QuantumRISC

Work Package 6, Deliverables 6.1 and 6.2

Evaluation, integration and demonstration of use cases

Version	1.0
Project Coordination	Fraunhofer Institute for Secure Information Technology
Date of preparation	December 4, 2023

SPONSORED BY THE

Federal Ministry
of Education
and Research

Authors

- Continental AG:
  - Maurice Heymann
  - Burak Selcuk
- Elektrobit Automotive GmbH:
  - Florian Grießer
  - Anahita Hamidi
  - Hannes Hennig
  - Lars Müller
- MTG AG:
  - Evangelos Karatsiolis

Projektkoordination
Norman Lahr
Fraunhofer Institute for Secure Information Technology
Advanced Cryptographic Engineering
Rheinstr. 75
D-64295 Darmstadt
Deutschland

Telefon +49 6151 869100
Fax +49 6151 869224
Mail norman.lahr@sit.fraunhofer.de

Contents

1. Executive Summary

The QuantumRISC project is funded by the German Federal Ministry of Education and Research (BMBF). Goal of the project is in short to “bring Post-Quantum Computing (PQC) from theory into pratice”. The project is divided into the following six work packages (WP) to analyse and improve schemes, and to identify and develop software, hardware acceleration, and feasible strategies for the use of post-quantum cryptographic applications on embedded devices:

- WP 1: Use Cases and Requirements
- WP 2: Analysis and Optimization of PQC Schemes
- WP 3: Development of Software Libraries
- WP 4: Development of Hardware Accelerators
- WP 5: Software-Hardware Co-Design
- WP 6: Demonstrator of Use Cases

This report focuses on WP 6 describing the implementation of use cases in automotive embedded controllers and the demonstrator architecture itself. The actual implementation was performed in the AUTOSAR classic environment, which is the de-facto standard for automotive embedded devices.

2. Terminology

This section provides an overview of abbreviations and terms used in this document.

Abbreviation	Name	Explanation
AUTOSAR	AUTomotive Open System ARchitecture	De-facto standard for automotive basic software.
CAN	Controller Area Network	Automotive serial bus standard.
CDO	Crypto Driver Object	The AUTOSAR Crypto Drivers allow defining of different Crypto Driver Objects (i.e. AES accelerator, SW component, etc), which shall be used for concurrent requests in different buffers [9].
CryIf	Crypto Interface	AUTOSAR module CryIf interfaces the underlaying Crypto modules.
Crypto	Crypto MCAL Module	AUTOSAR module Crypto implements cryptographic routines in software or hardware.
Csm	Crypto Service Manager	AUTOSAR module Csm manages the access to the underlaying CryIf and Crypto modules.
ECU	Electronic Control Unit	Computing unit which handles specific systems, like engine control, brake control, in a vehicle.
FlexRay	FlexRay	Automotive serial bus system.
KEM	Key Encapsulation Mechanism	Cryptographic primitive that allows to securely transmit a key.
LIN	Local Interconnect Network	Automotive serial bus system.
MCAL	Microcontroller Abstraction Layer	AUTOSAR layer classifying hardware devices drivers.
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen	Standard for automotive operating system, communications stack, and network management protocol (engl. “Open Systems and their Interfaces for the Electronics in Motor Vehicles”).

Table 2.1.: Overview of the terminology used in this report.

3. Introduction

3.1. Document structure

This report documents the results of work package 6 (WP 6) of the QuantumRISC project. The WP 6 demonstrates use cases identified and specified by the previous work packages. The selection of the demonstrated use cases is described in Section 3.2 and the embedded platforms used for demonstration are discussed in Section 3.4.

The WP6 consists of two deliverables:

- D 6.1 - Documentation of use cases and demonstrator
- D 6.2 - Documentation of the algorithm evaluation

The deliverable D 6.1 in Chapter 5 describes the implementation of use cases and their demonstration, while target specific implementation details are given. Chapter 6 discusses and evaluates the performance of the used algorithms, whereas the feasibility of the usage in the embedded automotive environment is considered.

3.2. Selection of use cases

This chapter delves into a range of automotive use cases, which are not only pertinent to the automotive industry but also discussed in-depth as part of Work Package 1 of the QuantumRISC Project. By examining these use cases, we aim to gain valuable insights into the diverse applications and challenges encountered within the automotive domain.

Secure Session Establishment

In the automotive industry, the effective management and control of numerous devices connected to a backend system over IP-based networks, be it local, wide-area, or mobile, necessitate the establishment of a secure channel to ensure confidential and authentic communication. This is of paramount importance for various crucial use cases, such as fleet management of cars or trucks, the collection and aggregation of sensor data. These use cases heavily rely on IP-based communication protocols due to their inherent scalability and ability to accommodate the required services. By employing secure channels, automotive companies can safeguard sensitive information, ensure the integrity and authenticity of data, and mitigate potential threats posed by unauthorized access or malicious activities. This is particularly critical in industries where the consequences of compromised communication could lead to severe financial losses, safety hazards, or disruptions in critical infrastructure. Secure communication

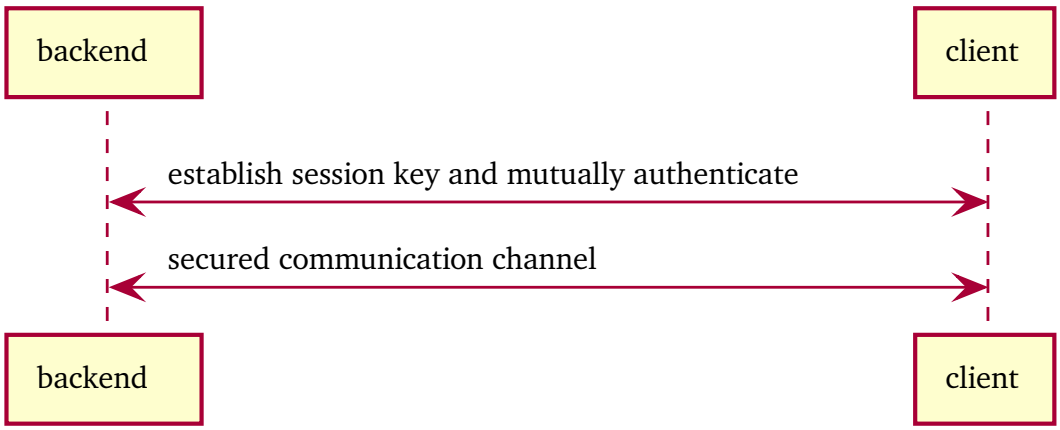


Figure 3.1.: Sequence diagram of the Session-based Secure Channel use case [21].

channels facilitate the secure transmission of data between backend systems and connected devices, allowing for effective management, monitoring, and control of various automotive applications. They enable real-time access to vehicle telemetry, remote diagnostics, software updates, and control commands, while maintaining data confidentiality, authentication, and integrity. In summary, establishing secure channels in the automotive domain is vital to ensure confidential and authentic communication, protect sensitive data, and enable efficient management and control of connected devices. By leveraging scalable IP-based communication protocols, automotive companies can effectively address the requirements of diverse use cases, providing a robust foundation for reliable and secure operations in the automotive world.

Secure Software Download

The use case of a secure download mechanism plays a vital role in ensuring that solely authenticated software is flashed onto an Electronic Control Unit (ECU). By utilizing asymmetric signatures, the mechanism effectively prevents attackers from flashing manipulated software onto an ECU. The signature, accompanied by the software to be flashed, undergoes the process of being signed with a private key within a trusted environment, typically the backend. The corresponding public key, essential for the verification process, must be enrolled in the ECU and securely stored in tamper-protected storage or has to be encrypted with a secured symmetric key. In the process of flashing software, the ECU typically initiates a dedicated bootloader, which serves as an intermediary stage. The bootloader, situated within the ECU, receives the software from a predetermined source, such as the communication unit or the onboard diagnosis interface. Subsequently, the software is downloaded into the ECU's memory, and the boot flags are adjusted accordingly. As an integral part of the bootloader, the secure software download mechanism performs a crucial signature verification step before proceeding with the flashing process. Only upon successful signature verification, indicating the authenticity and integrity of the software, does the bootloader proceed to flash the software onto the ECU. By implementing the secure download mechanism, which leverages asymmetric signatures and integrates it within the bootloader, the

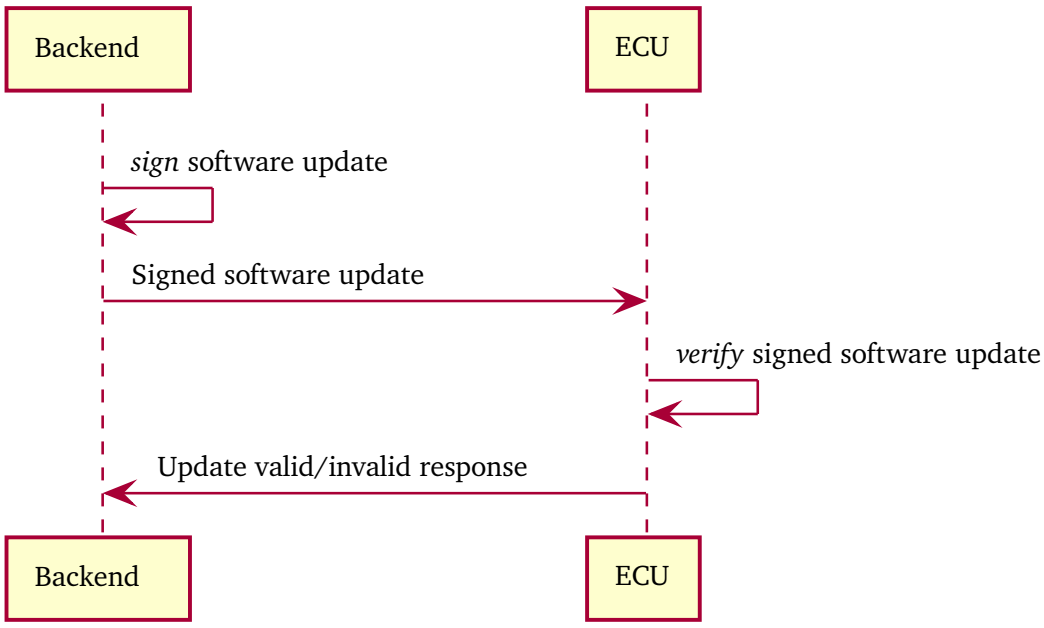


Figure 3.2.: Sequence diagram of the secure software download use case [21].

research ensures that exclusively authenticated software, validated through thorough signature verification, is flashed onto the ECU. This mechanism serves as a fundamental security measure, effectively mitigating the risk of manipulated software infiltration and upholding the integrity and reliability of the ECU's functionality.

Secure Access Control

Secure access control is a critical requirement in automotive ECUs to limit access to specific services and data, such as flashing operations. Only authorized entities, including manufacturers and car service stations, should be able to unlock these operations.

By implementing challenge-response schemes, only authorized entities can access specific services and data, safeguarding against unauthorized use and potential security breaches. Furthermore, the adoption of these cryptographic techniques, such as signatures, MACs, and encryption, enhances overall system security by ensuring the authenticity and integrity of communication. Additionally, the integration of these schemes with industry-standard architectures like AUTOSAR facilitates compatibility and promotes the adoption of secure access control mechanisms in automotive systems. Implementing and evaluating these schemes in a demonstrator allows for practical validation of their effectiveness and suitability in real-world automotive environments, paving the way for enhanced security measures in future automotive systems.

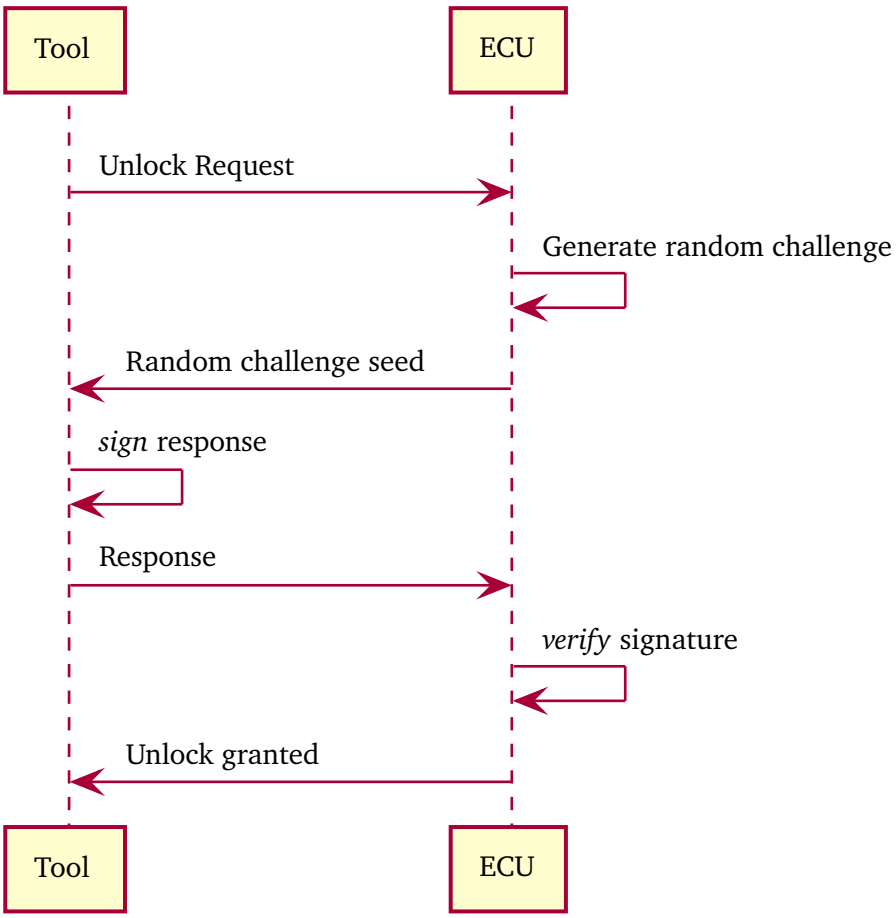


Figure 3.3.: Sequence diagram of the secure unlock use case [21].

