UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONA**TECH**

Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

Quantum Resistant Authenticated Key Exchange for OPC UA using Hybrid X.509 Certificates

A Master's Thesis
Submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by
Patrik Scheible

In partial fulfilment
of the requirements for the degree of
MASTER IN TELECOMMUNICATIONS ENGINEERING

Advisor:
Prof. Dr. Jorge L. Villar
Sebastian Paul, M.Sc.

Barcelona, April 2020

intentionally left blank

Title of the thesis:
Quantum Resistant Authenticated Key Exchange for OPC UA using Hybrid X.509 Certificates

Author:
Patrik Scheible

Advisor:
Prof. Dr. Jorge L. Villar (UPC)
Sebastian Paul, M.Sc. (Robert Bosch GmbH)

Abstract

While the current progress in quantum computing opens new opportunities in a wide range of scientific fields, it poses a serious threat to today's asymmetric cryptography. New quantum resistant primitives are already available but under active investigation. To avoid the risk of deploying immature schemes we combine them with well-established classical primitives to hybrid schemes, thus hedging our bets. Because quantum resistant primitives have higher resource requirements, the transition to them will affect resource constrained IoT devices in particular. We propose two modifications for the authenticated key establishment process of the industrial machine-to-machine communication protocol OPC UA to make it quantum resistant. Our first variant is based on Kyber for the establishment of shared secrets and uses either Falcon or Dilithium for digital signatures in combination with classical RSA. The second variant is solely based on Kyber in combination with classical RSA. We modify existing open-source software (open62541, mbedTLS) to integrate our two proposed variants and perform various performance measurements.

Acknowledgements

I would like to thank my supervisor at the Universitat Politecnica de Catalunya, Professor Jorge L. Villar, who provided excellent guidance during my work on this thesis.

I would like to express my gratitude to the Robert Bosch GmbH, which supported my thesis and I want to especially thank Sebastian Paul, who supervised my work, gave exceptional input and helped me to solve many many problems during the journey.

Finally I would like to thank my parents for their constant support throughout my studies.

Revision History and Approval

05.03.2020 First Draft

23.03.2020 Included Implementation of 'Variant Two'. Improved Charts.

17.04.2020 Corrected typos; Mentioned that Kyber is used as KEM in experiments. Improved bar charts in the results section.

Contents

List of Figures

List of Tables

1 Introduction

Today's industrial control systems (ICSs) often comprise a network of different components such as sensors and actuators, programmable logic controllers (PLCs), supervisory control and data acquisition (SCADA) systems as well as human machine interfaces (HMIs). Furthermore, manufacturing execution systems (MESs) and enterprise resource planning systems (ERPs) on higher levels enable production control and scheduling from a business perspective. New applications like data mining and machine learning allow to improve the production process but require to interconnect these systems on all levels, thus forming a cyber physical system (CPS). The industry is slowly becoming aware of the fact that ICSs suffer from the same security threats as classical computer networks and are starting to deploy security measures. An example is the vendor independent machine to machine (M2M) communication protocol OPC Unified Automation (OPC UA). It was designed from the ground up with security in mind and uses strong cryptography.

But recent advances in quantum computing start to threaten the security measures. Most of the currently used asymmetric cryptography schemes rely on the hardness of integer factorization and the discrete logarithm of large numbers. An algorithm, for quantum computers, to efficiently solve the two aforementioned problems, i.e. in polynomial time, already exists today [1]. The only missing part to render most current asymmetric crypto systems useless is a large scale quantum computer. However, it seems quite possible that such a large scale quantum computer will become reality within the next few decades. This threat has lead to active research in the field of quantum resistant crypto algorithms and their incorporation into protocols. But the effort focuses mainly on standard IT and doesn't consider the ICS devices' peculiarities such as small memory and reduced computing resources.

The following sections of this introduction will explain the development of industrial automation systems towards the Industrial Internet of Things (IIoT) with a special focus on security measures in place. Then, a quick overview of the OPC UA protocol and introduction to the threat of quantum computers on its security will be given.

1.1 Industrial Internet of Things

An ICS can be separated into the Field-, Direct Control-, Supervisory Control-, Production Control- and Production Scheduling levels as depicted in Figure 1. The information-flow between the levels used to be as follows: The field-, direct- and supervisory control level used to communicate via proprietary field bus protocols. The higher levels with the MESs and ERPs are located in the office IT network of the company. Data was transmitted manually from the supervisory control level to the higher layers, which could have been as simple as a worker reading gauge values and noting them down on a report sheet. Typical IT security threats, such as worms, viruses and trojans, did not apply to these systems.

However, with cheap sensor and network technology becoming more widely available, a tendency to deploy standard components and technology in ICSs emerged. Field bus protocols were replaced with Ethernet-like standards such as ProfiNet for PLC communication and

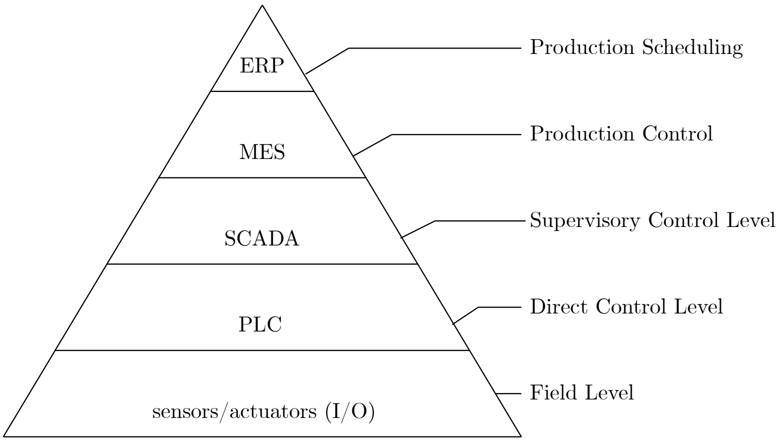


Figure 1: Automation pyramid showing at which logical level the functions are located in an industrial network.

standard Windows PCs are used to run SCADA software. In this thesis these networks will be referenced as industrial control system, industrial network or automation network synonymously. The rest of a company's network with typical services such as e-mail, web servers and access to the internet will be called office network, corporate network or business network.

Even though it was always considered a security risk to connect the automation to the corporate network, it is, in fact, done in practice [2], which is not surprising since in combination with big data analytics it will increase efficiency significantly [3, p. 4].

The term Industrial Internet of Things will be used for industrial networks that are connected to the office network.

Apart from the enormous advantages, connectivity undermines the traditional main security feature of these networks: Strict separation from the office IT and the internet [2], [4]. A growing number of incidents have shown that new strategies for industrial network security are needed. The incidents range from a worm intended for office networks but also making its way into the industrial network of a nuclear power plant [5] over malware gathering information specifically from ICSs [6] to the infamous Stuxnet sabotage attack [7]. A new concept coming up in the industry is "defense in depth" [4], [8]–[10], which uses not only perimeter security (i.e. strict separation of networks) but follows the strategy of "Prevent" – "Detect" – "Response", where the focus of this thesis on encryption and authentication falls in the first category "Prevent". An example of such an in-depth security building block is the use of the OPC UA protocol, which is designed for the communication between industrial devices and offers a range of security features.

1.2 OPC Unified Automation

OPC UA was first published in 2006 by the OPC Foundation, a consortium of the automation industry. It is a set of standards that define a data model for industrial communication that helps to increase vendor independent interoperability throughout all levels of the automation pyramid. Additionally, OPC UA describes different encodings for the data models as well as network protocols. Exemplary data which is exchanged using OPC UA are sensor values and set point values. For instance, think of an OPC UA enabled air conditioner system, that accepts temperature as a writeable value. It also senses the humidity, which it exposes for other devices as a readable value. A PC could connect to the air conditioner, display the humidity and let the user set the temperature via a graphical user interface (GUI) [11, p. 85]. Of course this is only a minimal example, in practice automation systems are usually more complex.

Currently OPC UA supports the data encodings *UA Binary*, *XML/text* and *JSON*. Each encoding uses different network transport layers, i.e. *XML/text* uses HTTP and *JSON* uses WebSockets, both on top of transport layer security (TLS). They are intended for the use in higher layer systems that typically use desktop PC hardware and can deal with high protocol overhead. For *UA Binary*, OPC UA defines its own transport layer *OPC UA TCP* and a service called *secure channel* to provide confidentiality and integrity [11, p. 198, 211]. The *secure channel* design is loosely based on TLS. Due to its very small resources footprint, compared to *XML/text* and *JSON*, *UA Binary* is used in the lower levels of the ICS.

From the early design phases on, OPC UA seriously considered IT security [12, p. 9]. Network traffic can be encrypted to provide confidentiality and all messages can be signed to provide integrity. The use of PKIs allows the devices to authenticate each other, which makes it much harder for an attacker to insert malicious devices into the network or for corrupted devices to impersonate others. Security evaluations regarding the protocol itself as well as its popular implementations have been carried out and couldn't find any serious flaws [13], [14].

The OPC UA protocol stack is currently available for Python, C#.Net, C and Java. Prototypical implementations in the scope of this thesis will make use of the open source library *open62541* which is implemented in C and published under the Mozilla Public License v2.0[1] [15]. Since there is already research work carried out for quantum resistant versions of TLS [16], [17] this thesis will focus on the security of the binary version of OPC UA only.

1.3 Post Quantum Cryptography

The asymmetric crypto algorithm used in OPC UA depends on the security profile selected, but the recommendation is RSA with a key size of 2048 bits. In this thesis, when RSA is mentioned without a key size, by default 2048 bits is assumed. The security of RSA can be reduced to the problem of factorizing large integers, which is believed to be hard because it can only be solved in subexponential time on a classical computer. However Peter Shor developed an algorithm that makes use of superposition in a potential quantum computer, thus solving the factorization problem in polynomial time in $O(logN)$, where $log(N)$ is the number of digits of N [1]. Running Shor's algorithm requires a quantum computer with at least $2 + \frac{3}{2}log(N)$ fault tolerant qubits [18]. For RSA 2048 ($log(N) = log(2^{2048}) = 2048$)

---

[1] Available from https://open62541.org/ and https://github.com/open62541/open62541

this would be 3074. However, physical qubits are not fault tolerant and many of them are needed in order to emulate one logical (fault tolerant) qubit. Thus running Shor's algorithm for large numbers requires several millions [19] if not hundreds of millions of qubits [20]. In contrast, the latest breakthrough in quantum computing, as of 2019, is a 53 qubit quantum computer [21]. Even though today's quantum computers are far from being powerful enough, from here on the term quantum computer will be used for a possible future large scale version that is powerful enough to break RSA.

The possibility of such a quantum computer has sparked interest in a field of research called quantum resistant or post quantum (PQ) cryptography. There are some algorithms available that are based on problems for which no efficient conventional nor quantum solution is found yet. These algorithms and the underlying mathematical problems are under active investigation and it is quite possible that cryptanalysis uncovers new weaknesses or bad parameter choices. However as research progresses, confidence into these new algorithms will grow and they will be ready to be used as a replacement of RSA.

The remaining question is: When should we start the transition to quantum resistant algorithms? To answer that question we have to consider the following [22]:

- The time we want the data to remain secret denoted as x. It is possible that an attacker records encrypted data today and decrypts it in the future when he becomes able to do so. We have to ensure that encryption happens only after time x, when the data has become irrelevant. Note that for authentication $x = 0$ since it can not be broken afterwards.
- The time y that we need for the transition to new cryptographic systems for all our devices. This value depends on the measures that we have to take. If our current hardware is capable of executing new algorithms, software updates might be sufficient. However, if new hardware is required this can be a more complex task. Especially in the automation industry, devices often are used for >15 years in order to get a reasonable return of investment.

If we say z is the time we have left until a quantum computer is available, then we have to ensure that $x + y < z$. x is a question of policy and to estimate y we have to investigate how complex the migration time is. But z is hard to predict. Michele Mosca, a renowned researcher in the field of quantum algorithms, estimates a $1/7$ chance to break RSA 2048 by 2026 and a $1/2$ chance to break it by 2031 [22].

This estimation shows that it is important to start investigating quantum resistant schemes and to analyse the effort that has to be taken to implement those in a large scale. For the automation industry it will be very beneficial to have prototypes of quantum resistant communication devices available as soon as possible in order to properly estimate the migration time y.

In this thesis, the terms quantum resistant or post quantum cryptography describe all schemes that are hard to break, even for an attacker with access to a quantum computer, opposed to conventional or classical cryptography, that is only hard to break on a conventional computer.

1.4 Hybrid Cryptography

As explained before, most quantum resistant crypto primitives are rather new and have not withstood many years of cryptanalysis yet. This leaves us in the situation that on one hand, we have the threat that in the not so distant future conventional crypto might be broken, on the other hand, there is a risk that quantum resistant schemes may suffer from teething problems.

A good compromise is a hybrid approach, i.e. to combine conventional and quantum resistant schemes in such a way that an attacker has to break both in order to break the system. For instance, this concept is very useful in key exchange protocols, where a key can be derived from multiple partial keys such that knowledge of less than all partial keys is completely useless for an attacker. We exchange each partial key using a different scheme. Also for signature schemes it is very easily applicable, by signing the message twice, each with a different scheme. Only when both signatures can be verified we consider the message authentic.

Note that the term hybrid is ambiguous in the context of cryptography. Typically it describes a system that combines symmetric with asymmetric schemes. However in the context of PQ crypto, as well as in this thesis, hybrid cryptography refers to the principle described above.

1.5 Scope and Goals

The primary aim of this work is to present a novel method for an authenticated quantum resistant hybrid key exchange in OPC UA. In order to reach this goal we will outline an exemplary PKI suitable for the industrial environment. Then we will evaluate different ways of using X.509 compliant quantum resistant hybrid certificates. Finally we will put these parts together to an authenticated key exchange method based on existing work about unauthenticated but quantum resistant hybrid key exchange for OPC UA. We will focus on the client/server communication model and not consider the publisher/subscriber model which has just very recently been introduced into OPC UA and has not been properly adopted by the protocol stack implementations yet.

Thus, the main research question is:

How can we design an authenticated hybrid key exchange that combines conventional and quantum-resistant cryptographic primitives for an OPC UA based industrial network, using hybrid X.509 compliant certificates?

In particular following questions shall be answered:

- How can we incorporate additional quantum resistant public keys into X.509 certificates while maintaining backwards compatibility, in the setup of an OPC UA secure session?
- How can we digitally sign certificates, used in the OPC UA secure channel setup, within a quantum-resistant PKI that utilizes conventional and post-quantum signature schemes.
- How can we use these certificates to authenticate a key exchange in OPC UA.
- What are the performance impacts of our proposed solution on currently used micro controller hardware?

2 State of the Art

This section shall give an overview over all the "building blocks" that are used to arrive at the goal of this work. We first have to take a detailed look at the conventional security features of OPC UA, in particular on how to setup secure channels. While OPC UA is intended for the use with PKIs, the standard leaves the actual design of said PKI open [11, p. 212]. Therefore we will point out the peculiarities of industrial PKIs in contrast to normal PKIs and survey different concepts in literature. Furthermore we will introduce relevant quantum resistant cryptographic primitives. Finally, different hybrid schemes will be explained, setting the stage for authenticated key exchange methods.

2.1 OPC UA

As explained in Section 1.2, OPC UA is not only a network protocol, but defines an information model for industrial process data and different ways of how to encode this data. It also describes ways of transmitting the encoded data through the network, which can be seen as the protocol part. For higher level systems, the data is transported using the standard protocols HTTP and WebSockets, both relying on TLS for security. For them, it seems appropriate to wait for quantum resistant versions of TLS to be used in the web and then adopt them into OPC UA.

However in order to avoid as much protocol overhead as possible, OPC UA defines its own transport layer based on TCP/IP and names it *OPC UA TCP* and basing security on its own secure channel layer. The data is encoded using the *UA Binary* encoding, defined in part 6 of the OPC UA standard [23]. For this protocol it is necessary to investigate a post quantum secure version on its own. Thus, this work will solely focus on *OPC UA TCP* with *UA Binary* encoding and will refer to it simply as OPC UA protocol.

2.1.1 Security of OPC UA

Part 2 of the OPC UA specification [10] gives an overview over the security goals that the standard considers and describes a threat model.

Then it defines the term *security profile* [10, p. 16] as a set which enumerates the security functionalities that a certain OPC UA product offers. A security profile can contain several *security policies*. A policy defines the concrete algorithms and cryptographic primitives that have to be used for signing, encryption and key derivation. For instance, "Basic256Sha256" stands for a cipher suite with RSA 2048, Sha256 and AES-CBC 256. Usually the administrator of an application decides which security policies he wants to enable. Additionally, each connection has one of the three security modes: "None", "Sign" and "SignAndEncrypt". According to the OPC UA specification, the security policy "None" is intended only for testing and the security mode "None" can only be used with this policy. When the security mode "Sign" is used, the session key exchange happens encrypted and all other communication is secured by a message authentication code (MAC) but not encrypted. This makes sense since in the industrial environment the goal integrity is usually much higher prioritised than confidentiality [24, p. 279].

2.1.2 Protocol Overview

Client – Server connections in OPC UA are organised in logical layers. In the highest layer there is a session between client and server. Users are authenticated and authorized per session. The session data is transmitted inside a secure channel. Besides integrity and confidentiality, the secure channel layer also provides authentication between the applications by the means of certificates. On the lowest level, the transport layer is responsible for the delivery of messages and functions such as data fragmentation. Figure 2 shows the layers.

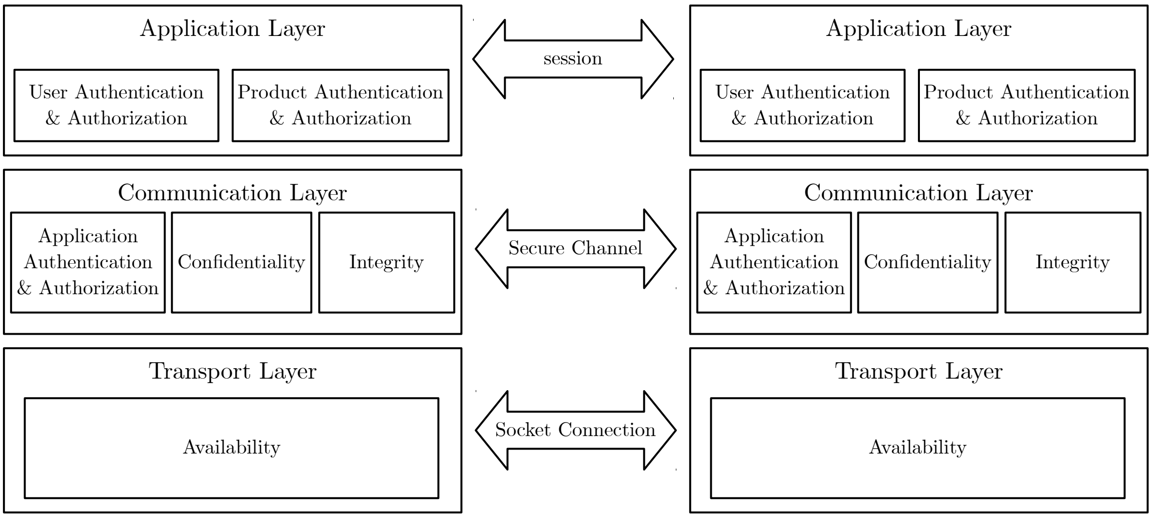


Figure 2: OPC UA communication layers [11, p. 211]

An OPC UA server offers different endpoints. They could provide different functionalities, or the same functionality with a different combination of security policies and security modes. For simplicity herein we will focus on very simple servers that just offer a single functionality and the endpoints only differ in their security configuration. For example, a server could offer two endpoints with the security policy "Basic256Sha256" and the security mode "Sign" for the first endpoint and with the same policy, but with the security mode "SignAndEncrypt" for the second endpoint. Both endpoints expose a single read only variable as their functionality. The server provides a certificate for each endpoint, even when each endpoint uses the same one. The client can then decide which endpoint he wants to connect to.

A client either knows available endpoints in advance or he queries them by sending an unencrypted *getEndpointsRequest* to the server. The server answers with a *getEndpointsResponse*, which contains all available endpoints as well as the server's certificate with his public key.

In detail, first the *OPC UA TCP* transport layer has to be established. Therefore, the two parties negotiate the maximum message length, called chunk size, by sending a HEL message from the client and replying with an ACK message from the server. This serves the purpose to initialize appropriately sized message buffers on each side and also determines the maximum chunk size. Before the *getEndpoint request* and *response* can be transmitted it is required to establish a secure channel first, however, it uses the security policy "None" which means no encryption and integrity checks are performed. Finally the connection is terminated again. The upper part of Figure 3 shows this in detail.

Subsequently, a new connection is established with a secure channel that uses a security policy different from "None", picked by the client from the available policies on that server.

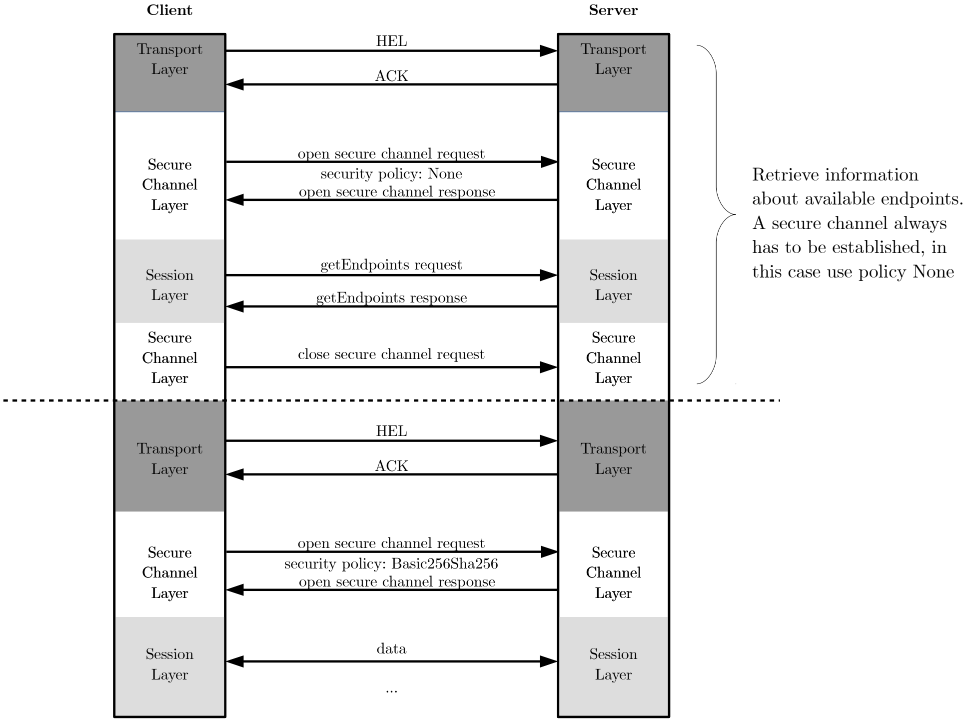


Figure 3: Detailed view of the exchanged messages in order to connect to a server.

On top of the secure channel, a session is established. The session is independent from the secure channel, which means that the secure channel can be closed and reopened without terminating the session.

2.1.3 Secure Channel in OPC UA

We will take a closer look at the secure channel: During the establishment of the channel, asymmetric cryptography is used to authenticate the applications and to derive a shared session key. Afterwards symmetric cryptography is used to encrypt and provide message authentication.

After the connection is established on the transport layer, we need to open a secure channel as shown in Figure 4. Thus, the client sends an *openSecureChannel* request to the server. This request contains the client's certificate, the security policy he wants to use (not shown in the figure), a random number called $nonce_{client}$, a thumbprint of the server's certificate tb_{server} and a signature s_1 over the whole message. $nonce_{client}$ and the signature s_1 are encrypted using the server's public key. The server receives that message and verifies the client's certificate and the thumbprint. Then, he uses his own private key to decrypt the message and verifies the signature using the client's public key from the certificate.

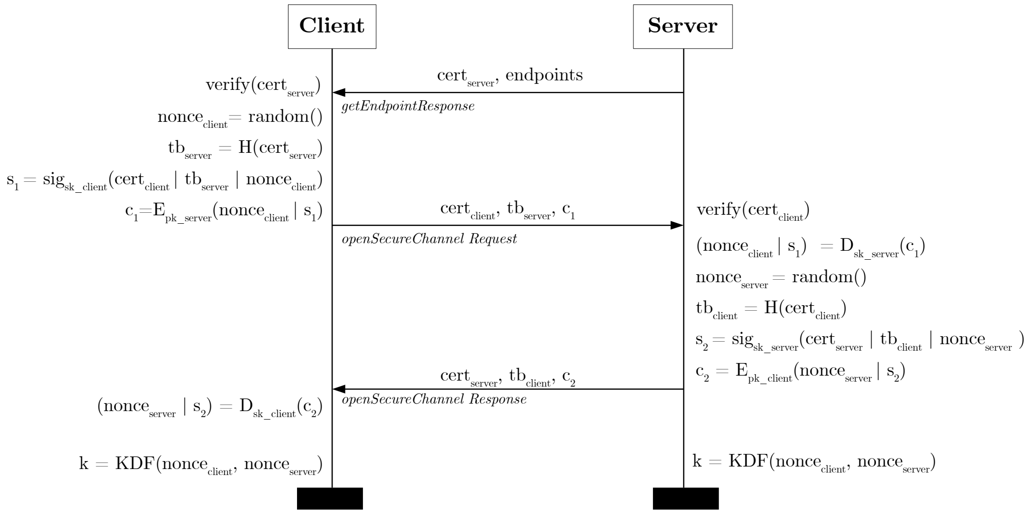


Figure 4: Key exchange in OPC UA. $H()$ is a cryptographic hash function, $E_{pk}()$ is an encryption function using the public key pk, sig_{sk} is a signature function using the private key sk, $D_{sk}()$ is a decryption function using the private key sk and $KDF()$ is a key derivation function.

Next, the server will generate his own random number $nonce_{server}$ and encapsulate it into a message, the same way as the client did, adding a signature and encrypting the packet using the client's public key and sends it to the client. Now both sides know $nonce_{client}$ and $nonce_{server}$ and use them as input to a key derivation function to generate the symmetric session key k.

2.2 X.509 Certificate Format

X.509 certificates have the purpose of binding an identity to a public key. This is done by writing the name and public key together into a file, then computing a hash over this file and attaching a signature of the hash at the end of the file. To verify a certificate, the verifier needs to know the public key of the signer of the certificate, also called issuer, which he could as well obtain via a certificate. This way, a chain of certificates can be constructed, however the last public key in the chain has to be trusted, thus it is called the trust anchor. The system of certificate chains forms a part of a PKI which is further described in Section 2.3.

This section explains the structure and file format used for X.509 certificates as specified by RFC 5280 [25]. In particular we refer to version 3 called X.509v3. For clearer notation we only refer to X.509 and mean X.509v3 implicitly. The format is described in Abstract Syntax Notation One (ASN.1) [26], which is comparable to a *struct* in the C programming language but more generic and independent from actual programming languages. Figure 5 gives a graphical representation of the important parts of the X.509 data structure: The certificate consists of three data fields: While *signatureAlgorithm* specifies which algorithm was used, *signatureValue* contains the actual signature over the binary representation of *tbsCertificate*. *tbsCertificate* itself is constructed from more data types and contains organizational information, the subject, i.e. the identity that is bound to a public key, the issuer, the public key of the subject and a field with extensions. The extension field consists of a sequence of extension objects, each having an ID, an attribute that defines if it is critical and the binary extension data. When a software parses the certificate it will check the ID of the extension and then decide how to interpret the binary extension value. If the extension ID is unknown and critical equals false, the extension is simply ignored but for unknown critical extensions the verification will fail. RFC5280 defines 15 standard extensions that are known by common software, however, it is possible to define custom extensions to be used with specialized tools.

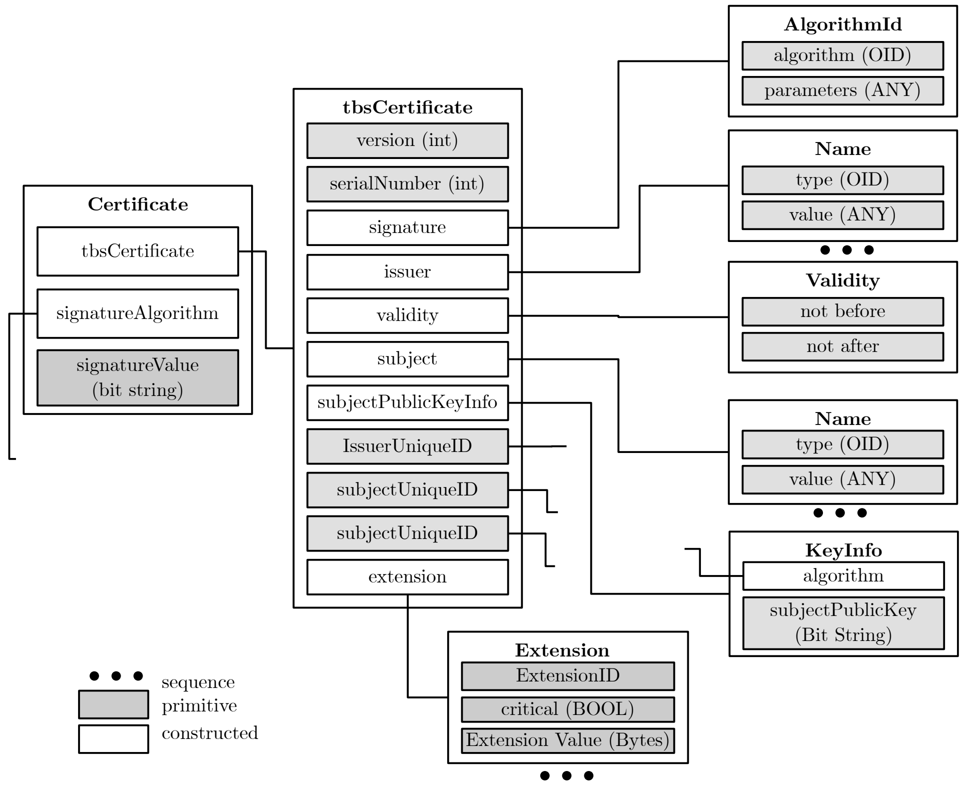


Figure 5: Structure of an X.509 certificate. The three dots mean that this type is repeated in a sequence $0 - n$ times. Primitive data types can be encoded directly and constructed data types consist of a set of primitive types.

Each data type can be encoded into a binary representation as defined by the Distinguished Encoding Rules (DER) [27]. Every container in Figure 5 is represented as a sequence of data. For example, the container *Certificate* is a sequence of the three objects inside, represented by their respective binary data. The binary representation of a certificate therefore is one byte that identifies a sequence followed by several bytes defining the length of the sequence. After that, the binary data of *tbsCertificate* follows. The end of *tbsCertificate* and the start of *signatureAlgorithm* can be determined by examining the first few bytes of *tbsCertificate*, which contain the length of this sequence.

To sign a certificate we must follow these steps:

1. setting the proper data in *tbsCertificate* (subject, signer, public key, etc.)
2. set the proper algorithm identifiers
3. encode *tbsCertificate* into its binary representation
4. calculate the signature form this binary data and write it to *signatureValue*

Later sections discuss the use of the *Extension* field to adopt this format for the use in hybrid public key schemes.

2.3 Public Key Infrastructures for IIoT

As mentioned before, parts of the OPC UA security depend on certificates. However the standard does not specify how certificates are signed [11, p. 212] and in general does not define the structure of a PKI. Of course the first idea would be to use the same PKIs that are already in use for the web. But it turns out that the requirements for industrial networks are different. For example, in the web, typically only the server is authenticated, whereas in an industrial network usually mutual authentication is desired [28].

Since awareness for IT security in industrial networks started to rise only in the past few years, there exist no well established best practices for industrial PKIs yet. The NIST's "Guide to Industrial Control Systems (ICS) Security" from 2011 [4] discusses mainly firewall configurations, i.e. network separation, as a means of technical security measures and does not consider PKIs as part of a cyber security strategy for industrial networks. On the contrary, more recent studies [9, p. 13] do start to demand PKIs for mutual authentication between devices, also as a result of the increasing demand for cloud services.

2.3.1 Differences between Classical and Industrial PKIs

Because PKIs have been well studied for classical IT environments, it is interesting to point out some differences of their industrial counter parts.

- In an industrial PKI, a certificate identifies a device, whereas a certificate classically identifies a person or organisation.
- The administrative overhead that is acceptable to sign a certificate is much lower. While it is feasible to check an ID card before signing a certificate, this approach would not scale very well if for every device that has to be replaced in a factory plant we would have to manually sign its certificate. In fact, in classical PKIs you can observe that only servers are equipped with certificates since it is considered too costly and intricate for every single user [29, p. 18]

This leads to problems mainly in provisioning of new devices in the factory plant. A possible solution is to consider two separate PKIs: The operator and the manufacturer PKI [28]. The manufacturer of the devices equips all his produced devices with a manufacturer certificate that contains data such as a public key, serial ID etc. and is signed by the manufacturer certificate authority (CA). During provisioning the new device can setup a secure connection with the operator PKI's registration authority (RA). The operator RA can verify the data of the new device using the manufacturer CA's public key and decides if the device can be trusted[2]. Now the new device has to generate a new key pair and send a certificate signing request to the RA. Once this certificate is signed, all other devices in the ICS will trust this device as well.

Independently from available solutions we can formulate following special requirements for an industrial PKI:

- Run in an isolated network, possibly without internet connection.
- Every end device needs a certificate (not only servers).
- Very little to no human interaction when provisioning certificates to new devices.
- Signature verification must be possible on resource constraint devices (regarding memory and CPU power).

2.3.2 Exemplary PKI

For the purpose of this thesis, we find ourselves in the situation that on one hand, there are no well established industrial PKI solutions yet, on the other hand, that a quantum resistant authentication scheme depends on such an infrastructure. Thus we will rely on evaluations done in research literature in order to sketch out a PKI scheme that can be used for evaluation in conjunction with quantum resistant hybrid certificates. In [30] the following three trust models are shown:

- A *Web of Trust* as it is used in pretty good privacy (PGP).
- The *Direct Trust Model* where all certificates are installed manually in a trust list.
- A *hierarchical PKI* where certificates are signed by a root CA and intermediate CAs.

And they come to the conclusion that the *Web of Trust* and the *Direct Trust Model* do not scale sufficiently [30]. Thus we will consider a typical hierarchical PKI with one root CA at the corporate level and intermediate CAs for each factory plant and assume that the additional requirements from 2.3.1 can be fulfilled by introducing concepts like the manufacturer and operator PKI.

---

[2] For instance the RA could have rules like "always trust model x from vendor y", it could know the serial numbers of all purchased devices or there could be a message that a human supervisor has to confirm. Additionally a log entry can be made to provide an audit trail.

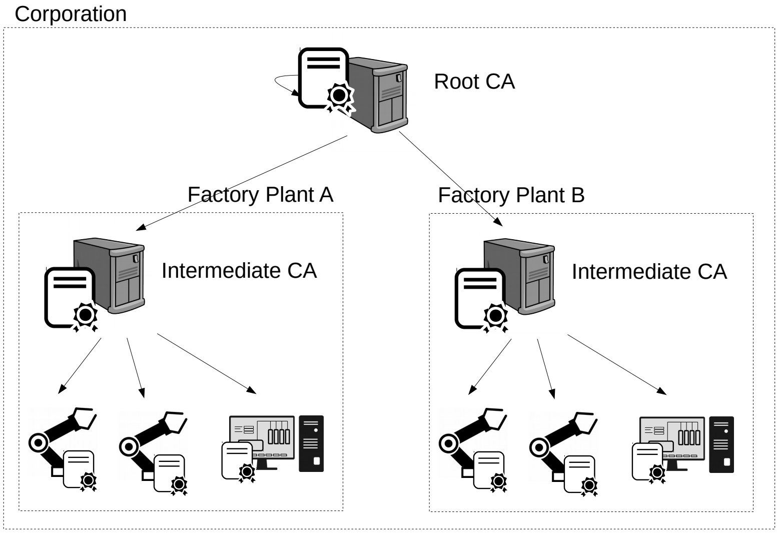


Figure 6: Exemplary industrial PKI for a company with multiple factory plants.

Figure 6 shows the exemplary industrial PKI that we consider for this thesis. The company operates one root CA. Every factory plant runs an intermediate CA that is signed by the root CA. All devices in the factory plant have certificates that are signed by the corresponding intermediate CA and need to have the root certificate installed. By caching their intermediate CA's certificate, the devices inside the factory plant, between which we expect the majority of communication, do not need to exchange long certificate chains. Only when devices between factory plant A and B need to communicate, they have to include their intermediate certificate into the chain.

2.4 Security Levels

When cryptographic primitive shall be compared or when primitives have to be parametrized to have an equivalent security, the need to quantize the security of algorithms arises. For example, when two or more primitives are combined in a hybrid scheme it is desired to have the same level of security for each algorithm.

This security level is commonly expressed in bits, meaning that an attacker has to perform at least 2^n computational steps in order to break a n-bit secure cipher or hash function with n bits of output. Symmetric ciphers for which the best known attack is a brute force search over the whole key space have a security level corresponding to the key length. For hash functions with n-bit output the preimage resistance security level is also n-bit, because on average we have to perform 2^n calculations, but the collision resistance is usually much lower due to the birthday paradox.

For asymmetric primitives there are more efficient attacks available than brute force searches. Therefore we have to relate the security level to the computational steps performed by these attacks. For example, to achieve roughly 128-bit security for RSA, a key length of 3072 bits is required [31, p. 63]. Actually breaking RSA, i.e. solving the RSA problem, is not harder than factoring the modulo $N = p \cdot q$ of the public key [32, p. 1065]. Similar, the Diffie-Hellmann (DH) key exchange primitive can be broken by solving the discrete logarithm problem [33].

The NIST has defined five security levels used in their PQ cryptography project (see Section 2.5.1). Level L1, L3 and L5 correspond to 128-bit, 192-bit and 256-bit security respectively. L2 and L4 are defined as algorithms that can be broken with the same computational resources as required to find a collision in SHA-256 and SHA-384 [34, p. 16]. These levels will be used throughout the thesis to compare cryptographic primitives.

Bruce Schneier recommends 128 bit security for the most valuable secrets [35] so at least for sensor data etc. this security level should be sufficient.

2.5 Post Quantum Cryptography

Today, there exist no algorithms that can solve integer factorization or the discrete logarithm problem in polynomial time on a classical computer. Nor is there a method to reduce the number of steps in a brute force search on the keyspace of, for instance Advanced Encryption Standard (AES) or hash functions.

However two algorithms, designed for quantum computers, tackle these problems, thus rendering crypto systems based on the above described problems insecure. The first one, Grover's search algorithm [36], allows to perform a brute force search with a square root speed up. This means that the complexity is reduced from $O(n)$ to $O(\sqrt{n})$ and the bit level security of symmetric crypto primitives such as AES or Secure Hash Algorithm (SHA) is halved. If we want to achieve an equivalent bit level security in a post quantum scenario we have to double the key sizes. This change in security parameters is feasible and therefore we can conclude that quantum computers with Grover's algorithm do not pose a serious threat to symmetric ciphers and cryptographic hash functions.

