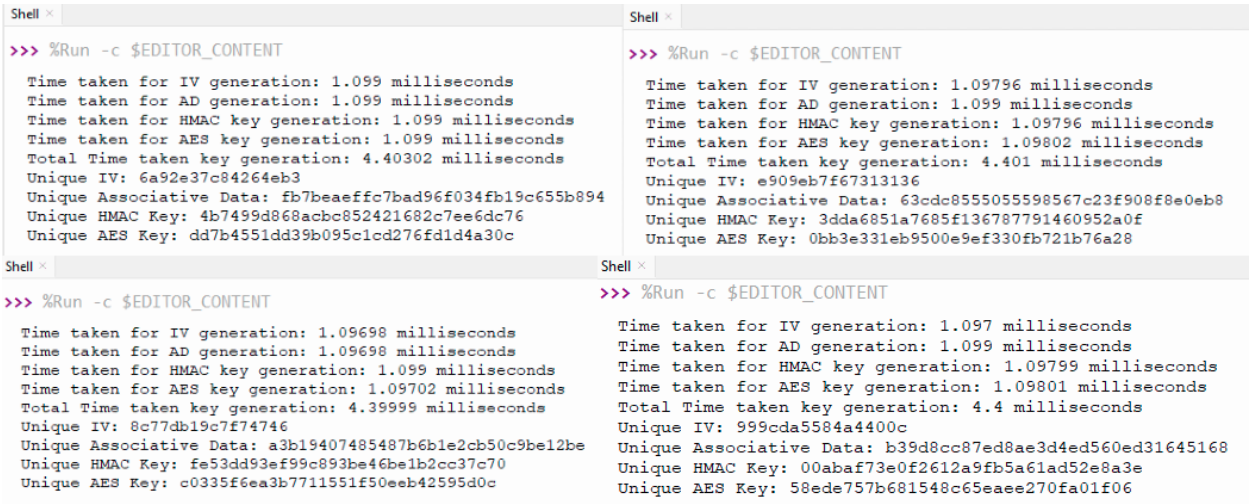

Figure 6.5: Execution times (ms) for nonce generation, HMAC, and encryption key generation

for IoMT as described in [160]. The comparison focuses solely on key generation time and key length. Table 6.4 shows that the scheme in [160] requires 800 ms to generate keys at the server and IoMT device, with a key length of only 64 bits. In contrast, the proposed method takes approximately 1.09 ms to generate a 96-bit nonce and 128-bit HMAC and AES keys.

The network performance analysis is conducted to evaluate the efficiency and reliability of the communication infrastructure within the system. Figure 6.6 illustrates the network topology and the performance

Table 6.4: Comparison of PUF-AEAD based on the key generation and PUF's statistical properties

Parameters	**[160]**	**[161]**	**PUF-AEAD**
Key generation time (ms)	800	NA	1.09
Key length (bits)	64 bits	64 bits	128 bits
PUF Uniqueness (%) (should be 50%)	48%	48.46%	49.6%
PUF Reliability (%)	1.7%	2.38%	2.67%

analysis. Each node in the network represents a combination of hardware components, including one RPP, one ESP32-D0WD, and an MPU6050 sensor. The analysis focuses on key metrics, including latency (μs), signal strength (dBm), noise (dBm), packet loss (%), and network throughput (Mbps). The network environment is simulated using the Python-based library NetworkX, where nodes represent individual devices and edges symbolize device-to-device communication channels. The size of each node indicates its power consumption, while the color gradient between nodes, ranging from red to blue, represents the strength of communication, with red indicating lower and blue indicating higher communication intensity.

1. **Latency**: The average latency observed across the network ranges from 0.075 μs to 9.654 μs. Lower latency values indicate faster response times and better real-time communication performance.

2. **Signal strength and noise**: The average signal strength is recorded between 10.156 dBm and 18.905 dBm, while the noise level ranges from 0.065 dBm to 3.379 dBm. Higher signal strength coupled with lower noise levels contribute to better communication quality and reliability.