3.3. Selection of PQC Algorithm

In this section, we delve into the selected post-quantum cryptography (PQC) algorithms, CRYSTALS-Dilithium and CRYSTALS-Kyber, and provide an overview of their functionality. CRYSTALS-Dilithium is a PQC signature scheme that offers secure digital signatures and CRYSTALS-Kyber is a PQC key encapsulation mechanism (KEM) for the secure establishment of a shared secret. Further details on the comparison of these algorithms can be found in work package 2 [19], providing insights into their performance, computational requirements, and resistance against potential attacks.

3.3.1. CRYSTALS-Dilithium - Signature Scheme

CRYSTALS-Dilithium [13] is a post-quantum cryptographic algorithm that belongs to the family of lattice-based cryptography [17]. It is designed to provide secure digital signatures that are resistant to attacks by both classical and quantum computers. The algorithm relies on the mathematical principles of lattice theory and hardness assumptions related to certain computational problems in lattices. In CRYSTALS-Dilithium, the underlying mathematical structure is a high-dimensional lattice, which

can be thought of as a periodic arrangement of points in space. These lattices possess unique mathematical properties that make certain computational problems, such as the Shortest Vector Problem (SVP) and the Learning With Errors (LWE) problem, computationally difficult to solve [13]. The security of CRYSTALS-Dilithium is based on the hardness of these lattice problems. Specifically, the algorithm utilizes a carefully constructed lattice and exploits the difficulty of finding short lattice vectors within this lattice. By introducing noise and randomness into the lattice, CRYSTALS-Dilithium creates a mathematical puzzle that can only be solved with knowledge of a secret key. The three procedures key generation, signing and verification are shortly introduced. For further information and more in-depth analysis, we refer readers to the paper titled "CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation" [13].

Key Generation
The Dilithium key generation process begins with the random generation of vectors ρ and K. These vectors play a crucial role in subsequent steps. With ρ generates a $k \times l$ matrix A each of whose entries is a polynomial in the ring $R_q = Z_q[X]/(X_n + 1)$. Afterwards, the algorithm samples random secret key vectors s_1 and s_2 [13]. Each coefficient of these vectors is an element of R_q with small coefficients of size at most η. Finally, the second part of the public key is computed as $t = As_1 + s_2$. All algebraic operations in this scheme are assumed to be over the polynomial ring R_q. Usually the calculation As_1 is accomplished with an inverse NTT (Number Theoretical Transform) operation. To accomplish the size reduction, the key generation algorithm outputs $(t_1, t_0) := Power2Round_q(t, d)$ as the public key instead of t. Further reduction can now be achieved with a CRH (Collision resistant hash) of ρ and t_1. The functions ExpandMask, ExpandA and CRH are usually implemented by a SHAKE-128 or SHAKE-256 [20] hashing algorithm. For the algorithm parameters see Table 3.1.

Algorithm 1 Key Generation [13]

1: $\rho \leftarrow \{0,1\}^{256}$
2: $K \leftarrow \{0,1\}^{256}$
3: $(s_1, s_2) \leftarrow S_\eta^l \times S_\eta^k$
4: $A \in R_q^{k \times l} := ExpandA(\rho)$
5: $t := As_1 + s_2$
6: $(t_1, t_0) := Power2Round_q(t, d)$
7: $tr \in \{0,1\}^{384} := CRH(\rho \parallel t_1)$
8: **return** $(pk = (\rho, t_1), sk = (\rho, K, tr, s_1, s_2, t_0))$

Signing Procedure
In the signing algorithm, a masking vector of polynomials, denoted as y, is generated. The coefficients of y are chosen to be less than γ_1. The parameter γ_1 is strategically determined to ensure that the resulting signature does not expose the secret key (achieving zero-knowledge), while still maintaining resistance against forgery. By computing Ay, the "high-order" bits of the coefficients in this vector are extracted and represented as w_1. Each coefficient w in Ay can be expressed in a canonical form

as $w = w_1 x 2\gamma_2 + w_0$, where the absolute value of w_0 is $\pm\gamma_2$. The challenge, denoted as c, is created by hashing the message and w_1. The resulting polynomial c is an R_q polynomial with exactly 60 ± 1's and the remaining coefficients set to 0. This distribution is chosen to ensure that c has a small norm and originates from a domain larger than 2^{256}. The potential signature, denoted as z, is then computed as the sum of y and the element-wise product of c and the secret key s_1. However, directly outputting z at this stage would compromise the security of the signature scheme by revealing the secret key. To address this issue, rejection sampling is employed. The parameter β is set as the maximum possible coefficient of cs_i. Since c has 60 ± 1's and the maximum coefficient in s_i is eta, it is evident that $\beta \pm 60\eta$. If any coefficient of z exceeds $\gamma_1 - \beta$, the signing procedure is rejected, and a restart is initiated. Additionally, if any coefficient in the low-order bits of $Az - ct$ exceeds $\gamma_2 - \beta$, the procedure is also restarted. The first check ensures security, while the second is necessary for both security and correctness. The signing procedure iterates through a while loop until both conditions are met. The parameter values are carefully selected to ensure that the expected number of repetitions is reasonably low, typically ranging between 4 and 7, depending on the specific implementations.

Algorithm 2 Signing Procedure [13]

Input: (sk, M)
$A \in R_q^{k \times l} := ExpandA(\rho)$
$\mu \in \{0,1\}^{384} := CRH(tr \parallel M)$
$\kappa := 0$
$(z, h) :=\perp$
$\rho' \leftarrow \{0,1\}^{384} := CRH(K \parallel \mu)$ (or $\rho' \leftarrow \{0,1\}^{384}$ for randomized signing)
while $(z, h) =\perp$ **do**
    $y \in S^l_{\gamma_1 - 1} := ExpandMask(\rho', \kappa)$
    $w := Ay$
    $w_1 := HighBits_q(w, 2\gamma_2)$
    $c \in B_{60} :=$ SHAKE-256$(\mu \parallel w_1)$
    $z := y + cs_1$
    $(r_1, r_0) := Decompose_q(w - cs_2, 2\gamma_2)$
    if $\|z\|_\infty \geq \gamma_1 - \beta$ or $\|r_0\|_\infty \geq \gamma_2 - \beta$ or $r_1 \neq w_1$ **then**
        $(z, h) :=\perp$
    else
        $h := MakeHint_q(-ct_0, w - cs_2 + ct_0, 2\gamma_2)$
        if $\|ct_0\|_\infty \geq \gamma_2$ or the # of 1's in h is greater than ω **then**
            $(z, h) :=\perp$
        end if
    end if
    $\kappa := \kappa + 1$
end while
return $\sigma = (z, h, c)$

Verification Procedure

The verifier initially computes w_1' as the "high-order" bits of Az - ct and proceeds to accept the signature if all coefficients of z are less than $\gamma_1 - \beta$, and if c matches the hash of the message and w_1'. Let us examine why the verification process is effective, specifically regarding the equality $HighBits(Az - ct, 2\gamma_2) = HighBits(Ay, 2\gamma_2)$.

$$HighBits(Ay, 2\gamma_2) = HighBits(Ay - cs_2, 2\gamma_2)$$

The key observation is that $Az - ct = Ay - cs_2$. Thus, our main objective is to demonstrate that $\|LowBits(Ay = cs_2, 2\gamma 2)\|_\infty < \gamma_2 = \beta$ for a valid signature. Given that the coefficients of cs_2 are smaller than β, adding cs_2 does not introduce any carries that would increase the magnitude of low-order coefficients to at least γ_2. As a result, the Equation holds true, ensuring correct verification of the signature.

Algorithm 3 Verify Procedure [13]

Input: (pk, M, $\sigma = (z, h, c)$)

1: $A \in R_q^{k \times l} := ExpandA(\rho)$
2: $\mu \in \{0,1\}^{384} := CRH(CRH(\rho \parallel t_1) \parallel M)$
3: $w_1' := UseHint_q(h, Az - ct_1 \cdot 2^d, 2\gamma_2)$
4: **return** $\|z\|_\infty < \gamma_1 - \beta$ and c = SHAKE-256$(\mu \parallel w_1')$ and # of 1's in h is $\leq w$

Public key	1184 (Bytes)
Secret key	2800 (Bytes)
Signature	2044 (Bytes)
#Signatures	∞
q	8380417
d	14
γ_1	$(q-1)/16$
γ_2	$\gamma_1/2$
(k, l)	$(4, 3)$
η	6
β	325

Table 3.1.: Parameter of Dilithium2 (medium).

3.3.2. CRYSTALS-Kyber - Key encapsulation mechanism (KEM)

CRYSTALS-Kyber [4] is a PQC key encapsulation mechanism that also belongs to the family of lattice-based cryptography like Dilithium. It is designed to provide a secure establishment of a shared secret and encryption functionalities in the presence of potential quantum adversaries. The algorithm utilizes mathematical structures called lattices and relies on the hardness of solving the learning-with-errors problem in module lattices (MLWE [16]). CRYSTALS-Kyber offers a balance between security and efficiency, making it suitable for implementation in resource-constrained environments.

The algorithm of Kyber is split into two layers. The lower layer of the algorithm consists of a public key encryption scheme (PKE) containing the functions for the key generation (KeyGen), encryption (Enc), and decryption (Dec). The upper layer of Kyber implements the KEM with the three analog functions. Therefore, the function PKE.Enc is the encryption function of the public key encryption scheme and KEM.Enc is the encapsulation function of the KEM. In the following we resolved the dependencies from the KEM layer to the PKE layer by implementing the PKE algorithm in the depicted KEM algorithm denoted by a comment on the right side in Algorithm 4, 5, and 6.

Key Generation

The simplified key generation of Kyber is shown in Algorithm 4. The secret key sk consists of a vector s of polynomials with small coefficients, randomly chosen. The public key pk consists of two components, a matrix of random polynomials denoted as A and a vector of polynomials represented as t. The matrix A is generated using a random seed ρ and the extendable output function (XOF) SHAKE-128. To compute the polynomial vector t, an additional error vector e is necessary. This error vector also contains polynomials with small coefficients. By performing a matrix-vector multiplication and addition $t = A \circ s + e$, we can obtain the value of t. The sampling of vector s and e consists of a CBD (Centered Binomial Distribution) function [4] and a SHAKE-256 operation used as a pseudo random function (PRF). The *Encode* function refers to the serialization of polynomials. For the parameters k, q, η_1, η_2 see Table 3.2. The security lies in the difficulty of reconstructing the secret value of s from the given pair (A, t). In fact, the recovery of s would require an attacker to solve the module-learning-with-errors (MLWE) problem, upon which this system is constructed. The MLWE problem is anticipated to be challenging, even for quantum computers, which is precisely why it is utilized in PQC.

Algorithm 4 KEM.KeyGen [4]

Output: Public key pk
Output: Secret key sk

1: $z \leftarrow \{0,1\}^{256}$
2: $d \leftarrow \{0,1\}^{256}$ ▷ Start PKE.KeyGen
3: $(\rho, \sigma) :=$ SHA3-512(d)
4: Generate matrix $\hat{A} \in R_q^{k \times k}$ in NTT domain
5: Sample $s \in R_q^k$ from B_{η_1}
6: Sample $e \in R_q^k$ from B_{η_1}
7: $\hat{s} := NTT(s)$
8: $\hat{e} := NTT(e)$
9: $\hat{t} := \hat{A} \circ \hat{s} + \hat{e}$
10: $pk := (Encode_{12}(\hat{t} \mod^+ q) \parallel \rho)$
11: $sk' := Encode_{12}(\hat{s} \mod^+ q)$ ▷ End PKE.KeyGen
12: $sk := (sk' \parallel pk \parallel$ SHA3-256$(pk) \parallel z)$
13: **return** (pk, sk)

Encapsulation Procedure

In the encapsulation step, first a random message m gets generated and is then encrypted using the underlying encryption function PKE.Enc of Kyber. Similar to other public key encryption systems, the message is encrypted using the public key pk and the resulting ciphertext c is decrypted using the secret key. The simplified encapsulation scheme is shown in Algorithm 5. For the encryption, first the matrix A gets generated using the seed ρ obtained by the public key. Then we need to generate some error and noise vectors r and e_1 and the error polynomial e_2, which are needed for the encryption and are sampled just like in the key generation Algorithm 4. The polynomials of r, e_1, and e_2 possess small coefficients, thus resulting in small errors. These random errors terms are freshly generated anew each time. The resulting ciphertext consists of the polynomial vector u and the polynomial v. The calculation of polynomial vector $u = A^T r + e_1$ adds noise to the matrix A. In the calculation of polynomial $v = t^T r + e_2 + Decompress_q(m, 1)$ the message m gets encrypted using t from the public key and some error terms. The $Compress$ and $Decompress$ functions of the algorithm are responsible for the scaling of the coefficients of the polynomials. This is done to perform the LWE error correction during the encryption and decryption, and to discard low-order bits in the ciphertext to reduce the size of the ciphertext [4]. During the encryption phase, the function $Decompress$ is applied to the message m to create error tolerance gaps prior to the adding of errors. The $Decode$ function refers to the deserialization of byte arrays to polynomials. The operations take place with NTT (number-theoretic transform) to perform efficient multiplications in R_q [4]. The shared key K is calculated using a chain of hash computations depended on the secret message m, the public key pk, and the ciphertext c. For the parameters n, k, et al. see Table 3.2.