On the contrary, Shor's algorithm is capable of factorizing large prime numbers and solving the discrete logarithm problem in polynomial time on a quantum computer [1]. Unlike the solution for symmetric primitives, increasing the key size is a practically ineffective counter measure. Thus, we have to consider all commonly used asymmetric cryptographic primitives as broken in a PQ scenario. Table 1 lists the most common algorithms and the effect of quantum computers on them. The only strategy left is to switch to new algorithms that do not depend on the difficulty of integer factorization and the discrete logarithm.

The most promising quantum resistant cryptography systems can be divided into four categories [37] that are explained in more detail in the following subsections:

- Code based
- Hash based
- Multivariate Polynomial based
- Lattice based

Table 1: Effects of Grover's and Shor's algorithm on commonly used cryptographic primitives [37].

Type	Scheme	Post-quantum security level
Public key encryption	RSA	Broken
	ECC	Broken
Signatures	RSA	Broken
	DSA	Broken
	ECDSA	Broken
Key exchange	DH	Broken
	ECDH	Broken
Symmetric key encryption	AES-128	64 bit
	AES-256	128 bit
Hash functions	SHA-256	128 bit
	SHA-3-256	128 bit

A fifth category, isogeny based, is also emerging, however is not considered in this thesis as it is still a very new field.

2.5.1 NIST Post-Quantum Cryptography Project

Different standardization organisations have started to address quantum secure cryptography. The ETSI's Quantum Safe Working Group [38] has published several studies on post quantum scenarios and on quantum safe algorithms, the IETF has looked into some proposals for the integration of quantum safe algorithms in protocols such as TLS and X.509 certificates [25], [39], [40] and the US NIST is running their Post-Quantum Cryptography standardization project [41].

The NIST PQ project's goal is to specify one or more publicly disclosed digital signature, public-key encryption and key-establishment algorithms that are secure even in the presence of a quantum computer by 2024. Therefore, they have asked the public for proposals for quantum secure key encapsulation methods (KEMs) and signature schemes. The proposals have undergone vivid discussions among the community of cryptography experts and the algorithms left in round 2, which is the current status of the project as of writing this thesis, are promising candidates for future standardized post quantum schemes. Table 2 shows the 17 KEMs and 9 signature schemes in round 2. Due to the progress of the NIST's project and the vast public attention it receives, this thesis focuses on the algorithms of round 2 of this project. Note that NIST did not consider stateful signature schemes.

The following sections will first give some basic insight into the mathematical foundations of simple examples of each family and will then discuss the specific properties of the NIST

Table 2: Encryption schemes of round 2 of the NIST's PQ crypto project.

Type	Name	Family
Public key encryption and key exchange	BIKE	Code
	Classic McEliece	Code
	CRYSTALS-KYBER	Lattice
	FrodoKEM	Lattice
	HQC	Code
	LAC	Lattice
	LEDAcrypt	Code
	NewHope	Lattice
	NTRU	Lattice
	NTRU Prime	Lattice
	NTS-KEM	Code
	ROLLO	Code
	Round5	Lattice
	RQC	Code
	SABER	Lattice
	SIKE	Isogeny
	Three Bears	Lattice
Signatures	CRYSTALS-DILITHIUM	Lattice
	FALCON	Lattice
	GeMSS	Multivariate
	LUOV	Multivariate
	MQDSS	Multivariate
	Picnic	-
	qTESLA	Lattice
	Rainbow	Multivariate
	SPHINCS+	Stateless hash

signature algorithms. The KEMs have already been evaluated for the use in OPC UA in a project [42] preceding this thesis and these results will be used. Therefore only the digital signature schemes of the NIST PQ project will be reviewed.

2.5.2 Code Based Methods

McEliece suggested the first public key crypto system based on coding theory [43]. The main idea is to have a general linear code as the public key. Random errors are added to the message and to recover it we need to decode. In general, decoding is believed to be not possible in polynomial time [44], however for special codes it is easy. Therefore, the private key contains information on how to easily decode. Following we give a quick summary how linear codes work [45, p. 159] and then explain how they can be used for cryptography.

Consider a vector space C of the dimension k over a binary finite field. This means that each vector has k entries (k dimensions) and each entry can have either the value 0 or 1 where $1 + 1 = 0$ (binary finite field). We can encode any binary message of length k as a vector in

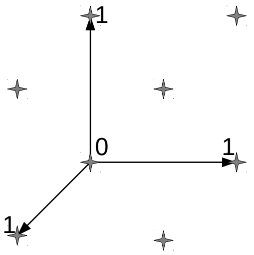


Figure 7: Example of a 3 dimensional vector space over a binary finite field. On each axis we can only be at either position 0 or 1. The stars mark all possible vectors in this space. Each star corresponds to a 3-bit message.

C and every vector in C represents a message. Figure 7 illustrates this.

$$\overbrace{\begin{pmatrix}0 & 1 & 1 & 0 & \cdots & 1\end{pmatrix}}^{k} \tag{1}$$

In order to obtain redundancy, the messages are encoded into vectors of length n with $n > k$. Thus the, codewords are vectors in an n dimensional space V. The number of all vectors in V is bigger than the number of vectors in C. Hence not all vectors in V are mapped to a message.

To easily convert between messages and their corresponding encoded vectors, the message can be seen as the coefficients of a basis in V. Let's say $\vec{b_1}, ..., \vec{b_k}$ are k linear independent vectors in V. Then we find the vector $\vec{c}$ that represents the message m by

$$\vec{c} = m_1\vec{b_1} + ... + m_k\vec{b_k} \tag{2}$$

where m_1 represents the first bit of the message etc. Mathematically C is a subspace of V with the basis $\vec{b_1}...\vec{b_k}$. An easy notation for (2) is to write the k basis vectors as rows in a matrix $\mathbf{G}$, called generator matrix, as shown in Figure 8.

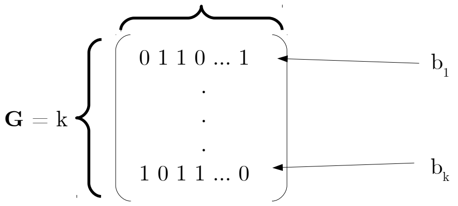


Figure 8: Generator matrix.

To retrieve the codeword from a message, the message has to be written as a row vector and be multiplied by the generator matrix:

$$\vec{m}\mathbf{G} = \vec{c} \tag{3}$$

During transmission, an error vector is added to the codeword ($\vec{c'} = \vec{c} + \vec{e}$). This means that the received vector is not inside the subspace C. To decode we assume that the weight[3] of the error vector, i.e. the number of '1' bits in the vector, is small because it is more likely that only a few errors have occurred. The goal is to find an error vector with a small weight that takes us back to a valid codeword.

It is possible to transform **G** into a $(n-k) \times n$ matrix **H** that can be used to check if a received encoded message is a vector in the code's subspace, i.e. if an error has occurred [45, p. 166]. Therefore the received message is written as a column vector and multiplied by **H**. The result of this operation is called syndrom.

$$\mathbf{H}\vec{c} = \vec{0} \tag{4}$$

For valid codewords, i.e. if the error vector has a weight of 0, the syndrom is $\vec{0}$. Otherwise, the syndrom only depends on the error vector and not on the message. Thus, if we want to correct errors with a weight $\leq t$ we can create a table with all error vectors and their corresponding syndroms. Assuming that the mapping between error vectors and syndroms is distinct, the procedure to correct errors is to calculate the syndrom of a received encoded message, matching the syndrom in the table, correct the error in the received encoded message and then decode. While this is a computationally complex task and might not be possible for an arbitrary code, there is a way of constructing codes that allow to correct up to t errors, called Goppa Codes [46]. It has been proven that efficient decoding algorithms for Goppa Codes exist.

In order to utilize linear coding for cryptography, the McEliece crypto system firstly creates a Goppa Code with a $k \times n$ generator matrix **G** that can correct up to t errors. Then a random invertible $k \times k$ matrix **S** and a random $n \times n$ permutation matrix **P** are created. The properties of a permutation matrix are that each row and column contains only a single '1' entry, the rest is '0'. When multiplied by a vector, the elements in the vector are permuted but not changed. Especially important is that a permutation does not change the weight of a vector.

The public key is a generator matrix that is calculated as the product of the three matrices:

$$\hat{\mathbf{G}} = \mathbf{SGP} \tag{5}$$

To encrypt a message, it has to be multiplied by $\hat{\mathbf{G}}$ and an error vector e with weight t has to be added.

$$c = m\hat{\mathbf{G}} + e \tag{6}$$

Because $\hat{\mathbf{G}}$, in contrast to **G**, is not a Goppa Code, it is computationally hard to decode c. On the other hand with the knowledge of **S**, **G** and **P** it is easy to decode as will be shown in the following, and therefore these three matrices constitute the private key. To decrypt,

---

[3] For example, the weight of the vector (000101) = 2 because two bits are set. We simply count the number of ones in a vector.

the ciphertext is first multiplied by the inverse of $\mathbf{P}$.

$$c' = c\mathbf{P}^{-1} = (m\hat{\mathbf{G}} + e)\mathbf{P^{-1}} = m\mathbf{SG} + e\mathbf{P}^{-1} \tag{7}$$

Since $\mathbf{P}^{-1}$ is a permutation matrix it will simply change the position of the '1' bits in the error vector, the weight remains the same. Thus we can rewrite (7) as:

$$c' = (m\mathbf{S})\mathbf{G} + e' \tag{8}$$

and using the Goppa Code $\mathbf{G}$ the error e' can be corrected and $(m\mathbf{S})$ can be decoded. Finally we multiply by the inverse of $\mathbf{S}$ to obtain m.

$$m = c'\mathbf{S}^{-1} = m\mathbf{SS}^{-1} \tag{9}$$

A problem of code based crypto systems is the large size of public keys. While variants of McEliece, such as the Niederreiter crypto system, can reduce the public key size, it still lies in the range of 100 kilobytes to several megabytes [37, p. 95]. On the other hand, encryption and decryption can be performed very fast, since matrix multiplications have a low computational complexity.

2.5.3 Hash Based Methods

Cryptographic hash functions have the advantage that they are not vulnerable to Shor's algorithm since they are not based on factorization or the discrete logarithm problem. The output size of the hash functions have to be selected large enough to withstand a brute force search with quadratic speed up due to Grover's algorithm. While there are no hash based public key encryption schemes available, it is possible to sign messages using a private key and verify the signatures with a public key entirely based on a generic hash function. As an example, the fundamentals of Lamport's signature system [47] are explained.

First we consider a system that can only sign a 1-bit message. This means the message is either 1 or 0. The secret key will consist of two random numbers s_0 and s_1. The public key comprises the hash values of these two random numbers, $p_0 = H(s_0)$ and $p_1 = H(s_1)$.

$$\begin{aligned} sk &= \{s_0, s_1\} \\ pk &= \{p_0 = H(s_0),\ p_1 = H(s_1)\} \end{aligned} \tag{10}$$

If the message '0' shall be signed, we reveal the secret key s_0 as the signature and if '1' shall be signed we reveal s_1. When we want to verify the signature we just have to hash the signature and compare it to p_0 for a '0' message or p_1 for a '1' message.

To sign messages of arbitrary length, we compute a hash over the message and sign the hash h that produces an output of b bits. h_i refers to the ith bit of the hash. To sign the hash we have to expand the 1-bit system described above to a b-bit system. Therefore, the private key s is computed as b pairs of random numbers i.e. two random numbers for each bit of h and $2b$ random numbers in total. s_i refers to the ith pair of random numbers.

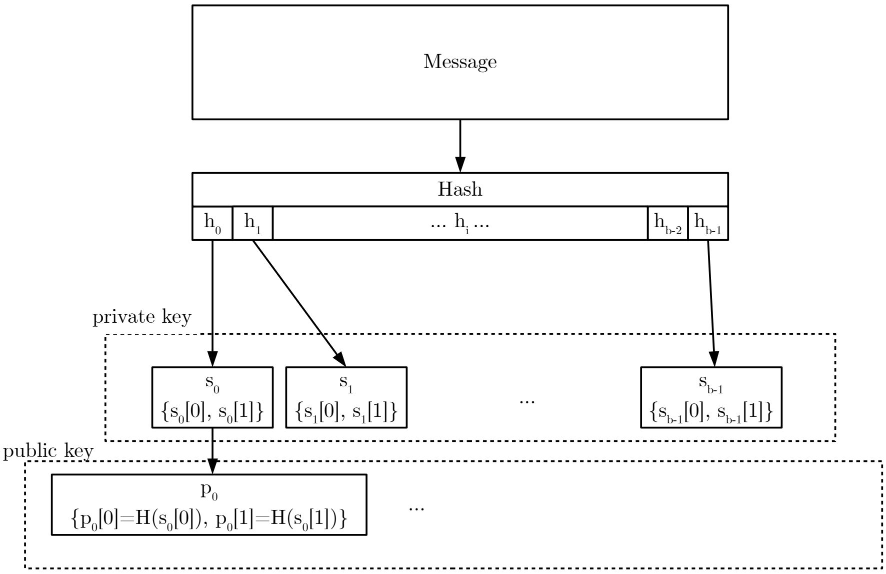


Figure 9: Generation of private and public key for a hash based signature system.

Then for the public key we take each random number of the private key and compute the hash of each. Analogous to the private key, the public key consists of b pairs of hashes, each associated with a bit of h. If we assume that always the same hash function is used, the public key consists of $2b$ hashes of length b-bit and therefore of $2b^2$ bits. See Figure 9 for a graphical representation.

Furthermore, to sign a hash h, we look at every single bit h_i and use the ith pair of random numbers of the secret key s_i as the signature. When $h_i = 0$ then sig_i is set to $s_i[0]$ and if $h_i = 1$ then sig_i is set to $s_i[1]$. Hence the signature consists of b secret numbers.

When a signature shall be verified, we take the ith number from the signature, look at h_i and pick $p_i[0]$ or $p_i[1]$ depending on h_i. If $H(sig_i) = p_i[h_i]$ then the verification was correct. Additionally it has to be ensured that the has h matches the hash of the message.

A major disadvantage of this signature scheme is that part of the private key is revealed with each signature. Therefore it is only secure when a key pair is used only for one signature. A solution to the problem is to generate enough key pairs, depending on how many messages we expect to sign in a certain amount of time. For example at a certificate authority we know the valid time of a root certificate and we can estimate how many certificates we expect to sign in this period.

When this scheme is combined with a Merkle hash tree [48, p. 227], it is possible to have only one public key for all the key pairs that were created. In Figure 10 we have created 8 key pairs and a hash tree for the public keys. The only key we have to distribute is p_{15}. In order to verify s_1 the signer of the message has to show the recipient p_1, p_2, p_{10} and p_{14}. If this works out until p_{15}, the verifier can trust the signature. After signing a message with s_1 we can never use it again and have to use s_2 for the next message, thus making the system stateful.

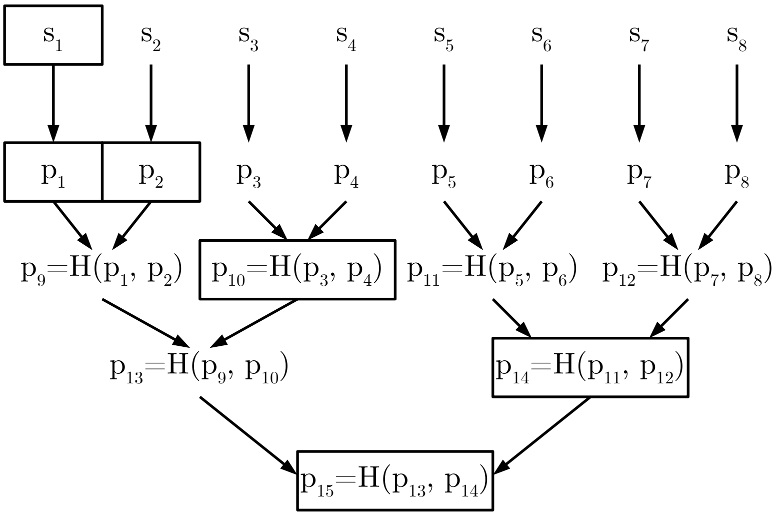


Figure 10: A Merkle tree of public keys.

The effort of maintaining a stateful system might be feasible in some scenarios, however it seems very impractical to be used to sign messages that are exchanged in IIoT applications.

Goldreich [49] suggested a stateless hash based crypto system which uses huge signature chains that can be created "on the fly". When a message is signed, a random chain can be selected and therefore the number of chains just has to be large enough to make it unlikely to use the same key pair twice.

SPHINCS+ SPHINCS+ [50] is the only stateless hash based signature scheme left in round two of the NIST PQ project. It improves the ideas introduced above and achieves small private and public keys, however generates rather large signatures. It can be used with different hash functions and the authors describe 36 different parameter sets, Table 3 summarizes key and signatures sizes for security levels L1, L3 and L5.

Table 3: Key and signature sizes of SPHINCS+ in bytes.

Security Level	Public-key	Private-key	Signature
L1 (128 bit)	32	64	8080 – 16976
L3 (192 bit)	48	96	17064 – 36664
L5 (256 bit)	64	128	29792 – 49261

2.5.4 Multivariate Polynomial Based Methods

Solving a set of multivariate quadratic polynomial equations over a finite field in general is NP-hard. This problem is called MQ problem or Multivariate Quadratics problem. But we can find instances of this problem that are easier to solve, thus creating trap doors and making them useful for public key cryptography [37, p. 193]. Following the wonderful explanations of [51, p. 162] this section gives a brief overview over the underlying problem and the construction of multivariate schemes.

Firstly, we define a multivariate quadratic polynomial over a finite field $\mathbb{F}_q$ as: A polynomial function of multiple variables of the form:

$$p(x_1, ..., x_n) = \sum_{1 \leq j \leq k \leq n} \gamma_{j,k} x_j x_k + \sum_{j=1}^{n} \beta_j x_j + \alpha \tag{11}$$

where $\gamma_{j,k}, \beta_j, \alpha \in \mathbb{F}_q$.

For example a multivariate quadratic of two variables ($n = 2$) would look like this:

$$\begin{aligned} p(x_1, x_2) &= \gamma_{1,1} x_1 x_1 + \gamma_{1,2} + \gamma_{2,2} x_2 x_2 + \beta_1 x_1 + \beta_2 x_2 + \alpha \\ &= \gamma_{1,1} x_1^2 + \gamma_{1,2} + \gamma_{2,2} x_2^2 + \beta_1 x_1 + \beta_2 x_2 + \alpha \end{aligned} \tag{12}$$

Basically the γ terms are every possible combination of variables where $x_1 x_2 = x_2 x_1$ and is combined to one term which is expressed by the constraint $j \leq k$ in the summation.

We can define a system of m multivariate quadratic polynomials of n variables

$$\begin{aligned} p_1(x_1, ..., x_n) &= \sum_{1 \leq j \leq k \leq n} \gamma_{1,j,k} x_j x_k + \sum_{j=1}^{n} \beta_{1,j} x_j + \alpha_1 \\ &\vdots \\ p_m(x_1, ..., x_n) &= \sum_{1 \leq j \leq k \leq n} \gamma_{m,j,k} x_j x_k + \sum_{j=1}^{n} \beta_{m,j} x_j + \alpha_m \end{aligned} \tag{13}$$

The set of polynomials is denoted as $\mathcal{P} = (p_1, ..., p_m)$. For encryption we represent the message as a vector $\vec{x} = (x_1, ..., x_n) \in \mathbb{F}_q$ and use $\mathcal{P}$ as the public key with $m = n$, i.e. the same number of polynomials as we have variables. We evaluate each polynomial of $\mathcal{P}$ with $\vec{x}$ as input. This yields n results, one for each polynomial, which we collect in the vector $\vec{y}$. Thus we simply write $\vec{y} = \mathcal{P}(\vec{x})$, which is the one way function. It is NP-hard to find $\vec{x}$ if only the ciphertext $\vec{y}$ and the polynomials are known.

However, now we must construct a trap door, i.e. a way to reverse that calculation with a private key. Therefore, we have to find a set of polynomials $\mathcal{P}'$ that are actually easy to invert and then transform them to a general instance of the problem. The method of constructing $\mathcal{P}'$ differs in the variants of multivariate public key crypto systems, however the transformation to $\mathcal{P}$ is done in the same way. We need to find the invertible affine transformations S and T. The tuple $(S, \mathcal{P}', T)$ compose the private key. The public key $\mathcal{P}$ is obtained by applying the affine transformations to $\mathcal{P}'$. Figure 11 illustrates the trap door.

For decryption we take the ciphertext $\vec{y}$ and apply the inverse affine transform $Y = T^{-1}(\vec{y})$ to it. Then we have to find X such that $\mathcal{P}'(X) = Y$. How this is done depends on the concrete scheme. Finally we do the inverse of the affine transform $\vec{x} = S^{-1}(X)$ to obtain the plain

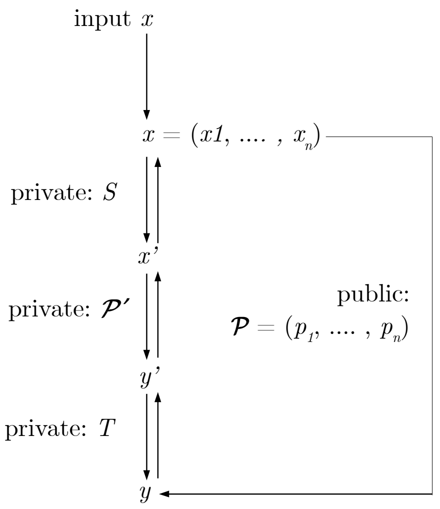


Figure 11: Graphical representation of the trapdoor. The left path via S, $\mathcal{P}'$ and T can be gone backwards, the path via $\mathcal{P}$ not [51, p. 163].

text.

$\mathcal{P}'(X) = Y$ is not always bijective, that means that for a given Y we can find multiple X. Therefore we have to find all possible X and decrypt to multiple $\vec{x}$. Via a checksum we have to ensure to select the correct $\vec{x}$.

An advantage of multivariate schemes, especially when used for digital signatures are the very small signature sizes. However, the public keys are very large. So these schemes are ideal for applications where public keys are rarely distributed, but many messages have to be signed.

GeMSS GeMSS [52] stands for 'A Great Multivariate Short Signature' and is among the candidates in round two of the National Institute of Standards and Technology (NIST) PQ project. It is based on the Hidden Field Equations (HFE) cryptosystem and offers very small signature sizes, however has very large public keys, as can be seen in Table 4. The author's claim that verification can be implemented rather fast.

Table 4: Key and signature sizes of GeMSS in bytes. Signature values are rounded to full bytes.

Security Level	Public-key	Private-key	Signature
L1 (128 bit)	352 188	13 438	33
L3 (192 bit)	1237 964	34 070	52
L5 (256 bit)	3040 700	75 893	72

LUOV LUOV [53] is the abbreviation for Lifted Unbalanced Oil and Vinegar. Compared to GeMSS it has much smaller public key sizes (see Table 5) but they are still large compared to, for instance, lattice based schemes. It is based on the UOV (Unbalanced Oil and Vinegar) crypto scheme that was proposed in 1997 but greatly reduces the public key size. The techniques that are used to reduce the key size are rather new and in 2019 some new attacks on LUOV were presented [54].

Table 5: Key and signature sizes of LUOV in bytes.

Security Level	Public-key	Private-key	Signature
L1 (128 bit)	11 500	32	239
L3 (192 bit)	35 400	32	337
L5 (256 bit)	82 000	32	440

MQDSS The MQDSS specification [55] does not explain the meaning of the scheme's name but one could guess it means Multivariate Quadratic Digital Signature Scheme. Two parameter sets are recommended: MQDSS-31-48 offering L1-L2 security and MQDSS-31-64 offering L3-L4 security. In contrast to other schemes the public key is very small, however the signatures tend to be large, see Table 6. This is achieved by using pseudo random functions to generate the keys, so they actually only require a seed for these functions.

In 2019 an attack on MQDSS was suggested [56] that also has been confirmed by the MQDSS team but requires further investigation.

Table 6: Key and signature sizes of MQDSS in bytes.

Security Level	Public-key	Private-key	Signature
L1-L2 (min. 128 bit)	46	16	20854
L3-L4 (min. 192 bit)	64	24	43728

Rainbow The Rainbow signature scheme [57] comes with three parameter sets: Ia, IIIc, Vc and has versions that compress the keys. Table 7 considers the compressed versions.

Table 7: Key and signature sizes of Rainbow in bytes.

Security Level	Public-key	Private-key	Signature
L1 (128 bit)	68 100	93 000	64
L3-L4 (min. 192 bit)	206 700	511 400	156
L5 (256 bit)	491 900	1 227 100	204

2.5.5 Lattice Based Methods

NTRU is one of the widely known public key crypto systems that is based on lattice problems. As an example, this section gives a high level overview of the system.

All mathematical operations in NTRU are based on the truncated polynomial ring

$$R = \mathbb{Z}[X]/(X^N - 1) \tag{14}$$

Thus, all the coefficients are integers and the highest possible degree is $N - 1$. Multiplication in the ring with symbol * is defined as a cyclic convolution with f and g being polynomials, f_i and g_j being the coefficients of these polynomials and k being the coefficient's index of the result.

$$f * g = \sum_{i+j \equiv k \bmod N}^{i,j} f_i \cdot g_j \mod q \tag{15}$$

All operations on the coefficients are performed modulo q, meaning the coefficients are $f_k < q$ for all f_k.

The parameters for the crypto systems are N, q, p for which $q > p$ and $gcd(p, q) = 1$, implying that q and p are coprime. In order to create a key pair, the parameters N, q, p has to be selected and a polynomial f with coefficients $\{-1, 0, 1\}$ have to be chosen at random, such that the inverse of f modulo p, called f_p, and the inverse modulo q, called f_q exist. Hence

$$f * f_q \equiv 1 \mod q \tag{16}$$

$$f * f_p \equiv 1 \mod p \tag{17}$$

The Euclidean algorithm can be used to calculate the inverses. Subsequently, another random polynomial g with coefficients $\{-1, 0, 1\}$ has to be chosen. The private key is f, the public key h is computed as

$$h \equiv pf_q * g \mod q \tag{18}$$

To encrypt a message it has to be converted to a polynomial m. Since the possible coefficients are $\{-1, 0, 1\}$, some kind of ternary encoding of the message comes to mind. Then a random polynomial r with coefficients $\{-1, 0, 1\}$ has to be chosen and the encrypted message is calculated as

$$e \equiv r * h + m \mod q \tag{19}$$

In order to decrypt the message, it is multiplied by the private key f.

$$\begin{aligned}
a &\equiv f * e && \bmod q \\
a &\equiv f * (r * h + m) && \bmod q \\
a &\equiv f * (r * pf_q * g + m) && \bmod q \\
a &\equiv f * pf_q * r * g + f * m && \bmod q \\
a &\equiv p \cdot r * g + f * m && \bmod q
\end{aligned} \tag{20}$$

where $f * f_q$ cancels out. Because the coefficients of the polynomials except f_p were selected to be small, mod q has no effect on them. Next a is written mod p instead of mod q.

$$\begin{aligned}
b &= a && \bmod p \\
b &= p \cdot r * g + f * m && \bmod p \\
b &= f * m && \bmod p
\end{aligned} \tag{21}$$

Note that $p \cdot r * g \equiv 0 \bmod p$ because any multiple of p mod p is 0. From here on, only a multiplication by f_p is needed to recover the message.

$$\begin{aligned}
c &= f_p * b && \bmod p \\
c &= f_p * f * m && \bmod p \\
c &= m && \bmod p
\end{aligned} \tag{22}$$

Note that the coefficients of m are in the range $[0, p)$ but they were originally selected from $\{-1, 0, 1\}$. Thus it is necessary to represent the coefficients in the range $[-p/2, p/2)$ to obtain the correct message.

qTESLA One lattice based signature scheme among the NIST submissions is qTESLA [58]. It is based on the Ring Learning With Errors (R-LWE) problem and offers a tight security reduction which means that it is provably secure. However the two available parameter sets impose large public keys as can be seen in Table 8.

Table 8: Key and signature sizes of qTESLA in bytes.

Security Level	Public-key	Private-key	Signature
L1 (128 bit)	14880	5184	2592
L3 (192 bit)	38432	12352	5664

Dilithium Dilithium [59] is a signature scheme based on the hardness of the module learning with errors problem. Table 9 shows key and signature sizes. Note that the bit security is lower than specified by the NIST for each level [60]. But the authors claim that their calculation of the bit levels follows a very conservative approach and therefore categorize their algorithm in the security categories as shown in the table. However this is still an ongoing discussion in the PQ project.

Table 9: Key and signature sizes of Dilithium in bytes.

Security Level	Public-key	Private-key	Signature
L1 (100 bit)	1184	2800	2044
L2 (141 bit)	1472	3504	2701
L3 (174 bit)	1760	3856	3366

FALCON The third lattice based signature scheme among the NIST round two candidates is FALCON. It is a derivate of NTRUsign and focuses on a small public key and signature size. It achieves this by using NTRU lattices; lattices of a certain structure that allow to be described with very little data. The key and signature sizes, shown in Table 10, are the smallest compared to other submissions in the NIST PQ project remaining in round two.

Table 10: Key and signature sizes of FALCON in bytes.

Security Level	Public-key	Private-key	Signature
L1 (114 bit)	897	1281	690
L5 (263 bit)	1793	2305	1330

2.5.6 Available Open Software Libraries

This section discusses the available implementations of the schemes that are currently in round 2 of the NIST's PQ challenge.

Reference Implementations Every algorithm that is submitted to the NIST PQ project must include a platform independent ANSI C reference implementation. The purpose of these implementations is to have a 'fair' performance comparison. A common interface for all algorithms is defined in form of a *api.h* file. All the reference implementations are available on the NIST's website, however most of them do not mention any kind of licensing. Some reference implementations, such as Falcon, are published under the MIT license.

These implementations are mainly for the purpose of demonstrating the algorithms and are not meant for productive usage.

Open Quantum Safe The Open Quantum Safe project [61] is an open source software library, available on Github, that implements selected algorithms mainly from the NIST PQ project. Currently 9 out of the 17 remaining KEMs and 7 out of 9 signature schemes of round two are implemented. Even though it is a C library, wrappers for C#, Go, C++ and Python are available. The project is published under the MIT license.

The OQS library was also included in an openSSL fork that allows to generate certificates using some of OQS' algorithms.

PQClean PQClean [62] is an open source project, also available on Github, that takes the reference implementations from the NIST project and provides clean implementations of them. They all have a consistent interface. All the algorithms are organized in folders and do not require any dependencies. When using PQClean, it is possible to only select the required algorithms and copy their folders and a common folder directly into a C project. Thus it is not meant to be build into a library binary but is meant to be used directly as the C source files.

Each algorithm is individually licensed, where most of them are public domain or under a MIT license.

PQM4 The PQM4 project [63] uses the implementations from PQClean and additionally provides optimized versions for the instruction set of the ARM Cortex-M4 CPU family. They also provide cross-platform optimized versions of some algorithms. Each algorithm included has the same license as in the PQClean project (either public domain or MIT).

2.6 Hybrid Key Exchange Mechanisms

In this section we define a KEM formally and explain the relevant security notions. A KEM is defined as a set of three algorithms

1. Key Generation
   $(pk, sk) \leftarrow \text{KeyGen}()$
2. Encapsulation
   $(c, k) \leftarrow \text{Encaps}(pk)$
3. Decapsulation
   $k \leftarrow \text{Decaps}(sk, c)$

The key generation algorithm returns a key pair consisting of a public key pk and a secret key sk and has no input. The input to the encapsulation function is a public key pk and it returns a ciphertext c and a shared secret key k. The decapsulation function receives a secret key sk and a ciphertext c as inputs and returns the shared secret key k or failed decapsulation. A client would call the *KeyGen* function and then send the public key pk to the server. Subsequently the server calls the *Encaps* function with the public key it received from the client, stores the shared secret key k for later symmetric encryption and sends the ciphertext c to the client. The client uses his private key sk and the received c as input to the *Decaps* function to obtain the shared secret key k that it now shares with the server.

2.6.1 Security Notions

The most common security notion for KEMs is indistinguishability under a certain attacker model, where indistinguishability is defined as an experiment or game that is played between a challenger and an attacker:

1. The challenger generates a key pair using $(pk, sk) \leftarrow \text{KeyGen}()$
2. The system calls $(c, k_0) \leftarrow \text{Encaps}(pk)$
3. The challenger samples k_1 uniformly random from the key space.
4. The attacker receives c and either k_1 or k_0 selected at random.
5. The goal for the attacker is now to be able to tell if he received the correct k_0 which corresponds to c or if he received k_1 which was selected at random. The attacker wins as soon as his probability of being correct is higher than simple guessing $(\frac{1}{2})$.

The attacker models define the abilities an attacker has during the above game. The most important models are the chosen-plaintext-attack (CPA) and the chosen-ciphertext-attack (CCA).

CPA The experiment is performed as above and the attacker has no additional information. He can see the secret key $k_{0/1}$ provided by the challenger and has to decide if it was the correct one, i.e. the one that was used to encrypt c. He is also able to call the *Encaps* function by himself using the public key pk generated by the system.

CCA In the CCA case the attacker has the additional ability to query an oracle to decrypt any ciphertext except for c (in this case winning would be trivial). For example the attacker could flip one bit in $c \rightarrow c'$ and the oracle would return him $\text{Decaps}(sk,\ c')$.

Since there are no powerful quantum computers available today it is useful to specifically model the additional abilities of a quantum attacker. There is a classical attacker, that is implicitly implied in the current models. Then there is an attacker that stores data today and uses a quantum computer in the future. And there is a scenario where a quantum computer is available and the attacker uses it during the whole attack. And in the far future there is the possibility that end users also use quantum computers and therefore the attacker can query the decapsulation oracle in superposition [64].

Bindel et. al. [65] introduced a new notation for the different kinds of attackers: X^yZ with $X, Z \in \{C, Q\}$ and $y \in \{c, q\}$ where C stands for classical and Q for quantum. X describes the ability of an adversary during the interaction with the oracle, y specifies if the adversary can interact with the oracle in superposition and Z indicates if the attacker has quantum capabilities after interaction with the oracle. Note that this fine distinction is only useful in the CCA case, for the CPA case it is sufficient to specify if the attacker can use a quantum computer or not. The practical attacker models are [66]:

C^cC The attacker is purely classical, this is the traditional scenario.

C^cQ The attacker is classical but will gain access to a quantum computer in the future, after he finished interacting with the decapsulation oracle.

Q^cQ The attacker has a quantum computer available, but can only interact classically with the decapsulation oracle. This scenario is applicable when only the attacker is quantum and the other parties use classical computers and is commonly referred to as the *post-quantum* setting [67, p. 365] [66].

Q^qQ This is a full quantum attacker that can also query the decapsulation oracle in superposition.

This thesis will focus on the first three models, leaving the full quantum attacker for future investigation. This is reasonable since we do not have any quantum computer in our system that an attacker could possibly query in superposition.

2.6.2 Combiners

Section 1.4 of the introduction motivated the use of hybrid crypto schemes: *Hedge the bets* when transitioning to new cryptographic primitives. The remaining question is how algorithms can be combined such that the overall security is not reduced.

To obtain a hybrid KEM, two normal KEMs shall be combined. The new hybrid KEM consists of the three algorithms $\mathrm{GenKey_h}()$, $\mathrm{Encaps_h}(pk_h)$ and $\mathrm{Decaps_h}(sk_h,\ c_h)$. Each of the hybrid algorithms will make use of the two inner KEM's functions $\mathrm{KeyGen_0}()$, $\mathrm{KeyGen_1}()$, $\mathrm{Encaps_0}(pk_0)$, $\mathrm{Encaps_1}(pk_1)$, $\mathrm{Decaps_0}(sk_0,\ c_0)$ and $\mathrm{Decaps_1}(sk_1,\ c_1)$.

Following we introduce two different methods of combining KEMs, typically called "combiner". The security of each of the methods has been proven in [66]. All combiners guarantee the same security promises as the strongest of the two used KEMs. For example using a $\mathrm{C^cC}$-IND-CCA secure and a $\mathrm{Q^cQ}$-IND-CCA secure scheme will guarantee $\mathrm{Q^cQ}$-IND-CCA security for the hybrid KEM. If, after further cryptanalysis, it turns out that the second KEM is not secure at all, the hybrid KEM still guarantees $\mathrm{C^cC}$-IND-CCA security.

XORthenMAC The XOR then MAC combiner is an enhancement of the simple XOR combiner, which generates two ciphertexts c_0, c_1 using the two KEMs and combines them as a tuple to the hybrid ciphertext $c^* = (c_0, c_1)$. The decapsulation will return two shared secrets $k^* = (k_0, k_1)$ which will be combined by XORing them. However, even if both KEMs are IND-CCA secure, the resulting hybrid KEM is only IND-CPA secure [68, p. 198]. In the IND-CCA case the adversary has access to a decapsulation oracle that will decapsulate any ciphertext but c^*. The attacker simply can call the oracle with $(c_0, c_1') \neq c^*$ and $(c_0', c_1) \neq c^*$ and receives $k_0 \oplus k_1'$ and $k_0' \oplus k_1$. He can select c_0' and c_1' such that he knows the corresponding key and then reconstruct k_h.

The XOR then MAC combiner prevents this kind of attack by attaching a MAC to the ciphertext. Algorithm 1 shows the procedure: In line 2, the public key is split up into two individual public keys for the inner KEMs. Each KEM's encapsulation function is called separately (line 3 and 4). Both secret keys are split up into a secret key and a MAC key (line 5, 6). This is done by simply splitting at a certain byte position, e.g. 16 bytes for $k_{i,mac}$ and 16 bytes for $k_{i,secret}$ in case k_0 was 32 bytes long. The two MAC keys are then concatenated in line 7 and the secret keys are combined using XOR. Finally a MAC is calculated over the two ciphertexts (line 10). Thus the returned hybrid ciphertext is $c_h = (c^*, \tau) = ((c_0, c_1), \tau)$.

Algorithm 1 XORthenMAC combiner

```
 1: procedure $\textsc{Encaps}_{\text{H}}(pk_h)$
 2:     $(pk_0, pk_1) \leftarrow pk_h$
 3:     $(c_0, k_0) \leftarrow \textsc{Encaps}_0(pk_0)$
 4:     $(c_1, k_1) \leftarrow \textsc{Encaps}_1(pk_1)$
 5:     $(k_{0,mac}, k_{0,secret}) \leftarrow k_0$
 6:     $(k_{1,mac}, k_{1,secret}) \leftarrow k_1$
 7:     $k_{mac} = k_{0,mac} || k_{1,mac}$
 8:     $k_{secret} = k_{0,secret} \oplus k_{1,secret}$
 9:     $c^* \leftarrow (c_0, c_1)$
10:     $\tau \leftarrow \text{MAC}(c^*, k_{mac})$
11:     return $((c^*, \tau), k_{secret})$
12: end procedure
```

The hybrid decapsulation function then uses the two ciphertexts c_0 and c_1 which it can obtain from c_h, decapsulates each using Decaps$_0$(c_0,sk_0) and Decaps$_1$(c_1,sk_1) and verifies the MAC using the decapsulated secret keys. Only if the MAC is correct, the secret k_h is returned. If at least one of the inner KEMs is IND-CCA secure, the attacker can not obtain k_{mac}. This is because for the decapsulation oracle to work, the attacker has to pass $((c', c_1), \tau)$ while it is not feasible to compute the correct τ without knowledge of k_{mac}, assuming an ideal MAC function.

dualPRF A dual PRF (dPRF) is a Pseudo Random Function (PRF) if at least one of its two inputs are random, i.e. dPRF(k, ·) and dPRF(·, x) are PRFs if k and x are random. The hybrid shared secret can be calculated from the $k_h = dPRF(k_0, k_1)$. Thus even if the attacker knows k_0 he cannot reconstruct k_1 from k_h. Attacks equivalent to the attack on the plain XOR combiner are not possible and therefore no MAC is required here.