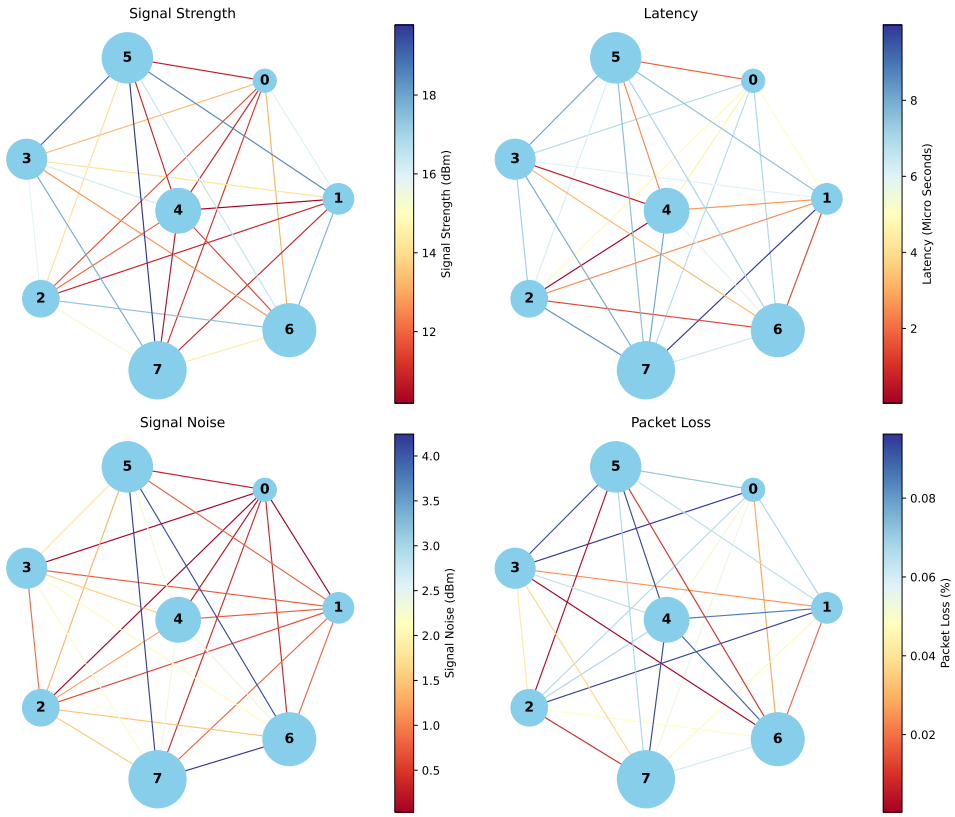

Figure 6.6: Network performance analysis in terms of signal strength, latency, single noise and packet loss

3. **Packet loss**: Packet loss percentages are minimal, ranging from 0.002% to 0.094%. Lower packet loss rates indicate robust data transmission and minimal data loss during communication.

4. **Network throughput**: The network throughput varies between 91.57 Mbps and 149.55 Mbps. Higher throughput values reflect the network's ability to transfer data effectively within a given time frame.

The network performance analysis demonstrates the effectiveness of the communication infrastructure in facilitating efficient and reliable data exchange among devices. The low latency, coupled with high signal strength and throughput, indicates a well-optimized device capable of effectively supporting various communication and services. Further optimizations can be explored to enhance performance and scalability for future deployments.

6.4 SUMMARY OF THE WORK

The chapter discusses the integration of PUFs with the AEAD algorithm to enhance the security of the IoV networks. The primary focus is on the proposed PUF-AEAD algorithm, which combines AES-CTR encryption and HMAC authentication using a nonce and associated data generated by a PUF device. This integration aims to address the resource constraints of IoV devices while ensuring robust security. The key

contributions of this research include the modification of AEAD with the PUF function and the implementation and performance analysis of the proposed algorithm. PUF-AEAD incorporates PUFs to generate unique and unpredictable nonces and keys, enhancing security against potential attacks. This modification addresses nonce reuse issues and introduces a dynamic key generation process. The algorithm is implemented using Raspberry Pi Pico and ESP32-D0WD microcontrollers, along with an MPU6050 gyroscope sensor. The analysis demonstrates significant improvements in encryption time, RAM utilization, current consumption, energy consumption, and overall security.

The proposed scheme leverages PUFs to dynamically generate secret cryptographic parameters, thus eliminating the need for static key storage in memory. This approach significantly enhances security by preventing key exposure via memory dumps, physical probing, or cloning attacks. The cryptographic key is derived at runtime from the PUF response using a fuzzy extractor and error correction codes (BCH), ensuring stability even under environmental variations. Additionally, the use of session-based nonces derived from PUF responses prevents key reuse and guarantees forward secrecy. To address potential weaknesses of PUFs, such as machine learning-based modeling, the protocol introduces challenge obfuscation using session-specific nonces, making it difficult for an attacker to generalize response behavior. Environmental inconsistencies and device aging are managed through ECC-based correction mechanisms and adaptive thresholding. Furthermore, power analysis and timing attacks are mitigated through side-channel noise injection and constant-cycle execution. Any physical tampering of the PUF circuit disrupts the electrical properties and leads to incorrect responses, enabling tamper detection. These combined countermeasures ensure the secure and reliable use of PUFs, even in adversarial or uncontrolled deployment environments.

The chapter provides a detailed analysis of nonce and key generation, where the nonce is generated using a PUF response and the current timestamp to ensure uniqueness. Encryption and authentication keys are derived from the raw key generated by the PUF device, adding an extra layer of security. The authenticated encryption process involves AES-CTR mode, where the plaintext is XORed with the encrypted nonce, and HMAC is used for authentication, ensuring the integrity and authenticity of the ciphertext. The security analysis shows that the proposed scheme is resilient against nonce reuse, replay attacks, man-in-the-middle attacks, and physical attacks. PUFs ensure that keys are tied to the unique physical characteristics of the device, making them difficult to clone or extract. The experimental setup involves the ESP32-D0WD and Raspberry Pi Pico, using the MPU6050 sensor for data collection. The system is programmed using the Arduino IDE on a Linux-based OS. Performance metrics include encryption time, RAM utilization, current consumption, and energy consumption. The results show that the proposed method is efficient and suitable for resource-constrained IoV devices. It consumes only 587 μJ energy to encrypt 1028 bytes of data. PUF-AEAD takes 1.09 ms to generate key of 128 bits length. PUF function used in PUF-AEAD provides 49.6% of uniqueness and 2.67% of reliability. Additionally, the network performance analysis evaluates the efficiency and reliability of the communication infrastructure within the system using key metrics such as latency, signal strength, noise, packet loss, and throughput. The analysis indicates a well-optimized communication infrastructure capable of supporting various IoV applications. Overall, the chapter presents a comprehensive approach to enhancing the security of IoV networks through the integration of PUFs and AEAD, demonstrating both theoretical and practical improvements in performance and security.