Algorithm 5 KEM.Enc [4]

Input: Public key pk
Output: Ciphertext c
Output: Shared key K

1: $m \leftarrow \{0,1\}^{256}$
2: $m \leftarrow$ SHA3-256(m)
3: $(\overline{K}, r) :=$ SHA3-512$(m \parallel$ SHA3-256$(pk))$
4: $\hat{t} := Decode_{12}(pk)$ ▷ Start PKE.Enc
5: $\rho := pk + 12 \cdot k \cdot n/8$
6: Generate matrix $\hat{A} \in R_q^{k \times k}$ in NTT domain
7: Sample $r \in R_q^k$ from B_{η_1}
8: Sample $e_1 \in R_q^k$ from B_{η_2}
9: Sample $e_2 \in R_q$ from B_{η_2}
10: $\hat{r} := NTT(r)$
11: $u := NTT^{-1}(\hat{A}^T \circ \hat{r}) + e_1$
12: $v := NTT^{-1}(\hat{t}^T \circ \hat{r}) + e_2 + Decompress_q(Decode_1(m), 1)$
13: $c_1 := Encode_{d_u}(Compress_q(u, d_u))$
14: $c_2 := Encode_{d_v}(Compress_q(v, d_v))$ ▷ End PKE.Enc
15: $c := c_1 \parallel c_2$
16: $K :=$ SHAKE-256$(\overline{K} \parallel$ SHA3-256$(c))$
17: **return** (c, K)

Decapsulation Procedure

In the decapsulation step, the ciphertext c gets decrypted with the secret key sk using the underlying decryption function PKE.Dec of Kyber. The simplified decapsulation scheme is shown in Algorithm 6. For the decryption, first the polynomial vector u and the polynomial v are extracted from the ciphertext c, and the secret polynomial vector s is extracted from the secret key sk. The decryption is achieved by subtracting the dot product of the transpose of s and u from v, as expressed by the equation $m' = v - s^T u$. The result m' is a noisy polynomial. To get rid of the added noise (resp. small errors) the function *Compress* is applied to the result to revert the added error tolerance from the *Decompress* during the encryption. After that, the function *Encode* is applied to convert the polynomial to a byte array. As in the encapsulation phase, the shared key K is computed using the encryption function PKE.Enc from Algorithm 5 (line 4-14) and the chain of hash computations. Kyber allows decapsulation failures to increase performance with a claimed failure probability of $\delta = 2^{-139}$ [4].

Algorithm 6 KEM.Dec [4]

Input: Ciphertext c
Input: Secret key sk
Output: Shared key K

$pk := sk + 12 \cdot k \cdot n/8$
$h := sk + 24 \cdot k \cdot n/8 + 32 \in \{0,1\}^{256}$
$z := sk + 24 \cdot k \cdot n/8 + 64$
$u := Decompress_q(Decode_{d_u}(c), d_u)$ ▷ Start PKE.Dec
$v := Decompress_q(Decode_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
$\hat{s} := Decode_{12}(sk)$
$m' := Encode_1(Compress_q(v - NTT^{-1}(\hat{s}^T \circ NTT(u)), 1))$ ▷ End PKE.Dec
$(\overline{K}', r') :=$ SHA3-512$(m' \parallel h)$
$c' := PKE.Enc(pk, m', r')$ ▷ Calls PKE.Enc() from Algo. 5
if c = c' **then**
    return $K :=$ SHAKE-256$(\overline{K}' \parallel$ SHA3-256$(c))$
else
    return $K :=$ SHAKE-256$(z \parallel$ SHA3-256$(c))$
end if

Public key	Secret key	Ciphertext	n	k	q	η_1	η_2	(d_u, d_v)	δ
800 (Bytes)	1632 (Bytes)	768 (Bytes)	256	2	3329	3	2	$(10, 4)$	2^{-139}

Table 3.2.: Parameter of Kyber512.

3.4. Hardware selection

This section examines the Infineon TriCore TC38xQP microcontroller and its significance in the automotive industry as well as the suitability for post-quantum cryptography applications. The microcontroller has been chosen for its widespread adoption in various projects, its ability to deliver sufficient performance, and its support for the AUTOSAR platform in automotive applications.

3.4.1. Infineon Tricore TC38xQP

The Infineon Tricore TC38xQP microcontroller has garnered significant attention and adoption within the automotive industry owing to its formidable features and capabilities tailored specifically for automotive applications. This microcontroller finds extensive application in diverse areas, including engine management systems, transmission control units, and safety systems such as Anti-Lock Braking Systems (ABS) and Electronic Stability Control (ESC). Renowned for its robust processing power, real-time execution environment, and fault tolerance, the TC38xQP empowers precise control and reliable operation of automotive systems. Notably, its low power consumption and compact form factor contribute to overall system efficiency. However, it is crucial to acknowledge

that the TC38xQP's adoption is accompanied by certain considerations. Due to its dual banking feature it is often used for Secure Software Downloads and Over the Air (OTA) updates. The microcontroller's higher cost compared to alternatives and the specialized knowledge and tools required for programming and debugging are factors that warrant attention. Nevertheless, the TC38xQP aligns seamlessly with AUTOSAR, a standardized software architecture extensively used in the automotive domain. This integration with AUTOSAR ensures portability, reusability, and maintainability of software components, streamlining development processes and reducing time-to-market. In essence, the TC38xQP emerges as a sought-after choice in the automotive industry, leveraging its versatility, real-time performance, and compatibility with AUTOSAR to deliver efficient and reliable automotive systems.
The TC38xQP is a 32-bit quad-core controller, with the following hardware details [2]:

- 300 MHz clock speed
- 10 MB Flash (ROM)
- 1.5 MB SRAM
- 1 Gbit Ethernet
- 12xCAN FD, 2xFlexRay, 24xLINs, 6xQSPI , 2xI^2C, 25xSENT 5xPSI, 1xHSSL, 3xMSC
- Support of floating and fixed point calculation

4. AUTOSAR

AUTOSAR is a partnership of vehicle manufacturers, suppliers, service providers and companies from the automotive electronics, semiconductor, and software industry. The consortium was founded in 2003 and develops open industry standards for automotive software architecture. The main goals of AUTOSAR incorporate the fulfilment of future vehicle requirements, increasing scalability and flexibility to integrate and transfer functions, re-use of software, and accelerate development and maintenance [5]. AUTOSAR specifies two main standards Classic Platform [7] and Adaptive Platform [6] besides other standards.

The consortium aims to standardize the interfaces between different modules within a software stack, seeking to achieve abstraction from the underlying hardware. This standardization enables faster development cycles and promotes higher code reusability by decoupling software components from specific hardware platforms. By providing a common framework and interface definitions, AUTOSAR fosters interoperability, modularization, and portability across various automotive systems, ultimately enhancing efficiency and flexibility in software development processes. In this research project we use the AUTOSAR Classic Platform standard to implement the chosen use cases with the specified cryptographic primitives. Classic Platform is based on the OSEK [14] standard. The code is directly executed from ROM. All applications have the same address space. Classic Platform is optimized for signal-based real-time communication and uses a fixed task configuration.

4.1. Architecture

AUTOSAR Classic Platform employs a layered architecture and consists of the following three basic layers, as shown in Figure 4.1:

- **Basic Software layer (BSW)**
  The Basic Software layer is the lowest layer in the architecture. It is a standardized software layer that provides standard ECU functionality e.g. OS, low level drivers, bus-communication, diagnostics, cryptography etc. The BSW is again structured in layers, the Microcontroller Abstraction Layer (MCAL), ECU Abstraction Layer, and Services Layer. Some services like Complex Drivers and partly System Services span over multiple layers. The provided services are accessible via the RTE abstraction layer.

- **Runtime Environment (RTE)**
  The RTE provides communication services to the application software of the Application layer. It is responsible for the inter- and intra-ECU information

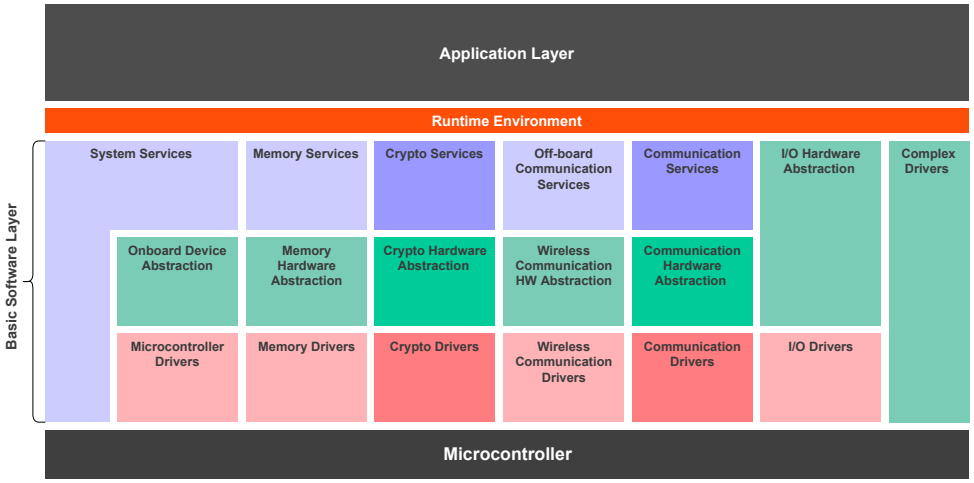


Figure 4.1.: AUTOSAR Classic Platform layered software architecture [8].

exchange. Thus it represents the full interface for applications and makes them independent from the mapping to a specific ECU.

- **Application layer**
  The Application layer is the highest layer in the architecture and contains the application software. The application software interacts with the RTE to use func-tionalities of the BSW layer. It consists of Software Components and/or Sensor/Actuator Components. Software Components are completely ECU independent, while the latter are dependent on the specific hardware.

4.2. Crypto Stack

The Basic Software layer contains a Crypto stack. The Crypto stack provides standardized cryptographic functionalities and services. Thus Software Components in the Application layer and system modules in the BSW layer don't need to implement cryptographic primitives themselves, but can use the centralized Crypto stack available via the RTE or in case of system modules directly. In AUTOSAR, a cryptographic primitive refers to a fundamental cryptographic operation or building block that is utilized to provide various security functionalities within the AUTOSAR software architecture. It serves as a foundational component for implementing cryptographic algorithms and protocols. Further details are given in Section 4.2.4.

The Crypto Stack consists of the Crypto Service Manager (Csm) [11], Crypto Interface (CryIf) [10], and the Crypto Driver (Crypto) [9] module. Figure 4.2 depicts the correlation of these modules in the Crypto stack, and their usage from the Application level. Csm and CryIf are using configured channels to communicate with the respective Crypto primitives, as further explained in the following subsequent sections.

The Key Manager (KeyM) module is additionally part of the Crypto Stack. The security-related communication modules like Secure Onboard Communication (SecOC), Transport Layer Security (TLS), and Internet Protocol Security (IPSec) belonging to

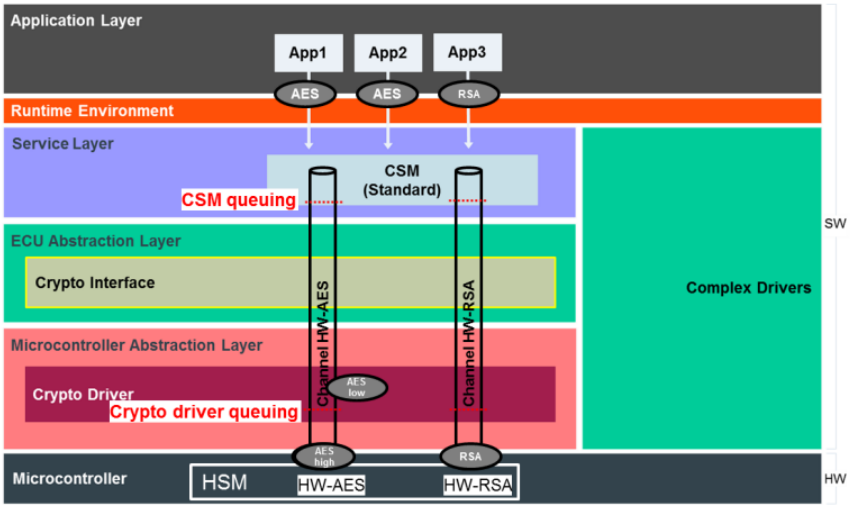


Figure 4.2.: Crypto Stack layered architecture [12].

the Communication Stack. Since they are not directly relevant to this research project we omit them. In this project, the primary focus was on the practical implementation of post-quantum algorithms, with a specific emphasis on the Crypto module within the MCAL layer.

4.2.1. Crypto Service Manager

The Csm module is in the Service Layer and therefore the highest module in the Crypto Stack. It supports algorithm independent service interface that makes it possible for different applications to use the same service but with different underlying primitives or schemes e.g. the service Hash with the primitive SHA2-256 or SHA3-513. The Csm provides an interface for the RTE which can be used by a Software Component or the API of the Csm can be directly called from a system module in the BSW layer. The Csm employs two different concepts, a direct API mainly for key management services and a job-based API mainly for cryptographic primitives. The main parts of the job-based concept consists of a job and a prioritized queue. A job is composed of a referenced cryptographic primitive, a referenced key, a process priority, and an assigned queue. The job-based concept enables that multiple independent jobs can be processed in separate prioritized queues. The direct API can only handle synchronous calls while the job-based API can handle both synchronous and asynchronous calls. Depending on the request of the Software Component the direct key management interface is called and the request is forwarded to the CryIf module or a job is created with the given data and pushed into the assigned queue.

4.2.2. Crypto Interface

The CryIf module lies between the Csm and Crypto module and provides abstraction from the Crypto Driver. The CryIf module links a Crypto Driver Object to the Csm

module via a channel. In detail, the Csm maps one queue to a CryIf channel and the CryIf maps one channel to a Crypto Driver Object. Through this abstraction the CryIf enables that one Csm can map to multiple Crypto Driver Objects. Direct interface calls are just forwarded to the specific Crypto Driver.

4.2.3. Crypto Driver

The Crypto Driver is an MCAL module and thus the lowest in the Crypto stack. It contains the actual cryptographic primitives and key management algorithms. It also provides the functionality for key storage and handling, which includes the definition of the key structure with key elements. An instance of a Crypto Driver is called a Crypto Driver Object (CDO). A CDO includes a specific set of primitives and has its own independent workspace. Multiple CDOs can be configured that are independent from each other. Only one primitive can be performed at a time per CDO. Each CDO is mapped via a CryIf channel to the Csm module. The public interface of the Crypto module is used by the CryIf to process/cancel jobs and for key management functions. Then the CDO forwards the data to the actual crypto routine. The cryptographic operations of a CDO can be implemented in hardware or in software.

4.2.4. Cryptographic primitive

Each CDO has a specific set of cryptographic primitives. A cryptographic primitive is specified by a service, family, and mode. A primitive can be further refined by configuring a secondary family.