However in the hybrid scenario we made the assumption that one of the two inner KEMs might be completely broken. Lets say KEM$_1$ is completely broken in such a way that the attacker is able to retrieve k_1 from $c_h = (c_0, c_1)$. Now it is conceivable that the attacker is able to find a $c_1^* \neq c_1$ that decapsulates to the same k_1. Querying the oracle with (c_0, c_1^*) is allowed and will return the correct k_h.

In order to mitigate this attack surface, the final shared secret is calculated as $k_h = PRF(dPRF(k_0, k_1), (c_0, c_1))$. This has been suggested and proven to be secure by [66, p. 15].

2.7 Certificates and Signatures

In this section, the security notions for signatures are discussed. Since the security of certificates, as described in Section 2.2, solely relies on signature algorithms, the same notions can be applied to certificates as well.

A signature scheme consists of three functions [65]:

- $(sk, vk) \leftarrow \text{KeyGen}()$: Returns a secret signing key sk and a public verification key vk.
- $\sigma \leftarrow \text{Sign}(sk,\ m)$: Takes a message m and the signing key sk as input and returns a signature σ.
- $\{0, 1\} \leftarrow \text{Verify}(vk,\ m,\ \sigma)$: Takes the verification key vk, the message m and the signature σ as input and returns 1 (true) or 0 (false) depending if the signature can be verified or not.

2.7.1 Security Notions

Existential Unforgeability under Chosen Message Attack (EUF-CMA) This notion of security is defined as an experiment in which the challenger generates a public/private key pair. The attacker gets access to the public key and then is allowed to query an oracle that will sign any message of the attackers choosing under the generated private key. At one point the attacker has to generate a signature for a new message that he didn't query previously from the oracle. If the signature can be verified under the previously generated public key the attacker wins the experiment. EUF-CMA security requires that no attacker exists that can win this experiment within a reasonable time.

Analogous to Section 2.6 the quantum capabilities can be modelled for the attacker, depending at which stage of the experiment he has access to a quantum computer and if the oracle can be queried in superposition. Using the X^yZ notation, $X \in \{C, Q\}$ determines if the attacker has access to a quantum computer when he can query the signing oracle, $y \in \{c, q\}$ describes whether the attacker can query the oracle in superposition and $Z \in \{C, Q\}$ indicates if the attacker can access a quantum computer after losing access to the oracle.

In order to query a signing oracle in superposition it is necessary that the oracle is implemented on a quantum computer. This thesis assumes that none of the components of a system is implemented on a quantum computer, therefore quantum access to the oracle is not considered.

Non-Separability When two signature schemes are combined to a hybrid scheme the new security notion of non-separability can be defined [65]. It describes if it is possible for an attacker to use a hybrid signature to produce a valid signature for one of the schemes the hybrid scheme was constructed of. For example let Σ' be a hybrid signature scheme composed of the two schemes Σ_1 and Σ_2. If a verifier accepts messages that are signed with Σ' or Σ_1 but acts differently, depending on what signature scheme has been used, an attacker could try to transform a message that was signed using Σ' into a message that was signed by Σ_1. If the attacker is not able to do that, Σ' is said to be 1-nonsep, and 2-nonsep if the same applies for Σ_2. More formally this can be defined in an experiment for τ-non-separability: The challenger generates a hybrid key pair and the attacker gets access to the public hybrid verification key vk_h. Then the attacker can query hybrid signatures for messages of his choosing from an oracle. Finally the attacker has to output a message m and a signature σ such that

1. The inner verify function $\text{Verify}_\tau()$ of scheme Σ_τ returns 1 (i.e. verification passed).
2. The hybrid verify function cannot distinguish the signature from a signature that was created using Σ_τ i.e. can be tricked into believing it deals with a legacy signature.

This property is especially important to avoid downgrade attacks.

2.7.2 Combiners

All methods of combining signature schemes that are described below use the same key generation method; The keys are concatenated: $sk_h \leftarrow (sk_0, sk_1)$ and $vk_h \leftarrow (vk_0, vk_1)$. The EUF-CMA security of each scheme has been proven under the condition that at least one of the underlying schemes is EUF-CMA secure [65].

Concatenation The most trivial way of combining signatures is to compute signatures for the message using the two underlying schemes and concatenate them: $\sigma_0 \leftarrow Sign_0(sk_0, m)$, $\sigma_1 \leftarrow Sign_1(sk_1, m)$, $\sigma_h \leftarrow (\sigma_0, \sigma_1)$.

This method does not provide any non-separability. An attacker can use σ_0 or σ_1 as valid signatures of m for each underlying scheme, without the possibility for a verifier to notice that the signature was extracted from a hybrid scheme and not generated by an underlying scheme.

Nesting In this combiner scheme, the message m is first signed by the first inner sign function: $\sigma_0 \leftarrow Sign_0(sk_0, m)$. The second inner sign function takes the message plus the first signature as input: $\sigma_1 \leftarrow Sign(sk_1, (m, \sigma_0))$. The hybrid signature again is the combination of both inner signatures $\sigma_h \leftarrow (\sigma_0, \sigma_1)$. This scheme is 1-non-separable but *not* 0-non-separable because σ_1 is not a valid signature for the message directly (without σ_0).

2.8 Authenticated Key Exchange

2.8.1 Bellare-Rogaway Model

Most pure KEMs are only designed to be secure against a passive attacker, who is only able to eavesdrop on communication but cannot interfere in message transmission. This attacker model is not realistic in most cases. In the internet for example, data packets are routed through many different networks, operated by untrusted third parties.

Bellare and Rogaway introduced a detailed model for key exchange protocols including an active attacker [69]. The model first defines a set of participants $\mathcal{U}$, and each participant

$U \in \mathcal{U}$ has a key pair (pk_U, sk_U) assigned. The model assumes that every participant knows the public key of all other participants. When a participant wants to exchange a key he runs a session of the protocol denoted as $\Pi^j_{U,V}$. This represents the jth run of a session for user U and intended communication partner V. Every participant can run multiple sessions in parallel. Each session has six associated variables:

1. *role* $\in$ {initiator, responder}
2. *status* $\in$ {running, accepted, rejected}
3. *sid* can be a number or undefined
4. *key_status* $\in$ {fresh, revealed}
5. K is the stablished session key or undefined
6. *tested* $\in$ {true, false}

When a key is to be established between two participants, both run a session of the protocol. Let the two sessions be $\Pi^i_{S,T}$ for participant S and $\Pi^j_{U,V}$ for participant U. Then $S = V$, $T = U$ and the *sid* of both session matches. After running the protocol the session key K of both sessions also match.

The adversary has full control over the network and has additional influence on the sessions of all honest parties. He specifically can use the following functions:

NewSession(U**,** V**,** *role***)** Create a new session running in U with intended partner V with U in the specified role (initiator or responder).

Send($\Pi^j_{U,V}$**,** m**)** Sends a message to the session. The session will react as if it had received the message from V. If the session's *status* changes to "accepted" and partner V's key has been revealed, this session is also marked as "revealed".

Reveal($\Pi^j_{U,V}$**)** If the session has the *status* "accepted", the session key K is returned and the session is marked as "revealed". If the *status* is not "accepted", "undefined" is returned.

Corrupt(U**)** Returns the long term secret key of U, sk_U. Also sets all sessions where U is involved to "revealed".

Test($\Pi^j_{U,V}$**)** This function returns either the session key K of the session or a key randomly sampled from the key space. It is used at the end of an experiment when the attacker has to distinguish the session key from a random key. Therefore it may only be called once.

Based on this model two security experiments can be defined [66], [70], [71]:

2.8.2 BR-Match security experiment

First the long term key pairs for each participant U is generated. Then the adversary receives all the public keys and has access to the five functions defined above. Subsequently the attacker looses access to the functions NewSession, Send, Test and only can use Reveal and Corrupt. At some point the adversary stops. He wins if any of the following is true:

- He was able to trick the participants in such a way that there are two paired sessions($\Pi.sid = \Pi'.sid$) that both are not in *status* "rejected" but that have derived different session keys ($\Pi.K \neq \Pi'.K$)
- There exist two paired sessions with different intended partners, i.e. $\Pi^j_{U,V}$ and $\Pi^i_{U',V'}$ that both have the same *sid*, one has the *role* "initiator" and the other has the *role* "responder" but $U \neq V'$ or $V \neq U'$.
- More than two sessions share the same *sid*.

2.8.3 BR-key-secrecy experiment

As in the previous experiment, key pairs for all the participants are generated and the adversary gets access to all the public keys. Then it is decided randomly if the test function shall return a random key or the correct key of the session when queried. The attacker does not know what was decided for obvious reasons. Then the adversary is granted access to the functions NewSession, Send, Reveal, Corrupt, Test. After a while, the attacker enters a second stage of the experiment and loses access to the functions NewSession, Send and Test. The adversary now has to say if the key that would be returned by the test function is correct or if it was a random key. The attacker is not allowed to test the key of a session that was revealed or that he will reveal later, i.e. if there is a session that has *tested*=true and *key_ status*=revealed the adversary looses the experiment.

In summary, both experiments are divided into two stages: In the first stage the attacker can use all the oracle functions, in the second stage he can only use corrupt and reveal. This allows to define attackers that have access to quantum computers only in the second stage of the experiment. They correspond to real active attackers that do not have access to a quantum computer yet, but might have in the future. The two stages allow to model two stage attackers (C^cC, C^cQ, Q^cQ) that have different quantum capabilities in each stage.

3 Methodology

The methodology section explains how hybrid quantum resistant certificates are designed and how a Python software, that can create these certificates, was implemented. Then, two modified versions of the OPC UA key exchange protocol, named 'Variant One' and 'Variant Two', are shown and we argue why these changes are necessary to make them secure against a Q^cQ attacker while maintaining at least the security of the already used classical key exchange method. We describe the implementation of both methods and explain how the performance of the schemes was measured.

3.1 PKI

All hybrid schemes proposed in the following rely on the exchange of multiple public keys per entity by the means of certificates, where entity refers to the server's or the client's identity. In particular following issues have to be addressed:

1. Two instead of one public key has to be bound to each entity.
2. The certificate has to be signed by a CA using a hybrid signature scheme.
3. Since the certificates are going to be used in the transition phase from conventional to PQ cryptography it is desired that the conventional signature is non-separable in order to mitigate the risk of downgrade attacks.
4. Certificates should be backwards compatible such that a legacy software can still make use of the conventional public key and verify the conventional signature.

The straight forward solution is to define a new certificate format that includes the entity's data, the conventional public key, the PQ public key and is signed with a PQ signature scheme. Over all these data fields including the PQ signature, the conventional signature is computed and added as shown in Figure 12. Both signatures are generated using the CA's hybrid private key, which consists of a quantum resistant and a classical private key.

The PQ signature is directly a valid signature for the data that is signed. However, the conventional signature is only valid for the data *and* the PQ signature and thus cannot be used for just the data, i.e. when an attacker removes the PQ signature, the conventional signature becomes invalid. Hence, we say the conventional signature is non-separable. This hybrid signature scheme corresponds to the nested combiner introduced in Section 2.7.2.

In a typical legacy scenario we can assume two types of systems:

1. Updated systems that are aware of hybrid signatures. They will expect either the combination of two signatures (hybrid) or a signature using only a classical scheme coming from legacy systems. They are equipped with hybrid certificates.
2. Legacy systems that are not aware of the hybrid scheme and that use conventional certificates.

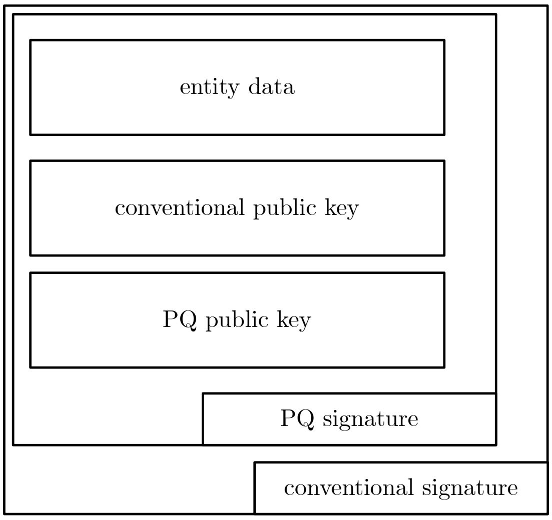


Figure 12: Straight forward approach to hybrid certificates

If an attacker removes the conventional signature and presents this forged certificate to system 1, it will detect that only the PQ part of the hybrid signature is present and will reject the certificate, since there are no systems that use PQ only signatures. System 2 will reject the certificate because it cannot make sense of the PQ signature. Hence we conclude that non-separability of the PQ signature is not important in our transition scenario.

On the contrary, if it was possible to separate the conventional signature, i.e. to remove the PQ signature, system 1 could be tricked into believing that it was presented with a legacy certificate and thus would accept it. This opens the possibility of downgrade attacks and justifies the need for non-separability of the conventional signature.

3.1.1 Hybrid X.509 Certificates

The certificate format described above is not compatible with the X.509 certificate format [25] as described in Section 2.2; X.509 was designed to strictly bind one public key to the subject and to be signed with exactly one signature scheme by the CA. Only the extension field within the *tbsCertificate* data allows to add custom data. Following we introduce and compare four possible designs for hybrid certificates that maintain compatibility with the X.509 certificate format.

Dual Certificates The most basic method is to issue two certificates, one using a conventional and the other using a quantum resistant public key [65]. Each of them is signed by the same CA but using a different signature scheme (conventional and PQ respectively). This method has the disadvantage that the two certificates can have different validity dates as well as that they have to be handled individually. All systems need to maintain both certificate versions and have to know when to use only a conventional certificate for legacy systems and when they have to use both. Also the file size of both certificates combined is larger than necessary because all the subject and issuer information is duplicated. It does not provide any non-separability.

One advantage is that after a transition to PQ-only cryptography, it is very easy to discard the conventional certificates.

Concatenation The Open Quantum Safe project concatenates keys and signatures [17]. Therefore a new hybrid algorithm is defined with a new Object Identifier (OID), for example the combination of RSA and Dilithium would get its own OID assigned. Then the two public keys are byte wise concatenated and now represent a "single" key of the new scheme and therefore can be added to a X.509 certificate. Software that processes these certificates must be aware of the new OID and would then know the signatures and public key lengths. With this information they could retrieve each individual key/signature. This approach does not offer any non-separability in a strict sense, both signatures are valid for the *tbsCertificate* data. However, the data contains two public keys which could be used to detect a missing signature. While this approach is compatible with X.509, it is not backwards compatible since legacy systems can not recognize the new OID.

Nested Certificates A derivate of the dual certificates is to embed one certificate into another certificate as a custom extension [65]. First, a certificate with a PQ public key and a PQ signature is generated, which we consider the inner certificate. Then a second certificate with a conventional public key is created, called the outer certificate. The byte representation of the inner certificate is stored in a custom extension of the outer certificate. Figure 13a illustrates the resulting hybrid certificate.

In this approach the, subject data is still duplicated. However the whole certificate is backwards compatible if the extension is flagged non-critical. A legacy software would ignore the custom extension with the inner PQ certificate and verify the outer conventional certificate. The inner certificate can be separated and used by itself, however the outer certificate is non-separable because the inner certificate is part of the signed data.

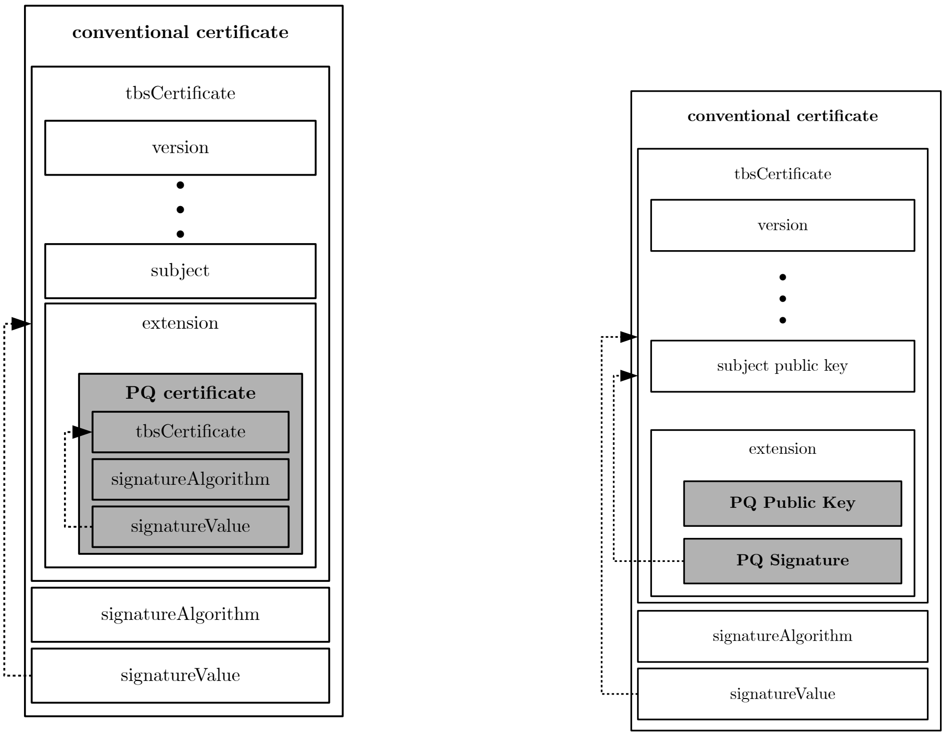
Custom Extension To avoid the overhead of the duplicated subject fields, it has been proposed to only store the additional public key and the additional signature in two custom extensions [72] as can be seen in Figure 13b. This leads to the problem that the additional signature must be computed over *tbsCertificate* but is part of the *tbsCertificate* data block itself. We consider finding such a signature impractical[4]. An experimental openSSL fork for hybrid certificates [73] solves this issue by filling the inner signature with zeros first. Then the inner signature is computed and replaces the zeros before the whole certificate is signed in the conventional way. When the inner PQ signature shall be verified, first the inner signature data is copied into a temporary buffer and then the

---

[4] This reduces to an interesting problem: Find a bit string x such that

$$\mathrm{sign}(data||x) = x.$$

It is not clear if such an x exists how it can be found. For this thesis we prefer to go the 'engineer' way as proposed in this section.



(a) Nested Hybrid Certificate

(b) Hybrid Certificate with extension fields

Figure 13: Two methods of using extension fields to created X.509 compatible hybrid certificates.

data in the certificate is replaced with zeros. Now the *tbsCertificate* data looks the same as when the signature was calculated. By flagging the extensions non-critical, a legacy system can verify the certificate as a conventional certificate. The inner signature can not be removed and therefore the conventional outer signature is non-separable.

Table 11 compares the four schemes. PKI Management describes the expected overhead due to having duplicated subject, issuer and validity information. The last proposed method of embedding the second public key and signature into a custom extension comes very close to the ideal hybrid certificate described in the introduction to this section and therefore was selected to be used in experiments in the scope of this thesis.

Table 11: Comparison of hybrid X.509 certificate schemes. ●= good; ○= bad; ◐= acceptable.

	Backwards Compatibility	Size	Non-Separability	PKI Management
Dual Cert.	◐	○	○	○
Concat.	○	●	◐	●
Nested	●	○	●	◐
Custom Ext.	●	◐	●	●

3.1.2 Implementation

Within the scope of this thesis, two available open source hybrid certificate software packages were investigated. However, as the following shows none of them are flexible enough to be used in this project.

The open quantum safe (OQS) project [17] offers an integration into openSSL. It can also create hybrid certificates, but it uses the 'Concatenation' method only. A fork of the OQS openSSL project [73] uses the preferred 'Custom Extension' method but is limited to the Picnic and qTesla signature algorithms that produce very large key and signature sizes (see Figure 20). Both projects are implemented in C and the last one also has a version in Java using the Bouncy Castle crypto library.

Due to these limitations we decided to create a new software package that can create hybrid certificates as needed for the thesis' experiments. For rapid implementation, the software is written mostly in Python 3.6. Figure 14 shows the components of the program that we named *ccreator*. A library that contains all the needed PQ cryptography functions was written in C by including the sources of the PQClean project. These C-files were compiled and linked into a shared object binary called *libhybrid_crypto.so*. To use it in Python, a *ctypes* wrapper was created that exposes a class for each of the algorithms Dilithium2, Dilithium3, Dilithium4, Falcon512 and Falcon1024 (see Section 3.3 for the selection of cryptographic primitives). Each class provides three methods to generate key pairs, to generate a signature and to verify a signature.

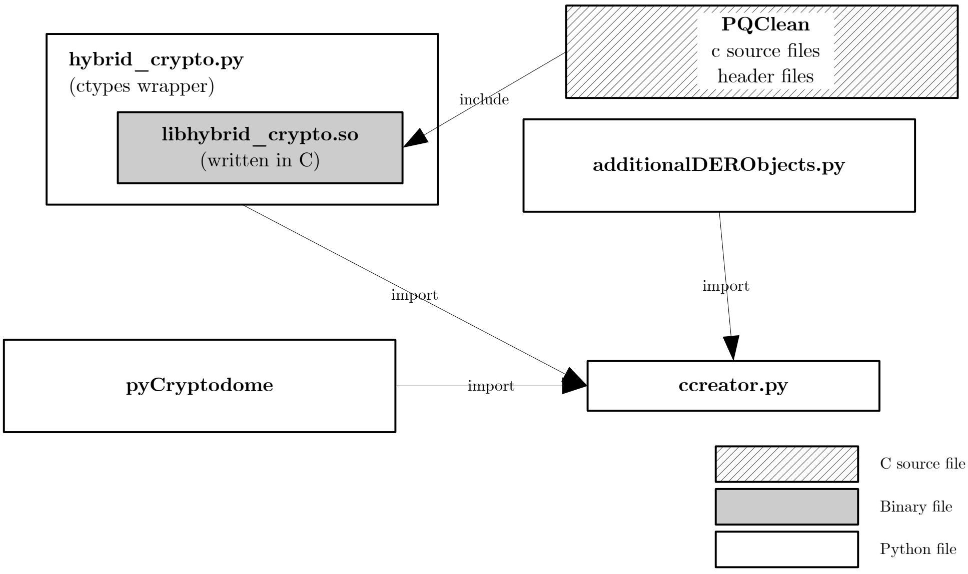


Figure 14: Architecture of the hybrid certificate creation program

The software must be able to create the X.509 certificate structure (as shown in Figure 5) from scratch and freely manipulate all the fields. For example, we want to be able to set the subject and issuer field. It is also required to represent a certificate as well as each field in its binary form according to DER. All reviewed cryptography libraries that are able to create X.509 certificates do not offer this flexibility[5]. Therefore classes for each constructed field were created. For instance a certificate is represented by the class *DerCertificate* and has the attributes *tbsCertificate*, *signatureAlgorithm* and *signatureValue* which in turn are also represented by their corresponding class. All the classes are inherited from the virtual class *DerObject*. Figure 15 shows a detail of the class diagram.

The base class *DerObject* has the virtual method *Encode()* that returns a byte-string representing the object. In the case of the *DerCertificate*, the *Encode()* method calls the implementation of the *Encode()* method of all of its attributes and concatenates them to a DER sequence[6].

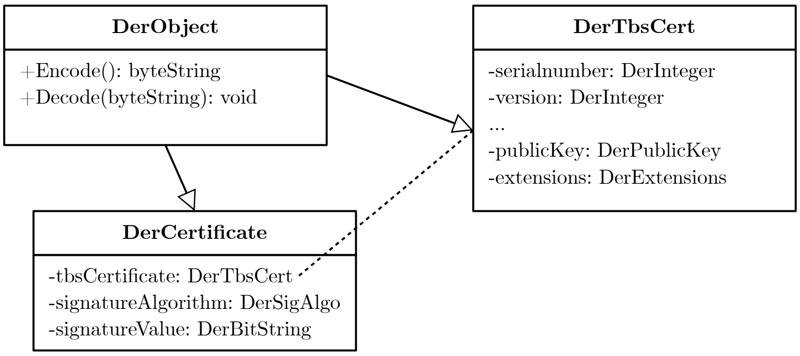


Figure 15: Simplified class diagram of the DER classes

For example, to compute the signature of a certificate, we could call the *Encode()* method of the *tbsCertificate* to retrieve the binary data that we want to sign, then pass this to the signature algorithm and write the result to the *value* attribute of the certificates *signatureValue* attribute as sketched out in Algorithm 2. Some of the more basic DER objects could be used from the publicly available pyCryptodome library and the additional DER objects were implemented in the file *additionalDERObjects.py*.

Algorithm 2 Signing a certificate object

```
$binaryMessage$ = certificate.tbsCertificate.Encode()
signAlgo = Dilithium2()
$signature$ = signAlgo.sign($binaryMessage$, $secret\_key$)
certificate.signatureValue.value = $signature$
```

Each algorithm is identified in X.509 by its unique OID. The OID is a hierarchical naming scheme managed by the International Telecommunications Union (ITU) and the International Organization for Standardization (ISO) [74]. Since no OIDs are specified yet for the algorithms in the NIST PQ project, we use arbitrary values for experiments within this thesis. They have

---

[5] N.B.: The Java library Bouncy Castle was found to offer the best functionality when dealing with low level certificate manipulation. However due to a lack of Java coding practice we decided against using this library.

[6] A DER sequence consists of a header byte that identifies it as a sequence, plus a few bytes indicating the length then followed by the binary representation of the objects in the sequence

to be replaced as soon as they are officially specified.

When the extension fields of an X.509 certificate are parsed, the extension is also identified by their OID. If a system does not recognize the OID of an extension, it will check another field that specifies if the extension is critical. If this is false, i.e. the extension is non-critical the system will ignore the extension. This will happen for example if a legacy system tries to parse any of the new extension OIDs of Table 12. Figure 16 illustrates where the extension OID and the algorithm OID is used.

Table 12: Additional OIDs used in this thesis.

	OID	**Object**
Extension Identifier	1.2.3.413	Hybrid Public Key Info
	1.2.3.412	Hybrid Signature
Algorithm Identifier	1.3.6.1.4.300.1	Dummy
	1.3.6.1.4.100.2	Dilithium 2
	1.3.6.1.4.100.3	Dilithium 3
	1.3.6.1.4.100.4	Dilithium 4
	1.3.6.1.4.200.1024	Falcon 1024
	1.3.6.1.4.200.512	Falcon 512
	1.3.6.1.4.300.1	Kyber 512
	1.3.6.4.300.3	Kyber 768
	1.3.6.4.300.5	Kyber 1024

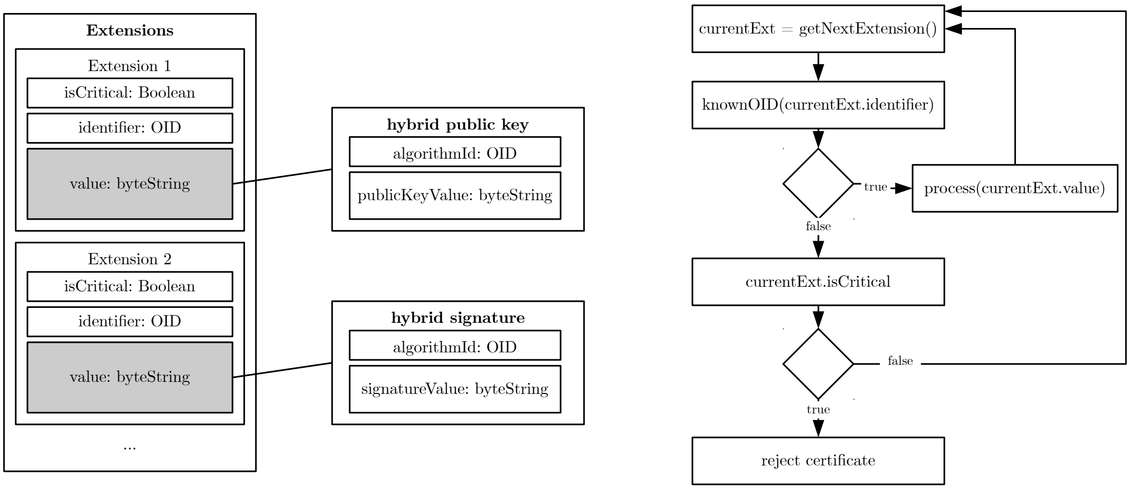


Figure 16: Shows the extension object that resides inside the tbsCertificate object. The value of an extension is just binary data. However with the knowledge of the extension identifier we know how to decode that data.

The following set of 7 certificates has been created for each signature algorithm that was tested.

- A self signed certificate for the root CA called *root*

- Two application certificates that are signed by *root* and are called *root_signed_1* and *root_signed_2*.
- Two intermediate CA certificates, called *intermediate_1* and *intermediate_2* each of them signed by *root*.
- Two application certificates signed by *intermediate_1* and *intermediate_2* called *intermediate_signed_1* and *intermediate_signed_2*.

3.2 Key Exchange Mechanism in OPC UA

In order to simplify the security analysis of the secure channel, we divided it into the key establishment phase and the data transmission phase. The goal of the first step, the key establishment, is to derive a secret key, shared by both parties while providing BR-Match and BR-Key-Secrecy security. In the second step this shared secret is used to provide authenticity and confidentiality to the data transmission. Figure 17 illustrates the two phases. Both steps are assumed to be secure against conventional attackers (C^cC) which was the initial design goal of OPC UA and which has been shown by several studies [13], [14], [75].

Another simplification is the assumption that as soon as an adversary becomes quantum, he has access to RSA private keys. This is reasonable since via factorization using Shor's algorithm the attacker calculates the private key from the public key.

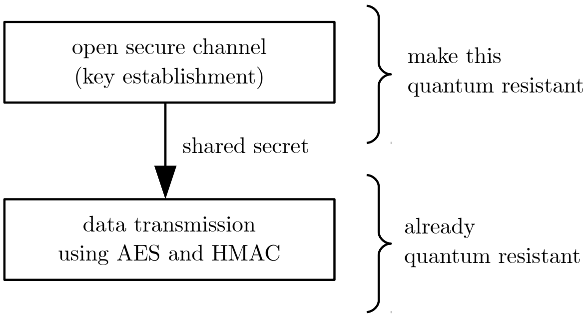


Figure 17: The secure channel can be seen as a two step process: First a shared secret key is established and then it is used in the secure channel.

First we analyse the second step, the data transmission through an established secure channel. 'Established' means that both parties already share the same secret key. When using the security policy *Basic256Sha256*, the following symmetric cipher suite is used:

- AES with 256 bit key length to encrypt and sign messages.
- Sha256 for hashes and as basis for Keyed Hash Message Authentication Code (HMAC).

Shor's algorithm is not applicable to symmetric schemes and Grover's algorithm is only able to half the bit level security. Therefore, the above primitives still provide 128 bit of security against a Q^cQ quantum attacker, which we consider sufficient. Hence, we can use the data transmission part of the secure channel as provided by OPC UA in a PQ scenario, given we select the proper protocol parameters. Hence, from here on we will focus the analysis on the key establishment process only.

The key establishment as described in Section 2.1.3 is based on the security of RSA. C^cQ-BR-Key-Secrecy is completely broken as following shows. In the experiment we have two parties that participate in the protocol: The client denoted as C and the server denoted as S. During setup of the experiment, public and private keys are generated for both parties. The server's key pair is denoted as (pk_S, sk_S) the client's key pair as (pk_C, sk_C).

The attacker will call **NewSession(C, S, "initiator")** which will cause C to generate a random value called $Nonce_C$, encrypt it using pk_S, sign it using sk_C and send it to S. The adversary stores the encrypted message for later. Upon reception the server will start its own session of the protocol, paired with C's session. He will generate a random number $Nonce_S$, encrypt it using pk_C, sign it using sk_S and send it to the client. Again the adversary intercepts and stores the message for later use. At this point the attacker is not quantum yet, therefore cannot decrypt the messages. The two sessions of the server and the client have become paired sessions. They both are using the same session identifier *sid* and have derived the same key K. The attacker finally calls **TEST**() on any of the two sessions and stores the resulting key. Now we enter the second stage of the experiment, where the attacker gains access to a quantum computer and therefore gains knowledge of sk_S and sk_C. He then proceeds to decrypt the two previously stored messages and uses $Nonce_C$ and $Nonce_S$ to derive the key. The attacker easily wins the experiment by comparing the derived key with the stored result from the test function.

C^cQ-BR-Match security is still given. The access to a quantum computer in the second stage does not give any advantage to the attacker since he is not allowed to interfere with the protocol any more because this would require access to the functions **NewSesssion()** and **Send()**. This insight becomes trivial considering that authentication cannot be broken retroactively.

A Q^cQ attacker however can also break BR-Match security. With knowledge of the private keys in the first stage of the experiment, he can decrypt, alter, re-encrypt and sign any message he wants. Thus he could for instance answer to all messages of a client pretending to be the server. Finally the client would have a session $\Pi^j_{C,S}$ while there is no corresponding session in the server, but the attacker would have a paired session $\Pi^i_{A,C}$.

Next, we propose two different additions and changes to the original OPC UA secure channel establishment mechanism in order to gain resistance to quantum attackers. The first version is very close to the original OPC UA approach and achieves BR-Key-Secrecy and BR-Match security. The second variant uses a generic authenticated key exchange method and achieves BR-Key-Secrecy, however BR-Match security can only be guaranteed under certain conditions. Both methods rely on a Q^cQ-IND-CPA KEM. The evaluation and selection of such a KEM was done in previous work and both variants are based on the results of this work [42].

Before the schemes are explained we introduce the notation used:

$cert_U^V$ Describes a certificate where $V \in \{H, Q, C\}$ specifies if it is a classical certificate (C), a PQ certificate (Q) or if it is a hybrid certificate (H). U states the subject of the certificate, usually S for server or C for client, but is not limited to those.

(pk_U^V, sk_U^V) Public/private key pair, where $V \in \{Q, C\}$ indicates if the key is used in a conventional or in a PQ scheme and $U \in \{S, C, E\}$ describes if the key is associated with the server (S), the client (C) or if it is an ephemeral key (E) that was just generated for a single connection.

c_U^V Cipher text that is generated by a KEM, where $V \in \{Q, C\}$ indicates if it is a classical or PQ scheme and U indicates who's private key was used to generated the ciphertext (client C or server S), i.e. who the intended recipient of the ciphertext is.

$H(\cdot)$ Cryptographic hash function.

$\sigma_U^V(\cdot)$ Signing function. $V \in \{Q, C\}$ determines if it is a conventional or a quantum resistant signing function and U specifies who's private key was used, i.e. who signed the message.

3.2.1 Variant One

In the previous work, it was proposed to add a quantum resistant KEM. Therefore, following steps are added to the original key establishment process.

- The client C generates a quantum resistant ephemeral key pair (pk_E^Q, sk_E^Q).
- C creates the openSecureChannelRequest message and includes pk_E^Q. The public key is also signed with the conventional signature scheme, because it is inside the message.
- The server S receives the openSecureChannelRequest and extracts pk_E^Q.
- S calls the Encaps(pk_E^Q) function of the KEM which will return a ciphertext c_E^Q and a secret value s. s serves the same purpose as the OPC UA nonces as they will be used to derive the shared key later on.
- S calculates a MAC in order to implement the XORthenMAC hybrid scheme and adds c_E^Q and the MAC to the openSecureChannelResponse. Both are included in the data that is signed using the classical scheme.
- C decapsulates c_E^Q and retrieves s, verifies the MAC and now knows the additional input for symmetric key derivation.

This scheme protects against a C^cQ attacker: In the first stage where the attacker is allowed to interact with the participants and their messages, the attacker has no access to a quantum computer and therefore conventional primitives are considered secure. In the second stage the attacker cannot interact with the messages any more and must rely on data collected in the first stage, however he has access to a quantum computer and can break conventional primitives which can be simplified as giving the attacker access to the conventional private keys. Since the Open Secure Channel Request (OSCRq) and the Open Secure Channel Response (OSCRp) are both secured with conventional signatures, the attacker can not alter or spoof messages in the first stage. But he can store all the messages that are exchanged and pass it to the second stage of the experiment. In the second stage the attacker can decrypt the server and client nonce but cannot decipher c_E^Q. Therefore, he is missing one input for the Key Derivation Function (KDF) and cannot obtain the shared secret and therefore C^cQ-BR-Key-Secrecy is provided. Since BR-Match security can only be broken in the first stage, the C^cQ has no advantage over a conventional C^cC attacker for whom the secure channel key establishment was designed and which is already assumed to be secure.

Next, we show that this scheme does not provide any BR-Match or BR-Key-Secrecy security against a Q^cQ attacker: Here the attacker has knowledge of the conventional private keys even in the first stage of the experiment using his quantum computer. This means that he can alter any message and recompute a valid signature. In the first stage, the attacker intercepts the OSCRq message. He then replaces pk_E^Q with $\widetilde{pk_E^Q}$, a new public key that the attacker has generated himself and thus knows the corresponding private key. The server then continues to create a secret $\tilde{s}$ and a ciphertext $\widetilde{c_E^Q}$ using $\widetilde{pk_E^Q}$. The attacker intercepts the OSCRp and is able to decrypt $\widetilde{c_E^Q}$ and thus can compute the same session key $\tilde{K}$ as the server. Then he uses the original public key pk_E^Q to encapsulate a new secret s in the ciphertext c_E^Q and sends this ciphertext within the OSCRp back to the client. The client will use s to derive a secret K that the attacker also can compute because he knows all the inputs to the KDF. Both sessions in the client and server will reach the status 'accepted' but the session keys are different, i.e. $K \neq \tilde{K}$. Therefore the attacker has broken BR-Match security. The attacker passes K and $\tilde{K}$ to the second stage of the experiment and calls the **test()** function. He can now compare the returned key to K or $\tilde{K}$ depending on which session he called the **test()** function and can always win the experiment. Thus BR-Key-Secrecy is also broken.

Following we propose a modification of the above protocol in order to achieve BR-Match and BR-Key-Secrecy security against a Q^cQ adversary. Therefore we analyse how the possibilities for an attacker change when we switch from C^cQ to Q^cQ. The important difference is that the C^cQ attacker cannot act actively because he is not able to break the conventional signature scheme. Thus if we replace this signature scheme with a Q^cQ-EUF-CMA secure scheme, the Q^cQ would find himself in the same situation during the first stage of the experiment as the C^cQ attacker in the unmodified protocol. To achieve that we propose to use a hybrid signature scheme to sign the OSCRq and the OSCRp utilizing a concatenation combiner. This combiner does not provide non-separability, however this security feature does not bring any advantage here. The additional quantum resistant signature is calculated over the original message and is then appended after the conventional signature. The additionally required public key for the quantum resistant signature scheme is transported by the means of a hybrid certificate as detailed in Secction 3.1. Figure 18 shows this approach schematically.

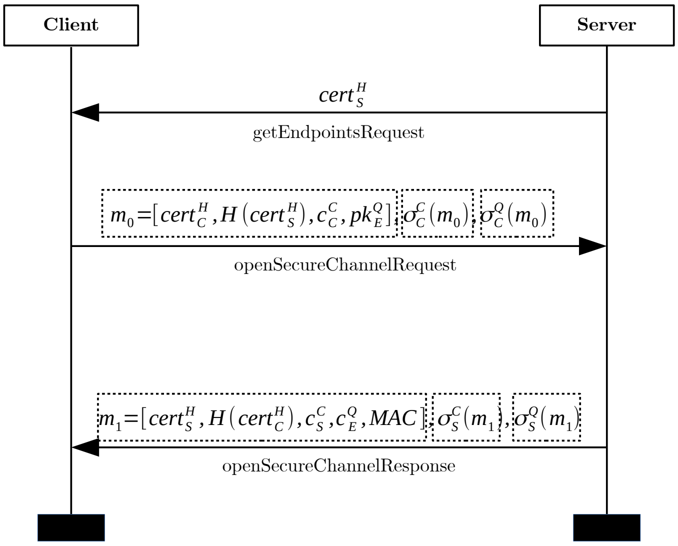


Figure 18: Keeping the addition of a hybrid KEM and adding quantum resistant hybrid signatures to the messages. The conventional certificates are replaced by quantum resistant certificates.

Summarizing the rational behind the protocol modification: On an abstract level we keep the protocol how it is, but replace the conventional primitives with hybrid primitives and therefore gain confidence that the protocol is still secure.

As Figure 18 also shows, each message contains a hash over the receiver's certificate, in OPC UA terminology called a *thumbprint*. This measure thwarts a specific identity mismatch attack [76]: Consider a client C, a legitimate server called S and a malicious server called M. M can use his identity and S' public key to obtain a certificate from a RA. This would work if we assume that the RA does not request a proof of possession of the corresponding private key. The attacker could now trick the client into connecting to him and present him his certificate with S' public key. C would think he is talking to M. M however can forward all traffic to S. Eventually C and S would agree on a shared secret, however C would think that he speaks to M[7]. However if the server S compares the thumbprint to his own certificate he would notice the attack and would reject the connection.

3.2.2 Variant Two

An alternative way of creating a hybrid authenticated key exchange protocol that only relies on a KEM [77] and does not require hybrid signatures is depicted in Figure 19. This is a generic key exchange protocol, where the hybrid certificate's PQ public key is actually the public key for the KEM and not a signature scheme key, as in the previous protocol. Once

---

[7] In the context of OPC UA this poses a real life risk: Imagine two companies A and B that run industrial systems and are competitors. Both companies are customers of a third company C which offers remote support services. A and B grant access for C to all their systems because they trust C. Now A calls C and asks them to remotely log into his system and issue a shutdown command. However, A forwards all commands to the system of company B. In summary, A has successfully tricked C to shutting down B's system.

the client has received and verified the server's certificate, he uses the server's public key to encapsulate the secret value s_C into the ciphertext c_S^Q. The server does the same with the client's public key pk_C^Q which he retrieves from the client certificate $cert_C^H$. Additionally, as in the previous protocol, an ephemeral secret is exchanged and the steps of the conventional OPC UA protocol, such as client nonce and server nonce creation and encapsulation into c_S^C and c_C^C, are performed (not shown in Figure 19).

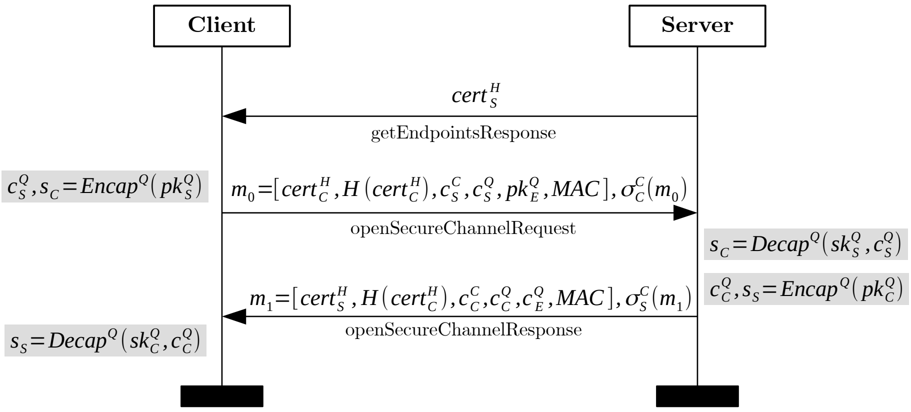


Figure 19: Additional steps in variant two of the key exchange protocol.

The combination of conventional and quantum resistant keys is done by utilizing the XOR then MAC technique. Therefore a MAC over the ciphertexts is computed. For the OSCRq, this is $MAC(c_S^C||c_S^Q, k_{mac,C})$ and for the OSCRp it is $MAC(c_C^C||c_C^Q||c_E^Q, k_{mac,S})$, where $||$ denotes byte wise concatenation. The symmetric MAC keys $k_{mac,C}$ and $k_{mac,S}$ are derived from the symmetric keys that are encapsulated. They are combined using XOR.

The main changes in this variant of the key establishment are:

- The size of hybrid certificates changes: Instead of the public key of a quantum resistant signature scheme it now contains the public key of a quantum resistant KEM.
- The messages that are exchanged between client and server do not have to be signed by a hybrid scheme. Thus we do save the bytes that were used for the quantum resistant signature part and save the computing time for hybrid signature generation.
- But we add an additional ciphertext from the KEM in each message.
- And we have to perform an extra encapsulation and decapsulation operation on server and client each.

As the results section will show this can lead to a gain in performance depending on the actually used schemes.

While the security of this key exchange is guaranteed in the Canetti-Krawczyk model [77] it is easy to see that it cannot hold BR-Match security against an Q^cQ attacker. Let's assume that in the first stage of the experiment, the adversary alters the ephemeral public key pk_E^Q to $\widetilde{pk_E^Q}$ in the OSCRq. In the OSCRp the adversary replaces $\widetilde{c_E^Q}$ with $c_E^Q = Encaps(pk_E^Q)$.

This would lead to server and client deriving different shared secret keys $\Pi^1_{C,S}.K \neq \Pi^1_{S,C}.K$ without them noticing, subsequently changing their status to 'accepted' and thus breaking BR-Match security. Note that this does not break BR-Key-Secrecy, because the attacker is still missing the secret values that were generated using client and server's long term public keys pk^Q_S and pk^Q_C.

3.3 Selection of Cryptographic Primitives

The described protocols rely on the following generic quantum resistant primitives:

- A KEM
- A signature scheme

In order to implement prototypes and to conduct experiments it is necessary to select concrete primitives. As mentioned in Section 2, the algorithms proposed in the NIST PQ project round 2 were considered.

For the KEM we did not do a separate evaluation and relied on the results of the predecessor project [42] that implemented and evaluated an unauthenticated key exchange in OPC UA. The Kyber KEM [77] with the parameter sets Kyber-512, Kyber-768 and Kyber-1024 were used. Each parameter set was selected to match with the security level of the used signature scheme.

As a primary selection criterion for the signature scheme we used the public key size and signature size of the scheme. This decision is based on a peculiarity of the OPC UA protocol that requires the messages in the openSecureChannel key establishment process to be transmitted in one message chunk [23, p. 48]. This means that no fragmentation in the OPC UA TCP transport layer can occur. Note that this effect is limited to the transport layer that is defined by OPC UA and thus fragmentation on any lower layer (such as the TCP or Ethernet layer) is allowed. However the standard does not specify the maximum size of such a chunk, it only requires the implementations to provide a chunk size of at least 8192 bytes but it could be more depending on the lower networking layers used. For the measurements we relaxed this hard criterion and assumed a chunk size of at least 16 kiB. However this 16 kiB also accounts for the payload data, the conventional signature data as well as the public key and ciphertext of the Kyber KEM. This allows a future work to also see which cryptographic schemes are only slightly above the maximum size and might be useful in combination with other protocol changes.

In proposed 'Variant One', adding a quantum resistant signature scheme requires an additional signature in each exchanged message and an additional signature as well as an additional public key in the certificate. Hence for the size comparison following metric was used:

$$publicKeySize + 2 \cdot signatureSize \tag{23}$$

Figure 20 clearly shows that Dilithium and Falcon are best suited considering their public key and signature size. Additionally a performance investigation study for authenticated

PQ handshakes in TLS comes to the same conclusion [78]. The L1 variants of MQDSS and SPHINCS+ have sizes that would fall within the 16 kiB limit, however, it was decided to exclude them from the experiments because they don't allow a margin for additional data and conventional signatures and public keys which also have to be included in the openSecureChannel messages. Table 13 shows the corresponding sizes in bytes.

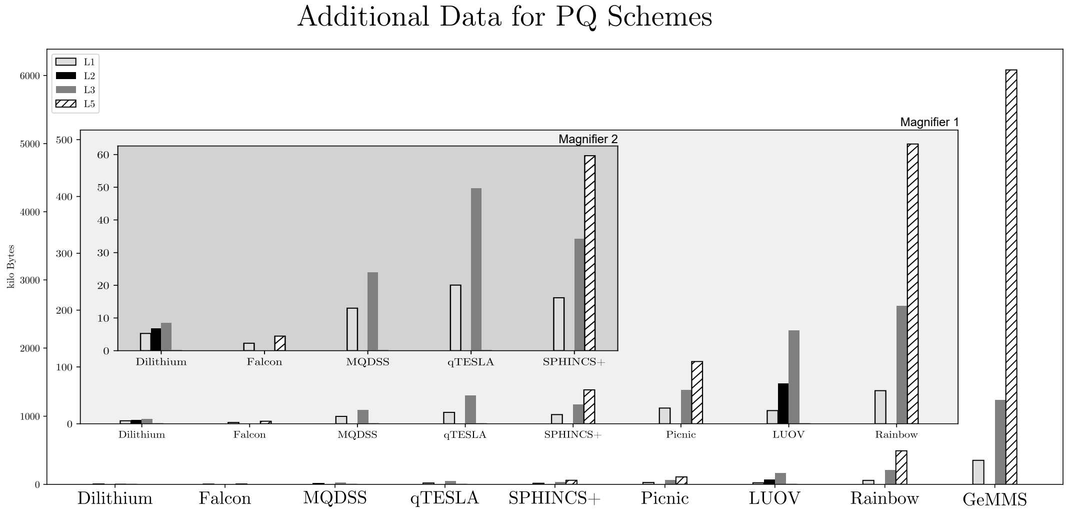


Figure 20: Additional data needed per PQ signature scheme and security level. Note that for each scheme only parameters for certain security levels are available.

Table 13: Additional data required per signature scheme. Size in bytes. Note that not all schemes offer parameters for all security levels.

	Dil.	**Falcon**	**GeMMS**	**LUOV**	**MQDSS**	**Picnic**	**qTESLA**	**Rainbow**	**SPHI.**
L1	5 272	2 112	352 446	23 239	13 004	27 636	20 064	58 272	16 192
L2	6 874	–	–	71 137	–	–	–	–	–
L3	8 492	–	1 238 786	164 440	23 892	59 548	49 760	207 056	34 176
L5	–	4 453	6 081 835	–	–	109 528	–	492 344	59 648

Our conclusion is to use Dilithium and Falcon with all their available parameter sets to build prototypes for the above presented key exchange protocols and conduct the performance evaluation based on them. The parameter sets that provide higher security levels than needed (>128 bit) are investigated because the NIST PQ project is still ongoing and it can be possible that the security levels might change in the future.

3.4 Prototype Implementation

3.4.1 Variant One

The authenticated hybrid key establishment scheme was implemented into the open source OPC UA library *open62541*, which is available on Github. The source code contains an example folder which in turn contains an encrypted server and client. The client establishes a secure channel with the server, opens a new session and retrieves the current time and date.

Then the client closes the secure channel and terminates. This setup was used for all the conducted experiments.

The unauthenticated quantum resistant key exchange was already implemented and only had to be merged into the most current version of *open62541*. Modifications that were implemented within the scope of this thesis are:

- Handling of hybrid certificates. For this we can take advantage of the modular structure of *open62541*. Internally, a function pointer references a single method that verifies certificates. The function itself resides in a plugin, which is realised in an external c-file. A new function to verify hybrid certificates was created and the function pointer was adjusted accordingly.
- All secure channel related functions are organized in a security policy. Therefore, a security policy data structure object acts as an interface providing function pointers to signing, verify, encryption, etc. functions and additionally stores context data such as private and public keys. In the concrete implementation this is a C-struct with function pointers and context data variables. The server can support multiple security policies. The OSCRq sent by the client contains a field that specifies the security policy the client wants to use and the server selects the matching security policy and from there always uses the callback functions associated with the policy when cryptographic functions are needed. We copied the existing security policy *Basic256Sha256*, renamed it to *Hybrid* and modified some of the function pointers to point to new functions that are described in the following.
- *Open62541* makes use of a data structure that represents a channel. A pointer to this data structure is passed as a parameter to all relevant functions. The channel stores all context data of a channel such as the exchanged certificates, derived symmetric keys, a pointer to the used security policy, etc.
- A function that signs messages. The original asymmetric function was copied and modified in such a way that it adds another quantum resistant signature at the end of the message buffer. The size of the message buffer, located in the security policy was adjusted accordingly.
- Analogue to the signing function, a verification function that recognizes hybrid signatures and verifies them had to be implemented and referenced in the *Hybrid* security policy.
- Minor changes in the core source code due to changed interfaces, for example to support two private and public keys.
- All cryptographic functions were implemented in a separate library file, that was also used by the certificate creation Python program (see Section 3.1.2).

Figure 21 shows an overview of the modular design. The library libhybrid_crypto.a is the same as the one used for the certificate creator Python software but it is statically linked into the OPC UA server and client. The plugins have access to certain secure channel context objects that contain data such as the root certificates and the private keys that are needed, however, this is not shown in Figure 21.

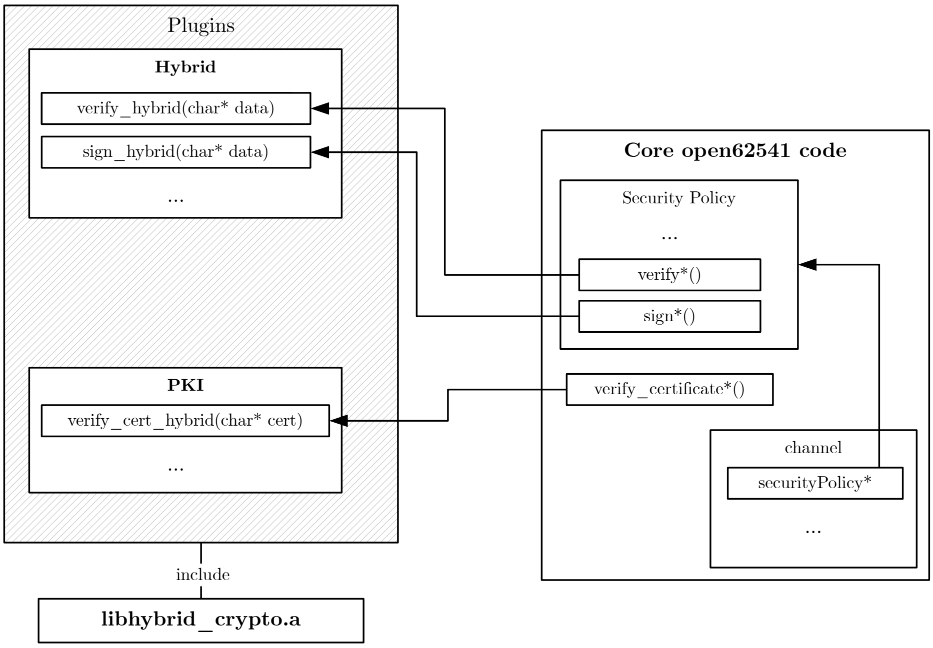


Figure 21: Modular structure of *open62541*.

libhybrid_crypto.a This library is mainly an interface wrapper around the signing functions from the PQClean library. The relevant source code of PQClean was directly copied into the libhybrid_crypto project folder.

The listing shows the interface of the library, exposing a keypair generation, a verify and a sign function for each signature scheme in each parameter set.

```
int dilithium2_keypair(uint8_t *pk, uint8_t *sk);
int dilithium2_sign(uint8_t *sig, size_t *siglen, const uint8_t *m, size_t
   ↪ mlen, const uint8_t *sk);
int dilithium2_verify(const uint8_t *sig, size_t siglen, const uint8_t *m,
   ↪ size_t mlen, const uint8_t *pk);

int dilithium3_keypair(uint8_t *pk, uint8_t *sk);
int dilithium3_sign(uint8_t *sig, size_t *siglen, const uint8_t *m, size_t
   ↪ mlen, const uint8_t *sk);
int dilithium3_verify(const uint8_t *sig, size_t siglen, const uint8_t *m,
   ↪ size_t mlen, const uint8_t *pk);
```

```
int dilithium4_keypair(uint8_t *pk, uint8_t *sk);
int dilithium4_sign(uint8_t *sig, size_t *siglen, const uint8_t *m, size_t
    ↪ mlen, const uint8_t *sk);
int dilithium4_verify(const uint8_t *sig, size_t siglen, const uint8_t *m,
    ↪ size_t mlen, const uint8_t *pk);

int falcon1024_keypair(uint8_t *pk, uint8_t *sk);
int falcon1024_sign(uint8_t *sig, size_t *siglen, const uint8_t *m, size_t
    ↪ mlen, const uint8_t *sk);
int falcon1024_verify(const uint8_t *sig, size_t siglen, const uint8_t *m,
    ↪ size_t mlen, const uint8_t *pk);

int falcon512_keypair(uint8_t *pk, uint8_t *sk);
int falcon512_sign(uint8_t *sig, size_t *siglen, const uint8_t *m, size_t
    ↪ mlen, const uint8_t *sk);
int falcon512_verify(const uint8_t *sig, size_t siglen, const uint8_t *m,
    ↪ size_t mlen, const uint8_t *pk);
```

Open62541 uses the cryptographic library *mbedtls* [79] for all security relevant operations including the conventional verification of certificate chains. Therefore, the certificate chain and the trusted root certificates are passed to a verification function provided by *mbedtls*. The verification function of *mbedtls* allows to provide a callback function as a parameter that will be called after each certificate in the chain was verified. We use this mechanism to implement the hybrid signature verification functionality. Inside the callback function we have access to the current and the previous certificate in the chain. To verify a certificate we have to retrieve the public key from the previous certificate and verify the PQ signature in the current certificate. Note that we do not need to check if issuer and subject match since these kind of checks were already performed by *mbedtls* when the callback function is called. When there is no previous certificate we arrived at the end of the chain and have to decide whether we are dealing with a trusted root certificate. This is also done by *mbedtls*. The flowchart in Figure 22 shows the chain verification process.

Hybrid Signature Verification Whenever *open62541* receives a message on the network socket that was signed using an asymmetric method (i.e. during secure channel establishment), the signature size is retrieved from the security policy object and the message is divided into message part and signature part on byte string level. Then those two byte strings are passed as parameters to the asymmetric verification function of the current security policy. In this function, we separate the concatenated signature into the conventional and PQ signature based on fixed offsets and call the corresponding verification function for each signature on the message. Only if both verification functions return true, the hybrid verification function

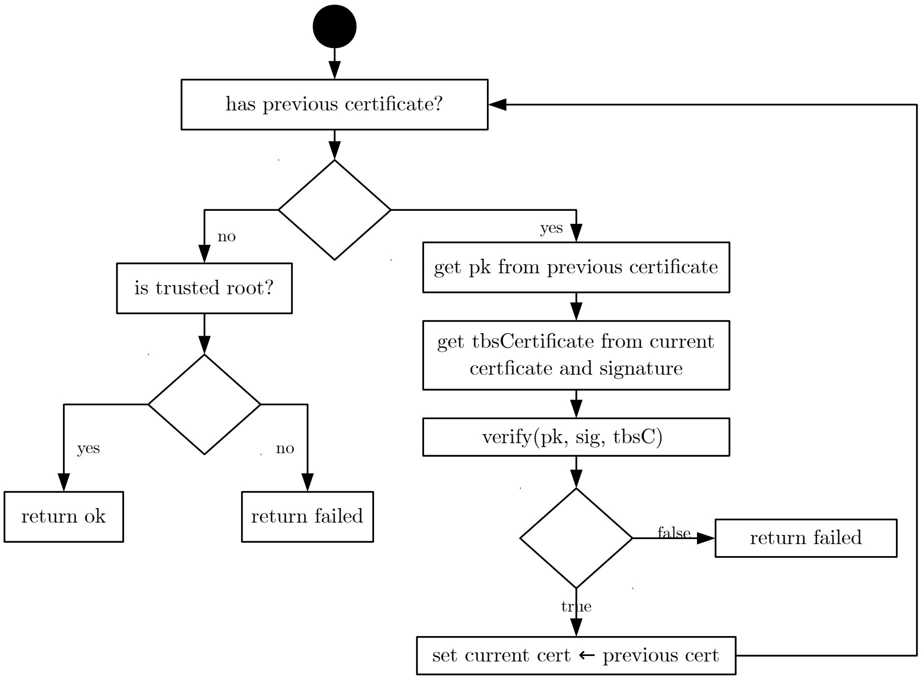


Figure 22: Process of verification of a certificate chain. Note that comparing issuer of the current certificate matches the subject of the previous certificate is performed by mbedtls automatically.

returns true. Figure 23 illustrates the process, which relates to the concatenation combiner explained in Section 2.7.2.

Appendix E shows the implementation in detail.

3.4.2 Variant Two

The basis for the implementation of our 'Variant Two' of the key establishment is the *open62541* code with the modifications for an unauthenticated quantum resistant key exchange from the predecessor project [42] which uses Kyber [77]. The code modifications can be divided into following parts:

Modified Certificates While in RSA the same public key can be used for signatures and the key exchange, this is in general not possible for the used PQ schemes. A key pair is specifically dedicated for a signature scheme or a KEM. Thus, the quantum resistant part of the hybrid device certificate must contain a KEM public key. However the CA certificates, that sign the device certificate, must contain a PQ public key for a signature scheme. Figure 24 illustrates this. These new device certificates are created using the *ccreator* Python tool described in Section 3.1.2.

Access to Key Pairs The functions in *open62541* that create and process the OSCRq and the OSCRp need access to the new quantum resistant key pairs. In particular, the functions that create the OSCRq and OSCRp have to access the long term private key and

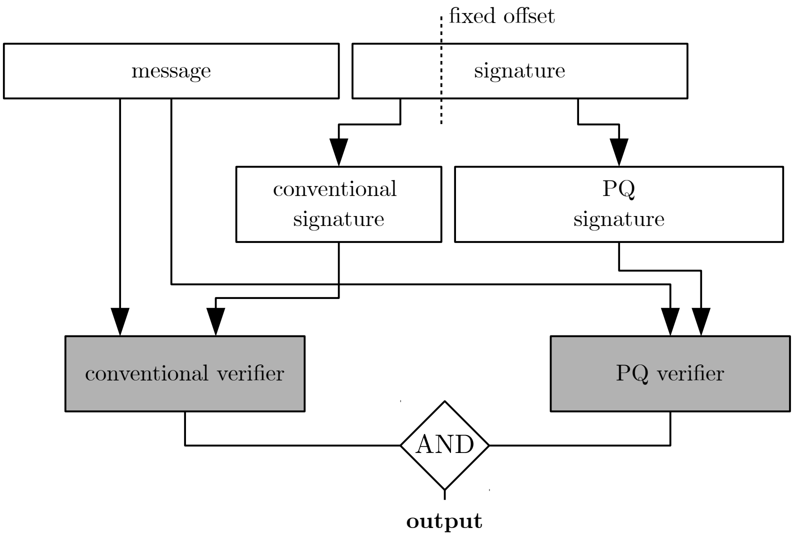


Figure 23: The hybrid signature verification function receives the message and the signature as parameters. The signature is then separated into the conventional and the quantum resistant part. Each signature is passed to their corresponding verifier function together with the message. Both have to return true.

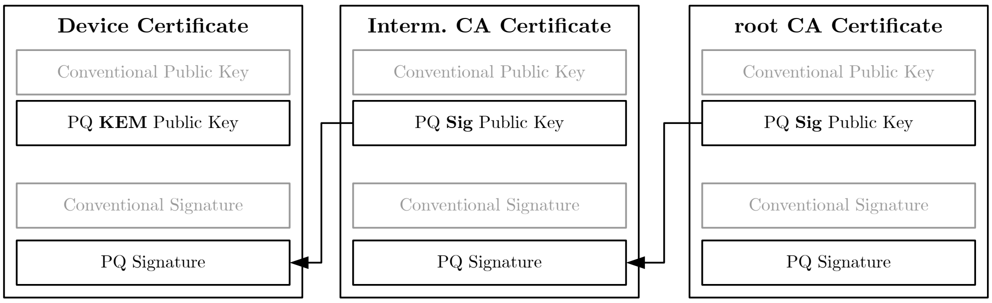


Figure 24: The PQ part of the device certificate contains a KEM public key. The issuer certificates that signs the public key use public keys of a signature scheme.

the functions that process the received messages need access to the long term public key of their communication partner. The function names and where they are located in the *open62541* source code are listed in Table 14.

All of these functions have access to the remote certificate[8] via the channel data structure: `channel->remoteCertificate`. The public key for encapsulation can be extracted from this certificate.

To access the private key, a new variable in the security policy data structure was introduced. Then, the local private key for the KEMs is passed to the program as a command line argument and assigned to the variable in the security policy. All of the functions in Table 14 have access to the current security policy and thus to the private key via the channel data structure: `channel->securityPolicy->postQuantumModule->privateKey`.

---

[8] For the client, the server certificate is considered the remote certificate. Analogous, for the server, the client certificate is considered the remote certificate.

Table 14: Functions in *open62541* that need access to private or public keys of the PQ KEM.

File	Function	Side	Purpose
ua_client_connect.c	openSecureChannel(...)	Client	Create OSCRq
ua_securechannel_manager.c	UA_SecureChannelManager_open(...)	Server	Process OSCRq
au_securechannel_manager.c	UA_SecureChannelManager_open(...)	Server	Create OSCRp
ua_client_connect.c	processDecodedOPNResponse(...)	Client	Process OSCRp

Additional Channel Properties The channel data structure requires additional properties. These are the additional long term shared secrets that are set after decapsulation in the functions that process the OSCRq and OSCRp. Also the new data fields in the OSCRq and OSCRp messages are represented by variables in the channel data structure. The functions in Table 14 do not create the messages directly but just compute the value for the additional fields, store them in the channel data structure and before transmission, the data is copied from the channel data structure.

Asymmetric Security Header Before transmission of the OSCRq and the OSCRp, the asymmetric security header is added to the message [23]. In our implementation we decided to store additional data that is required for the quantum resistant key exchange in this header since it requires less changes in the *open62541* source code. Figure 25 shows the additional fields.

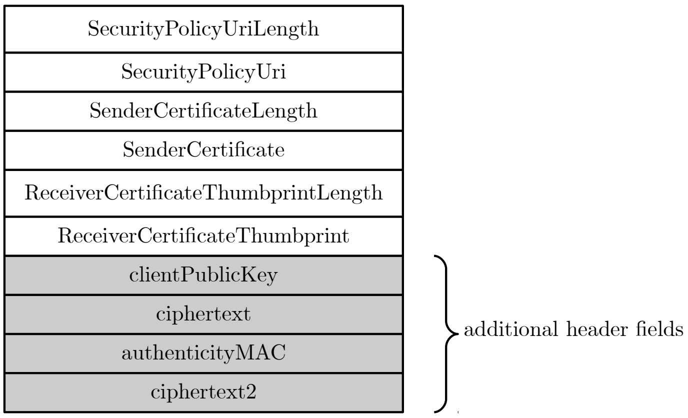


Figure 25: Modified asymmetric security header.

The asymmetric security header of theOSCRq contains the ephemeral public key and the ciphertext that was created using the encapsulation function with the server's public key. AuthenticityMAC is a MAC over the ciphertext and the conventional client nonce, using the XOR of the client nonce and the shared secret as the key as described in Section 3.2.2. Ciphertext2 remains empty.

The OSCRp's asymmetric security header contains a ciphertext that was created using the clients public key and a second ciphertext based on the ephemeral public key. The authenticityMAC is computed over the concatenation of both ciphertexts and the server nonce. The MAC key is the XOR of the local and remote symmetric signing key. These keys are derived from classical client and server nonce, the ephemeral shared secret and both long term secrets.

Appendix F shows the implementation in detail.

3.5 Measurement Setup

One part of this thesis' result is to show the feasibility of hybrid quantum resistant cryptography within the OPC UA protocol. While it is easy to theorize about packet sizes and CPU cycles it is still reassuring to build a running system, proofing that no pitfalls in the protocol or in the used implementation were overlooked.

In addition to that, we want to perform measurements that allow a closer look at our system, especially to see where it differs in resource requirements from the current implementation without any quantum resistant algorithms in place. This will help future implementers to identify weak points that can potentially cause problems when ported to other platforms.

Especially two criteria are evaluated: Sizes of certificates/sizes of network packets and the CPU requirements to execute signing and verification.

3.5.1 The Test System

The performance measurements are conducted using two Raspberry Pi 3 Model B that are connected via a 100 Mbps Ethernet connection and are utilizing an ARM Cortex-A53 microprocessor with 1.2 GHz. Both systems run the Raspbian Buster Linux distribution with kernel version 4.19. The operating system is expected to have a slight effect on the measurements, however this reflects a real world scenario that we can also expect when using an OPC UA server in practice since embedded systems on PLCs often also run Linux.

3.5.2 Software Under Evaluation

The implementation of the hybrid quantum resistant version of *open62541*, described in Section 3.4, produces two binary files when compiled: The server and the client executables. Via compiler flags, different versions of each, the server and client, can be compiled. For the test following versions were created:

1. Original, unmodified implementation to achieve some baseline measurements for later comparison.
2. Only the unauthenticated quantum resistant KEM is in place. No hybrid signatures are used. This is the variant that is only secure against passive C^cQ attacker and was used in the predecessor project [42].
3. A version for each signature scheme and 'Variant One' of the modified key establishment protocol:
    (a) Dilithium 2

Table 15: Versions of the built executables.

	Label	PQ KEM	Hybrid Sign Scheme	Certificate
	EXE_01	–	RSA	RSA
'Variant One'	EXE_02	Kyber 512 (unauth.)	RSA	RSA
	EXE_03	Kyber 512 (unauth.)	RSA+Dilithium 2	RSA+Dilithium2
	EXE_04	Kyber 512 (unauth.)	RSA+Dilithium 3	RSA+Dilithium3
	EXE_05	Kyber 512 (unauth.)	RSA+Dilithium 4	RSA+Dilithium4
	EXE_06	Kyber 512 (unauth.)	RSA+Falcon 512	RSA+Falcon512
	EXE_07	Kyber 512 (unauth.)	RSA+Falcon 1024	RSA+Falcon1024
'Variant Two'	EXE_08	Kyber 512	RSA	RSA+Dilithium2
	EXE_09	Kyber 768	RSA	RSA+Dilithium3
	EXE_10	Kyber 1024	RSA	RSA+Dilithium4
	EXE_11	Kyber 512	RSA	RSA+Falcon512
	EXE_12	Kyber 1024	RSA	RSA+Falcon1024

   (b) Dilithium 3
   (c) Dilithium 4
   (d) Falcon 512
   (e) Falcon 1024

4. A version for each signature scheme and 'Variant Two' of the key establishment.

Table 15 assigns a label to each executable for later reference in test cases.

All binaries were build directly on the Raspberry Pi systems using the *gcc* compiler and *cmake*. Following optimization flags have been applied:

```
-O3 -mcpu=cortex-a53 -mfpu=neon-fp-armv8 -mfloat-abi=hard
-funsafe-math-optimizations
```

A detailed description of the compilation process is given in Appendix A.

3.5.3 CPU Cycle Counter

The runtime of certain parts of a program can be evaluated by reading the Cycle Counter Register of the CPU [80]. The register is automatically incremented with each clock cycle. Thus before we run a function (or part of a function) that we want to measure, we read this register and store the value in memory. After the measurement, subtracting the current value of the register from the stored value yields the passed clock cycles, which we can print out on the terminal for later evaluation. Since we know the clock speed of the CPU (1.2 GHz), we can calculate the run time in seconds as

$$t = \frac{n}{f_C} \tag{24}$$

where n is the cycle count and f_C being the clock frequency in GHz. To keep the interference from the operating system as low as possible, the processes of the server and client are given the highest possible priority (-20) using the `nice` command.

Additionally to avoid effects of multiple CPU cores the `taskset` command was utilized to ensure that the whole process always runs on a single CPU core.

3.5.4 Measurement Points

Figure 26 shows at which steps in the secure channel establishment timing measurements were taken according to the methodology explained in Section 3.5.3. Following the measurements are further detailed.

① At first, the client verifies the server certificate, which is either pre-installed or was obtained in the previous getEndpoints step.

② Then the OSCRq is created. This includes ephemeral key pair generation for the PQ KEM.

③ Then the OSCRq is signed.

④ The transmission time of the OSCRq is expected to increase due to the larger included client certificate as well as the longer PQ signature.

⑤ The server first verifies the client's certificate.

⑥ Then the retrieved public key is used to verify the OSCRq's signature.

⑦ The server then creates the OSCRp, which includes the encapsulation process of a shared secret.

⑧ Before transmission the message is signed (either hybrid or conventional, depending on the test setup).

⑨ The OSCRp is transmitted with a higher packet size as in the conventional case due to the additional signature and the encapsulated shared secret.

⑩ The received message signature is verified.

⑪ To eventually open the secure channel, the shared secret has to be extracted by the KEM.

All the measurements were averaged over 100 runs. Therefore a test script was deployed, see Appendix B for further details.

3.5.5 Software Configuration

The set of measurement points described above was measured for 17 combinations of different versions of the executable and different certificates.

1. The *"Baseline"* setup uses the unmodified version of open62541 (EXE_01) with conventional RSA certificates.

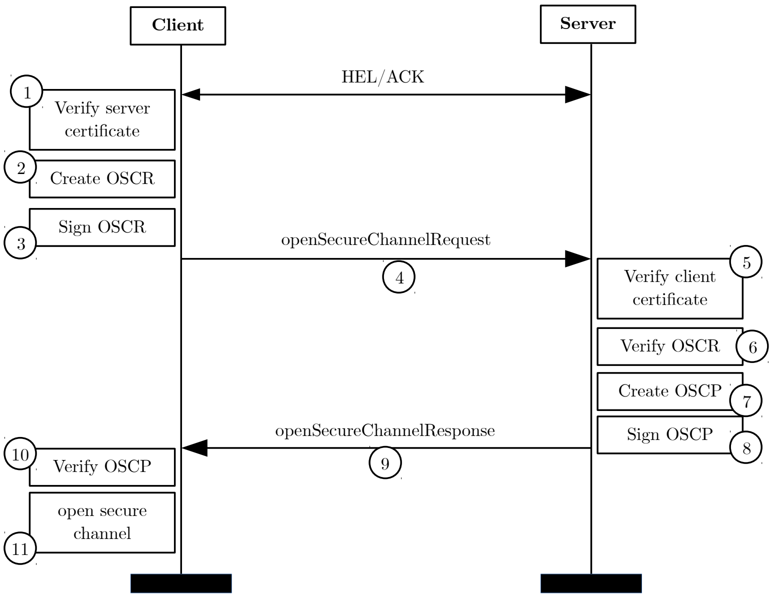


Figure 26: The steps of secure channel establishment. The runtime will be measured at the numbered points.

2. The *"Only KEM"* setup uses executable EXE_02 with the hybrid KEM in place but without any hybrid certificates or signatures.

3. Then five setups follow, where the unmodified open62541 (EXE_01) was used, but all the different versions of hybrid certificates were provided. This resembles a test to see if the hybrid certificates are actually backwards compatible and if they cause any unexpected effects on legacy systems. These setups are called

   - Compatibility Dilithium 2
   - Compatibility Dilithium 3
   - Compatibility Dilithium 4
   - Compatibility Falcon 512
   - Compatibility Falcon 1024

4. The next five setups use the hybrid certificates and 'Variant One' of the quantum resistant authenticated key establishment and their corresponding executables (EXE_03 - EXE_07). They are named:

   - Dilithium 2 – Var 1
   - Dilithium 3 – Var 1
   - Dilithium 4 – Var 1
   - Falcon 512 – Var 1
   - Falcon 1024 – Var 1

5. The last five setups use the 'Variant Two' key establishment also in combination with the five different hybrid certificates:
    - Dilithium 2 – Var 2
    - Dilithium 3 – Var 2
    - Dilithium 4 – Var 2
    - Falcon 512 – Var 2
    - Falcon 1024 – Var 2

In summary we have 17 different test setups and measure 11 measurement points in each setup. Each test result is the average over 100 runs of the test.

4 Results

This sections shows the test results. While one goal of the work was to provide an actual prototypical implementation of hybrid quantum resistant cryptographic schemes in OPC UA, the other part was to show the performance impacts. Therefore the runtime of different elements of the protocol stack is reviewed in detail. Additionally, sizes of certificates and messages are compared.

4.1 CPU Cycles

Firstly it is interesting to compare the total runtime of the key establishment process of the secure channel. The chart in Figure 27 shows the overall runtime for each setup in milliseconds. The first thing to notice is that the 'Only KEM' setup takes just 6.9 % longer than the 'Baseline' setup.

All the 'Compatibility' setups differ less than 0.5 % from the 'Baseline' setup. This proves that, considering CPU cycle time, the hybrid certificates can be used for legacy systems without noticeable negative effects[9].

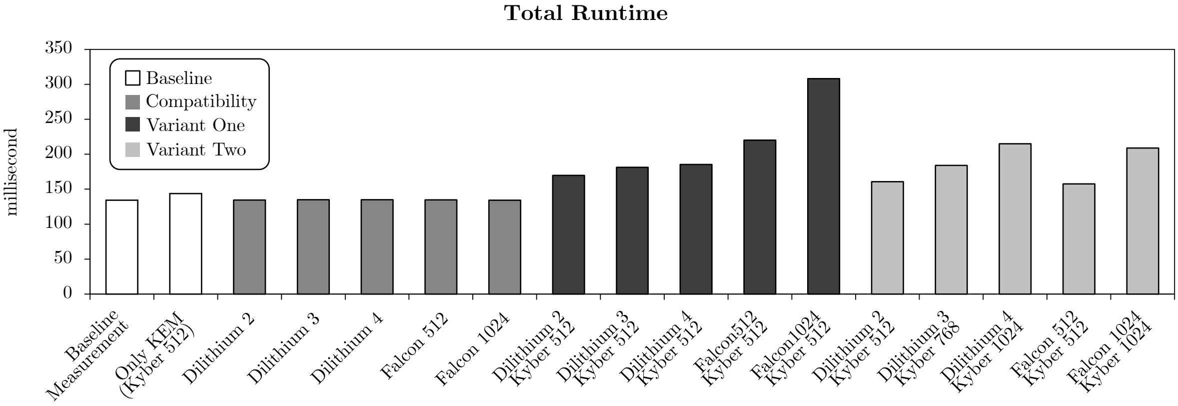


Figure 27: The total time consumed for the key establishment. Only a single certificate directly signed by the CA and no chains were used (chain length 1).