- **CryptoPrimitiveService** is the basic class of cryptographic operations e.g. “Encrypt” or “SignatureGenerate”.
- **CryptoPrimitiveAlgorithmFamily** specifies the cryptographic family of a service e.g. AES or RSA.
- **CryptoPrimitiveAlgorithmMode** specifies the mode of the family. Different modes can be available for one family e.g. ECB or CBC mode for the family AES.
- **CryptoPrimitiveAlgorithmSecondaryFamily** can be configured if the cryptographic primitive needs an another cryptographic primitive like a “Hash” algorithm, e.g. SHA2_512 for service “SignatureGenerate” with family RSA.

The context of a cryptographic primitive is stored in the **workspace** of a CDO. This context contains all runtime data which is needed by the primitive during its processing. Each CDO has its own workspace. The workspace is shared between the primitives. Therefore, a CDO can only perform one cryptographic primitive at a time.

The input and output information of a cryptographic primitive is stored in a specific data type called `Crypto_JobPrimitiveInputOutputType` (see Listing 4.1) defined by AUTOSAR. The Csm is the first module of the Crypto stack which receives the request for a Crypto Service as a function call. It receives the inputs and outputs as

function parameters for a specific Service and then fills the `Crypto_JobPrimitiveInputOutputType` accordingly and forwards the request further through the Crypto stack.

Listing 4.1: AUTOSAR 4.3.1 type definition of Crypto_JobPrimitiveInputOutputType.

```
Crypto_JobPrimitiveInputOutputType = {
    inputPtr                  : const uint8*
    inputLength               : uint32
    secondaryInputPtr         : const uint8*
    secondaryInputLength      : uint32
    tertiaryInputPtr          : const uint8*
    tertiaryInputLength       : uint32
    outputPtr                 : uint8*
    outputLengthPtr           : uint32*
    secondaryOutputPtr        : uint8*
    secondaryOutputLengthPtr  : uint32*
    verifyPtr                 : Crypto_VerifyResultType*
    mode                      : Crypto_OperationModeType
    cryIfKeyId                : uint32
    targetCryIfKeyId          : uint32
}
```

A cryptographic primitive is processed via a job. The state machine of a job is depicted in Figure 4.3. After initialization of the Crypto Driver, the job is in the `IDLE` state and no cryptographic primitive is processed at the moment. A job can be called with the operation modes `START`, `UPDATE`, and `FINISH`.

- `START:` A new request of a cryptographic primitive is triggered with operation mode `START`. All previous requests of the same job are canceled.
- `UPDATE:` In the operation mode `UPDATE` the cryptographic primitive expects input data. The job can be called multiple times with this operation mode consecutively to process large data. For some services like “Encrypt” or the service “AEADEncrypt” ouput data is also possible.
- `FINISH:` This operation mode informs the Crypto Driver that all input data has been fed and the cryptographic primitive can finalize calculations. The result of the cryptographic operation is stored in the output buffers and job state is switched to `IDLE`.

If an error occurs during the operation modes the job state switches to `IDLE` and all internal data like input data and intermediate results are removed.

The call of each operation mode individually is called the “Streaming approach”. Another possible operation mode is `SINGLECALL` which is a concatenation of `START`, `UPDATE`, and `FINISH`. The operation mode `SINGLECALL` could improve performance, because of less API calls and is intended for small data inputs, since it cannot call `UPDATE` multiple times like in the “Streaming approach”.

All jobs are being processed via the function `Crypto_ProcessJob(objectId, job)`. This function performs the cryptographic primitive that is configured in the job

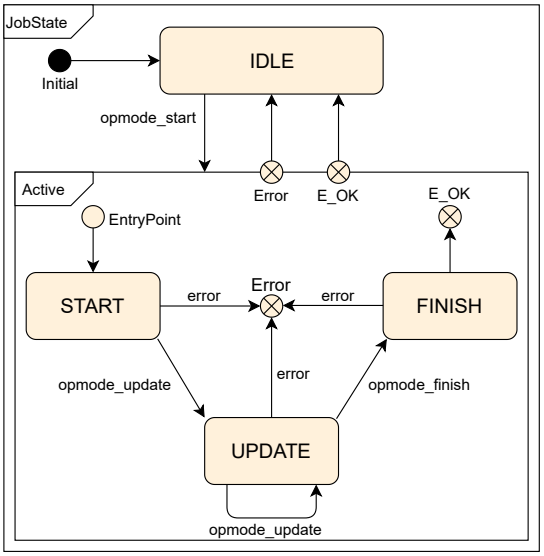


Figure 4.3.: AUTOSAR Crypto job state machine simplified [9].

parameter. It provides a single interface for the Singlecall and Streaming approach. The operation mode is set in the `Crypto_JobPrimitiveInputOutputType` (see Listing 4.1) which is referenced by the job parameter.

A cryptographic primitive can be processed **synchronously** or **asynchronously**. For synchronous processing the caller is blocked until the job is fully completed by the CDO. Whereas for asynchronous processing the caller only initiates the processing of a job but is then free for other tasks. The Crypto Driver Object informs the Caller when the job is finished via "Callback" functions.

4.3. Communication Stack

The Basic Software Layer contains a Communication Stack (Com Stack). The Com Stack provides services for communication via hardware interfaces e.g. inter ECU communication. It is capable of various communication and network protocols like Ethernet/IP, CAN, LIN, and FlexRay. A simplified depiction is illustrated in Figure 4.4. For this project the AUTOSAR Ethernet Stack and TCP/IP Stack is used for the communication with the Server.

In the lowest layer the **Ethernet Stack** is used. It contains the Ethernet Driver (Eth Driver) in the Microcontroller Abstraction Layer and the Ethernet Interface (Eth Interface) in the ECU Abstraction Layer. The Ethernet Stack represents the software driver of the Ethernet hardware interface. Above the Ethernet Stack the **TCP/IP Stack** is employed in the Services Layer. It receives the IP datagrams from the Ethernet Stack and contains modules for IP v4/v6, ICMP, ARP, UDP, TCP, and DHCP protocol.

The **Socket Adapter (SoAd)** module is above the TCP/IP Stack and converts UDP/TCP sockets to PDUs. A PDU (Protocol Data Unit) is a basic unit for data transfer. It is used as an abstraction of the specific communication protocols.

One central module of the Com Stack is the **PDU Router (PDUR)**. Its main responsibility

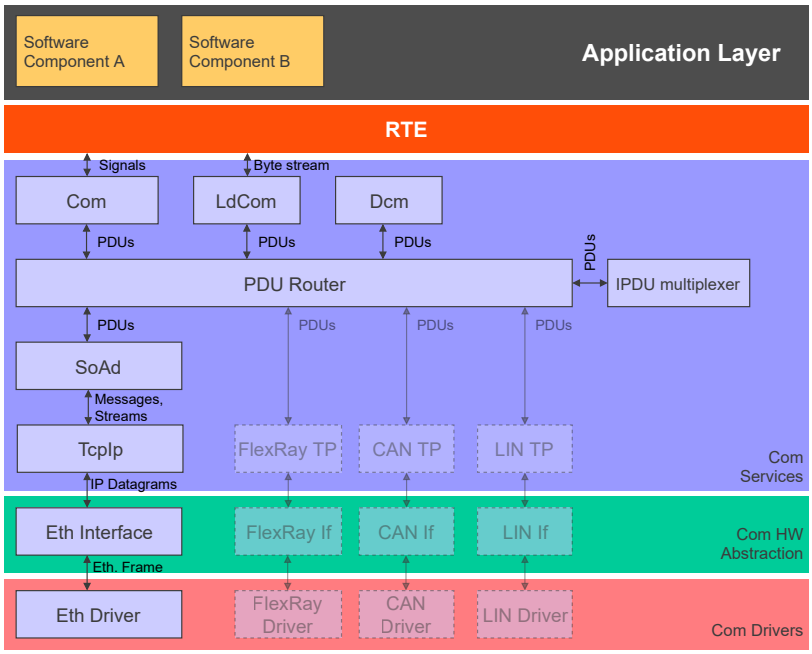


Figure 4.4.: AUTOSAR Classic Platform simplified Communication Stack.

is the routing of PDUs. The PDU Router has various routing operations like unicast from one module to another module, multicast from Com module to communication interface modules, and gateway functionality from a communication interface module to other communication interface modules. All modules below the PDUR are network protocol dependent, while all modules above are network protocol independent.

The **Communication (Com)** module is a network protocol independent module. It is the contact point for a Software Component via RTE and is responsible for the conversion of Signals to PDUs called signal packing. A Signal is a basic communication object like a primitive data type (e.g. int, char, etc.). Signals can also be grouped to automatically transmit complex data types like C-structures. The **Large Data Communication (LdCom)** module is similar to the Com module, but with the difference that it can handle large messages efficiently, without the overhead of Com signal packing. The data received by the RTE is a byte stream and can be directly converted to PDUs without signal packing, filtering, internal buffering, or transmission modes. The LdCom module is mainly used for Ethernet communication.

5. D6.1 - Demonstrator description

5.1. Goals

The demonstrator should visualize the results of post-quantum cryptography used in real-world embedded hardware devices, e.g. using *Tricore* or *RISC-V* chips. Thereby, the user should be able to understand the benchmarking results without further experience in cryptography. In our demonstrator, the PQC algorithms are integrated into the crypto-stack of the AUTOSAR Classic platform, that is also used in production. Thus, we can provide an almost real-world scenario on which the evaluation results are based. Besides the PQC algorithms, the user has the possibility to start pre-quantum algorithms such as RSA or ECC. The benchmarking results are illustrated using an easily-understanding diagram, showing how many microseconds each operation of the executed algorithms has taken. This allows the user to understand the differences between pre- and post-quantum algorithms and their individual pros and cons. To better understand the use cases, additional information is provided such as sequence diagrams and a detailed description including the threat model.

5.2. Architecture

The demonstrator consists of three components as visualized in fig. 5.1, namely a backend, a frontend and a target device. All these components have been containerized and can easily be started using Docker-Compose. The frontend communicates with the backend to send use cases requests, which are forwarded to the selected target by the server. That target executes the respective use case, whereby both the backend and the embedded device perform some cryptographical computations, e.g., verification/generation of a digital signature. The target device returns the measured execution time on each request, while the backend aggregates those information and its own timings and sends them back to the frontend for visualization.

The backend is responsible for generating the keys that are necessary to perform the cryptographic algorithms used in the respective use cases. For the sake of simplicity (and since this is a demo environment), the backend-server also generates the public-private key pairs for the targets, whereas in a production environment the target's keys would need to be generated in a more secure way (e.g., by itself or embedded at production). Furthermore, each target receives the same key-pair which is okay in a benchmarking scenario, but would be highly insecure in the real-world. Those keys are distributed to the target once it connects to the server. All messages are serialized using *protobuf* (cf. section 5.5.1), whereby the connection to the target uses TCP and the connection to the frontend is based on UDP.

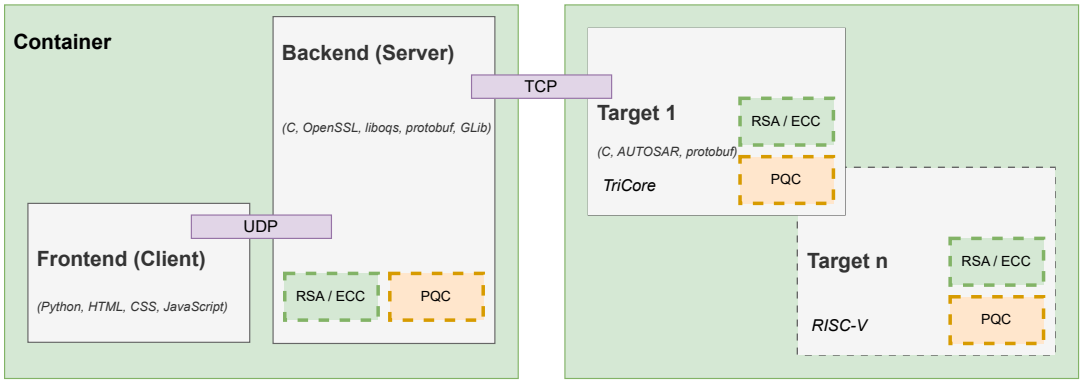


Figure 5.1.: Architecture of the Demonstrator

The backend has a dedicated UDP socket for the frontend and accepts simultaneous target connections using TCP sockets. We designed the backend in a dynamic way that allows an arbitrary number of target devices that can be reached by the frontend. A backend-target connection is established by sending an initialization message from the target device to the server. This message contains an unique identifier (UID) that is used to map the frontend requests to the respective target device. Once the server detected a new device, it generates a public-private key-pair and sends it to the target.

The backend-frontend connection is initiated in a similar way by sending a use case request messages to the specific server. This request basically contains the target that should execute the algorithms, the algorithms that should be benchmarked and the use case. The server executes the request and returns the timings of the algorithms to the frontend. The frontend displays a visual chart based on the timings to give a comparison between the algorithms.

5.3. AURIX TC38xQP Demonstrator

The integration of the Aurix Tricore TC38xQP chip into an automotive system involves various components and processes to establish efficient communication and functionality. The chip is connected to the system using a 1Gbit Ethernet interface. Notably, the use of two different Ethernet transceivers, namely Lantiq and Realtek, leads to the implementation of two distinct software versions to accommodate the specific requirements of each transceiver. In addition to the Ethernet connection, the chip is also connected to a test PC via the Lauterbach debugger, enabling debugging and testing functionalities during the integration process. For the realization of the basic functionaility the EB tresos Basic Software is used. All modules are based within the context of AUTOSAR, specifically leveraging AUTOSAR Release 4.0.3, 4.3.0 and AUTOSAR Release 4.2.2 for MCAL modules. This integration ensures compliance with standardized interfaces and protocols, facilitating seamless interoperability with other AUTOSAR components. The integration process is facilitated by utilizing EB Tresos Studio, a tool that streamlines the integration and utilization of these modules within the overall system architecture. This integration enhances the system's functionality

and capabilities.