Figure 28 shows the proportion of each measurement point with respect to the total runtime. In the 'Baseline' setup, the majority of the time is consumed for signature generation. 'Variant One' (Dilithium 4 and Falcon 1024 are shown in the chart) shows the same characteristic: The most time is consumed during signature generation. For 'Variant Two', creation of the OSCRq and OSCRp as well as 'Open Secure Channel, Client' steps take up a larger portion of the time. In those steps, the KEM key generation, encapsulation and decapsulation happens. The transmission times of the network packets take up such a small portion of the time (0.028 % in the 'Baseline' setup and 0.046 % in the 'Falcon 1024 - Var 1' setup) that they cannot be shown in the chart.

The full data that was measured can be found in Table 16 in Appendix C and the proportions of all setups are shown in Figure 40 in Appendix D.

---

[9] This is only true regarding the CPU requirements. The size limits for the larger certificates have to be evaluated separately

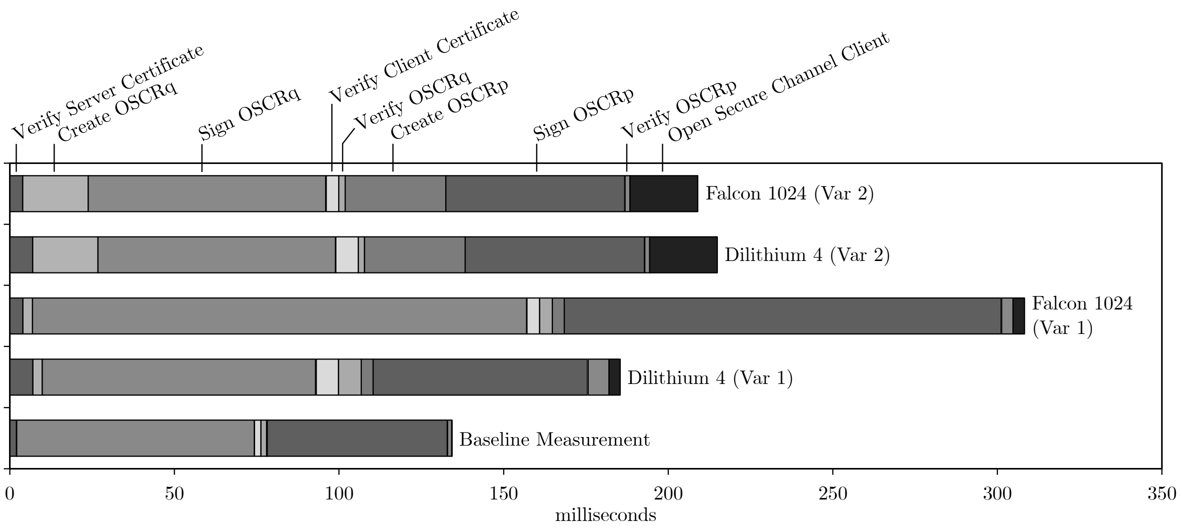


Figure 28: Proportion of the single steps in the key establishment for 5 exemplary setups. Sending of messages takes up such a small percentage that it is not shown in this chart.

4.1.1 Verification of Certificates

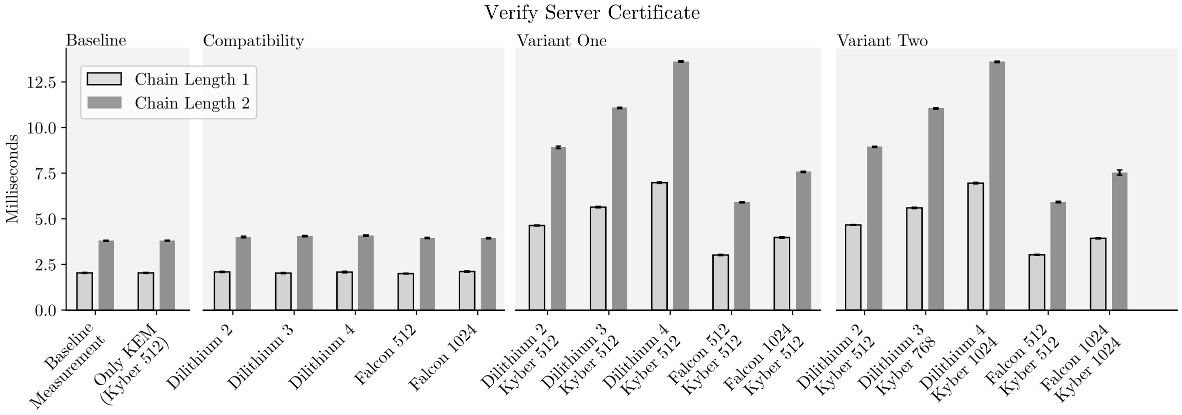


Figure 29: Verification of the server certificate at the client (measurement point ①). Average over 100 measurements.

Figure 29 shows the time that a client needs to verify the server certificate. This corresponds to measurement point ①. The results show the sets of measurements for a certificate that is directly signed by a root CA (Chain length: 1) and with an intermediate CA (chain length: 2), where two certificates were transmitted and verified. As expected, verifying a chain with two certificates takes about twice as long as verifying a chain with only a single certificate. We can observe that Falcon performs here better than Dilithium, i.e. is faster at signature verification, but we also have to consider that the impact of the verification process on the overall time for a key exchange is rather small (3.8 % in case of Dilithium 4 and 'Variant One').

We also observe that the "compatibility" setups have no more than 2.2 % difference from the "Baseline" setup. The only additional computational work in the "compatibility" mode is that the conventional signature signs the inner quantum resistant signature as well, therefore longer certificates have to be verified. For the legacy verifier, the message that has to be verified simply contains more generic data.

The only difference between 'Variant One' and 'Variant Two' regarding this measurement is that the certificates contain different public keys, but are signed with the same hybrid methods. Thus we cannot see any difference in Figure 29 between the variants when the same hybrid signature scheme was used.

At measurement point ⑤ the client certificate is verified by the server. There is no difference in the way the certificate is verified, thus the diagrams look almost the same. It can be seen in Figure 41 in Appendix D.

4.1.2 Creation of Messages

Figure 30 shows the time that the client needs to create a OSCRq, without computing the signature, which corresponds to measurement point ②. At the 'Baseline' and the 'Compatibility' measurements this is done in about 0.05 ms since here only the client nonce has to be generated.

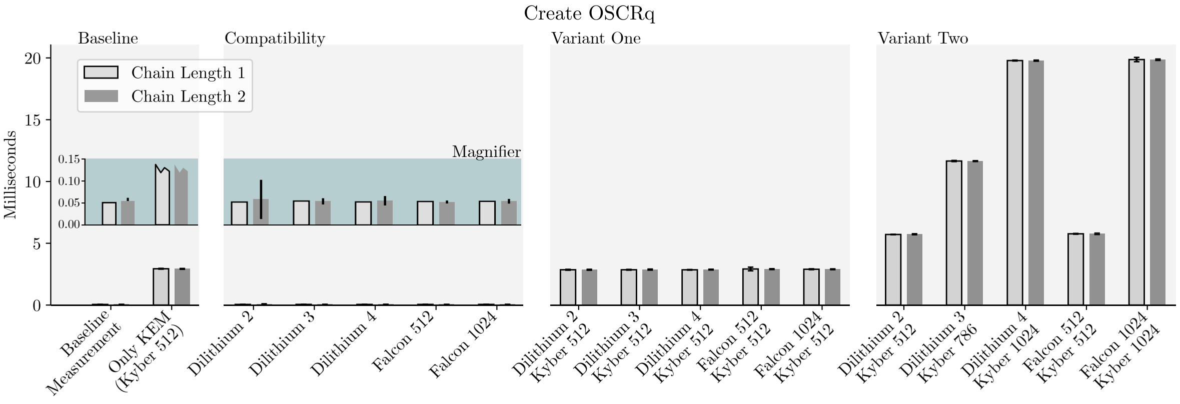


Figure 30: Measurement point ②

In the 'Only KEM' and all of the 'Variant One' setups the ephemeral key pair of Kyber-512 is generated, which increases the runtime to almost 3 ms. In the 'Variant Two' setups, additional to the ephemeral key pair generation, a shared secret is encapsulated. And different parameters for Kyber are used according to Table 15, which increases the runtime as well.

Figure 31 shows the equivalent measurement point at the server, where the OSCRp message is created, also without the signature. All the runtimes are higher than on the client. In 'Variant One' the server has to encapsulate a shared secret instead of creating a key pair. In 'Variant Two' the server has to perform two encapsulations and has to create a MAC.

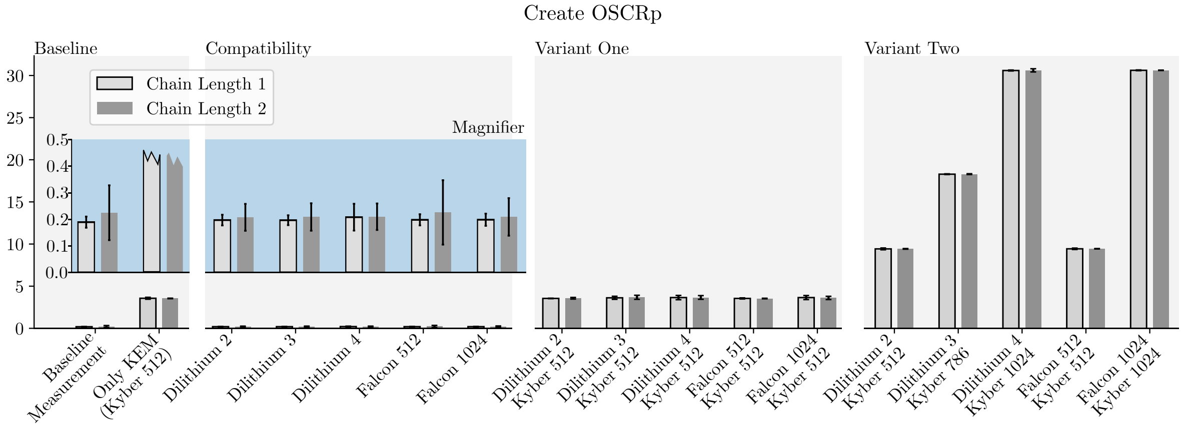


Figure 31: Measurement point ⑦

4.1.3 Signing of Messages

The signing of the OSCRq ③ and of the OSCRp ⑧ is already very CPU intensive when classical RSA is used. The overhead that is introduced by Dilithium is 19.5 % compared to the "Baseline" measurement. Falcon 1024 introduces a significant overhead of 140 % more runtime. Figure 32 shows a chart of the results. As expected, there are only small differences regarding the chain length of the certificates. The only effect of the chain length is that more data has to be signed.

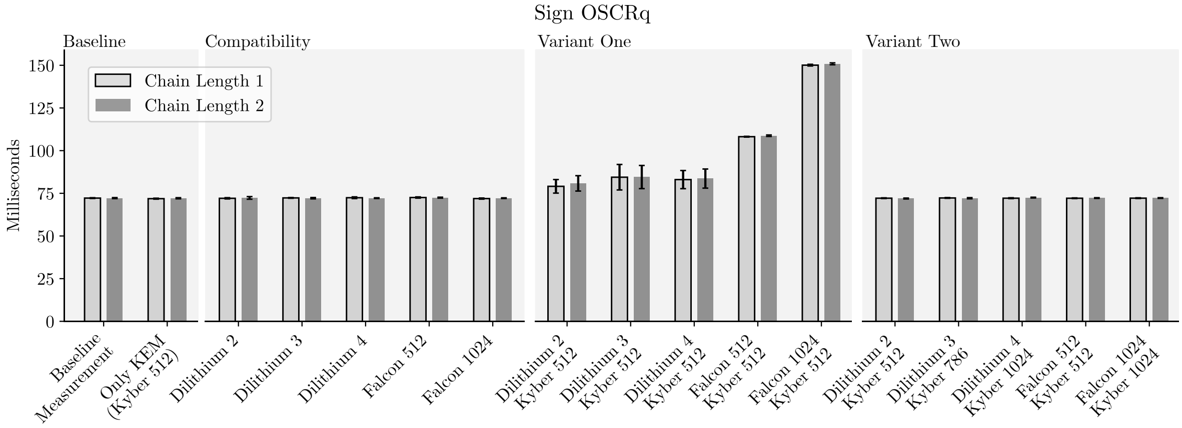


Figure 32: Runtime of the signing of a OSCRq.

Signing with the Dilithium variants has a high standard deviation due to the non deterministic algorithm. This has to be taken into account when it shall be applied for real time applications.

The 'Variant Two' measurements only use the same RSA signatures as the baseline measurement and therefore we don't see any difference.

On the server side at measurement point 8 there are no differences as can be seen in Figure 42 in Appendix D.

4.1.4 Transmission Times

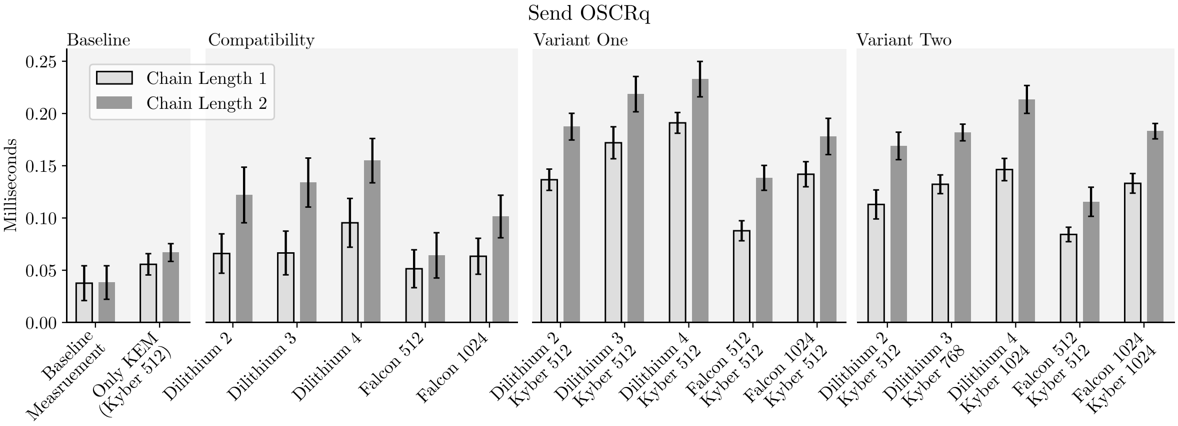


Figure 33: Measurement point ④

The chart in Figure 33 shows the transmission times of the OSCRq for all the setups and chain lengths at measurement point ④. We can see that the 'Compatibility' setups are affected due to the larger certificates that are included in the OSCRq. 'Variant Two' performs slightly better than 'Variant One' because the encapsulated KEM secrets are smaller than the additional quantum resistant signatures of 'Variant One'. In general, the transmission time rises depending on the signature size of the used PQ scheme.

However as it was shown in Figure 27, the overall effect on the key exchange process is small: Between 0.2 % and 0.01 % depending on the used signature scheme. This is still an interesting result if you consider that there might be other systems that have more limited data rates, such as long distance radio links.

The transmission of the OSCRp from the server to the client at measurement point ⑨ (Figure 43 in Appendix D) shows the same characteristics as Figure 33, considering the standard deviation illustrated by the error bars.

4.1.5 Verification of Messages

The verification time for an OSCRq as seen in Figure 34 is only indirectly dependent on the chain length. In either case only one signature (hybrid in the case of 'Variant One') has to be verified, but in case of chain length 2, there is more data in each message to be verified. The verification time for the Dilithium 4 parameter set in 'Variant One' is 301 % longer than for the baseline measurement. In contrast to signature generation, the effect on the overall process is small (83 ms for signing a message compared to 6.8 ms for verification in case of Dilithium 4 and 'Variant One').

'Variant Two' does not use hybrid signatures but still keeps the conventional RSA signatures. Thus the verification times only differ slightly from the 'Baseline' setup. Small variation come from the changed sized of the messages that are signed.

The verification time for the OSCRq is equivalent to the OSCRp, plotted in Figure 44 in Appendix D, since both messages have roughly the same byte size.

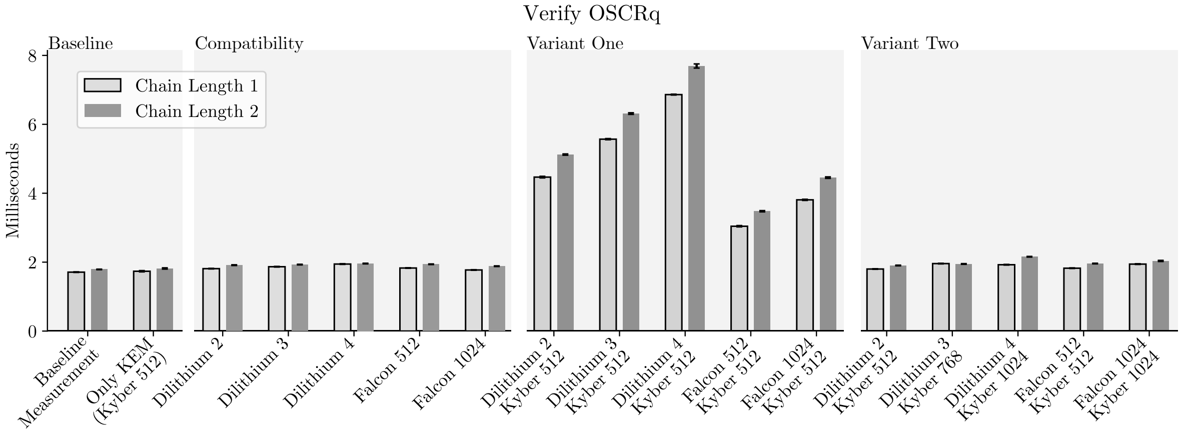


Figure 34: Measurement point ⑥.

4.1.6 Derive the Shared Secret Key at Client

In Figure 35 we see the time it takes for the client to derive the shared secret after the OSCRp arrived (measurement point ⑪). In the 'Baseline' and 'Compatibility' setups this process only involves key derivation using server and client nonce.

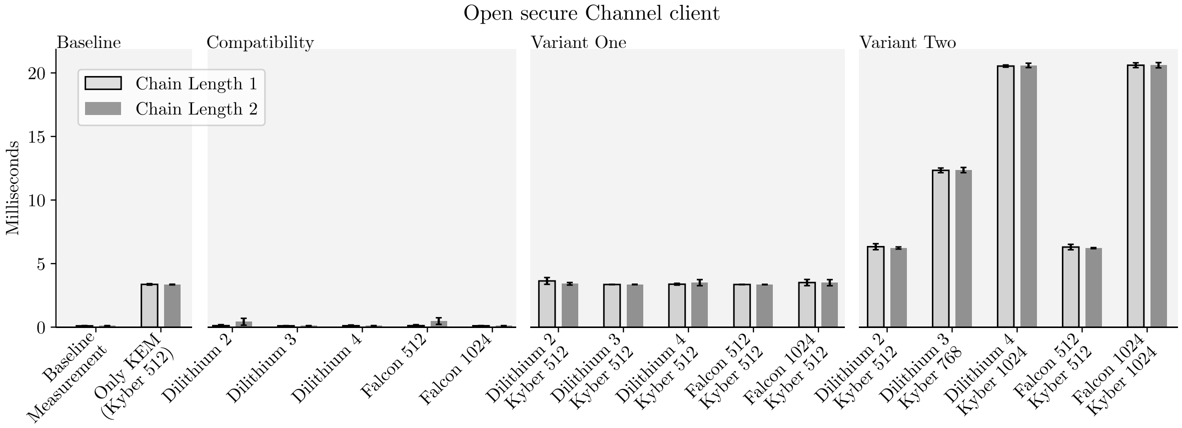


Figure 35: Measurement point ⑪.

In 'Variant One' and the 'Only KEM' setup, Kyber-512 is used and a shared secret has to be decapsulated. Signatures are not verified here and therefore no correlation with the used PQ signature scheme shows.

'Variant Two' involves two decapsulation processes: Decapsulation of the ephemeral shared secret and decapsulation of the long term server shared secret. The 'Dilithium – Var 2' and 'Falcon – Var 2' use Kyber-512 and therefore take about twice the time of 'Variant One'. The other signature schemes in 'Variant Two' are combined with other parameter sets of Kyber, thus the decapsulation process is longer.

4.2 Sizes

4.2.1 Certificate Sizes

Firstly we compare the sizes of hybrid certificates in Figure 36. For comparison, it contains a conventional RSA certificate with 906 bytes. The actual sizes of certificates can vary in practice, depending on the length of the subject and issuer data. However we assume this effect to be in the range of ±100 bytes. For the chart, every certificate has exactly the same issuer and subject information.

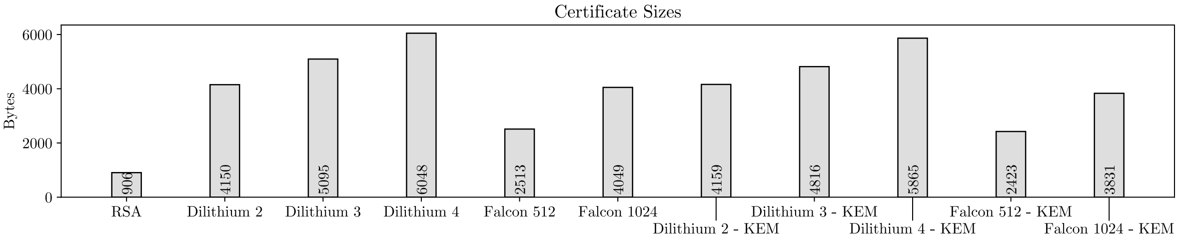


Figure 36: Sizes of hybrid certificates. RSA is a conventional RSA with 2048 bit key, non hybrid, certificate for comparison. The others are the quantum resistant schemes combined with RSA 2048.

The certificates marked with KEM contain a quantum resistant KEM public key instead of the public key of a quantum resistant signature scheme and are used in 'Variant Two'. Because the public keys of Kyber are slightly smaller than the public keys of the used signature schemes, these certificates are a little bit smaller than their counterparts.

Considering that a message chunk in OPC UA allows at least 8 kiB, it seems feasible to include all of the tested certificates. However this is only the size of the certificates itself, the messages will also include protocol overhead and payload data. Reasonable certificate chains can only be realised with Falcon 512. Note that the limit of 8 kiB for a message chunk is not a hard limit, but depends on the lower transport and networking layers used. Thus, in the experiment, we can exceed this limit because we control the implementation that was used. But we have to expect incompatibilities in practice.

4.2.2 Message Sizes

The sizes of exchanged messages in the key establishment are measured with Wireshark. Therefore only the messages GetEndpointResponse, OSCRq and OSCRp, as explained in Section 2.1.2 are considered. The GetEndpointsRequest does not change with the changing cipher suites and thus is not part of the results. As for the runtime measurements in Section 4.1, certificate chains of length 1 and length 2 are measured.

GetEndpointResponse Figure 37 shows the message lengths of the GetEndpointsResponse for a chain length of 1, meaning no intermediate CA was used. The chart compares the sizes between 'Variant One' and 'Variant Two' of the hybrid key exchange. For comparison, classical RSA, as used in the original OPC UA implementation, is shown as well.

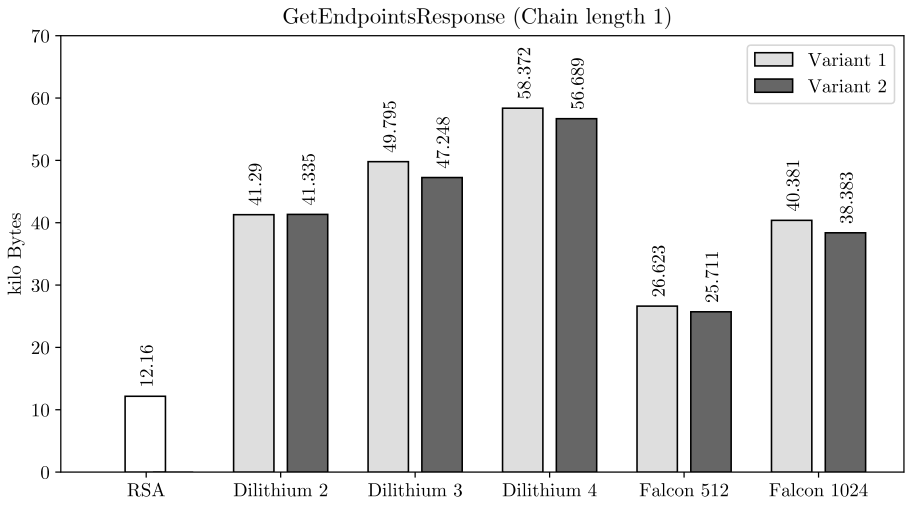


Figure 37: Get endpoints response message sizes. RSA is the standard implementation of OPC UA, all others are hybrid versions in 'Variant One' and 'Variant Two'.

The chart shows very large packet sizes compared to the certificate sizes. In OPC UA every endpoint uses its own certificate. This means that even if every endpoint uses actually the same certificate, a copy is individually included in every endpoint description. The standard configuration of *open62541* creates 7 endpoints, and we add two endpoints for the hybrid security policy, resulting in 9 endpoints. Therefore we expect the message size to have 9 times the certificate size plus some overhead for additional information about the endpoints, which the chart confirms.

Since the certificates of 'Variant Two' are smaller, also the messages become smaller. The biggest difference can be observed for hybrid certificates using Dilithium 3, where the message for 'Variant Two' is 5.4 % smaller compared to 'Variant One'. When two certificates are used in a chain (chain length 2), the message's sizes roughly double, as can be seen in Figure 45 in Appendix D.

These message sizes exceed the chunk size of 8 kiB by far. However, only the OSCRq and OSCRp are limited to one chunk, for the GetEndpointsResponse fragmentation is allowed.

OSCRq and OSCRp The sizes of OSCRq messages, that are sent from client to server, for a chain length of 1 are shown in Figure 38 and for a chain length of 2 are shown in Figure 39. Both charts compare the message sizes of 'Variant One' and 'Variant Two' and include the message size of classical RSA used in standard OPC UA for reference.

Both charts show that when Dilithium was used for the quantum resistant signature scheme, 'Variant Two' results in a smaller message sizes, but when Falcon is used, 'Variant One' produces smaller message sizes. This is due to Falcon's public keys being smaller than the Kyber KEM public keys, while the Dilithium public keys are larger. The horizontal lines in the charts show the 8 kiB chunk size limit. With a chain length of 1, as illustrated in Figure 38, the limit is exceed by Dilithium 4 and also by Dilithium 3 when used in 'Variant One'.

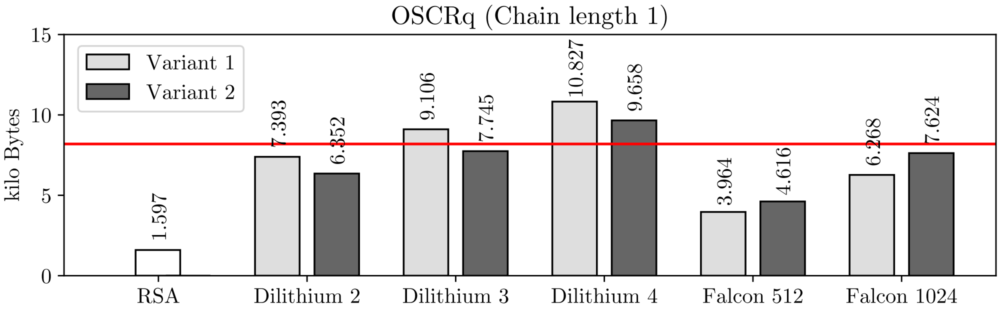


Figure 38: Size of the OSCRq sent by the client with a certificate chain length of 1. The line shows the 8 kiB chunk size limit.

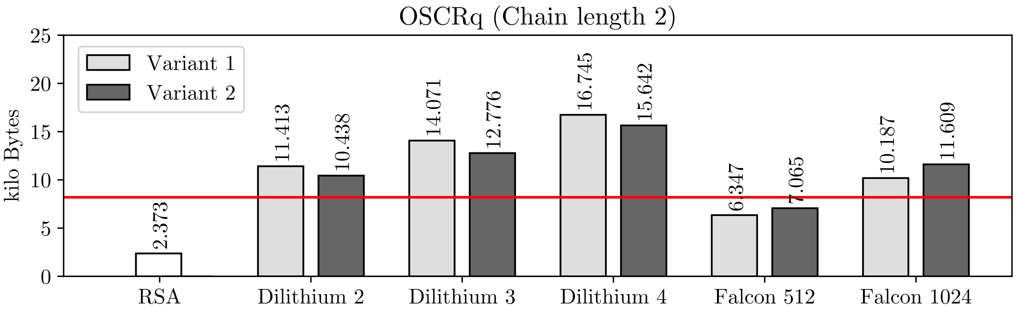


Figure 39: Size of the OSCRq sent by the client with an intermediate certificate included (chain length 2). The line shows the 8 kB chunk size limit.

When using a chain length of 2, i.e. when an intermediate CA certificate is included in the chain, only Falcon 512 can guarantee that the message will fit into one message chunk. To deploy any other scheme or parameter set one either has to find another way to transmit intermediate certificates or might adjust the chunk size limit of OPC UA accordingly.

The OSCRp which is shown in Figure 46 and Figure 47 in Appendix D exhibits the same characteristics as the OSCRq.

5 Budget

The main outcome of the thesis are several software artefacts that are only based on open source project that can be used free of charge. For the measurements that where conducted, two Raspberry Pi 3 micro computers were used.

Costs:

Item	**Quantity**	**Cost**
Raspberry Pi 3	2	50 EUR
Power Supply	2	10 EUR
Cat5e Cable 2m	2	7 EUR
Total		**134 EUR**

6 Environment Impact

The cryptographic schemes that were shown affect the computation time of the involved microprocessors. This means that their power consumption will most likely increase when we transition to hybrid quantum resistant schemes is made. To give a reasonable estimate on the effect of different cryptographic schemes on the power consumption, one needs a dedicated study that surveys the overall number of deployed cryptographic devices in the field and from there could try to estimate the increased power consumption due to quantum resistant cryptography.

7 Conclusion and Outlook

All 9 quantum resistant signature schemes that are remaining in round two of the NIST PQ project were evaluated regarding their suitability for OPC UA. Falcon and Dilithium are found to be the most suitable candidates due to their small public key and signature sizes. Here it turns out that transmission time of large data packets is, at least in the case of Ethernet, no issue, however limitations of message sizes in the protocols are the most restricting factor. This is also a hint for future protocol standardizers that it might be worth to allow more overhead data in order to be more flexible in the choice of quantum resistant cryptographic algorithms.

Furthermore this thesis shows how backwards compatible hybrid quantum resistant X.509 certificates can be created and a prototype in Python was implemented that actually proofs their feasibility. While this is indispensable for the transition phase towards quantum resistant cryptography it seems appropriate to design a new version of X.509 that actually allows easy use of multiple public keys and signatures and renounce backwards compatibility.

Based on these certificates, two methods for a hybrid quantum resistant and authenticated key exchange that withstand a Q^cQ attacker are proposed. While the 'Variant Two' promises minor performance improvements over 'Variant One', the 'Variant One' has stronger security arguments and is closer oriented on the existing key establishment method of OPC UA. Both variants are integrated into the open source OPC UA protocol stack *open62541* and proof practical feasibility and allow to estimate performance impacts beyond theoretical calculations.

Regarding size constraints, we demonstrate that Falcon is the most suitable signature scheme to be deployed in OPC UA due to its small signature and public key size, whereas Dilithium still has acceptable public key and signature sizes and performs better in terms of CPU usage, which becomes more important on resource constrained processors. However, one has to consider that in the future the computational cost of both schemes could decrease due to optimization in the implementations as well as due to specialized cryptographic hardware. But the sizes of signatures and public keys can be expected to be fixed at least when a scheme becomes standardized. For a future work it would be interesting to investigate the same issue from a viewpoint where all size constraints a dropped and only the computing performance is considered.

Finally, with the performance measurements we are able to provide insights to vendors of IoT and embedded systems to estimate if their systems are capable of running hybrid quantum resistant schemes and if not what changes are required.

It is worth noting that all the considered signature schemes are still under evaluation by the NIST and it remains to be seen which ones will become actual standards. This thesis suggest that in case neither Falcon nor Dilithium become part of the standard, some major changes to the OPC UA protocol become necessary to remain secure in a post quantum world.

Bibliography

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer", *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[2] E. P. Leverett, "Quantitatively assessing and visualising industrial system attack surfaces", *University of Cambridge, Darwin College*, vol. 7, 2011.

[3] A. Gilchrist, *Industry 4.0: The industrial internet of things.* Apress, 2016.

[4] K. Stouffer, J. Falco, and K. Scarfone, "Guide to industrial control systems (ICS) security", *NIST special publication*, vol. 800, no. 82, p. 16, 2011.

[5] K. Poulsen, "Slammer worm crashed Ohio nuke plant net", *The Register*, vol. 20, 2003.

[6] D. Hentunen and A. Tikkanen, *Havex Hunts For ICS/SCADA Systems*, 2014. [Online]. Available: https://www.f-secure.com/weblog/archives/00002718.html.

[7] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon", *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[8] M. Krotofil and D. Gollmann, "Industrial control systems security: What is happening?", in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, 2013, pp. 670–675.

[9] BMWi, Bundesministerium für Wirtschaft und Energie, *IT-Sicherheit für Industrie 4.0.* Official Study of the German Federal Ministry for Economic Affairs and Energy, 2016. [Online]. Available: https://www.bmwi.de/Redaktion/DE/Publikationen/Studien/it-sicherheit-fuer-industrie-4-0.pdf?__blob=publicationFile&v=4.

[10] DIN IEC 62541-2, *OPC UA Part 2 - Services Release 1.04 Specification*, 2017.

[11] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC Unified Architecture.* Berlin: Springer, 2009, ISBN: 978-3-540-68899-0.

[12] OPC Foundation, *Part 1 - Overview and Concepts*, 22.11.2017.

[13] BSI, *OPC UA Security Analysis: 2017-04-24*, Bonn, 2017. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/OPCUA/OPCUA.html.

[14] M. Puys, M.-L. Potet, and P. Lafourcade, "Formal Analysis of Security Properties on the OPC-UA SCADA Protocol", in *Computer Safety, Reliability, and Security*, A. Skavhaug, J. Guiochet, and F. Bitsch, Eds., vol. 9922, Cham: Springer International Publishing, 2016, pp. 67–75.

[15] F. Palm, S. Grüner, J. Pfrommer, M. Graube, and L. Urbas, "Open source as enabler for OPC UA in industrial automation", in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2015, pp. 1–6.

[16] E. Crockett, C. Paquin, and D. Stebila, *Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH*, 2019.

[17] D. Stebila and M. Mosca, “Post-quantum key exchange for the internet and the open quantum safe project”, in *International Conference on Selected Areas in Cryptography*, 2016, pp. 14–37.

[18] J. A. Smolin, G. Smith, and A. Vargo, “Oversimplifying quantum factoring”, *Nature*, vol. 499, no. 7457, p. 163, 2013.

[19] C. Gidney and M. Ekerå, “How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits”, *arXiv preprint arXiv:1905.09749*, 2019.

[20] A. Fruchtman and I. Choi, “Technical roadmap for fault-tolerant quantum computing”, *NQIT Technical Roadmap*, 2016.

[21] F. Arute, K. Arya, R. Babbush, *et al.*, “Quantum supremacy using a programmable superconducting processor”, *Nature*, vol. 574, no. 7779, pp. 505–510, 2019. DOI: 10.1038/s41586-019-1666-5.

[22] M. Mosca, “Cybersecurity in an era with quantum computers: Will we be ready?”, *IEEE Security & Privacy*, vol. 16, no. 5, pp. 38–41, 2018.

[23] DIN IEC 62541-6, *OPC UA Part 6 - Mappings*, 2017.

[24] M. Cheminod, L. Durante, and A. Valenzano, “Review of security issues in industrial networks”, *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 277–293, 2012.

[25] D. Cooper, S. Santesson, S. Farrell, *et al.*, “RFC 5280: Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile”, *IETF, May*, 2008.

[26] I. Rec, “X. 680”, in *Abstract Syntax Notation One (ASN. 1)-Specification of Basic Notation*, 1994.

[27] ——, “X. 690”, *Specification of ASN*, vol. 1, 1994.

[28] S. Hausmann and S. Heiss, “Usage of public key infrastructures in automation networks”, in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, 2012, pp. 1–4.

[29] A. Lioy, M. Marian, N. Moltchanova, and M. Pala, “PKI past, present and future”, *International Journal of Information Security*, vol. 5, no. 1, pp. 18–29, 2006.

[30] A. Fernbach and W. Kastner, “Certificate management in OPC UA applications: An evaluation of different trust models”, in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, 2012, pp. 1–6.

[31] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, “Recommendation for key management part 1: General (revision 3)”, *NIST special publication*, vol. 800, no. 57, pp. 1–147, 2012.

[32] R. L. Rivest and B. Kaliski, “RSA problem”, *Encyclopedia of Cryptography and Security*, pp. 1065–1069, 2011.

[33] B. den Boer, “Diffie-Hellman is as strong as discrete log for certain primes”, in *Proceedings on Advances in cryptology*, 1990, pp. 530–539.

[34] NIST, Ed., *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*, 2016. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf (visited on 05/02/2020).

[35] B. Schneier, *Applied cryptography: Protocols, algorithms, and source code in C*. john wiley & sons, 2007.

[36] L. K. Grover, "A fast quantum mechanical algorithm for database search", in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[37] D. J. Bernstein, J. Buchmann, and E. Dahmen, *Post-quantum cryptography*. Berlin Heidelberg: Springer-Verlag, 2009.

[38] *Quantum-Safe Cryptography (QSC)*. [Online]. Available: https://www.etsi.org/technologies/quantum-safe-cryptography (visited on 05/03/2020).

[39] P. E. Hoffman, "The transition from classical to post-quantum cryptography", *draft-hoffman-c2pq-05 (work in progress)*, 2019.

[40] M. Ounsworth and M. Pala, *Composite Keys and Signatures For Use In Internet PKI*, 2019.

[41] *Post-Quantum Cryptography*. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography (visited on 05/03/2020).

[42] Sebastian Paul and Esther Guerin, *Hybrid OPC UA: Enabling Post-Quantum Security for the Industrial Internet of Things*, Bosch Internal Report, 2019.

[43] R. J. McEliece, "A public-key cryptosystem based on algebraic", *Coding Thv*, vol. 4244, pp. 114–116, 1978.

[44] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the inherent intractability of certain coding problems (corresp.)", *IEEE Transactions on Information Theory*, vol. 24, no. 3, pp. 384–386, 1978.

[45] G. O'Regan, "Coding Theory", in *Mathematics in Computing*, Springer, 2013, pp. 155–169.

[46] E. Berlekamp, "Goppa codes", *IEEE Transactions on Information Theory*, vol. 19, no. 5, pp. 590–592, 1973.

[47] L. Lamport, *Constructing digital signatures from a one-way function*, 1979.

[48] R. C. Merkle, "A certified digital signature", in *Conference on the Theory and Application of Cryptology*, 1989, pp. 218–238.

[49] O. Goldreich, *The fundamental of cryptography: Basic applications*, 2004.

[50] D. J. Bernstein, A. Hülsing, S. Kölbl, *et al.*, "The SPHINCS+ signature framework", in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2129–2146.

[51] Gerard Tel, *Cryptography in Context*, 2008. [Online]. Available: https://www.staff.science.uu.nl/~tel00101/liter/Books/CrypCont.pdf (visited on 01/03/2020).

[52] A. Casanova, J.-C. Faugère, G. Macario-Rat, *et al.*, "Gemss: A great multivariate short signature", *Submission to NIST*, 2017.

[53] W. Beullens, A. Szepieniec, F. Vercauteren, and B. Preneel, "LUOV: Signature scheme proposal for NIST PQC project", 2017.

[54] J. Ding, Z. Zhang, J. Deaton, K. Schmidt, and F. Vishakha, "New attacks on lifted unbalanced oil vinegar", in *The 2nd NIST PQC Standardization Conference*, 2019.

[55] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe, *MQDSS Specification*, 2019. [Online]. Available: http://mqdss.org/files/MQDSS_Ver2.pdf (visited on 02/03/2020).

[56] D. Kales and G. Zaverucha, "Forgery Attacks on MQDSSv2. 0", 2019.

[57] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang, *Rainbow*, 2019. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/Rainbow-Round2.zip (visited on 02/03/2020).

[58] E. Alkim, P. S. Barreto, N. Bindel, P. Longa, and J. E. Ricardini, "The Lattice-Based Digital Signature Scheme qTESLA", *IACR Cryptology ePrint Archive*, vol. 2019, p. 85, 2019.

[59] L. Ducas, T. Lepoint, V. Lyubashevsky, *et al.*, "Crystals–dilithium: Digital signatures from module lattices", 2018.

[60] Panos Kampanakis, *Classical Sec Levels for Falcon and Dilithium*, 2019. [Online]. Available: https://groups.google.com/a/list.nist.gov/forum/#!searchin/pqc-forum/dilithium$20security%7Csort:date/pqc-forum/BG6lcVGe_90/4Ihx-HaTAwAJ (visited on 02/03/2020).

[61] D. Stebila and M. Mosca, "Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project", in *Selected Areas in Cryptography - SAC 2016*, R. Avanzi and H. Heys, Eds., Cham: Springer, 2017, pp. 14–37, ISBN: 9783319694535. DOI: 10.1007/978-3-319-69453-5{\textunderscore}2.

[62] *PQClean*. [Online]. Available: https://github.com/PQClean/PQClean (visited on 02/03/2020).

[63] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen, *PQM4*. [Online]. Available: https://github.com/mupq/pqm4 (visited on 02/03/2020).

[64] D. Boneh and M. Zhandry, "Secure signatures and chosen ciphertext security in a quantum computing world", in *Annual Cryptology Conference*, 2013, pp. 361–379.

[65] N. Bindel, U. Herath, M. McKague, and D. Stebila, "Transitioning to a Quantum-Resistant Public Key Infrastructure", in *Post-Quantum Cryptography*, T. Lange and T. Takagi, Eds., Cham: Springer International Publishing, 2017, pp. 384–405, ISBN: 978-3-319-59879-6.

[66] N. Bindel, J. Brendel, M. Fischlin, B. Concalves, and D. Stebila, "Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange", *Cryptology ePrint Archive*, vol. 2018/903, no. Report 2018/903, 2018. [Online]. Available: https://eprint.iacr.org/2018/903.

[67] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the Fujisaki-Okamoto transformation", in *Theory of Cryptography Conference*, 2017, pp. 341–371.

[68] F. Giacon, F. Heuer, and B. Poettering, "KEM combiners", in *IACR International Workshop on Public Key Cryptography*, 2018, pp. 190–218.

[69] M. Bellare and P. Rogaway, "Entity authentication and key distribution", in *Annual international cryptology conference*, 1993, pp. 232–249.

[70] C. Brzuska, "On the foundations of key exchange", PhD thesis, Technische Universität, 2013.

[71] J. Brendel, M. Fischlin, and F. Günther, "Breakdown Resilience of Key Exchange Protocols and the Cases of NewHope and TLS 1.3", *IACR Cryptology ePrint Archive*, vol. 2017, p. 1252, 2017.

[72] N. Bindel, J. Braun, L. Gladiator, T. Stöckert, and J. Wirth, "X.509-Compliant Hybrid Certificates for the Post-Quantum Transition", *Journal of Open Source Software*, vol. 4, no. 40, p. 1606, 2019, ISSN: 2475-9066. DOI: 10.21105/joss.01606.

[73] Luca Gladiator, *Hybrid Certificates in OpenSSL*, GitHub, 2019. [Online]. Available: https://github.com/CROSSINGTUD/openssl-hybrid-certificates/blob/OQS-OpenSSL_1_1_1-stable/HybridCert_technical_documentation.pdf.

[74] J. Larmouth, *ASN. 1 complete*. Morgan Kaufmann, 2000.

[75] S. Cavalieri and F. Chiacchio, "Analysis of OPC UA performances", *Computer Standards & Interfaces*, vol. 36, no. 1, pp. 165–177, 2013, ISSN: 09205489.

[76] W. Diffie, P. C. van Oorschot, and M. J. Wiener, "Authentication and authenticated key exchanges", *Designs, Codes and Cryptography*, vol. 2, no. 2, pp. 107–125, 1992, ISSN: 0925-1022. DOI: 10.1007/BF00124891.

[77] J. Bos, L. Ducas, E. Kiltz, *et al.*, "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM", in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2018, pp. 353–367, ISBN: 978-1-5386-4228-3. DOI: 10.1109/EuroSP.2018.00032.

[78] D. Sikeridis, P. Kampanakis, and M. Devetsikiotis, *Post-quantum authentication in TLS 1.3: A performance study*, 2020.

[79] *mbed TLS*. [Online]. Available: https://tls.mbed.org/ (visited on 01/03/2020).

[80] matthew, *Using the Cycle Counter Registers on the Raspberry Pi 3*, 2017. [Online]. Available: https://matthewarcus.wordpress.com/2018/01/27/using-the-cycle-counter-registers-on-the-raspberry-pi-3/ (visited on 06/02/2020).

Abbreviations

AES	Advanced Encryption Standard
ASN.1	Abstract Syntax Notation One
CA	Certificate Authority
CCA	chosen-ciphertext-attack
CPA	chosen-plaintext-attack
CPS	Cyber Physical System
CPU	Central Processing Unit
DER	Distinguished Encoding Rules
DH	Diffie-Hellmann
dPRF	dual PRF
ERP	Enterprise Resource Planning System
ETSI	European Telecommunications Standards Institute
EUF-CMA	Existential Unforgeability under Chosen Message Attack
GUI	Graphical User Interface
HFE	Hidden Field Equations
HMAC	Keyed Hash Message Authentication Code
HMI	Human Machine Interface
HTTP	Hyper Text Transfer Protocol
ICS	Industrial Control System
IETF	Internet Engineering Task Force
IIoT	Industrial Internet of Things
ISO	International Organization for Standardization
ITU	International Telecommunications Union
KDF	Key Derivation Function
KEM	Key Encapsulation Method
M2M	Machine To Machine

MAC	Message Authentication Code
MES	Manufacturing Execution System
NIST	National Institute of Standards and Technology
OID	Object Identifier
OID	Object Identifier
OPC UA	OPC Unified Automation
OPC	Open Platform Communications
OQS	Open Quantum Safe
OSCRp	Open Secure Channel Response
OSCRq	Open Secure Channel Request
PC	Personal Computer
PGP	Pretty Good Privacy
PKI	Public Key Infrastructure
PLC	Programmable Logic Controller
PQ	Post Quantum
PRF	Pseudo Random Function
RA	Registration Authority
RSA	Rivest Shamir Adelman
SCADA	Supervisory Control And Data Acquisition
SHA	Secure Hash Algorithm
TLS	Transport Layer Security
XML	Extensible Markup Language

A Compilation of open62541

Open62541 comes with a cmake file that builds the OPC UA stack as a static library. Two new software projects (server, client) were created for this thesis that also use cmake and include the open62541 cmake project. This means when the server or client is build, the open62541 library is automatically build as well and all the binaries are links.

The listing shows the CMakeLists.txt (cmake configuration file) that was used for the client, the server uses an identical file just with a different project name.

```
cmake_minimum_required(VERSION 2.8)

set(CMAKE_CXX_FLAGS_RELEASE "-O3 -mcpu=cortex-a53 -mfpu=neon-fp-armv8 -mfloat
    ↪ -abi=hard -funsafe-math-optimizations")

add_subdirectory("../hybrid_crypto_test/hybrid_lib/" "build_hybrid_lib/")
add_subdirectory("../open62541" "build_open62541/")

project(opc_ua_client)
add_executable(${PROJECT_NAME} "main.c" "common.h")

INCLUDE_DIRECTORIES("../open62541/include/" "../open62541/deps" "../
    ↪ hybrid_crypto_test/hybrid_lib/")

link_directories( "build_open62541/bin/" "build_hybrid_lib/")
target_link_libraries(${PROJECT_NAME} mbedcrypto mbedtls mbedx509 open62541
    ↪ hybrid_crypto)

set_property(TARGET ${PROJECT_NAME} PROPERTY C_STANDARD 99)
```

Following CMake flags were added to the open62541 CMakeLists.txt in order to be able to build different versions without changing the source code:

- `HYBRID_CERTIFICATE_VERIFICATION (ON/OFF)`
- `HYBRID_KEXV1_DILITHIUM_2 (ON/OFF)`
- `HYBRID_KEXV1_DILITHIUM_3 (ON/OFF)`
- `HYBRID_KEXV1_DILITHIUM_4 (ON/OFF)`
- `HYBRID_KEXV1_FALCON_512 (ON/OFF)`
- `HYBRID_KEXV1_FALCON_1024 (ON/OFF)`
- `HYBRID_KEM_OPEN (ON/OFF)`

B Measurement Script

To automate the test runs where a server is started and a client connects, two bash scripts were used.

For a test, the proper server executable has to be run on the server Raspberry Pi and the proper certificate files have to be copied to the server folder. Then on the client Raspberry Pi, the proper client executable has to be launched. It will connect to the server, output relevant measurement data and terminate. The server also outputs measurement data, but will not automatically terminate but will wait for a new connection. However for the next test case probably a different server executable is required and therefore the server has to be stopped and a new server has to be launch.

To automate this process, server and client each have a bash script. The server's script launches the server in the background and pipes the output data into a file for later evaluation. Then a netcat server is started that pauses the script until a connection to the netcat server is established and again terminated. The script then proceeds to terminate the server via the kill command and launch the next server.

The client script launches the client and redirects the output to a file as well. Once the client process terminates, a netcat command is sent to the server script. Then the client script waits a second to give the server enough time to kill and start the new server process and the proceeds to connect with the next client executable.

Part of the server measurement script

```
# --- TEST CASE ------------------------------------------
test_case_num=001

# Copy the proper certificates to the binary folder
sh /home/pi/code/certificates/classical/install_certs.sh root_signed

# Run the server, the script sets the parameters
sh run_server.sh 1 > $test_results/test_case_$test_case_num.txt &



echo test case $test_case_num started...
nc -l -p 4444

kill -9 $(pgrep opc_ua_server)
```

Part of the client measurement script

```
# --- TEST CASE ------------------------------------------
test_case_num=001

# Copy the proper certificates to the binary folder
```

```
sh /home/pi/code/certificates/classical/install_certs.sh root_signed

# Run the server, the script sets the parameters
sh run_client.sh 1 Basic256Sha256 > $test_results/
    ↪ test_case_$test_case_num.txt



echo
echo --- test $test_case_num done ----------------
```

C Measurement Result Data

Table 16: Measured data with one certificate (no chain). Average over 100 measurements. All values are in milliseconds.

	Verify Server Certificate	Create OSCRq	Sign OSCRq	Send OSCRq	Verify Client Certificate	Verify OSCRq	Create OSCRp	Sign OSCRp	send OSCRp	Verify OSCRp	Open secure channel, client
Baseline Measurement	2,034	0,051	72,201	0,038	1,948	1,710	0,189	54,716	0,070	1,300	0,104
Only KEM	2,034	2,938	71,867	0,056	1,950	1,736	3,566	54,764	0,067	1,328	3,364
Compatiblility Dilitihium 2	2,088	0,052	72,078	0,066	1,985	1,812	0,197	54,645	0,136	1,374	0,119
Compatiblility Dilitihium 3	2,025	0,055	72,313	0,067	1,910	1,867	0,197	54,980	0,139	1,394	0,107
Compatibility Dilithium 4	2,080	0,053	72,414	0,095	1,969	1,945	0,208	54,426	0,137	1,416	0,113
Compatibility Falcon 512	1,996	0,054	72,544	0,051	1,900	1,830	0,198	54,552	0,099	1,338	0,119
Compatibility Falcon 1024	2,109	0,054	71,915	0,063	1,992	1,773	0,198	54,617	0,136	1,371	0,109
Dilithium 2 - Var 1	4,631	2,856	79,089	0,137	4,483	4,468	3,547	62,784	0,147	4,046	3,635
Dilithium 3 - Var 1	5,640	2,856	84,438	0,172	5,479	5,569	3,621	64,881	0,230	5,100	3,355
Dilithium 4 - Var 1	6,981	2,852	83,037	0,191	6,791	6,863	3,650	65,084	0,249	6,350	3,379
Falcon 512 - Var 1	3,015	2,920	108,165	0,088	2,938	3,043	3,548	90,548	0,097	2,553	3,355
Falcon 1024 - Var 1	3,975	2,899	150,088	0,142	3,864	3,809	3,649	132,716	0,149	3,423	3,508
Dilithium2 - Var 2	4,664	5,712	72,146	0,113	4,507	1,801	9,425	54,468	0,118	1,410	6,328
Dilithium3 - Var 2	5,597	11,657	72,284	0,132	5,481	1,958	18,289	54,666	0,150	1,441	12,335
Dilithium4 - Var 2	6,953	19,786	72,144	0,146	6,803	1,925	30,592	54,395	0,165	1,486	20,539
Falcon512 - Var 2	3,027	5,766	72,120	0,084	2,928	1,825	9,435	54,621	0,095	1,373	6,300
Falcon1024 - Var 2	3,928	19,868	72,163	0,133	3,819	1,944	30,614	54,302	0,148	1,442	20,607

D Measurement Charts

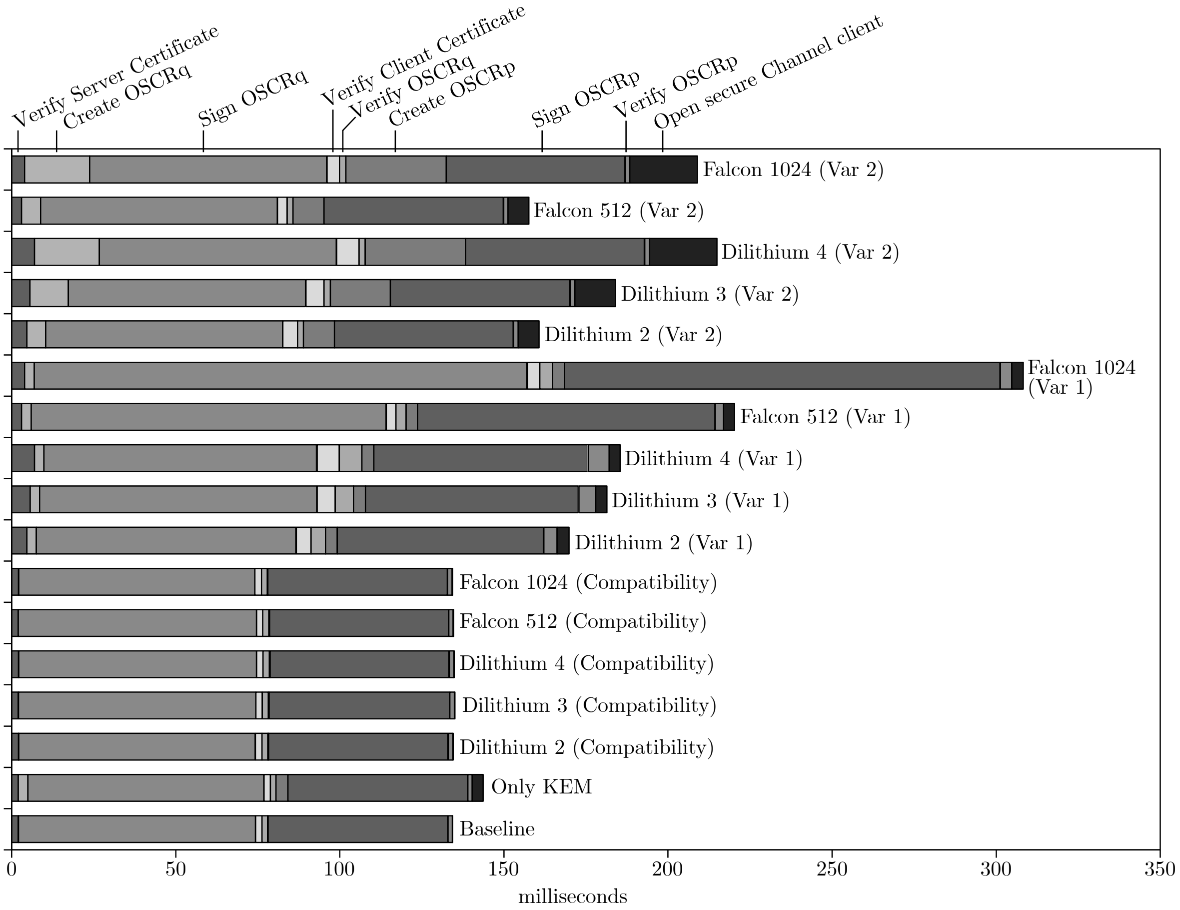


Figure 40: Proportion of steps during a key establishment process for all setups with a certificate chain length of 1.

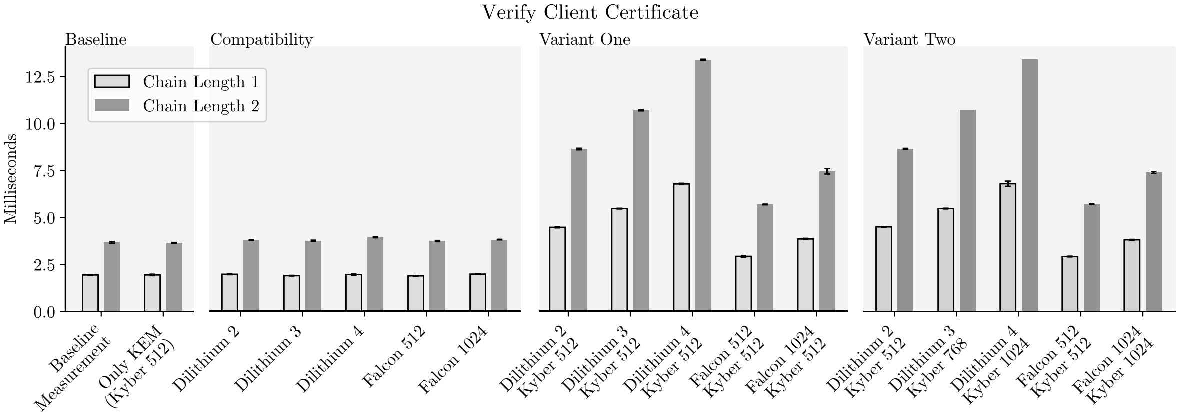


Figure 41: Verification of the client certificate at the server (measurement point ⑤)

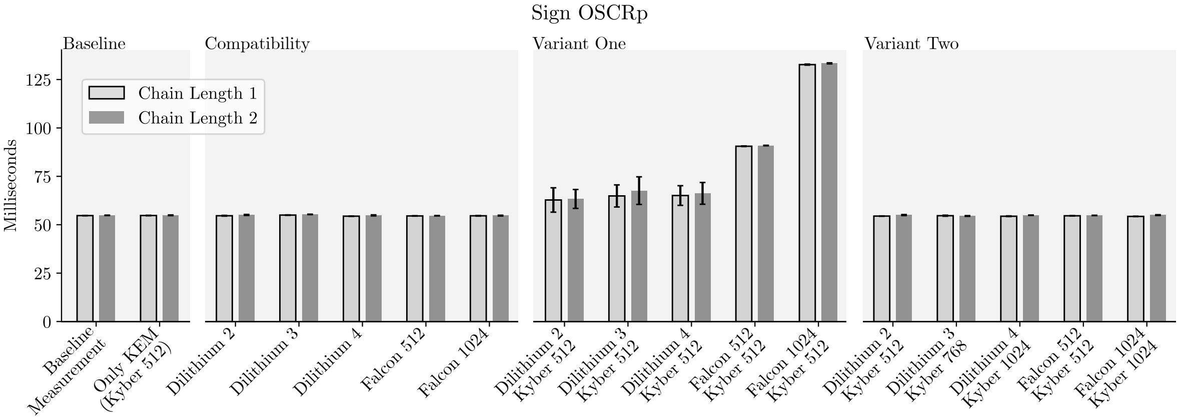


Figure 42: Signing of the OSCRp by the server (measurement point ⑧).

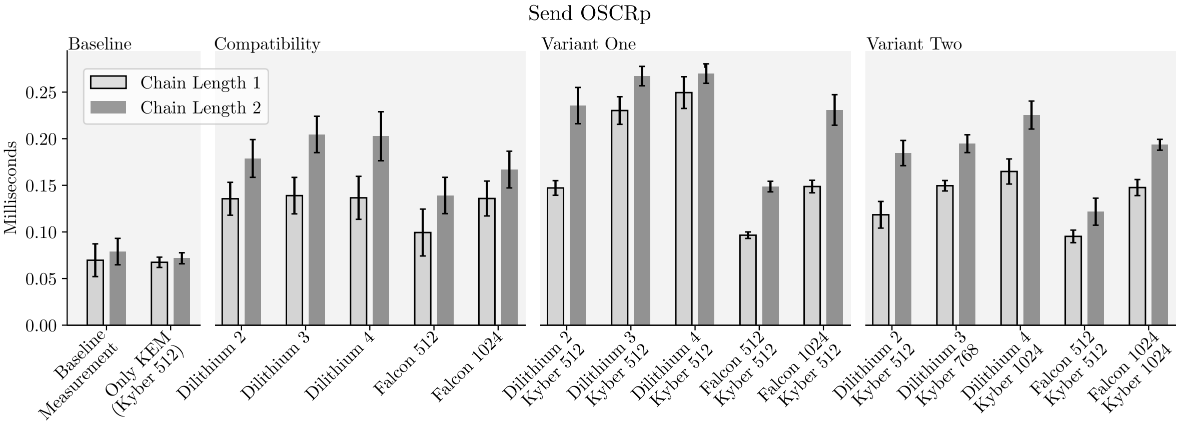


Figure 43: Transmission time of the OSCRp from server to client (measurement point ⑨).

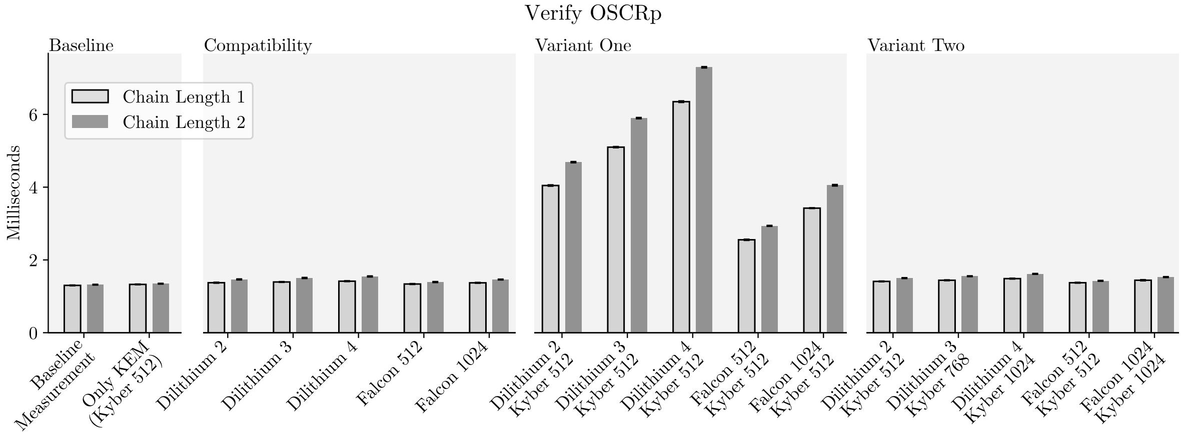


Figure 44: Verification of the OSCRp at the client (measurement point ⑩).

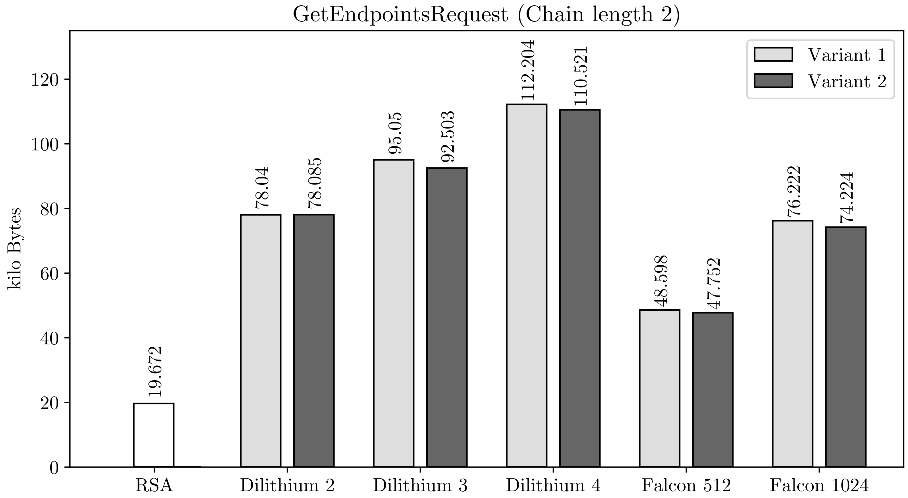


Figure 45: Get endpoints response, measured with Wireshark with a chain of two certificates (device and CA certificate).

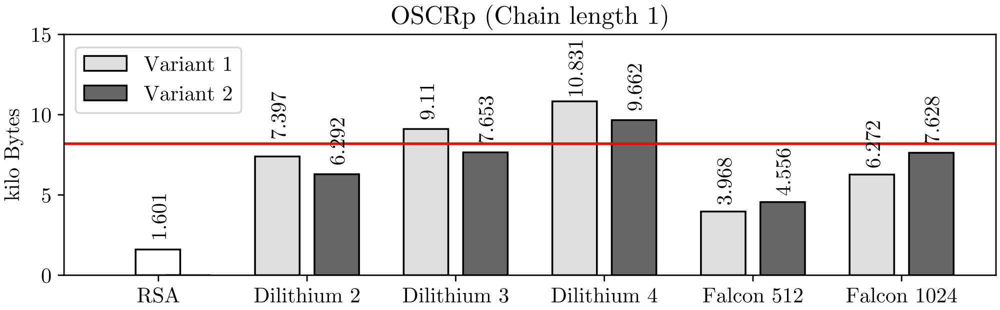


Figure 46: Data that are transmitted during the transmission of a OSCRp with a single certificate in the chain.

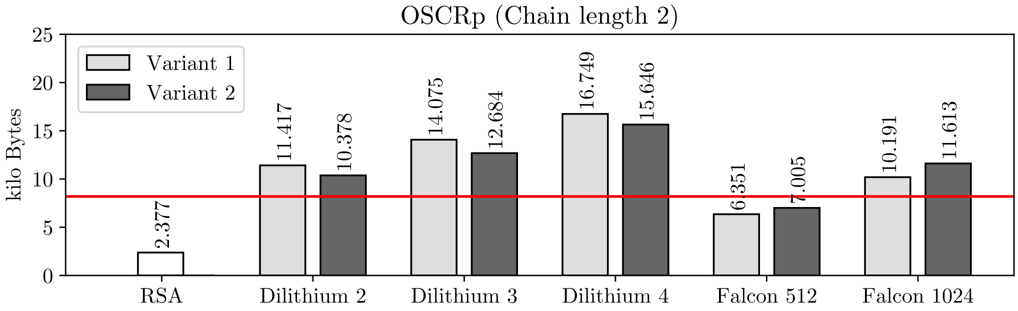


Figure 47: Size of a OSCRp message for different setups with one intermediate certificate in the chain.

E Implementation Details Variant One

Setting up the Project

Freitag, 24. Januar 2020 15:01

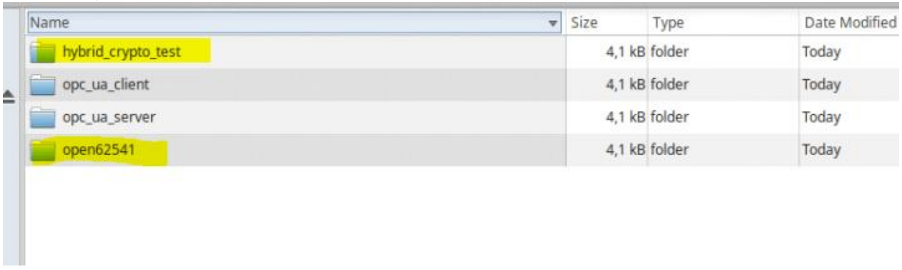
The goal of this documentation is to show the process of changing open62541 to support PQ X.509 compatible certificates and sign asymmetrically encrypted messages using the keys in these certificates.

We need:

- The hybrid certificate files. How to generate them is explained in a different document
- The hybrid_lib project files. They are explained in a different documentation
- A copy of open62541 from the github repository
- QtCreator
- All steps are explained on a Linux machine

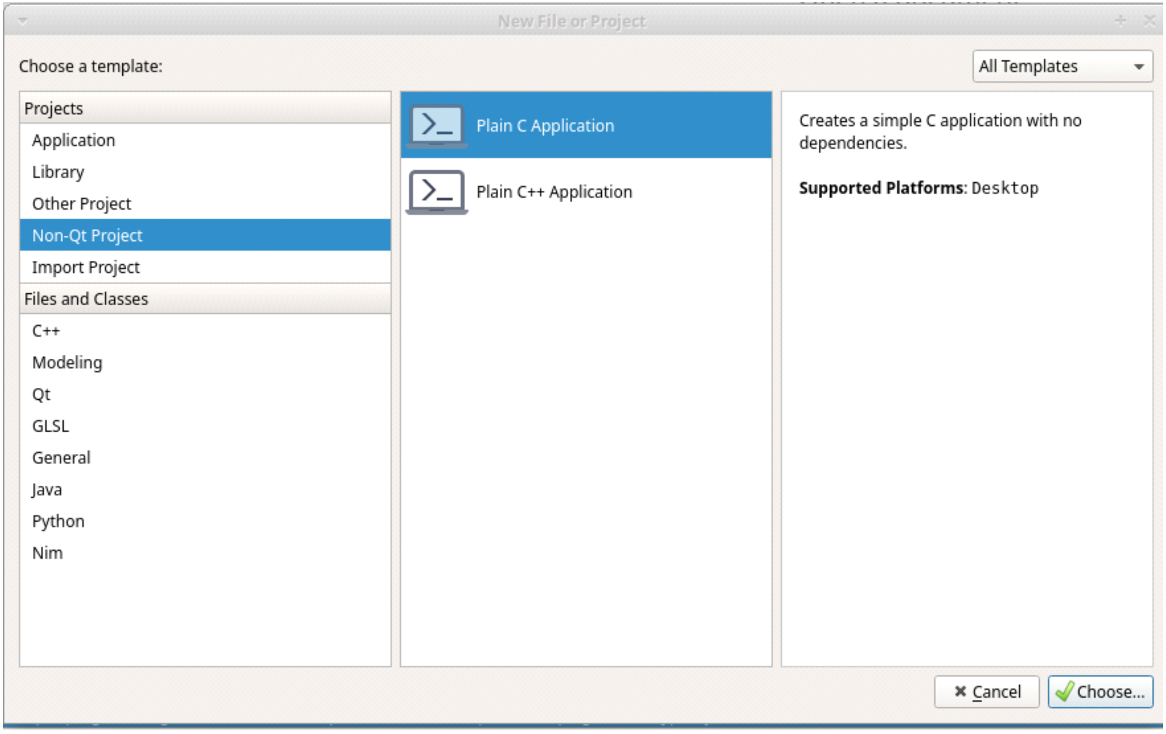
1. Creating the Project folders

You have to copy both, the open62541 project folder and the hybrid_crypto_test folder into the same directory.



Name	Size	Type	Date Modified
hybrid_crypto_test	4,1 kB	folder	Today
opc_ua_client	4,1 kB	folder	Today
opc_ua_server	4,1 kB	folder	Today
open62541	4,1 kB	folder	Today

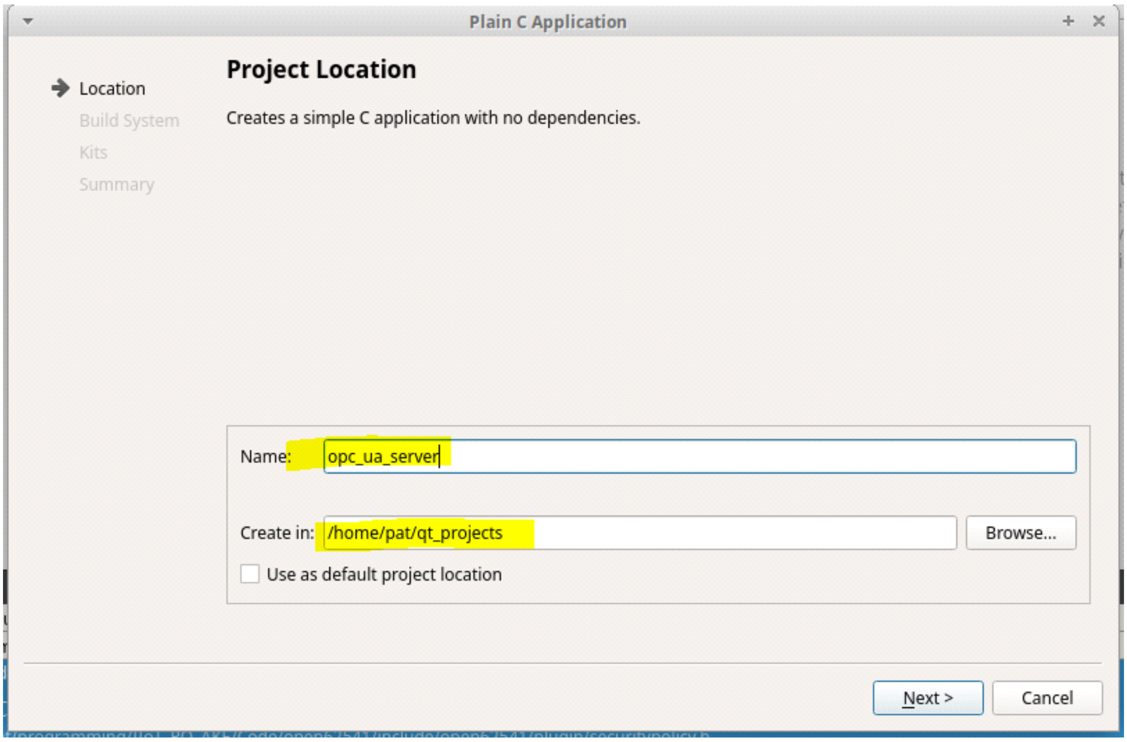
Then you have to start QtCreator and create two new projects as in the following screenshots:



Creates a simple C application with no dependencies.

Supported Platforms: `Desktop`

Name one project "opc_ua_server" and the other "opc_ua_client". Place both in the same folder as you copied the previous projects



Plain C Application

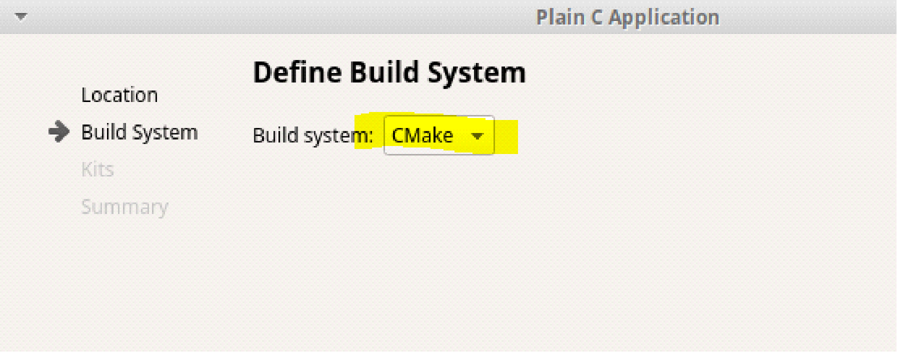
Project Location

Creates a simple C application with no dependencies.