The system employs a task-based architecture consisting of eight tasks that handle various system functions. Notably, the Communication task and Rte time task require a stack size of 8192 bytes to ensure smooth execution and efficient utilization of system resources. To enhance the system's robustness, an integration of MicroOs, a micro-kernel for AutoCore Os, is employed. This integration provides interrupt and exception handling capabilities, further bolstering the system's reliability.

In terms of security, PQC (Post-Quantum Cryptography) algorithms are integrated into the Crypto Software Module provided by Elektrobit. These algorithms, based on Crypto Module Version 2, enable secure communication within the automotive system. Further information are provided in the section Implementation of traditional cryptographic primitives

To optimize message handling, the integration of the protobuf library into AUTOSAR is adopted. The open-source library nanopb[3] is utilized, offering lightweight and memory-efficient message serialization and deserialization. Notably, this integration eliminates the need for explicit memory allocation as all binary messages of unspecified length are handled seamlessly by the integrated callback functions.

The communication flow within the system follows a defined sequence. After initialization, the system attempts to establish a connection with a specific IP address (e.g., 192.168.88.73) and a designated port. Upon successful connection, the subsequent chapter Demonstrator Sequence is executed, enabling the system to proceed with its intended operations. Throughout the communication process, all communication events are directed to the callback function of the LdCom Module. Furthermore, the integration of the Crypto API and protobuf components further enhances the communication capabilities of the system, ensuring secure and efficient data exchange.

5.4. Demonstrator Sequence

An abstract overview about the software flow and communication between server, client and target is given in fig. 5.2. The server starts by generating the necessary keys for RSA, ECC, Kyber and Dilithium. Since this is a non-productive demonstration environment, it has been decided that the public-private key pairs for the target are also generated by the backend. After the key generation, the server opens a UDP socket for the frontend and in another thread a TCP socket. The TCP socket creates one thread per target once it detects a new connection. The new threads awaits an initialization message containg the UID of the target. Afterwards, the backend sends the packed public-private keypairs to the target and stores the target's UID in a list. After unpacking them, the target is in an idle state awaiting a use case sent by the backend. The frontend connects to the UDP socket of the backend and after the user selected a use case, it sends this information along with the selected algorithms to the backend. The backend unpacks the received message and iteratively processes each algorithm. To do so, it performs the server-side computation, selects the respective target and sends a request to it. The target performs the target-side computation of the received algorithm, sends the time needed for the execution to the backend and returns to the idle state where it awaits a

new use case from the backend.

The server receives the measured time of the target and adds it to the total computation time. If there are any more algorithms that should be benchmarked, they will be processed as described before. If that is not the case, the total execution time is sent back to the frontend.

The frontend receives this information and visualizes it in a user-friendly diagram. Finally, it returns to a state awaiting user input.

5.5. Server / Backend

In detail, the server is implemented in C and has been containerized to be executed in any environment, whereby the image is based on *Ubuntu*. It requires *OpenSSL* (min. version 1.1.1f) to perform cryptographic operations in use cases where RSA and ECC algorithms are used. For the PQC algorithms, the open source library *liboqs*[1] is integrated since it supports Kyber512 and Dilithium2 by default. The communication between the devices are handled by standard UDP and TCP network sockets. The server offers one UDP network socket for the frontend and TCP sockets for the targets. The messages within this communication are serialized with *protobuf-c*[2], which is a pure C implementation of *Protocol Buffers*.

5.5.1. Communication protocol

The communication between server, frontend and target is performed via network socket, where the data is serialized with *protobuf*. By using protobuf, the data is structured in a previously defined format and interfaces are automatically generated by using the protobuf compiler. Therefore, the allowed messages, types and lengths are equivalent in all software components - independent of OS and programming language.

The protobuf-c compiler takes the *.proto*-file and generates one header and one source code file containing C-*structs* and *functions*. Within the implementation, one can use the generated functions to initialize, pack, unpack and free the messages. The transformation between host and endianess is also covered by protobuf. However, protobuf does not define how to send the data over the network. Therefore, the size of the messages has to be transmitted prior to the data. By using the `MSG_PEEK` option of POSIX sockets, those bytes representing the length of the protobuf message can be retrieved without accepting the message.

Listing 5.1 shows general messages of our protobuf definition that are used in the demonstrator implementation. The frontend creates a *Front2Back* message, which defines the use case, the plaintext size, the algorithms that shall be executed and which target devices to be used. During the execution, the server creates several `Back2Target` messages to send data to the target. The data types `KeyDist`, `SSD`, `SAC`, `SSE` are other protobuf structures that contain the information for respective use cases or for the key distribution. The boolean `endConnection` defines whether the

[1] https://github.com/open-quantum-safe/liboqs
[2] https://github.com/protobuf-c/protobuf-c

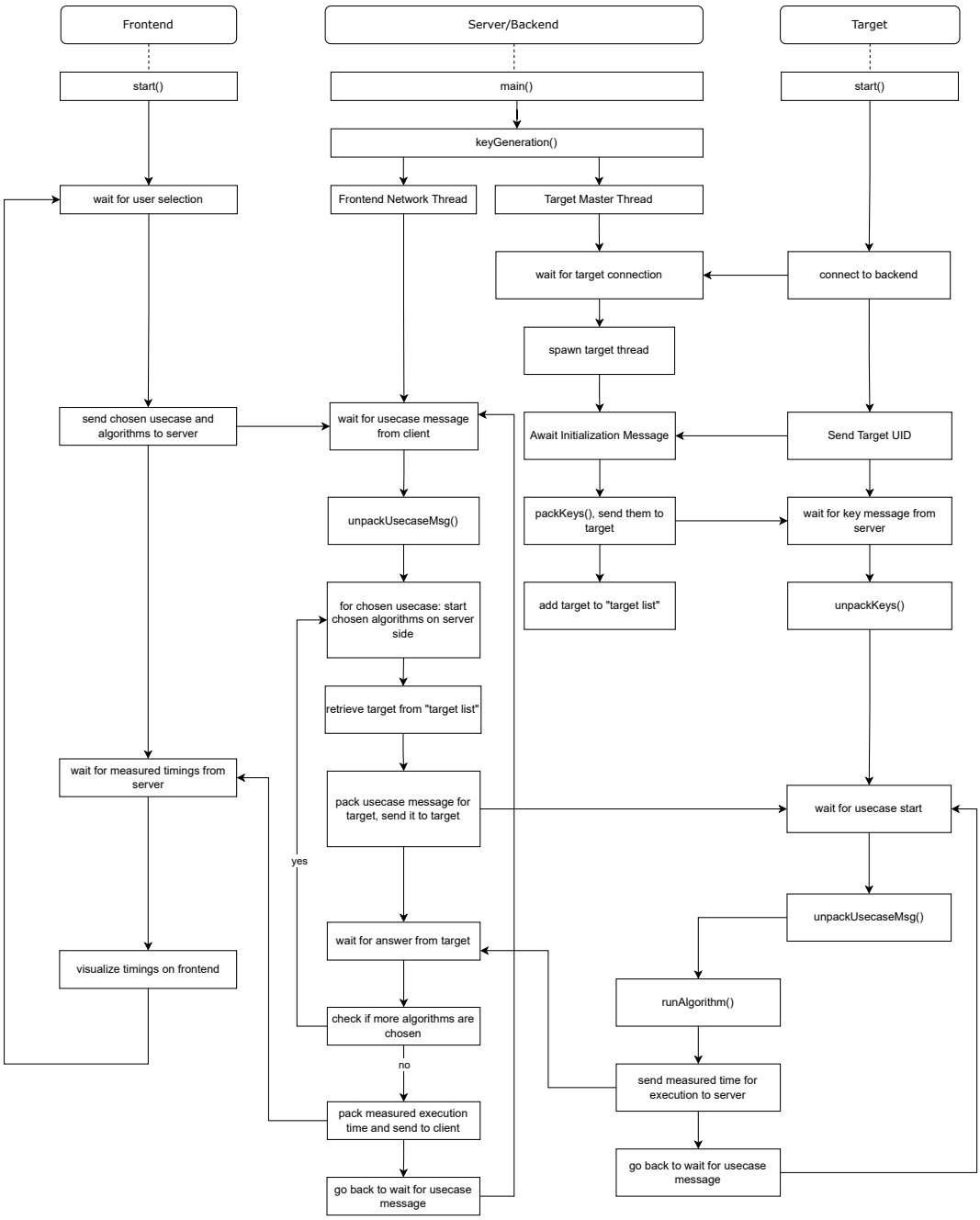


Figure 5.2.: Sequence diagram of the components

connection between target and backend should be terminated. The target uses the `Target2Back` structure to return the measured time of its executed algorithm. Those timings are stored within `Timing` structures and returned by using the `Back2Front` type. The keyword `repeated` allows multiple values of an element (like an array), where `oneof` allows only one element of the given types.

Listing 5.1: Cutout of protobuf definition in qrisc_demo.proto

```
1  syntax = "proto3";
2
3  message Front2Back {
4    string usecase = 1;               // SSD, SAC, SSE
5    uint32 msgSize = 2;               // message size in bytes
6    repeated Algorithm algoName = 3;  // RSA, ECC, ECDH, Kyber512, ↩
     Dilithium2
7    string target = 4;                // target names (tricore)
8  }
9
10 message Back2Front {
11   string usecase = 1;
12   repeated Timing results = 2;
13 }
14
15 message Back2Target {
16   oneof usecase {
17     KeyDist keyDist = 1;
18     SSD ssd = 2;
19     SAC sac = 3;
20     SSE sse = 4;
21     bool endConnection = 5;
22   };
23 }
24
25 message TargetInitRequest {
26   string targetName = 1;      // target name (e.g. tricore)
27 }
28
29 message Target2Back {
30   Algorithm algoName = 1;     // RSA, ECC, ECDH, Kyber512, Dilithium↩
     2
31   uint32 exectime = 2;
32 }
33
34 message Timing {
35   Algorithm algoName = 1;     // RSA, ECC, ECDH, Kyber512, Dilithium↩
     2
36   uint32 timingSign = 3;
37   uint32 timingVerify = 4;
38   uint32 timingCom = 5;
39   uint32 timingEnc = 6;
40   uint32 timingDec = 7;
41 }
```

5.6. Frontend

The frontend is an independent application, whereby the backend communication has been implemented in Python and the graphical interface in HTML, CSS and JavaScript. It uses the Python library *Eel*[3], offering a standalone web service supporting callbacks between Python and JavaScript code, so that interactions triggered on the user interface can be handled by Python functions and the other way around, python functions can access the web interface.

5.6.1. User Interface and Visualization

Figure 5.3 shows a screenshot of the frontend, where the user can start use cases, change the parameters and inspect the benchmarking results as horizontal bar chart. The user can choose the use case and the possible algorithms that are compatible with it. The connected targets are displayed in a drop-down menu, whereby they are currently statically inserted - this may be changed to only display the devices currently connected to the backend. The *Update Size* is only available for the use case *Secure Software Update*, whereby this parameter defines the size of the plaintext that will be signed by the server. The plaintext itself is randomly generated on server-side.

After the use case request has been handled by the server and a response has been received, the benchmarking results are visualized in a horizontal bar chart as seen in the figure. For the use cases, where digital signature algorithms are used, the chart gives a comparison between *communication delay*, *verification time* and *signing time*. The communication delay is the time delta between sending and receiving the message. Signing and verification time is the measured time for the respective algorithm process. For the *Secure Channel Establishment* use case, instead of the Signing and Verification time, the time taken for *Encapsulation* and *Decapsulation* are being benchmarked and shown in the diagram. However, Elliptic-Curve-Diffie-Hellman does in fact not encapsulate or decapsulate the shared secret as Kyber does, but it's measured time for the computation of the shared secret is displayed there as comparison to Kyber.

The values are presented in microseconds since a presentation in milliseconds would hide communication delay and verification time as they are (at least within the same network) neglectably small. Because the signing time for RSA and for Dilithium2 are significantly larger than communication and verification time, the x-axis has been limited to 2000 microseconds. The detailed time value can be read from the description of the respective bar.

5.6.2. Communication with Server

As written before, the client runs as Python script, where the user interface logic is implemented by JavaScript. The library *Eel* allows to use callbacks between Python and JavaScript, hence a use case request can be triggered from the website as Python function. In Listing 5.2 the `start_usecase()` function is given as example, where the annotation `@eel.expose` informs Eel that this function should be exposed to JavaScript

[3] https://github.com/python-eel/Eel

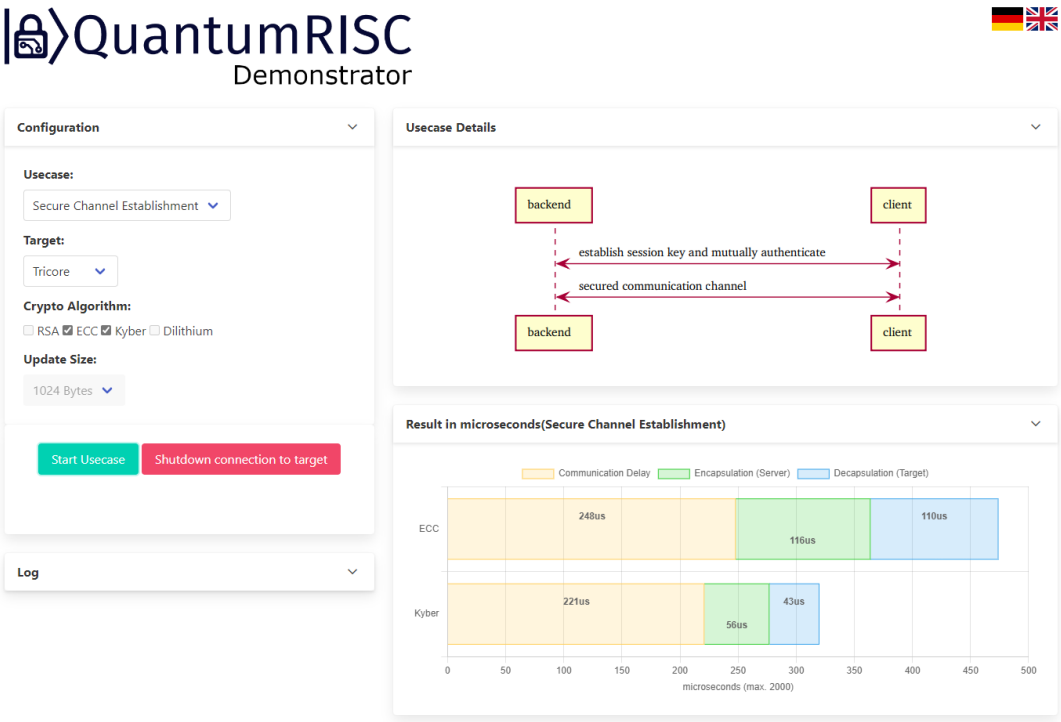


Figure 5.3.: Screenshot of the frontend

code. This function can then be started as regular call as given in Listing 5.3. As argument in the last brackets of the Javascript code, another function can be provided as callback.