Name: opc_ua_server

Create in: /home/pat/qt_projects

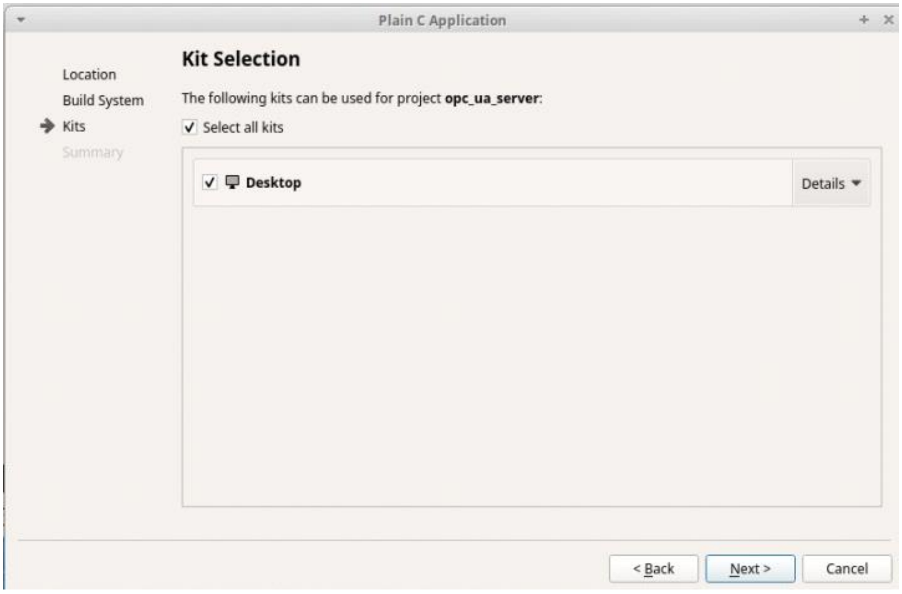
☐ Use as default project location



Plain C Application

Define Build System

Build system: CMake



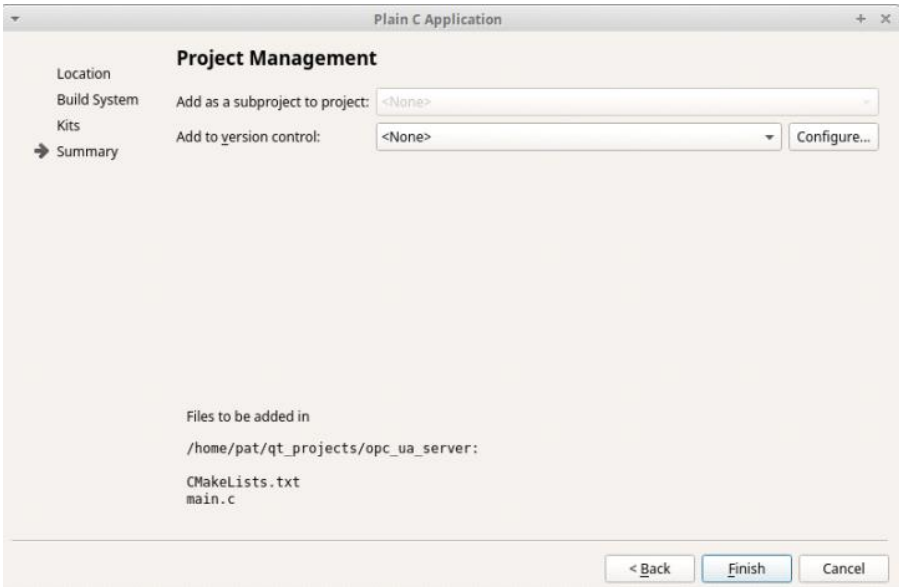
Plain C Application

Kit Selection

The following kits can be used for project **opc_ua_server**:

☑ Select all kits

☑ **Desktop**



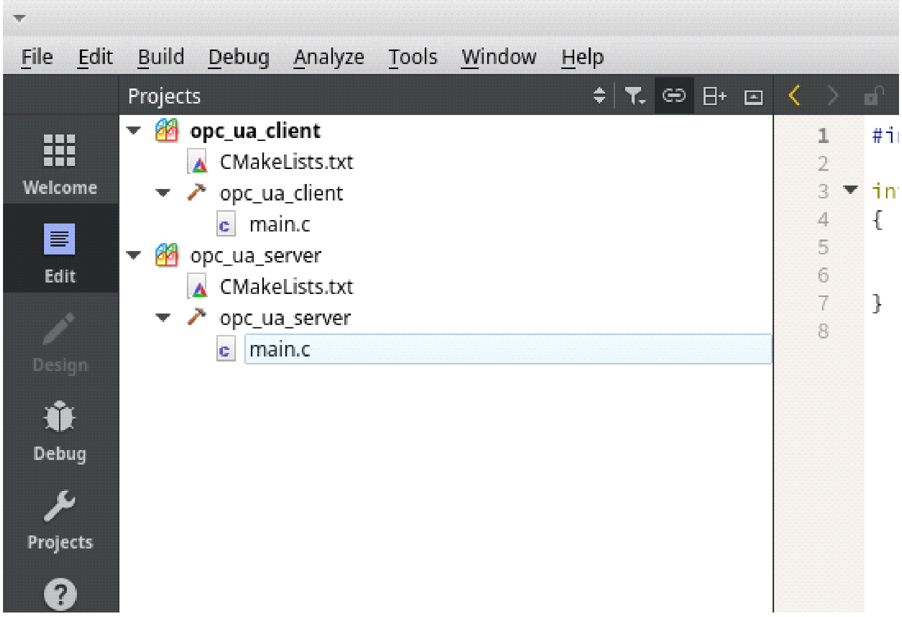
Project Management

Add as a subproject to project: <None>

Add to version control: <None>

Files to be added in

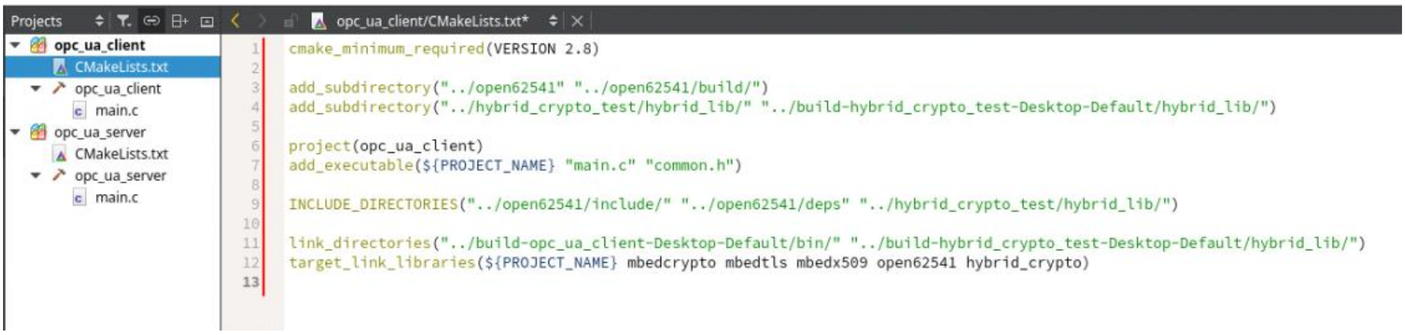
```
/home/pat/qt_projects/opc_ua_server:

CMakeLists.txt
main.c
```



2. Change the CMake files so that you have access to all the sub projects

Add the subdirectories that also contain CMake projects. The second folder specifies the build folders. Also the link and include directories are defined here. Note that mbedtls is already installed on the system as a static library --> The lib and include files are in a standard directory.

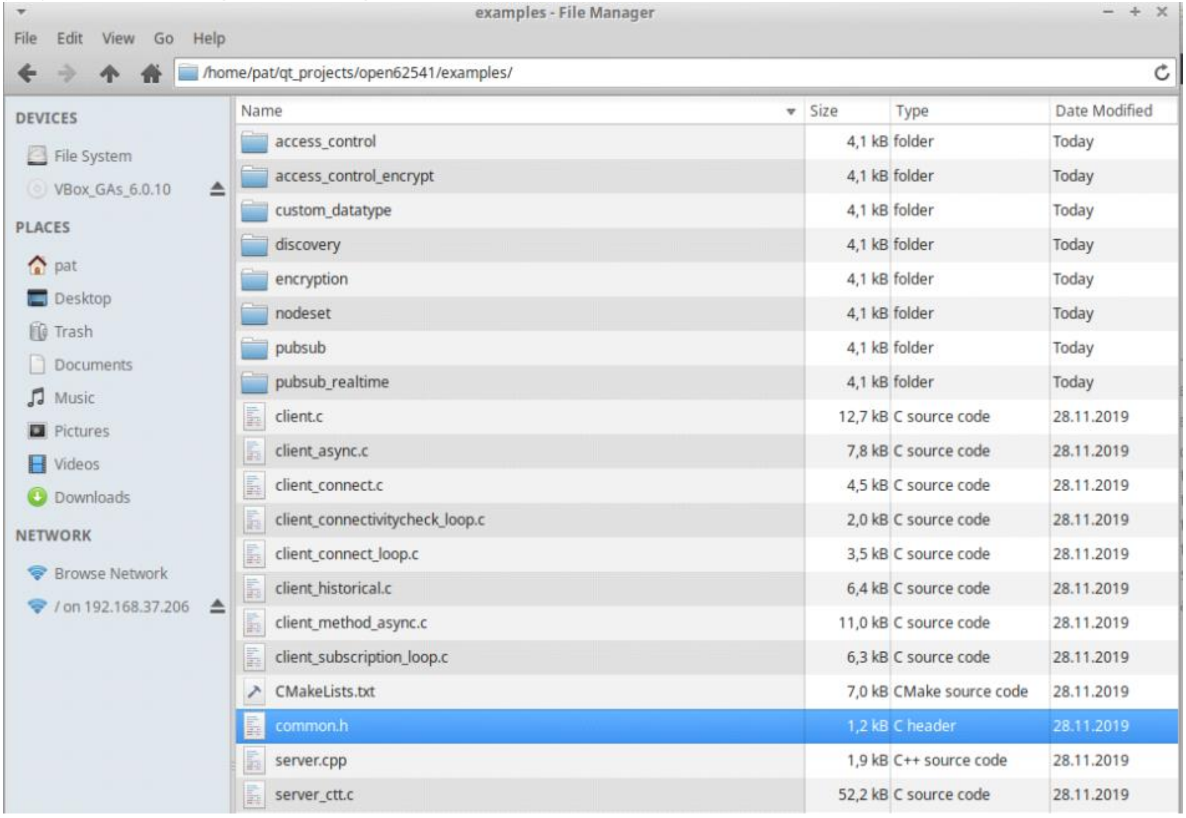
```
cmake_minimum_required(VERSION 2.8)

project(opc_ua_server)

add_subdirectory("../open62541" "../open62541/build/")
add_subdirectory("../hybrid_crypto_test/hybrid_lib/" "../build-hybrid_crypto_test-Desktop-Default/hybrid_lib/")

add_executable(${PROJECT_NAME} "main.c" "common.h")
INCLUDE_DIRECTORIES("../open62541/include/" "../open62541/deps" "../hybrid_crypto_test/hybrid_lib/")

link_directories("../build-opc_ua_server-Desktop-Default/bin/" "../build-hybrid_crypto_test-Desktop-Default/hybrid_lib/")
target_link_libraries(${PROJECT_NAME} mbedcrypto mbedtls mbedx509 open62541 hybrid_crypto)
```



```
cmake_minimum_required(VERSION 2.8)

add_subdirectory("../open62541" "../open62541/build/")
add_subdirectory("../hybrid_crypto_test/hybrid_lib/" "../build-hybrid_crypto_test-Desktop-Default/hybrid_lib/")

project(opc_ua_client)
add_executable(${PROJECT_NAME} "main.c" "common.h")

INCLUDE_DIRECTORIES("../open62541/include/" "../open62541/deps" "../hybrid_crypto_test/hybrid_lib/")

link_directories("../build-opc_ua_client-Desktop-Default/bin/" "../build-hybrid_crypto_test-Desktop-Default/hybrid_lib/")
target_link_libraries(${PROJECT_NAME} mbedcrypto mbedtls mbedx509 open62541 hybrid_crypto)
```

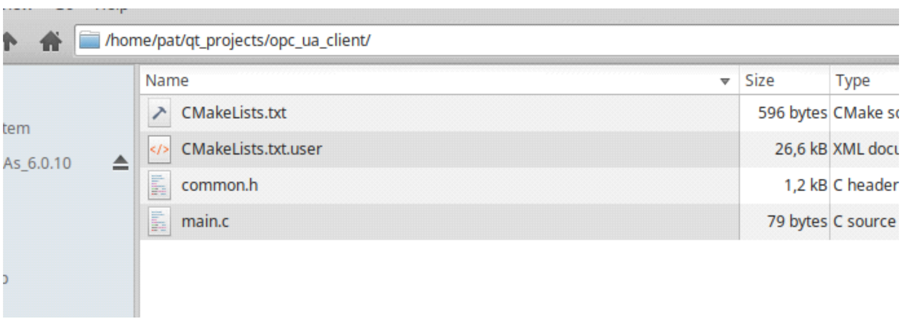
Notice that in the "add_subdirectory" command, the build folder has to be specified. If it doesn't exist yet, create it, or try if it will be automatically created when running CMake.

Then copy a file called "common.h" from the open62541 examples code into each of the two created project folders using a file manager



/home/pat/qt_projects/open62541/examples/

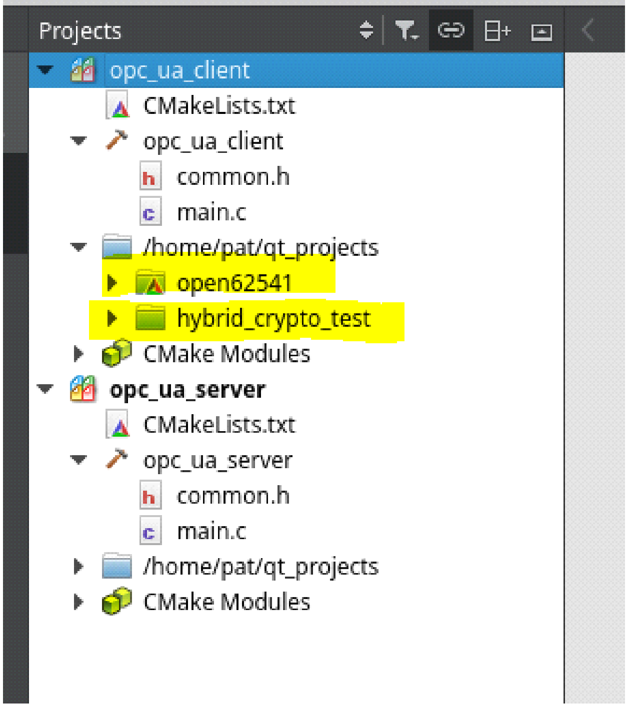
Name	Size	Type	Date Modified
access_control	4,1 kB	folder	Today
access_control_encrypt	4,1 kB	folder	Today
custom_datatype	4,1 kB	folder	Today
discovery	4,1 kB	folder	Today
encryption	4,1 kB	folder	Today
nodeset	4,1 kB	folder	Today
pubsub	4,1 kB	folder	Today
pubsub_realtime	4,1 kB	folder	Today
client.c	12,7 kB	C source code	28.11.2019
client_async.c	7,8 kB	C source code	28.11.2019
client_connect.c	4,5 kB	C source code	28.11.2019
client_connectivitycheck_loop.c	2,0 kB	C source code	28.11.2019
client_connect_loop.c	3,5 kB	C source code	28.11.2019
client_historical.c	6,4 kB	C source code	28.11.2019
client_method_async.c	11,0 kB	C source code	28.11.2019
client_subscription_loop.c	6,3 kB	C source code	28.11.2019
CMakeLists.txt	7,0 kB	CMake source code	28.11.2019
common.h	1,2 kB	C header	28.11.2019
server.cpp	1,9 kB	C++ source code	28.11.2019
server_ctt.c	52,2 kB	C source code	28.11.2019



/home/pat/qt_projects/opc_ua_client/

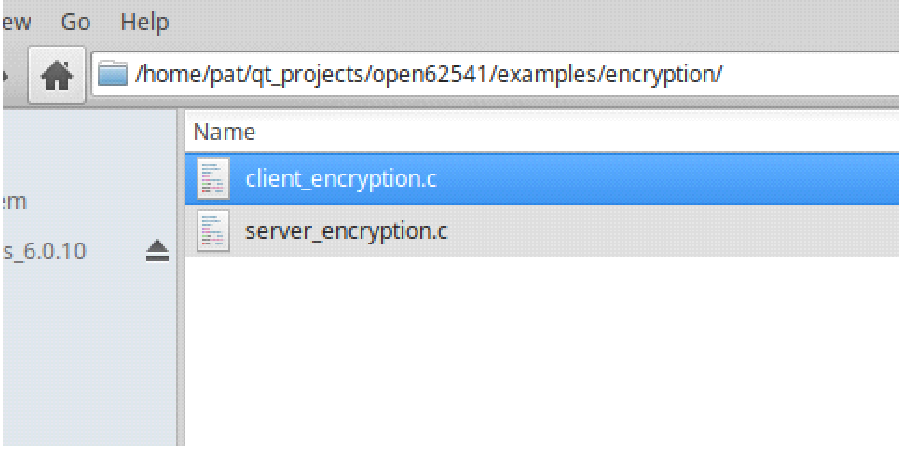
Name	Size	Type
CMakeLists.txt	596 bytes	CMake sc
CMakeLists.txt.user	26,6 kB	XML docu
common.h	1,2 kB	C header
main.c	79 bytes	C source

When the CMake files are saved, QtCreator is loading after that you can see the two other projects in the file tree.



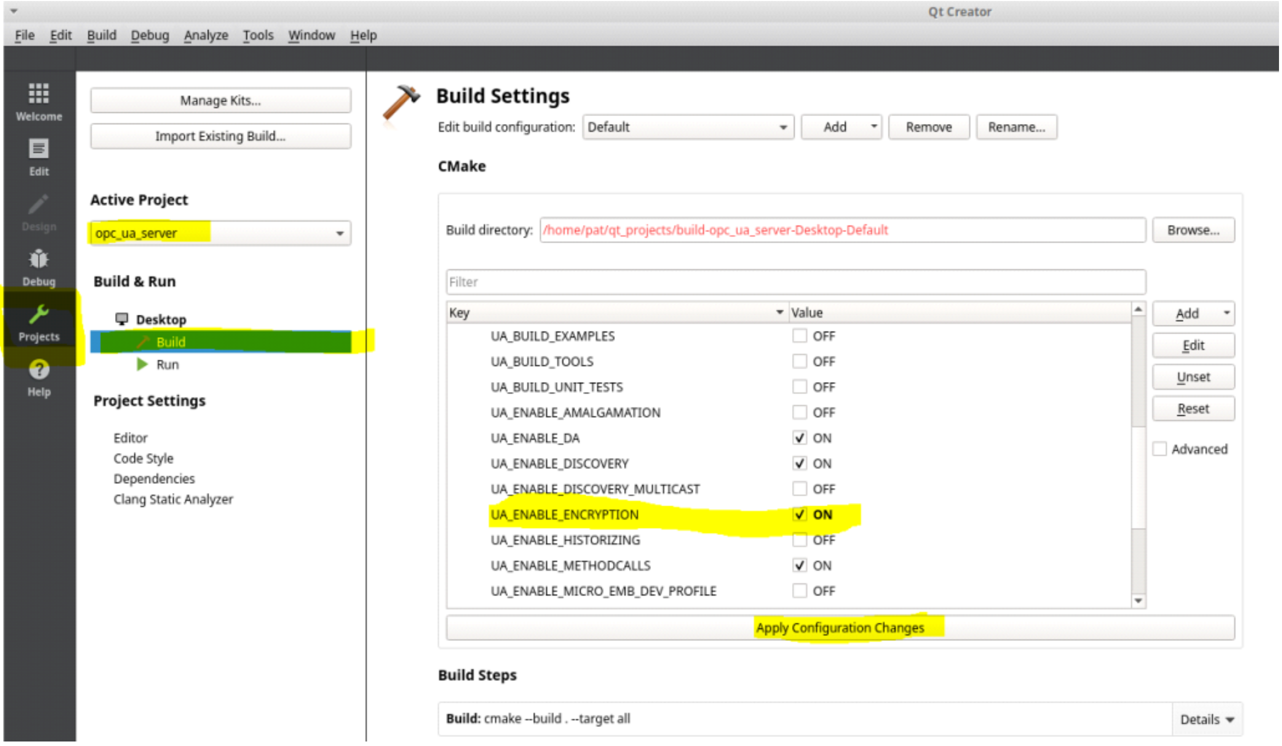
3. Add the sample code for server and client

Go the the examples folder of open62541 and look for server_encryption.c and client_encryption.c. Open each file with a text editor and copy the code to the main function of each project (server to server, client to client). Replace everything in the original main.c file.



4. Configure the Builds

On the left, click Projects, select active project (first server later client) and set the tick at UA_ENABLE_ENCRYPTION and click "Apply Configuration Changes". Do the same for the client.



Active Project: opc_ua_server

Edit build configuration: Default

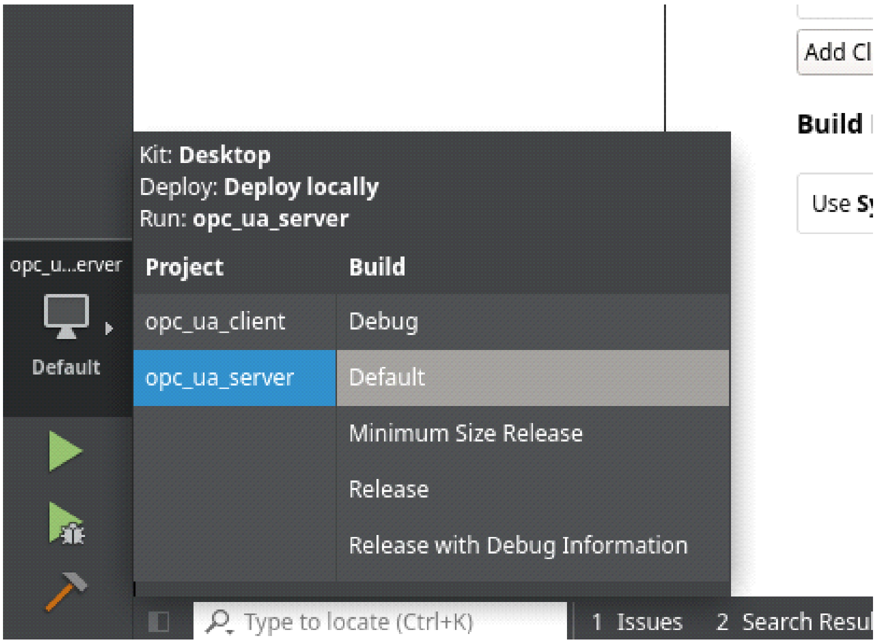
Build directory: /home/pat/qt_projects/build-opc_ua_server-Desktop-Default

Key	Value
UA_BUILD_EXAMPLES	☐ OFF
UA_BUILD_TOOLS	☐ OFF
UA_BUILD_UNIT_TESTS	☐ OFF
UA_ENABLE_AMALGAMATION	☐ OFF
UA_ENABLE_DA	☑ ON
UA_ENABLE_DISCOVERY	☑ ON
UA_ENABLE_DISCOVERY_MULTICAST	☐ OFF
UA_ENABLE_ENCRYPTION	☑ ON
UA_ENABLE_HISTORIZING	☐ OFF
UA_ENABLE_METHODCALLS	☑ ON
UA_ENABLE_MICRO_EMB_DEV_PROFILE	☐ OFF

Build: cmake --build . --target all

5. Build

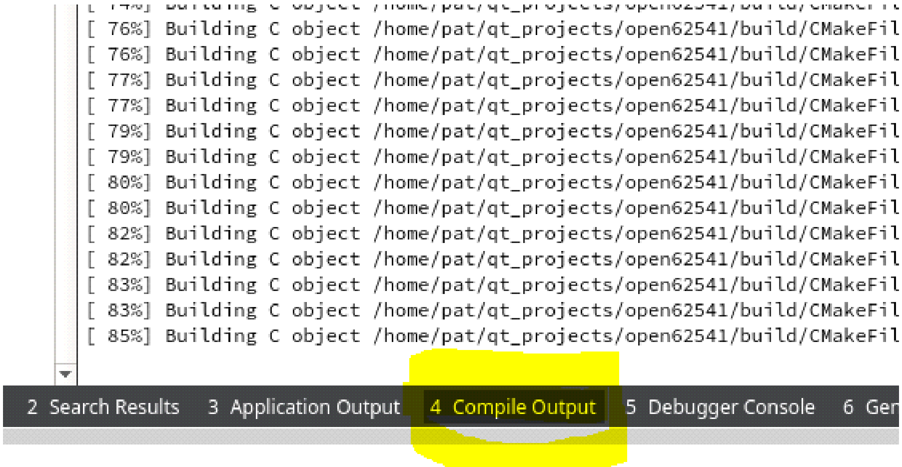
Set the active project to server and click the "Run" button (green triangle). This will build and run the server.



Kit: **Desktop**
Deploy: **Deploy locally**
Run: **opc_ua_server**

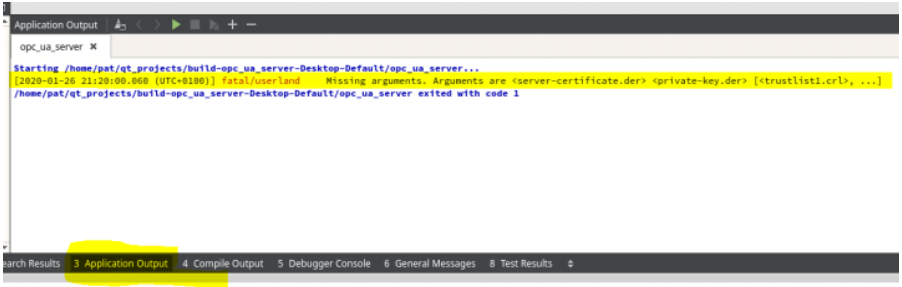
Project	Build
opc_ua_client	Debug
opc_ua_server	Default
	Minimum Size Release
	Release
	Release with Debug Information

Open the compile window to see the progress. The building and linking steps will take a few seconds



```
[ 76%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 76%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 77%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 77%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 79%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 79%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 80%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 80%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 82%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 82%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 83%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 83%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
[ 85%] Building C object /home/pat/qt_projects/open62541/build/CMakeFil
```

When the compilation step is done, the "Application Output" view will open automatically and show the programs output. Currently this is just an error message because we didn't supply the correct parameters. The important point is to notice that the program compiles and runs.



```
Starting /home/pat/qt_projects/build-opc_ua_server-Desktop-Default/opc_ua_server...
[2020-01-26 21:20:00.060 (UTC+0100)] fatal/userland     Missing arguments. Arguments are <server-certificate.der> <private-key.der> [<trustlist1.crl>, ...]
/home/pat/qt_projects/build-opc_ua_server-Desktop-Default/opc_ua_server exited with code 1
```

Now repeat the same steps for the client. You should see this output:

```
Starting /home/pat/qt_projects/build-opc_ua_client-Desktop-Default/opc_ua_client...
[2020-01-26 21:23:06.004 (UTC+0100)] fatal/userland     Arguments are missing. The required arguments are <opc.tcp://host:port> <client-certificate.der> <client-private-key.der> [<trustlist1.crl>, ...]
/home/pat/qt_projects/build-opc_ua_client-Desktop-Default/opc_ua_client exited with code 1
```

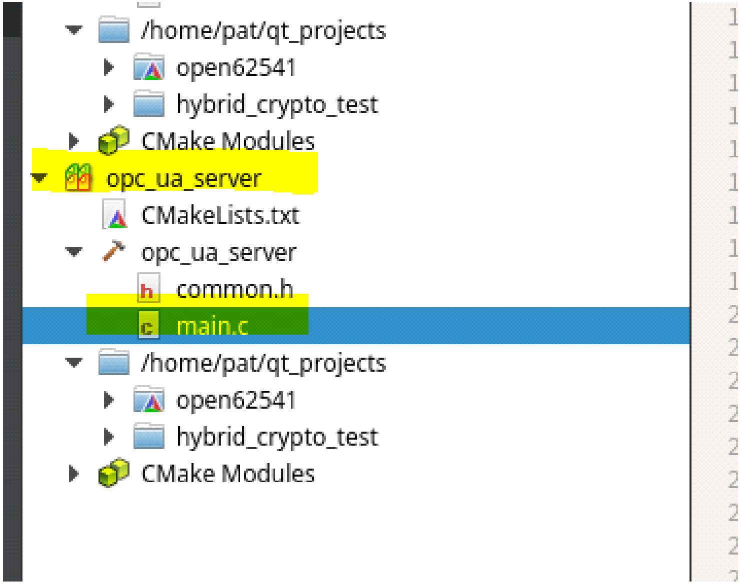
Adding Hybrid Private Key as Parameter

Sonntag, 26. Januar 2020 21:23

The first step in order to change open62541 to work with hybrid certificates is to supply the hybrid private key as an parameter. The parameter will be a file name that contains the binary private key (depending on the scheme used later). For now we only care about passing another binary file.

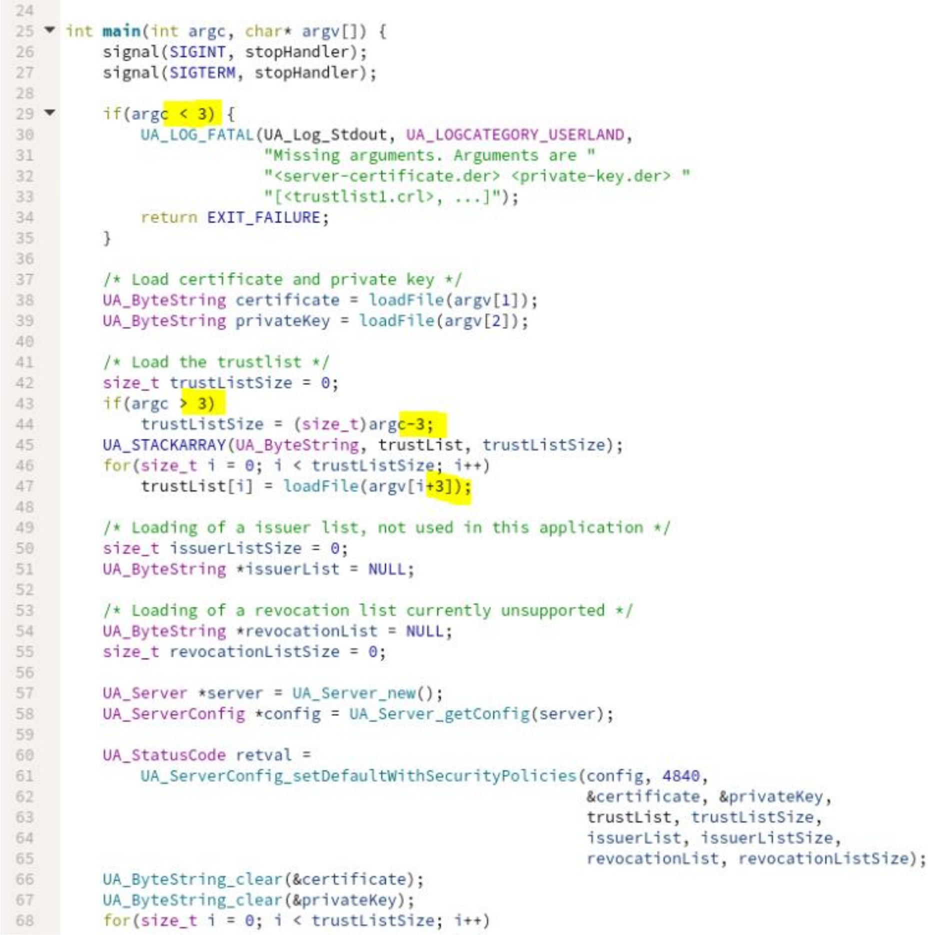
1. Server

In the server project, open the main.c file.



Change the number of parameters (3 in the original) to 4 since we will add an additional parameter.

Original



```
int main(int argc, char* argv[]) {
    signal(SIGINT, stopHandler);
    signal(SIGTERM, stopHandler);

    if(argc < 3) {
        UA_LOG_FATAL(UA_Log_Stdout, UA_LOGCATEGORY_USERLAND,
                     "Missing arguments. Arguments are "
                     "<server-certificate.der> <private-key.der> "
                     "[<trustlist1.crl>, ...]");
        return EXIT_FAILURE;
    }

    /* Load certificate and private key */
    UA_ByteString certificate = loadFile(argv[1]);
    UA_ByteString privateKey = loadFile(argv[2]);

    /* Load the trustlist */
    size_t trustListSize = 0;
    if(argc > 3)
        trustListSize = (size_t)argc-3;
    UA_STACKARRAY(UA_ByteString, trustList, trustListSize);
    for(size_t i = 0; i < trustListSize; i++)
        trustList[i] = loadFile(argv[i+3]);

    /* Loading of a issuer list, not used in this application */
    size_t issuerListSize = 0;
    UA_ByteString *issuerList = NULL;

    /* Loading of a revocation list currently unsupported */
    UA_ByteString *revocationList = NULL;
    size_t revocationListSize = 0;

    UA_Server *server = UA_Server_new();
    UA_ServerConfig *config = UA_Server_getConfig(server);

    UA_StatusCode retval =
        UA_ServerConfig_setDefaultWithSecurityPolicies(config, 4840,
                                                       &certificate, &privateKey,
                                                       trustList, trustListSize,
                                                       issuerList, issuerListSize,
                                                       revocationList, revocationListSize);
    UA_ByteString_clear(&certificate);
    UA_ByteString_clear(&privateKey);
    for(size_t i = 0; i < trustListSize; i++)
```

New:

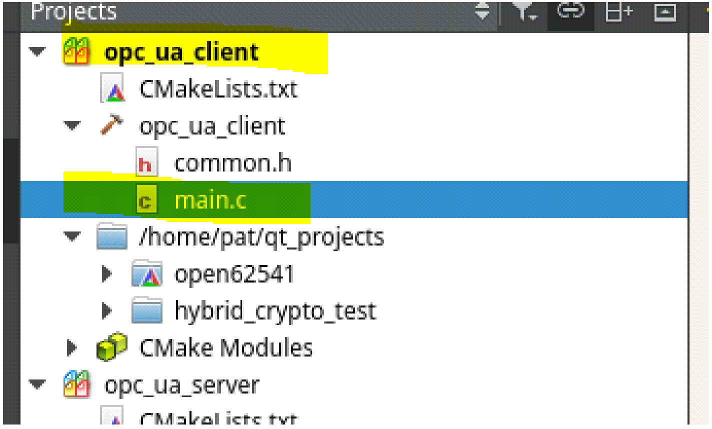
```
int main(int argc, char* argv[]) {
    signal(SIGINT, stopHandler);
    signal(SIGTERM, stopHandler);

    if(argc < 4) {
        UA_LOG_FATAL(UA_Log_Stdout, UA_LOGCATEGORY_USERLAND,
                     "Missing arguments. Arguments are "
                     "<server-certificate.der> <private-key.der> <hybrid-private-key.bin> "
                     "[<trustlist1.crl>, ...]");
        return EXIT_FAILURE;
    }

    /* Load certificate and private key */
    UA_ByteString certificate = loadFile(argv[1]);
    UA_ByteString privateKey = loadFile(argv[2]);
    UA_ByteString hybridPrivateKey = loadFile(argv[3]);

    /* Load the trustlist */
    size_t trustListSize = 0;
    if(argc > 4)
        trustListSize = (size_t)argc-4;
    UA_STACKARRAY(UA_ByteString, trustList, trustListSize);
    for(size_t i = 0; i < trustListSize; i++)
        trustList[i] = loadFile(argv[i+4]);

    /* Loading of a issuer list, not used in this application */
    size_t issuerListSize = 0;
    UA_ByteString *issuerList = NULL;

    /* Loading of a revocation list currently unsupported */
    UA_ByteString *revocationList = NULL;
    size_t revocationListSize = 0;

    UA_Server *server = UA_Server_new();
    UA_ServerConfig *config = UA_Server_getConfig(server);

    UA_StatusCode retval =
        UA_ServerConfig_setDefaultWithSecurityPolicies(config, 4840,
                                                       &certificate, &privateKey,
                                                       trustList, trustListSize,
                                                       issuerList, issuerListSize,
                                                       revocationList, revocationListSize);

    UA_ByteString_clear(&certificate);
    UA_ByteString_clear(&privateKey);
```

Additions:

- change the error message, that is displayed when too few arguments are passed.
- Add a UA_ByteString for the hybrid private key.
- Add the hybrid private key as a pointer to the parameters of the function

2. Client

Open the main.c file in the client project



We also have to change the number of arguments

Original

```
12
13  #include "common.h"
14
15  #define MIN_ARGS 4
16
17  int main(int argc, char* argv[]) {
18      if(argc < MIN_ARGS) {
19          UA_LOG_FATAL(UA_Log_Stdout, UA_LOGCATEGORY_USERLAND,
20                       "Arguments are missing. The required arguments are "
21                       "<opc.tcp://host:port> "
22                       "<client-certificate.der> <client-private-key.der> "
23                       "[<trustlist1.crl>, ...]");
24          return EXIT_FAILURE;
25      }
26
27      const char *endpointUrl = argv[1];
28
29      /* Load certificate and private key */
30      UA_ByteString certificate = loadFile(argv[2]);
31      UA_ByteString privateKey  = loadFile(argv[3]);
32
33      /* Load the trustList. Load revocationList is not supported now */
34      size_t trustListSize = 0;
35      if(argc > MIN_ARGS)
36          trustListSize = (size_t)argc-MIN_ARGS;
37      UA_STACKARRAY(UA_ByteString, trustList, trustListSize);
38      for(size_t trustListCount = 0; trustListCount < trustListSize; trustListCount++)
39          trustList[trustListCount] = loadFile(argv[trustListCount+4]);
40
41      UA_ByteString *revocationList = NULL;
42      size_t revocationListSize = 0;
43
44      UA_Client *client = UA_Client_new();
45      UA_ClientConfig *cc = UA_Client_getConfig(client);
46      cc->securityMode = UA_MESSAGESECURITYMODE_SIGNANDENCRYPT;
47      UA_ClientConfig_setDefaultEncryption(cc, certificate, privateKey,
48                                           trustList, trustListSize,
49                                           revocationList, revocationListSize);
50
51      UA_ByteString_clear(&certificate);
52      UA_ByteString_clear(&privateKey);
53      for(size_t deleteCount = 0; deleteCount < trustListSize; deleteCount++) {
54          UA_ByteString_clear(&trustList[deleteCount]);
55      }
```

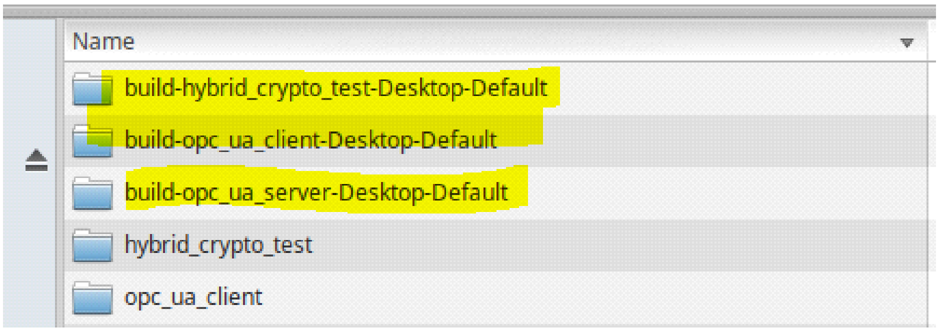
With changes

```
11  #include <stdlib.h>
12
13  #include "common.h"
14
15  #define MIN_ARGS 5
16
17  int main(int argc, char* argv[]) {
18      if(argc < MIN_ARGS) {
19          UA_LOG_FATAL(UA_Log_Stdout, UA_LOGCATEGORY_USERLAND,
20                       "Arguments are missing. The required arguments are "
21                       "<opc.tcp://host:port> "
22                       "<client-certificate.der> <client-private-key.der> <hybrid-private-key.bin> "
23                       "[<trustlist1.crl>, ...]");
24          return EXIT_FAILURE;
25      }
26
27      const char *endpointUrl = argv[1];
28
29      /* Load certificate and private key */
30      UA_ByteString certificate = loadFile(argv[2]);
31      UA_ByteString privateKey  = loadFile(argv[3]);
32      UA_ByteString hybridPrivateKey = loadFile(argv[3]);
33
34      /* Load the trustList. Load revocationList is not supported now */
35      size_t trustListSize = 0;
36      if(argc > MIN_ARGS)
37          trustListSize = (size_t)argc-MIN_ARGS;
38      UA_STACKARRAY(UA_ByteString, trustList, trustListSize);
39      for(size_t trustListCount = 0; trustListCount < trustListSize; trustListCount++)
40          trustList[trustListCount] = loadFile(argv[trustListCount+MIN_ARGS]);
41
42      UA_ByteString *revocationList = NULL;
43      size_t revocationListSize = 0;
44
45      UA_Client *client = UA_Client_new();
46      UA_ClientConfig *cc = UA_Client_getConfig(client);
47      cc->securityMode = UA_MESSAGESECURITYMODE_SIGNANDENCRYPT;
48      UA_ClientConfig_setDefaultEncryption(cc, certificate, privateKey,
49                                           trustList, trustListSize,
50                                           revocationList, revocationListSize);
51
52      UA_ByteString_clear(&certificate);
```

(the screenshot above has an error. The index in line 32 must be 4 and not 3!!)