Listing 5.2: Exposed function in Python

```
@eel.expose
def start_usecase(addr, port, usecase, target, algos, msglen):
  try:
    ret_timings, ret_timings_len = createFront2BackMsg(addr, port,
    usecase,
    target, algos, msglen)
    return usecase, ret_timings, ret_timings_len
  except ConnectionRefusedError:
    eel.handleConRefused()
```

Listing 5.3: Call of exposed function in JavaScript

```
eel.start_usecase(server_addr, server_port, usecase, target, algos,
msglen)(usecaseCallback);
```

Inside the `start_usecase()` function, the arguments are checked and a UDP socket with the given server IP address and port number is created and connected to. The respective IP and port are retrieved from the environment variables, that are set by the Docker container. If the connection is refused, an exception is raised, which calls an

error handler that has been implemented in JavaScript to inform the user. Otherwise, a `Front2Back` protobuf message is created and sent via the UDP connection. Then, the client switches into a blocking state until it receives an answer from the server. This message is unpacked as `Back2Front` message, containing the timings of the processed algorithms. Those values are passed to the JavaScript code for further processing and visualization on the user interface.

5.7. Target - Tricore TC38xQP

The target consists of a specific hardware board on which the cryptographic operation of the corresponding use case is performed. The target communicates with the server to determine the use case, exchange runtime measurements, inputs, and outputs of the cryptographic operation.

5.7.1. Implementation of traditional cryptographic primitives

Traditional cryptographic primitives are primitives that are currently defined by AUTOSAR which do not contain PQC primitives. This section describes how these primitives are implemented in an AUTOSAR (version 4.3.1) environment. We only focus on a signature scheme and a key exchange algorithm.

Signature Scheme

For the signature scheme we employ the RSASSA-PKCS1-v1.5 scheme for signature generation and verification. The signature scheme RSASSA-PKCS1-v1.5 is implemented according to RFC 8017 [18]. The essential part of the configuration of the cryptographic primitive for signature generation is shown in Listing 5.4. For the secondary family other hash algorithms could be configured, ranging from SHA1 to SHA2 and SHA3 with different hash sizes. The primitive for the secondary family doesn't have to be configured explicitly.

Listing 5.4: SignatureGenerate RSASSA-PKCS1-v1.5 config.

```
CryptoPrimitive:
  CryptoPrimitive_SIGNATUREGENERATE_RSA_RSASSAPKCS1V15:
    CryptoPrimitiveService:                   SIGNATURE_GENERATE
    CryptoPrimitiveAlgorithmFamiliy:          ALGOFAM_RSA
    CryptoPrimitiveAlgorithmMode:             ALGOMODE_RSASSA_PKCS1_v1_5
    CryptoPrimitiveAlgorithmSecondaryFamiliy: ALGOFAM_SHA2_512

CryptoKeyElement:
  CRYPTO_KE_SIGNATURE_KEY:
    Size:   1293 (Bytes)
    Format: CRYPTO_KE_FORMAT_BIN_RSA_PRIVATEKEY
```

The configuration of the RSASSA-PKCS1-v1.5 signature verification scheme, see Listing 5.5, is analogous to the signature generation, with the exception of the format

and size of the “CryptoKeyElement”. The key format of the public key and private key adheres to RFC 8017 [18] chapter 3.1. and 3.2. second representation respectively.

Listing 5.5: SignatureVerify RSASSA-PKCS1-v1.5 config.

```
CryptoPrimitive:
  CryptoPrimitive_SIGNATUREVERIFY_RSA_RSASSAPKCS1V15:
    CryptoPrimitiveService:                   SIGNATURE_VERIFY
    CryptoPrimitiveAlgorithmFamiliy:          ALGOFAM_RSA
    CryptoPrimitiveAlgorithmMode:             ALGOMODE_RSASSA_PKCS1_v1_5
    CryptoPrimitiveAlgorithmSecondaryFamiliy: ALGOFAM_SHA2_512

CryptoKeyElement:
  CRYPTO_KE_SIGNATURE_KEY:
    Size:   270 (Bytes)
    Format: CRYPTO_KE_FORMAT_BIN_RSA_PUBLICKEY
```

AUTOSAR specifies that for the service Crypto_SignatureGenerate the input buffer Plaintext is required in the operation mode `UPDATE` and optional in the mode `FINISH`. The generated output buffer Signature is required in the `FINISH` mode. The key handling is done separately.

	Crypto_SignatureGenerate:	**Crypto_SignatureVerify:**
Input:		
	- Plaintext	- Plaintext - Signature
Key:		
	- RSA_PRIVATEKEY	- RSA_PUBLICKEY
Output:		
	- Signature	- Verification result

For Crypto_SignatureVerify the Plaintext is required in the operation mode `UPDATE` and optional in the mode `FINISH`. The Signature is required in mode `FINISH` and the Verification result is also required in `FINISH` mode. Since both services are cryptographic primtives they are processed as jobs via the general interface Crypto_ProcessJob and not as direct function calls in the Crypto Driver module. This means that also all input and output information are stored in the data type Crypto_JobPrimitiveInputOutputType and not handled as function parameters.

Key Exchange

For the key exchange we employ an Elliptic Curve Diffie-Hellman (ECDH) with Curve X25519. The ECDH primitive is implemented according to RFC 7748 [15]. The key exchange consists of the two services *Crypto_KeyExchangeCalcPubVal* and *Crypto_KeyExchangeCalcSecret*. Crypto_KeyExchangeCalcPubVal calculates the public value from own secret. The public value is exchanged with the other peer of the key exchange. Crypto_KeyExchangeCalcSecret calculates the shared secret with the exchanged public value. A snippet of the configuration can be found in Listing 5.6. The main part of the key exchange configuration is the configuration of the keys.

The keys (“CryptoKeys”) are provided by the Crypto Driver Object. A CryptoKey can be referenced by a job in the Csm. One CryptoKey consists of one or more key elements. To map the “CryptoKeyElements” to a CryptoKey the container “CryptoKeyType” is used. One CryptoKey references exactly one CryptoKeyType. A CryptoKeyType consists of references to one or more CryptoKeyElements. A CryptoKeyElement is used to store the data and contains all relevant information like read/write access, maximum size of the element, init value after startup, or format of the key element.

Listing 5.6: Key Exchange ECDH config simplified.

```
CryptoKey:
  CryptoKey_KeyExchange_X25519:
    CryptoKeyTypeRef: CryptoKeyType_KeyExchange_X25519

CryptoKeyType:
  CryptoKeyType_KeyExchange_X25519:
    CryptoKeyElementRef:
        CRYPTO_KE_KEYEXCHANGE_SHAREDVALUE
        CRYPTO_KE_KEYEXCHANGE_OWNPUBKEY
        CRYPTO_KE_KEYEXCHANGE_PRIVKEY
        CRYPTO_KE_KEYEXCHANGE_ALGORITHM
        CRYPTO_KE_KMNCOMMON_WORKSPACE

CryptoKeyElement:
  CRYPTO_KE_KEYEXCHANGE_SHAREDVALUE:
    Id:          1
    Size:        32 (Bytes)
    Format:      CRYPTO_KE_FORMAT_BIN_OCTET
  CRYPTO_KE_KEYEXCHANGE_OWNPUBKEY
  CRYPTO_KE_KEYEXCHANGE_PRIVKEY
  CRYPTO_KE_KEYEXCHANGE_ALGORITHM
  CRYPTO_KE_KMNCOMMON_WORKSPACE
```

For the key exchange we need to configure several key elements. To specify which key exchange shall be used we need to set the specific key exchange protocol in the key element CRYPTO_KE_KEYEXCHANGE_ALGORITHM. With this we signal that the whole key is used for the specified key exchange algorithm. The key element CRYPTO_KE_KEYEXCHANGE_PRIVKEY represents the private key and is used to calculate the public value CRYPTO_KE_KEYEXCHANGE_OWNPUBKEY and shared secret CRYPTO_KE_KEYEXCHANGE_SHAREDVALUE. The private key has to be set before calling the services Crypto_KeyExchangeCalcPubVal or Crypto_KeyExchange-CalcSecret.

As mentioned earlier a common workspace is shared between all the cryptographic primitives for runtime data. Since this workspace is shared only between primitives that follow the job-based approach, all key management services like key exchange don’t have a common workspace defined by AUTOSAR. To get around this, we introduced another workspace which is shared between the configured key management services. The size of this extra workspace is configured via the key element CRYPTO_KE-_KMNCOMMON_WORKSPACE.

Since the two services Crypto_KeyExchangeCalcPubVal and Crypto_KeyExchange-

CalcSecret are key management services and hence don't follow the job-based approach, they are only available via a synchronous call with the operation mode `SingleCall`. This also means that they are implemented without the job state machine (see Figure 4.3).

	Crypto_KeyExchangeCalcPubVal:	**Crypto_KeyExchangeCalcSecret:**
Input:		
	- CryptoKeyId	- CryptoKeyId - PartnerPublicValue
Key:		
	- CryptoKey_KeyExchange_X25519	- CryptoKey_KeyExchange_X25519
Output:		
	- PublicValue	- SharedSecret

Both services have the input parameter "CryptoKeyId" with which they can access the configured key CryptoKey_KeyExchange_X25519. The service Crypto_KeyExchange-CalcPubVal calculates the public value and returns it via the output parameter PublicValue and also saves it to the key element CRYPTO_KE_KEYEXCHANGE_OWNPUBKEY. For the service Crypto_KeyExchangeCalcSecret, the shared secret is not returned directly by the function, but written to the key element CRYPTO_KE_KEYEXCHANGE_SHAREDVALUE instead.

5.7.2. Implementation of PQC primitives

This section explains how a PQC primitive for a signature scheme and key encapsulation mechanism (KEM) can be implemented in an AUTOSAR (version 4.3.1) environment. KEMs are not considered by AUTOSAR so far, only a key exchange API is specified. Our solution adds a new key management API for the encapsulation and decapsulation of a KEM. The key generation of a KEM can be performed by the already existing key generation interface of AUTOSAR. For the PQC signature scheme, our solution implements the signature generation and verification as a cryptographic primitive using the job-based approach. The key generation of the signature scheme is implemented using the existing key generation interface of AUTOSAR.

Signature Scheme

For the signature scheme we implement the PQC primitive "CRYSTALS-Dilithium" [13], more precise the parameter set Dilithium II from round 2 of the NIST PQC competition[4]. Dilithium consists of three functions: key generation, signature generation, and signature verification.

Before we have a look into the function implementations we add a new algorithm family id for the primitive Dilithium (see Listing 5.7), which is used in the configuration to enable the usage of Dilithium as a crypto service.

[4]https://github.com/pq-crystals/dilithium/tree/round2

Listing 5.7: Dilithium algorithm family id.

```
#if !(defined CRYPTO_ALGOFAM_DILITHIUM)
#define CRYPTO_ALGOFAM_DILITHIUM  0xE1U
#endif /* #if !(defined CRYPTO_ALGOFAM_DILITHIUM) */
```

The **key generation** function of Dilithium generates a secret key and a public key for the signature scheme. It is implemented in the general AUTOSAR function Crypto_KeyGenerate(uint32 cryptoKeyId) which centralizes all key generation algorithms of different primitives. Crypto_KeyGenerate is part of the direct key management API and therefore does not follow the job-based approach. To use the key generation, we need to first set the AUTOSAR-defined key element CRYPTO_KE_KEYGENERATE-_ALGORITHM, which specifies which key generation algorithm shall be used when calling the general function Crypto_KeyGenerate. Its data consists of the prior defined CRYPTO_ALGOFAM_DILITHIUM. Then the key has to be set valid with the function Crypto_KeyValidSet() and Crypto_KeyGenerate() can be called. The result of this function is stored in the configured key elements (see Listing 5.8). The public key is stored in the key elements `RHO` and `T1` and consists of 1184 bytes total. The secret key is stored in the key elements `RHO`, `K`, `TR`, `S1`, `S2`, and `T0` and consists of 2800 bytes total. AUTOSAR defines a key element index which maps key elements to a key element id of a crypto service. For Dilithium we extend this key element index with the key elements of the public key and secret key.

In addition to these key elements we have to configure another set of key elements and a crypto primitive since Dilithium uses a random function in its key generation phase. Three 32 byte random values are needed for `RHO`, `K`, and `S1,S2` sampling. For this we employ the crypto primitive *RANDOM_AES_CTRDRBG* with the key elements `RANDOM_ALGORITHM`, `CIPHER_KEY`, and `RANDOM_SEED_STATE`. These key elements also have to be set prior to the call of Crypto_KeyValidSet and Crypto_KeyGenerate.

Listing 5.8: KeyGenerate Dilithium config.

```
CryptoPrimitive:
  CryptoPrimitive_RANDOMGENERATE_AES_CTRDRBG:
    CryptoPrimitiveService:                   RANDOM
    CryptoPrimitiveAlgorithmFamiliy:          CRYPTO_ALGOFAM_AES
    CryptoPrimitiveAlgorithmMode:             CRYPTO_ALGOMODE_CTRDRBG
    CryptoPrimitiveAlgorithmSecondaryFamiliy: CRYPTO_ALGOFAM_NOT_SET

CryptoKeyElement:
  CRYPTO_KE_KEYGENERATE_ALGORITHM
  CRYPTO_KE_DILITHIUM_BUF01
  CRYPTO_KE_DILITHIUM_KEY_RHO
  CRYPTO_KE_DILITHIUM_KEY_K
  CRYPTO_KE_DILITHIUM_KEY_TR
  CRYPTO_KE_DILITHIUM_KEY_S1
  CRYPTO_KE_DILITHIUM_KEY_S2
  CRYPTO_KE_DILITHIUM_KEY_T0
  CRYPTO_KE_DILITHIUM_KEY_T1
  CRYPTO_KE_RANDOM_ALGORITHM
  CRYPTO_KE_CIPHER_KEY
```

```
20  CRYPTO_KE_RANDOM_SEED_STATE
```

The **signature generation** is implemented as a cryptographic primitive. Therefore we have to add the primitive to the configuration and specify the service, family, and mode, see Listing 5.9. For the service we specify the value SIGNATURE_GENERATE which is a defined crypto service by AUTOSAR, so we don't need to add a new service for Dilithium. For the family we specify CRYPTO_ALGOFAM_DILITHIUM which we already introduced, see above. For the mode we specify CRYPTO_ALGOMODE_NOT_SET since we don't employ a special mode. The secondary family is currently not configured.

To generate the signature the secret key is used. The secret key consists of the elements `RHO`, `K`, `TR`, `S1`, `S2`, and `T0`. For the signature generation we have to set the elements of the secret key and set the key to valid like we seen before in the key generation. Since the signature generation is a cryptographic primitive and therefore follows the job-based approach, we set the plaintext and its length in the Crypto_JobPrimitiveInputOutputType (4.1) in the fields inputPtr and inputLength respectively. Finally, we call Crypto_ProcessJob() to process the cryptographic primitive and the resulting signature will be written to the ouputPtr and outputLengthptr fields of the job.

The signature generation of Dilithium is only available with a synchronous processing and only in the operation mode `SingleCall`. This means that the state machine of the crypto primitive is not able to process the operation modes Start, Update, and Finish individually but rather the whole calculations are processed within one call and the caller is blocked till the request is finished.

Listing 5.9: SignatureGenerate Dilithium config.

```
1 CryptoPrimitive:
2   CryptoPrimitive_SIGNATUREGENERATE_DILITHIUM_NOTSET:
3     CryptoPrimitiveService:                  SIGNATURE_GENERATE
4     CryptoPrimitiveAlgorithmFamiliy:         CRYPTO_ALGOFAM_DILITHIUM
5     CryptoPrimitiveAlgorithmMode:            CRYPTO_ALGOMODE_NOT_SET
6     CryptoPrimitiveAlgorithmSecondaryFamiliy: CRYPTO_ALGOFAM_NOT_SET
```

The implementation of **signature verification** of Dilithium is analogous to the signature generation of Dilithium, with the service SIGNATURE_VERIFY and the public key consisting of `RHO` and `T1`. The signature verification takes the plaintext and the signature as input and together with the public key calculates the verification result.

Dilithium uses the extendable-output functions SHAKE-128 and SHAKE-256 [20] for various purposes like expanding the public matrix A from a seed, sampling the secret vectors $s1$ and $s2$, and as a collision resistant hash (CRH). AUTOSAR currently only allows to configure one secondary primitive for a crypto service via the configuration parameter SecondaryFamiliy. Since Dilithium employs two secondary primitives, this can't be mapped with AUTOSAR at the moment. Therefore, the two secondary primitives SHAKE-128 and SHAKE-256 are implemented as internal functions of Dilithium.

	Crypto_SignatureGenerate:	Crypto_SignatureVerify:
Input:		
	- Plaintext	- Plaintext - Signature
Key:		
	- Dilithium_PRIVATEKEY	- Dilithium_PUBLICKEY
Output:		
	- Signature	- Verification result

Key encapsulation mechanism (KEM)

For the KEM we implement the PQC primitive "CRYSTALS-Kyber" [4], more precise the parameter set Kyber512 of round 3 of the NIST PQC competition[5]. Kyber consists of three functions: key generation, key encapsulation, and key decapsulation. Similar to Dilithium, we first have to add a new algorithm family id for the primitive Kyber (see Listing 5.10), to add the crypto service Kyber.

Listing 5.10: Kyber algorithm family id.

```
1    #if !(defined CRYPTO_FAM_KYBER)
2    #define CRYPTO_FAM_KYBER  0xE2U
3    #endif /* #if !(defined CRYPTO_FAM_KYBER) */
```

The **key generation** function of Kyber generates a secret key and public key. Similar to Dilithium, it is implemented in the AUTOSAR specified function *Crypto_KeyGenerate(uint32 cryptoKeyId)* and therefore accessible via the direct key management API. To generate the key pair we need to first set the key element CRYPTO_KE_KEYGENERATE-_ALGORITHM with the value of CRYPTO_ALGOFAM_KYBER, set the key valid with the function Crypto_KeyValidSet() and finally call Crypto_KeyGenerate(). The output is stored in the configured key elements (see Listing 5.11). The public key KYBER_KEY-_PUBLIC consists of a size of 800 bytes and the secret key KYBER_KEY_SECRET of a size of 1632 bytes. We extend the key element index with the public and secret key for Kyber.

Since Kyber utilizes two 32 byte random values in the key generation phase, we configure the crypto primitive RANDOM_AES_CTRDRBG with the key elements `RANDOM-_ALGORITHM`, `CIPHER_KEY`, and `RANDOM_SEED_STATE`.

Listing 5.11: KeyGenerate Kyber config.

```
1 CryptoPrimitive:
2   CryptoPrimitive_RANDOMGENERATE_AES_CTRDRBG:
3     CryptoPrimitiveService:                    RANDOM
4     CryptoPrimitiveAlgorithmFamiliy:           CRYPTO_ALGOFAM_AES
5     CryptoPrimitiveAlgorithmMode:              CRYPTO_ALGOMODE_CTRDRBG
6     CryptoPrimitiveAlgorithmSecondaryFamiliy: CRYPTO_ALGOFAM_NOT_SET
7
8 CryptoKey_Kyber_KeyGen:
```

[5] https://github.com/pq-crystals/kyber/tree/v3.0

```
  CryptoKeyElement:
    CRYPTO_KE_KEYGENERATE_ALGORITHM
    CRYPTO_KE_KYBER_KEY_PUBLIC
    CRYPTO_KE_KYBER_KEY_SECRET
    CRYPTO_KE_RANDOM_ALGORITHM
    CRYPTO_KE_CIPHER_KEY
    CRYPTO_KE_RANDOM_SEED_STATE
```

For the **key encapsulation** of Kyber we introduce a new interface *Crypto_KeyEncapsulateCalcEnc()* (see Listing 5.12), because AUTOSAR does not consider a key encapsulation mechanism in its standard so far. Since key encapsulation is a key management algorithm, this new interface is part of the key management API and therefore does not follow not the job-based approach like signature generation for Dilithium. Which means that this key management service is only callable in a blocking mode resp. has a synchronous processing. Since this is a new interface of the Crypto Driver module, we also have to adapt the CryIf and Csm module of the Crypto stack to make this key management service accessible to other components in the AUTOSAR architecture. Therefore, we add the function Csm_KeyEncapsulateCalcEnc() in the Csm module and the function CryIf_KeyEncapsulateCalcEnc() in the CryIf module.

Listing 5.12: KeyEncapsulateCalcEnc interface.

```
FUNC(Std_ReturnType, CRYPTO_CODE) Crypto_KeyEncapsulateCalcEnc
(
        uint32  CryptoKeyId,
        uint32  CryptoRandomKeyId,
  const uint8*  PartnerPublicValuePtr,
        uint32  PartnerPublicValueLength,
        uint8*  CiphertextPtr,
        uint32* CiphertextLengthPtr
);
```

For the configuration of the key encapsulation (see Listing 5.13), we first set the AUTOSAR-defined key element CRYPTO_KE_KEYEXCHANGE_ALGORITHM with the value of the algorithm family id of Kyber (see Listing 5.10). The original intent of the key element CRYPTO_KE_KEYEXCHANGE_ALGORITHM is to decide which key exchange algorithm shall be used. Since a KEM can be seen as a unidirectional key exchange, we use this key element for our purposes. After we set the key to valid, we can call the key encapsulation function Crypto_KeyEncapsulateCalcEnc() with the public key of the peer as an input parameter with a size of 800 bytes. The result of this function is the ciphertext to be transferred to the peer and the shared secret. The ciphertext is returned via the function parameter `CiphertextPtr` consisting of 768 bytes. The shared secret is stored in the AUTOSAR-defined key element CRYPTO_KE_KEYEXCHANGE_SHAREDVALUE with a size of 32 bytes. Again we can utilize a key element original intended for key exchange service.

Additional, we have to configure and set the crypto primitive RANDOM_AES_CTRDRBG with its key elements, since the encapsulation needs to generate a 32 byte random secret.

Listing 5.13: KeyEncapsulateCalcEnc Kyber config.

```
CryptoPrimitive:
  CryptoPrimitive_RANDOMGENERATE_AES_CTRDRBG:
    CryptoPrimitiveService:                    RANDOM
    CryptoPrimitiveAlgorithmFamiliy:           CRYPTO_ALGOFAM_AES
    CryptoPrimitiveAlgorithmMode:              CRYPTO_ALGOMODE_CTRDRBG
    CryptoPrimitiveAlgorithmSecondaryFamiliy: CRYPTO_ALGOFAM_NOT_SET

CryptoKey_Kyber_Enc:
  CryptoKeyElement:
    CRYPTO_KE_KEYEXCHANGE_ALGORITHM
    CRYPTO_KE_KEYEXCHANGE_SHAREDVALUE
    CRYPTO_KE_KMNCOMMON_WORKSPACE
CryptoKey_RandomGenerateAESCTRDRBG:
  CryptoKeyElement:
    CRYPTO_KE_RANDOM_ALGORITHM
    CRYPTO_KE_CIPHER_KEY
    CRYPTO_KE_RANDOM_SEED_STATE
```

For the **key decapsulation** of Kyber, we also introduce a new key management interface *Crypto_KeyEncapsulateCalcDec()* (see Listing 5.14) just like for the encapsulation. Therefore we also have to add the new interface in the CryIf and Csm module accordingly.

Listing 5.14: Key management interface for Kyber.

```
FUNC(Std_ReturnType, CRYPTO_CODE) Crypto_KeyEncapsulateCalcDec
(
        uint32 CryptoKeyId,
  const uint8* CiphertextPtr,
        uint32 CiphertextLength
);
```

For the configuration of the decapsulation, we can use the AUTOSAR-defined key elements CRYPTO_KE_KEYEXCHANGE_PRIVKEY and- KEYEXCHANGE_SHAREDVALUE initially intended for Key Exchange. A snippet of the configuration is given in Listing 5.15. To use the key decapsulation, we have to set the key elements CRYPTO_KE-_KEYEXCHANGE_ALGORITHM and CRYPTO_KE_KEYEXCHANGE_PRIVKEY and call Crypto_KeyEncapsulateCalcDec() with the ciphertext as an input parameter. The resulting shared secret is stored in the key element CRYPTO_KE_KEYEXCHANGE-_SHAREDVALUE.

Listing 5.15: KeyEncapsulateCalcDec Kyber config.

```
CryptoKey_Kyber_Dec:
  CryptoKeyElement:
    CRYPTO_KE_KEYEXCHANGE_ALGORITHM
    CRYPTO_KE_KEYEXCHANGE_PRIVKEY
    CRYPTO_KE_KEYEXCHANGE_SHAREDVALUE
    CRYPTO_KE_KMNCOMMON_WORKSPACE
```

Kyber uses various hash functions of the Keccak family during its key generation, encapsulation, and decapsulation phase. These hash functions include SHA3-256 and

SHA3-512, SHAKE-128 as a XOF, and SHAKE-256 as a PRF and KDF. The hash functions are implemented as internal functions of Kyber.

Implementation Aspects

For the key encapsulation mechanism Kyber, our solution adds a new key management API for the encapsulation and decapsulation. Another idea would be to use the existing key management API of the key exchange for the encapsulation and decapsulation of a KEM. In this case, the encapsulation could be mapped to the function Crypto_KeyExchangeCalcPubVal and the decapsulation could be mapped to the function Crypto_KeyExchangeCalcSecret. However, the function parameters are not congruent and thus a misuse of the parameters and additional key elements has to be taken into account.

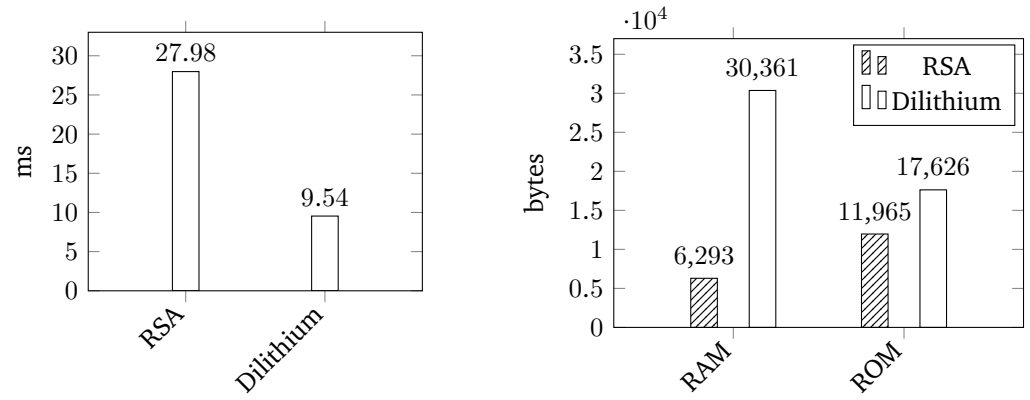
Both Dilithium and Kyber use multiple secondary primitives for various hash applications. Even though AUTOSAR does not support multiple secondary primitives, as already discussed before, Dilithium and Kyber use exclusively symmetric primitives based on Keccak. One implementation improvement would be to combine the symmetric primitives of Dilithium and Kyber.

6. D6.2 - Evaluation of implemented schemes

In this chapter we give an overview of the runtime measurements, the memory consumption, and the feasibility of the implemented primitives regarding to the selected use cases, as well as other evaluation aspects.

6.1. TriCore Performance Measurements

For the performance measurements, the 32-bit AURIX™ TriCore™ microcontroller TC387QP from Infineon is used, for more information see Chapter 3.4. The performance measurements depend on the hardware and therefore the results are depending on the specific test environment. During the runtime measurements, the TC38x CPU frequency is clocked at 300 MHz. The actual runtime is measured using the System Timer Module (STM), which is clocked at 100 MHz. We give all runtime values as an average over 100 iterations. In the following, we measured only the runtime of the primitive itself in the Crypto modul of an AUTOSAR environment. The start and stop time was therefore measured directly before and after the actual algorithm. That means without setting the necessary keys and extracting the result in the keys of the primitive. All available metrics can be found in Table A.1.



(a) runtime measurement

(b) memory consumption

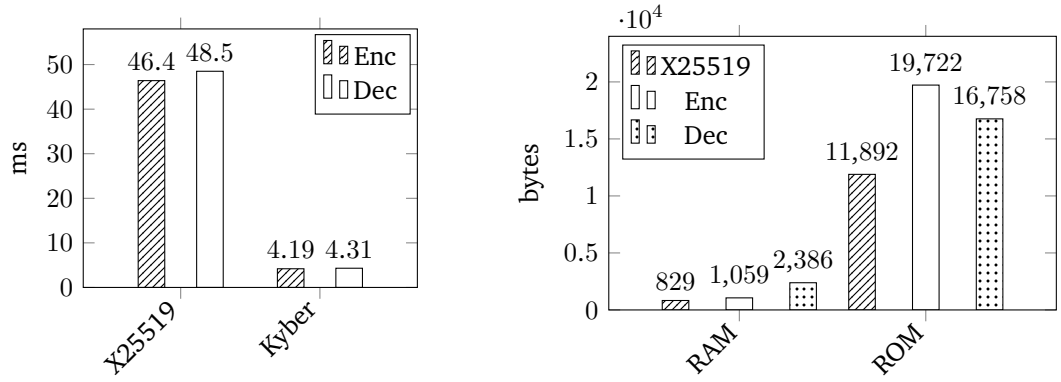
Figure 6.1.: Runtime measurement (in ms) and memory consumption (in bytes) of signature verification of RSA-PKCS1V15 and Dilithium.

6.1.1. Use case: Secure Software Download and Secure Access Control

Since the two use cases Secure Software Download and Secure Access Control employ the same cryptographic primitive, they are discussed together here. For this use case, the signature verification of RSA-PKCS1V15 (RSA) and Dilithium is compared. The runtime measurement and total memory consumption is shown in Figure 6.1. RSA-PKCS1V15 uses the hash algorithm SHA2-256 as a secondary primitive and the runtime was measured with a 2048 bit modulus. The Dilithium implementation uses the parameter set II of the second round of the NIST PQC competition. Both RSA and Dilithium verify a message of 1024 bytes. For the signature verification, Dilithium is 2.9 times faster than RSA as can be seen in the Figure 6.1a. In return Dilithium needs significantly more memory than RSA, especially RAM (see Figure 6.1b). Dilithium needs 4.8 times more RAM than RSA.

6.1.2. Use case: Secure Session Establishment

For this use case, the key exchange protocol ECDH with X25519 is compared to the key encapsulation mechanism Kyber. The Kyber implementation uses the parameter set Kyber512. For the key exchange X25519 the functions KeyExchangeCalcPubVal (CalcPublic) and KeyExchangeCalcSecret (CalcSecret) need to be processed on the embedded device. In this use case, the microcontroller is just responsible for the decapsulation of the key encapsulation mechanism Kyber. This is not a suitable comparison, more generally the comparison of a key exchange protocol and a key encapsulation mechanism in this regard needs to be taken with a grain of salt. Another reason is that it is not clear what is expected from the embedded device in an automotive environment, since a KEM is until now not considered in AUTOSAR. Because of that, we compared both the encapsulation and decapsulation of Kyber to the key exchange X25519, see Figure 6.2.



(a) runtime measurement

(b) memory consumption

Figure 6.2.: Runtime measurement (in ms) and memory consumption (in bytes) of key exchange protocol X25519 and KEM Kyber encapsulation (Enc) and decapsulation (Dec). (a) In case of X25519 “Enc” means CalcPublic and “Dec” CalcSecret.

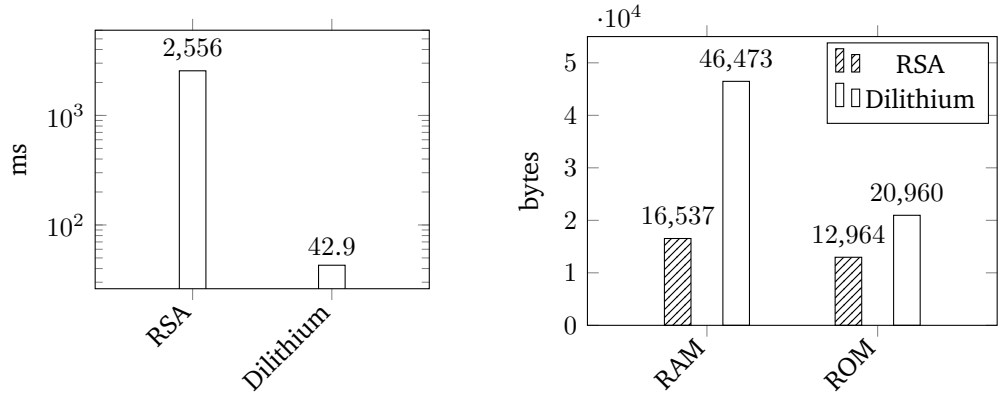
As for the runtime, Kyber encapsulation is compared to X25519 CalcPublic roughly 11 times faster, the same holds for decapsulation and CalcSecret. If we just compare the decapsulation of Kyber to X25519 CalcPublic and CalcSecret, then Kyber is roughly 22 times faster. The key generation is not included, since the use case assumes that the keys are pre-generated.

For the RAM consumption, X25519 and Kyber encapsulation need roughly the same, as can be seen in Figure 6.2b. The decapsulation requires more than twice the amount of RAM then the encapsulation. For the ROM consumption, the encapsulation of Kyber needs roughly 1.7 times the amount of ROM than the key exchange X25519 and the decapsulation needs 1.4 times the amount of ROM. Like Dilithium, Kyber is also faster buts needs more memory then the traditional primitives for these use cases.

6.1.3. Further aspects

In the following, we compare the **signature generation** of RSA and Dilithium, as the signature generation is not part of the given use cases above. We use the same parameters mentioned in section 6.1.1 and employ again a message size of 1024 bytes. The comparison of the runtime and memory consumption is shown in Figure 6.3. As for the runtime, Dilithium is roughly 60 times faster than RSA (Figure 6.3a with logarithmic scale). In return, Dilithium needs more memory than RSA especially more RAM. Dilithium has roughly 3 times the RAM consumption than RSA.

The signature generation of RSA is significantly slower than the signature generation of Dilithium. If we compare the signature generation of RSA to the signature verification of RSA we also see a big difference in runtime by a magnitude. This is because of the costly operations of the RSA signature generation [18].



(a) runtime measurement (b) memory consumption

Figure 6.3.: Runtime measurement (in ms) and memory consumption (in bytes) of signature generation of RSA-PKCS1V15 and Dilithium. Runtime measurement (a) is depicted with logarithmic scale.

Dilithium uses *rejection sampling* during its signing procedure which leads to big differences in runtime for different inputs with the same length. Because of rejection sampling, the signing algorithm can - under specific conditions - reject and restart

the signing procedure to fulfill security and correctness reasons [13]. Therefore the signing process takes more time whenever it rejects, which is highly dependent on the given input. We measured the signature generation of Dilithium with three different test vectors of the same length and could see a difference with up to 5 times more runtime. The value given in Figure 6.3a and Table A.1 is an average of the measured performance.

The current implementation of Dilithium and Kyber is in pure software and is not using any hardware accelerators. However, future implementations could make use of hardware accelerated cryptographic primtives provided by the Hardware Security Module (HSM) of the given microcontroller. This transition would enhance the system's security and performance capabilities by the usage of HW accelerators. One aspect to improve would be symmetric primitives used by Dilithmum and Kyber [13], [4]. Both make use of Keccak based primitives like SHA3 and SHAKE in various sizes. Therefore, these symmetric primitives would be candidates for hardware accelerated implementations. Further, the variant "Kyber-90s" of Kyber uses AES-256 and SHA2 instead of the Keccak based primitives SHA3 and SHAKE. The variant "Dilithium-AES" of Dilithium uses also AES-256. Therefore, another possibility for future implementations would be to implement the variants Kyber-90s and Dilithium-AES, and make use of a hardware accelerated AES-256 and SHA2 implementation. However, the Infineon AURIX TC3xQP HSM is only capable of hardware-accelerated SHA2-256 and AES-128 [1]. Therefore it is only partly suitable for the variant Kyber-90s to enhance the symmetric primitive SHA2-256.

A. Performance Measurements

Primitive	ms	Cycles	RAM (bytes)	ROM (bytes)
RSA SigGen	2556.0	255600049	16537	12964
RSA SigVerify	27.989	2798881	6293	11965
X25519 CalcPublic	46.403	4640328	829	11892
X25519 CalcSecret	48.508	4850844		
Dilithium KeyGen	9.597	959729	52141	27832
Dilithium SigGen	42.898	4289792	46473	20960
Dilithium SigVerify	9.541	954100	30361	17626
Kyber KeyGen	3.525	352519	5485	20506
Kyber Enc	4.192	419248	1059	19722
Kyber Dec	4.315	431471	2386	16758

Table A.1.: All TC387QP metrics of primitives implemented in AUTOSAR.

Bibliography

[1] Infineon Technologies AG. *AURIX™ 32-bit microcontrollers for automotive and industrial applications - Product Brochure*. **version** 1.0. URL: https://www.infineon.com/dgdl/Infineon-TriCore_Family_BR-ProductBrochure-v01_00-EN.pdf?fileId=5546d4625d5945ed015dc81f47b436c7.

[2] Infineon Technologies AG. *TC38x 32-Bit Single-Chip Microcontroller*. **version** 1.2. URL: https://www.infineon.com/dgdl/Infineon-TC38x-DataSheet-v01_02-EN.pdf?fileId=5546d4626f229553016fb316e6cf748b.

[3] Petteri Aimonen. *nanopb: Protocol Buffers for Embedded Systems*. https://github.com/nanopb/nanopb. Accessed: 30 July 2023. 2011.

[4] Roberto Avanzi **andothers**. *CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation*. NIST Post-Quantum Cryptography. version 3.0. **october** 2020. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[5] AUTOSAR Consortium. *AUTOSAR About*. Accessed: June 19, 2023. URL: https://www.autosar.org/about.

[6] AUTOSAR Consortium. *AUTOSAR Adaptive Platform*. URL: https://www.autosar.org/standards/adaptive-platform.

[7] AUTOSAR Consortium. *AUTOSAR Classic Platform*. URL: https://www.autosar.org/standards/classic-platform.

[8] AUTOSAR Consortium. *AUTOSAR Layered Software Architecture*. **version** 4.3.1. 2017. URL: https://www.autosar.org/fileadmin/standards/R4-3/CP/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf.

[9] AUTOSAR Consortium. *Specification of Crypto Driver*. **version** 4.3.1. 2017. URL: https://www.autosar.org/fileadmin/standards/R4-3/CP/AUTOSAR_SWS_CryptoDriver.pdf.

[10] AUTOSAR Consortium. *Specification of Crypto Interface*. **version** 4.3.1. 2017. URL: https://www.autosar.org/fileadmin/standards/R4-3/CP/AUTOSAR_SWS_CryptoInterface.pdf.

[11] AUTOSAR Consortium. *Specification of Crypto Service Manager*. **version** 4.3.1. 2017. URL: https://www.autosar.org/fileadmin/standards/R4-3/CP/AUTOSAR_SWS_CryptoServiceManager.pdf.

[12] AUTOSAR Consortium. *Specification of Crypto Service Manager*. **version** R21-11. 2021. URL: https://www.autosar.org/fileadmin/standards/R21-11/CP/AUTOSAR_SWS_CryptoServiceManager.pdf.

[13] Léo Ducas **andothers**. *CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation*. NIST Post-Quantum Cryptography. round 2. **march** 2019. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions.

[14] International Organization for Standardization. *ISO 17356-3:2005(en) Road vehicles — Open interface for embedded automotive applications — Part 3: OSEK/VDX Operating System (OS)*. International Standard. Accessed: July 3, 2023. **january** 2005. URL: https://www.iso.org/standard/40079.html.

[15] Adam Langley, Mike Hamburg **and** Sean Turner. *Elliptic Curves for Security*. RFC 7748. **january** 2016. DOI: 10.17487/RFC7748. URL: https://www.rfc-editor.org/info/rfc7748.

[16] Adeline Langlois **and** Damien Stehle. *Worst-Case to Average-Case Reductions for Module Lattices*. Cryptology ePrint Archive, Paper 2012/090. 2012. URL: https://eprint.iacr.org/2012/090.

[17] Daniele Micciancio **and** Oded Regev. *Lattice-based Cryptography*. 2008. URL: https://cims.nyu.edu/~regev/papers/pqc.pdf.

[18] Kathleen Moriarty **andothers**. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. **november** 2016. DOI: 10.17487/RFC8017. URL: https://www.rfc-editor.org/info/rfc8017.

[19] Marcel Müller **and** Michael Meyer. "QuantumRISC WP2 Report: Analysis and Optimization of PQC schemes". 2022. URL: https://quantumrisc.org/results/quantumrisc-wp2-report.pdf.

[20] NIST. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. **august** 2015. DOI: https://doi.org/10.6028/NIST.FIPS.202.

[21] David Noack **andothers**. *QuantumRISC WP1 Report: Use Cases and Requirements*. 2020. URL: https://quantumrisc.org/results/quantumrisc-wp1-report.pdf.