3. Adding the Arguments when Running the Project

When building the projects in a previous step, the build folders were created



Name

- build-hybrid_crypto_test-Desktop-Default
- build-opc_ua_client-Desktop-Default
- build-opc_ua_server-Desktop-Default
- hybrid_crypto_test
- opc_ua_client

Copy all the certificates and private keys to each, the server and client build folder. Decide which certificates you want to use (for example hybrid RSA/Dilithium3 certificates). Then copy following files to Server build folder:

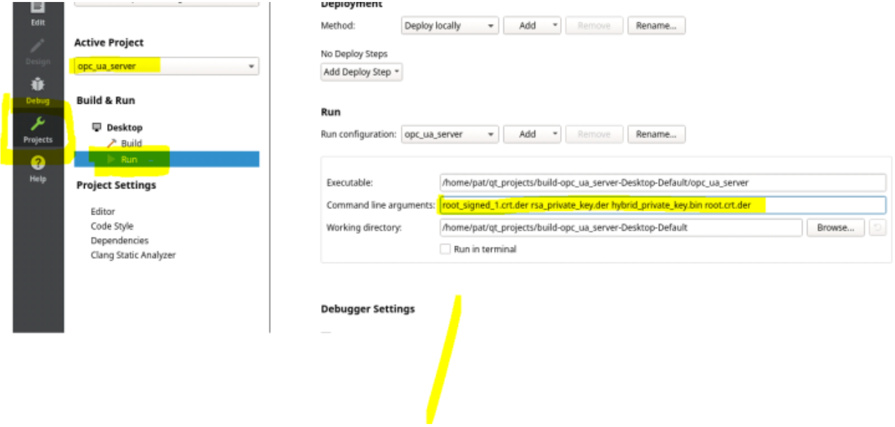
- Root_signed_1.crt.der
  Certificate for the server that was signed by the root certificates
- Rsa_private_key.der
  The RSA private key for the public key in root_signed_1.crt.der
- Hybrid_private_key.bin
  the dilithium3 private key for the public key in root_signed_1.crt.der
- Root.crt.der
  Certificate with the public key to verify root_signed_1.crt.der

Client build folder:

- Root_signed_2.crt.der
- Rsa_private_key.der
- Hybrid_private_key.bin
- Root.crt.der

Note that root.crt.der is the same file for both folders. The private keys have the same files names however contain different private keys (corresponding to the certificate each).

In QtCreator go to the projects settings and set the command line arguments



Active Project: opc_ua_server

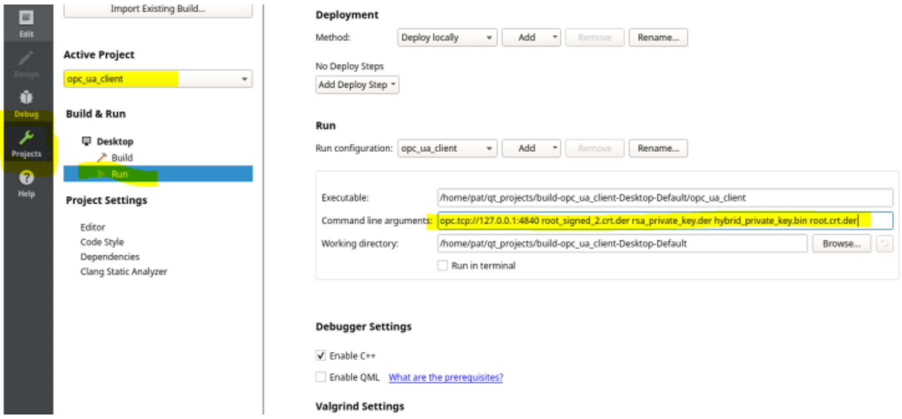
Method: Deploy locally

Run configuration: opc_ua_server

Executable: /home/pat/qt_projects/build-opc_ua_server-Desktop-Default/opc_ua_server

Command line arguments: root_signed_1.crt.der rsa_private_key.der hybrid_private_key.bin root.crt.der

Working directory: /home/pat/qt_projects/build-opc_ua_server-Desktop-Default



Active Project: opc_ua_client

Method: Deploy locally

Run configuration: opc_ua_client

Executable: /home/pat/qt_projects/build-opc_ua_client-Desktop-Default/opc_ua_client

Command line arguments: opc.tcp://127.0.0.1:4840 root_signed_2.crt.der rsa_private_key.der hybrid_private_key.bin root.crt.der

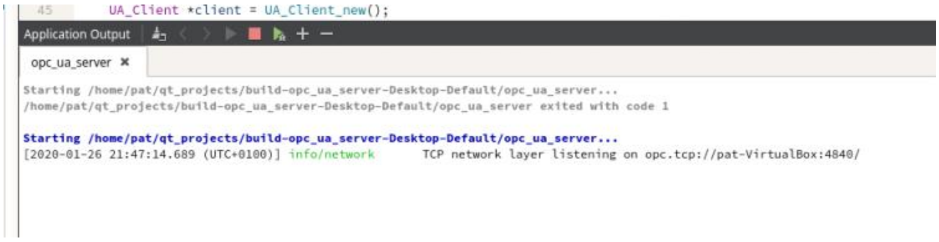
Working directory: /home/pat/qt_projects/build-opc_ua_client-Desktop-Default

☐ Run in terminal

☑ Enable C++

☐ Enable QML

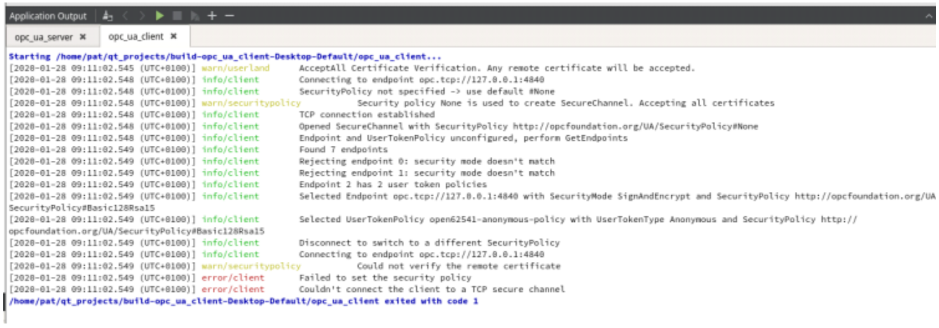
To test run first the server and then the client and see if they connect. Notice that even though hybrid certificates are used, so far open62541 is just ignoring the hybrid part and uses them as standard x.509 certificates.



```
UA_Client *client = UA_Client_new();
```

```
Starting /home/pat/qt_projects/build-opc_ua_server-Desktop-Default/opc_ua_server...
/home/pat/qt_projects/build-opc_ua_server-Desktop-Default/opc_ua_server exited with code 1

Starting /home/pat/qt_projects/build-opc_ua_server-Desktop-Default/opc_ua_server...
[2020-01-26 21:47:14.689 (UTC+0100)] info/network      TCP network layer listening on opc.tcp://pat-VirtualBox:4840/
```

When the client connects you should see following output with an error:



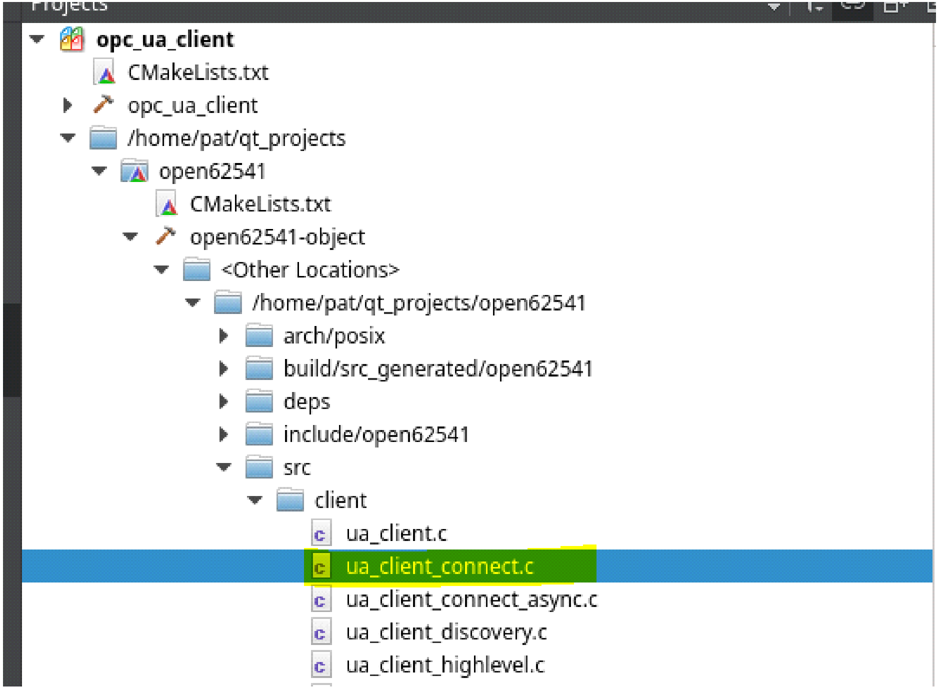
```
Starting /home/pat/qt_projects/build-opc_ua_client-Desktop-Default/opc_ua_client...
[2020-01-28 09:11:02.545 (UTC+0100)] warn/userland      AcceptAll Certificate Verification. Any remote certificate will be accepted.
[2020-01-28 09:11:02.548 (UTC+0100)] info/client        Connecting to endpoint opc.tcp://127.0.0.1:4840
[2020-01-28 09:11:02.548 (UTC+0100)] info/client        SecurityPolicy not specified -> use default #None
[2020-01-28 09:11:02.548 (UTC+0100)] warn/securitypolicy        Security policy None is used to create SecureChannel. Accepting all certificates
[2020-01-28 09:11:02.548 (UTC+0100)] info/client        TCP connection established
[2020-01-28 09:11:02.548 (UTC+0100)] info/client        Opened SecureChannel with SecurityPolicy http://opcfoundation.org/UA/SecurityPolicy#None
[2020-01-28 09:11:02.548 (UTC+0100)] info/client        Endpoint and UserTokenPolicy unconfigured, perform GetEndpoints
[2020-01-28 09:11:02.549 (UTC+0100)] info/client        Found 7 endpoints
[2020-01-28 09:11:02.549 (UTC+0100)] info/client        Rejecting endpoint 0: security mode doesn't match
[2020-01-28 09:11:02.549 (UTC+0100)] info/client        Rejecting endpoint 1: security mode doesn't match
[2020-01-28 09:11:02.549 (UTC+0100)] info/client        Endpoint 2 has 2 user token policies
[2020-01-28 09:11:02.549 (UTC+0100)] info/client        Selected Endpoint opc.tcp://127.0.0.1:4840 with SecurityMode SignAndEncrypt and SecurityPolicy http://opcfoundation.org/UA
SecurityPolicy#Basic128Rsa15
[2020-01-28 09:11:02.549 (UTC+0100)] info/client        Selected UserTokenPolicy open62541-anonymous-policy with UserTokenType Anonymous and SecurityPolicy http://
opcfoundation.org/UA/SecurityPolicy#Basic128Rsa15
[2020-01-28 09:11:02.549 (UTC+0100)] info/client        Disconnect to switch to a different SecurityPolicy
[2020-01-28 09:11:02.549 (UTC+0100)] info/client        Connecting to endpoint opc.tcp://127.0.0.1:4840
[2020-01-28 09:11:02.549 (UTC+0100)] warn/securitypolicy        Could not verify the remote certificate
[2020-01-28 09:11:02.549 (UTC+0100)] error/client       Failed to set the security policy
[2020-01-28 09:11:02.549 (UTC+0100)] error/client       Couldn't connect the client to a TCP secure channel
/home/pat/qt_projects/build-opc_ua_client-Desktop-Default/opc_ua_client exited with code 1
```

This happens because the signature size is too big.

Fixing to Make it Work with Hybrid Certificates

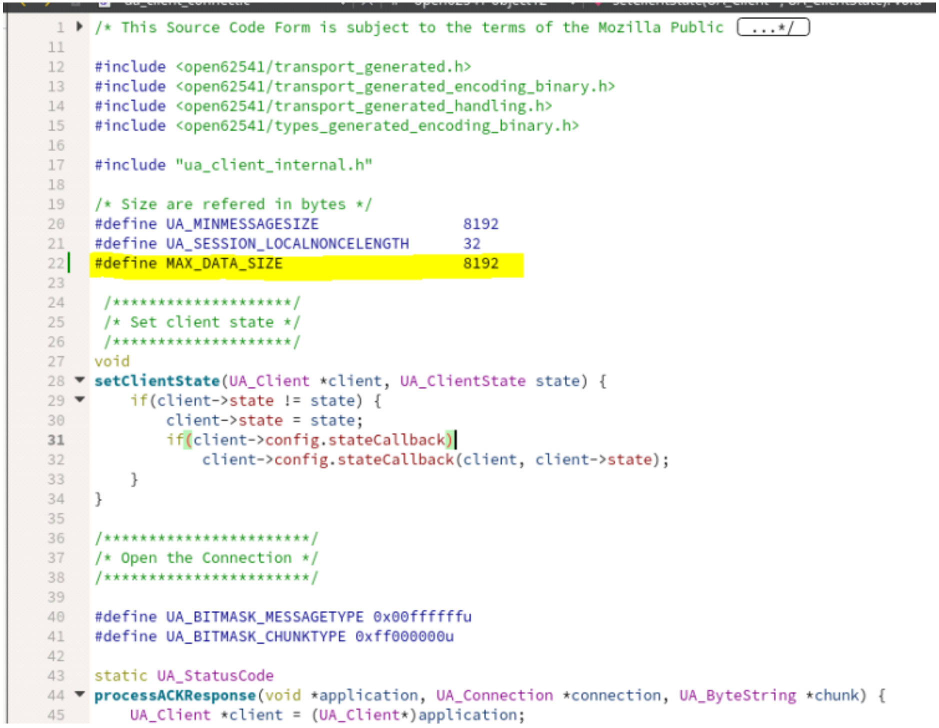
Dienstag, 28. Januar 2020 09:12

In either the server or the client project go to the file ua_client_connect.c



```
opc_ua_client
    CMakeLists.txt
  opc_ua_client
  /home/pat/qt_projects
    open62541
      CMakeLists.txt
      open62541-object
        <Other Locations>
          /home/pat/qt_projects/open62541
            arch/posix
            build/src_generated/open62541
            deps
            include/open62541
            src
              client
                ua_client.c
                ua_client_connect.c
                ua_client_connect_async.c
                ua_client_discovery.c
                ua_client_highlevel.c
```

Change the define for MAX_DATA_SIZE to 8192 (or larger). This is the size against which the certificate is checked when a session is activated and that causes an error with large signatures.



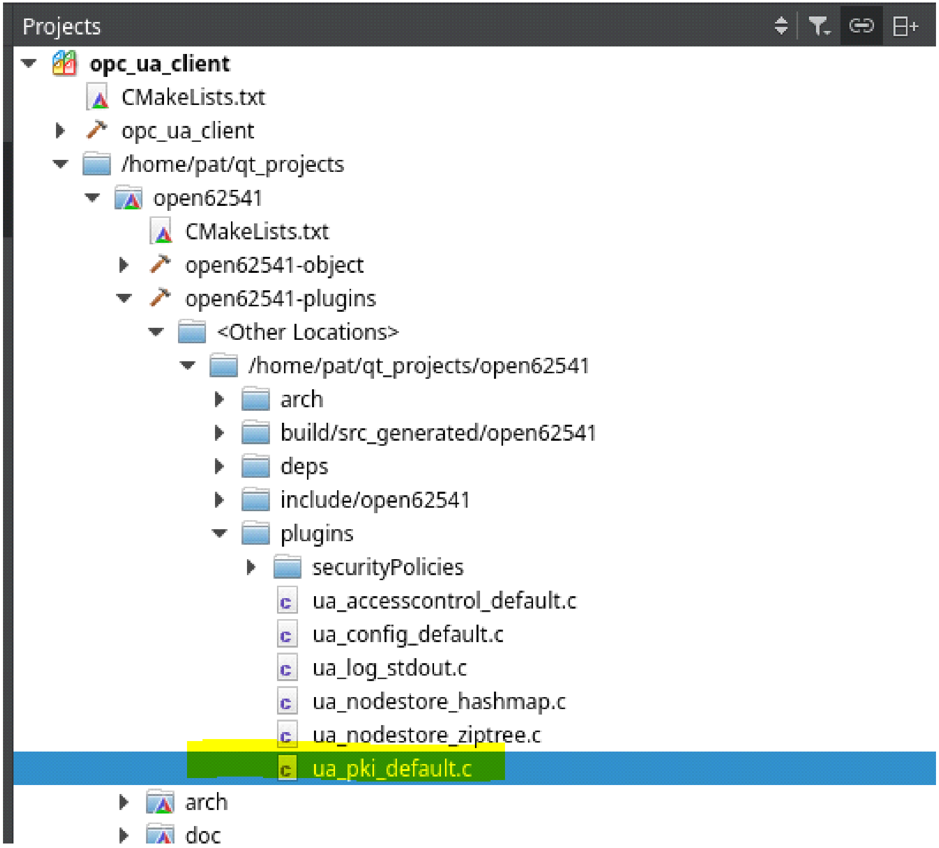
```
/* This Source Code Form is subject to the terms of the Mozilla Public ...*/

#include <open62541/transport_generated.h>
#include <open62541/transport_generated_encoding_binary.h>
#include <open62541/transport_generated_handling.h>
#include <open62541/types_generated_encoding_binary.h>

#include "ua_client_internal.h"

/* Size are refered in bytes */
#define UA_MINMESSAGESIZE                8192
#define UA_SESSION_LOCALNONCELENGTH      32
#define MAX_DATA_SIZE                    8192

 /********************/
 /* Set client state */
 /********************/
void
setClientState(UA_Client *client, UA_ClientState state) {
    if(client->state != state) {
        client->state = state;
        if(client->config.stateCallback)
            client->config.stateCallback(client, client->state);
    }
}

/***********************/
/* Open the Connection */
/***********************/

#define UA_BITMASK_MESSAGETYPE 0x00ffffffu
#define UA_BITMASK_CHUNKTYPE 0xff000000u

static UA_StatusCode
processACKResponse(void *application, UA_Connection *connection, UA_ByteString *chunk) {
    UA_Client *client = (UA_Client*)application;
```

Save and rebuild both projects. (Note: I had some problems when rebuilding that the file was not actually rebuild. So make sure with the debugger that MAX_DATA_SIZE has actually the value you want it to be!!)

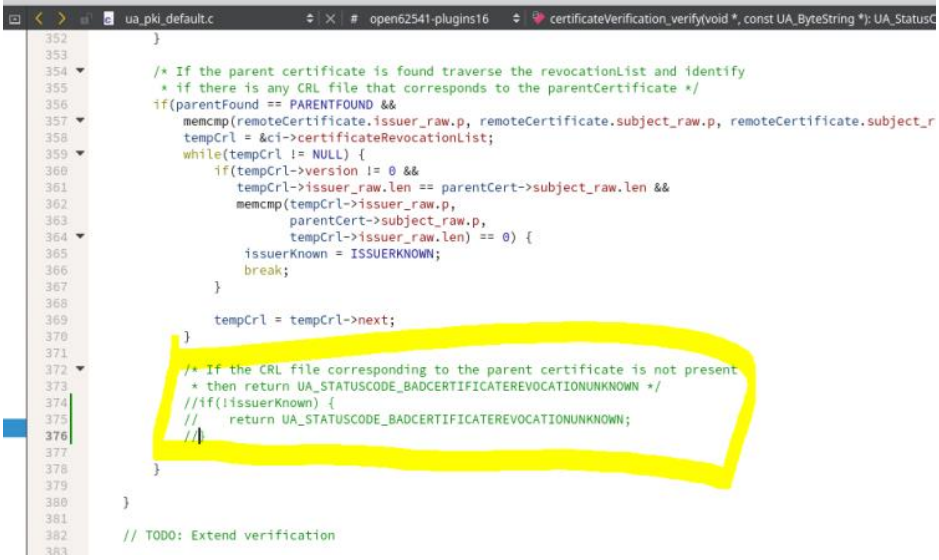
When testing if the changes worked there is another issue with the root certificate. Checking will fail because there is no revocation list. So to check if this change worked you have to remove the root.crt.der from the command line arguments for server and client.

Remove the revocation List Check

Look for the file ua_pki_default.c in the plugins folder.

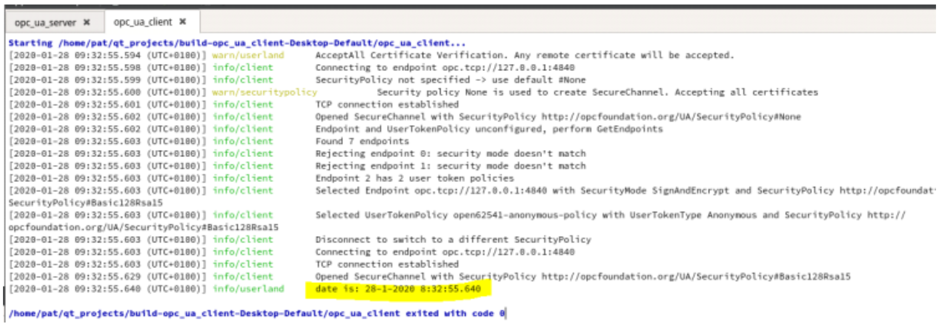


Find the function certificateVerification_verify() and comment out the following:



```
    }

    /* If the parent certificate is found traverse the revocationList and identify
     * if there is any CRL file that corresponds to the parentCertificate */
    if(parentFound == PARENTFOUND &&
        memcmp(remoteCertificate.issuer_raw.p, remoteCertificate.subject_raw.p, remoteCertificate.subject_r
        tempCrl = &ci->certificateRevocationList;
        while(tempCrl != NULL) {
            if(tempCrl->version != 0 &&
               tempCrl->issuer_raw.len == parentCert->subject_raw.len &&
               memcmp(tempCrl->issuer_raw.p,
                      parentCert->subject_raw.p,
                      tempCrl->issuer_raw.len) == 0) {
                issuerKnown = ISSUERKNOWN;
                break;
            }

            tempCrl = tempCrl->next;
        }

        /* If the CRL file corresponding to the parent certificate is not present
         * then return UA_STATUSCODE_BADCERTIFICATEREVOCATIONUNKNOWN */
        //if(!issuerKnown) {
        //    return UA_STATUSCODE_BADCERTIFICATEREVOCATIONUNKNOWN;
        //}

    }

}

// TODO: Extend verification
```

After compiling you should be able to run server and client successfully.

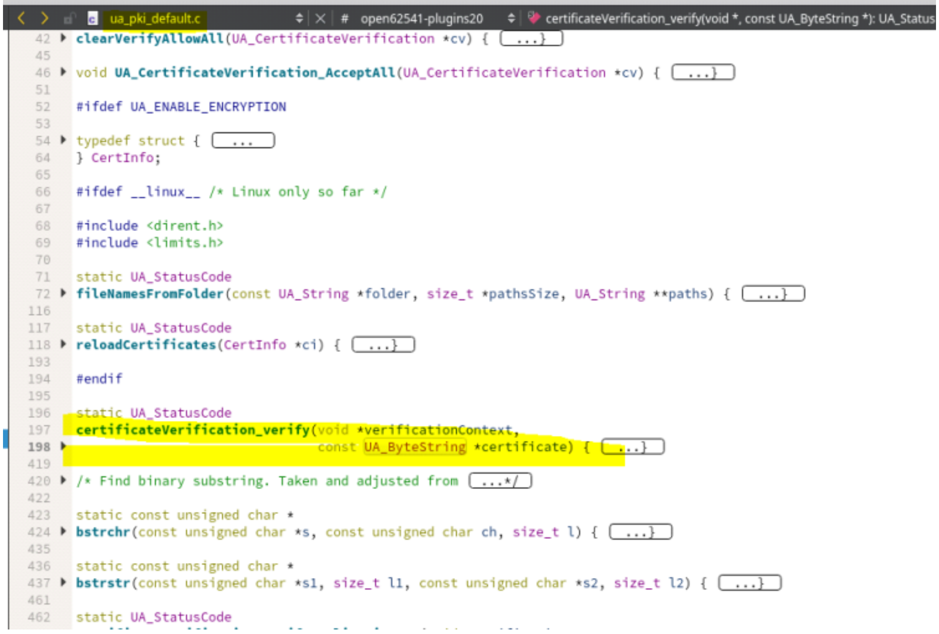


```
Starting /home/pat/qt_projects/build-opc_ua_client-Desktop-Default/opc_ua_client...
[2020-01-28 09:32:55.594 (UTC+0100)] warn/userland      AcceptAll Certificate Verification. Any remote certificate will be accepted.
[2020-01-28 09:32:55.598 (UTC+0100)] info/client        Connecting to endpoint opc.tcp://127.0.0.1:4840
[2020-01-28 09:32:55.599 (UTC+0100)] info/client        SecurityPolicy not specified -> use default #None
[2020-01-28 09:32:55.600 (UTC+0100)] warn/securitypolicy        Security policy None is used to create SecureChannel. Accepting all certificates
[2020-01-28 09:32:55.601 (UTC+0100)] info/client        TCP connection established
[2020-01-28 09:32:55.602 (UTC+0100)] info/client        Opened SecureChannel with SecurityPolicy http://opcfoundation.org/UA/SecurityPolicy#None
[2020-01-28 09:32:55.602 (UTC+0100)] info/client        Endpoint and UserTokenPolicy unconfigured, perform GetEndpoints
[2020-01-28 09:32:55.603 (UTC+0100)] info/client        Found 7 endpoints
[2020-01-28 09:32:55.603 (UTC+0100)] info/client        Rejecting endpoint 0: security mode doesn't match
[2020-01-28 09:32:55.603 (UTC+0100)] info/client        Rejecting endpoint 1: security mode doesn't match
[2020-01-28 09:32:55.603 (UTC+0100)] info/client        Endpoint 2 has 2 user token policies
[2020-01-28 09:32:55.603 (UTC+0100)] info/client        Selected Endpoint opc.tcp://127.0.0.1:4840 with SecurityMode SignAndEncrypt and SecurityPolicy http://opcfoundat
SecurityPolicy#Basic128Rsa15
[2020-01-28 09:32:55.603 (UTC+0100)] info/client        Selected UserTokenPolicy open62541-anonymous-policy with UserTokenType Anonymous and SecurityPolicy http://
opcfoundation.org/UA/SecurityPolicy#Basic128Rsa15
[2020-01-28 09:32:55.603 (UTC+0100)] info/client        Disconnect to switch to a different SecurityPolicy
[2020-01-28 09:32:55.603 (UTC+0100)] info/client        Connecting to endpoint opc.tcp://127.0.0.1:4840
[2020-01-28 09:32:55.603 (UTC+0100)] info/client        TCP connection established
[2020-01-28 09:32:55.629 (UTC+0100)] info/client        Opened SecureChannel with SecurityPolicy http://opcfoundation.org/UA/SecurityPolicy#Basic128Rsa15
[2020-01-28 09:32:55.640 (UTC+0100)] info/userland      date is: 28-1-2020 8:32:55.640

/home/pat/qt_projects/build-opc_ua_client-Desktop-Default/opc_ua_client exited with code 0
```

Add Hybrid Certificate Verification

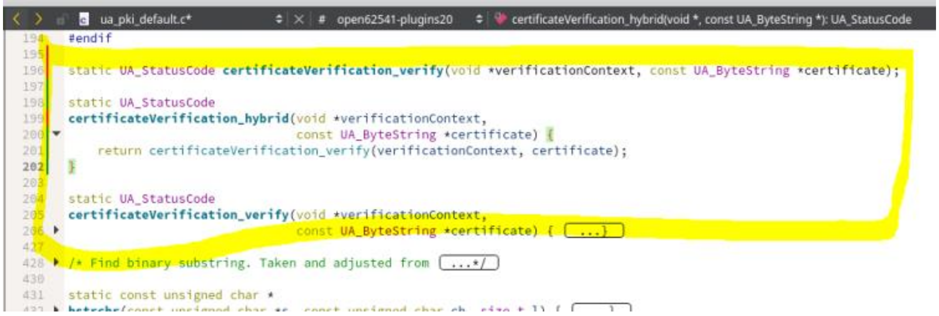
Dienstag, 28. Januar 2020 09:34

File: ua_pki_default.c



```c
clearVerifyAllowAll(UA_CertificateVerification *cv) { ...}

void UA_CertificateVerification_AcceptAll(UA_CertificateVerification *cv) { ...}

#ifdef UA_ENABLE_ENCRYPTION

typedef struct { ...
} CertInfo;

#ifdef __linux__ /* Linux only so far */

#include <dirent.h>
#include <limits.h>

static UA_StatusCode
fileNamesFromFolder(const UA_String *folder, size_t *pathsSize, UA_String **paths) { ...}

static UA_StatusCode
reloadCertificates(CertInfo *ci) { ...}

#endif

static UA_StatusCode
certificateVerification_verify(void *verificationContext,
                               const UA_ByteString *certificate) { ...}

/* Find binary substring. Taken and adjusted from ...*/

static const unsigned char *
bstrchr(const unsigned char *s, const unsigned char ch, size_t l) { ...}

static const unsigned char *
bstrstr(const unsigned char *s1, size_t l1, const unsigned char *s2, size_t l2) { ...}

static UA_StatusCode
```

This is the original verification function. Add another function:



```c
#endif

static UA_StatusCode certificateVerification_verify(void *verificationContext, const UA_ByteString *certificate);

static UA_StatusCode
certificateVerification_hybrid(void *verificationContext,
                               const UA_ByteString *certificate) {
    return certificateVerification_verify(verificationContext, certificate);
}

static UA_StatusCode
certificateVerification_verify(void *verificationContext,
                               const UA_ByteString *certificate) { ...}

/* Find binary substring. Taken and adjusted from ...*/

static const unsigned char *
```

This function so far just calls the original function. Note that the function prototype for the original function was added above.

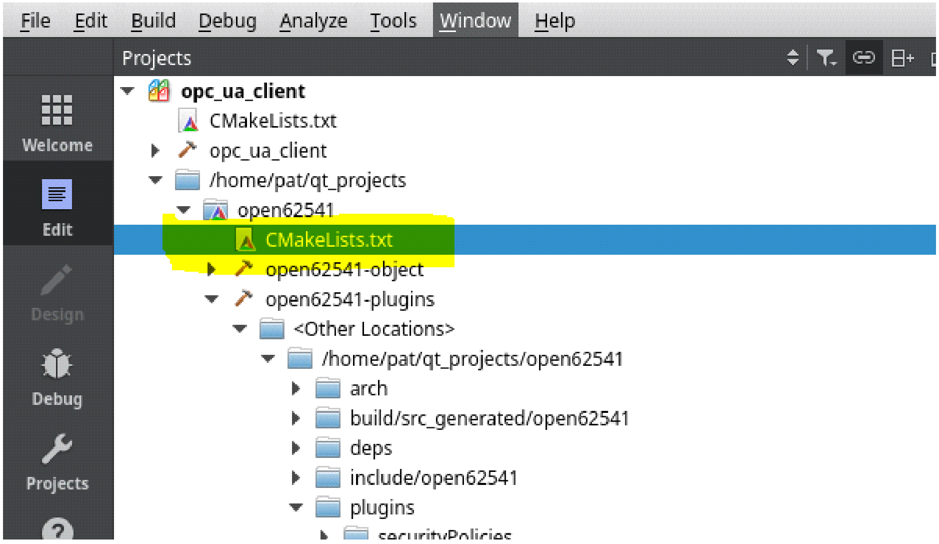
Go to the function UA_CertificateVerificationTrustList() and change the callback for the verifyCertificate function to the hybrid version

```
UA_StatusCode
UA_CertificateVerification_Trustlist(UA_CertificateVerification *cv,
                                      const UA_ByteString *certificateTrustList,
                                      size_t certificateTrustListSize,
                                      const UA_ByteString *certificateIssuerList,
                                      size_t certificateIssuerListSize,
                                      const UA_ByteString *certificateRevocationList,
                                      size_t certificateRevocationListSize) {
    CertInfo *ci = (CertInfo*)UA_malloc(sizeof(CertInfo));
    if(!ci)
        return UA_STATUSCODE_BADOUTOFMEMORY;
    memset(ci, 0, sizeof(CertInfo));
    mbedtls_x509_crt_init(&ci->certificateTrustList);
    mbedtls_x509_crl_init(&ci->certificateRevocationList);
    mbedtls_x509_crt_init(&ci->certificateIssuerList);

    cv->context = (void*)ci;
    if(certificateTrustListSize > 0)
        cv->verifyCertificate = certificateVerification_hybrid;
        //cv->verifyCertificate = certificateVerification_verify;
    else
        cv->verifyCertificate = verifyCertificateAllowAll;
    cv->clear = certificateVerification_clear;
    cv->verifyApplicationURI = certificateVerification_verifyApplicationURI;

    int err = 0;
    for(size_t i = 0; i < certificateTrustListSize; i++) { ...}
    for(size_t i = 0; i < certificateIssuerListSize; i++) { ...}
    for(size_t i = 0; i < certificateRevocationListSize; i++) { ...}

    return UA_STATUSCODE_GOOD;
error:
    certificateVerification_clear(cv);
```

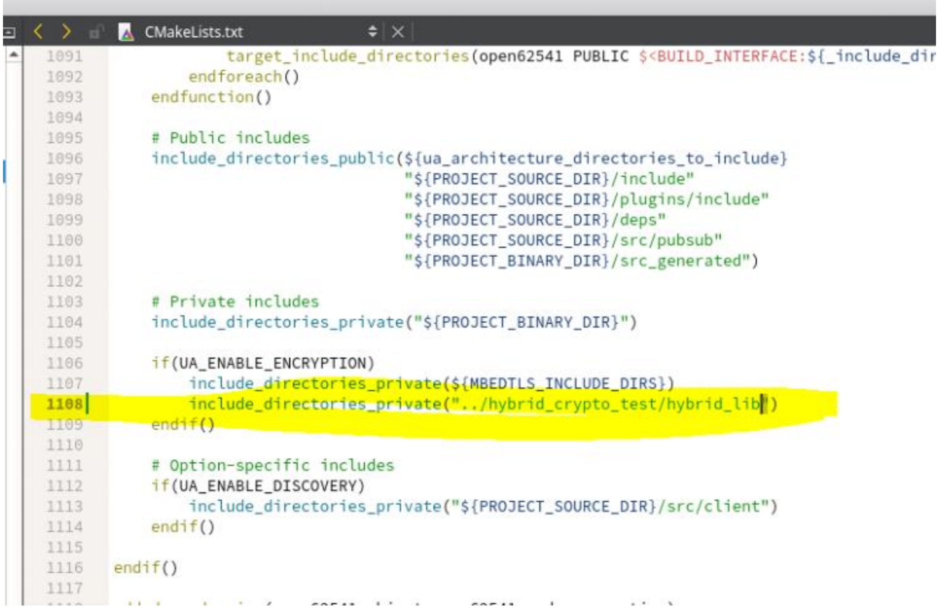
At this point you should be able to compile and run server and client successfully. You also can check with a debugger that actually the hybrid certificate verification function is called.

Adding the Hybrid Verification Logic

Edit the CMake file of the open62541 project



Add the line to include the hybrid crypto library into open62541

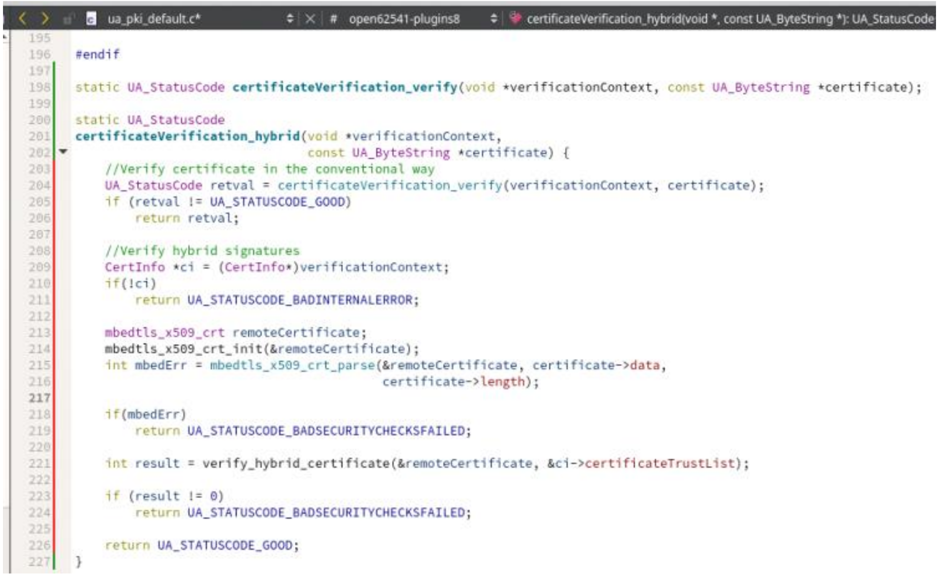


```
            target_include_directories(open62541 PUBLIC $<BUILD_INTERFACE:${_include_dir
        endforeach()
    endfunction()

    # Public includes
    include_directories_public(${ua_architecture_directories_to_include}
                               "${PROJECT_SOURCE_DIR}/include"
                               "${PROJECT_SOURCE_DIR}/plugins/include"
                               "${PROJECT_SOURCE_DIR}/deps"
                               "${PROJECT_SOURCE_DIR}/src/pubsub"
                               "${PROJECT_BINARY_DIR}/src_generated")

    # Private includes
    include_directories_private("${PROJECT_BINARY_DIR}")

    if(UA_ENABLE_ENCRYPTION)
        include_directories_private(${MBEDTLS_INCLUDE_DIRS})
        include_directories_private("../hybrid_crypto_test/hybrid_lib")
    endif()

    # Option-specific includes
    if(UA_ENABLE_DISCOVERY)
        include_directories_private("${PROJECT_SOURCE_DIR}/src/client")
    endif()

endif()
```

Then add the hybrid_crypto.h file to ua_pki_default.c. Make sure that the file is recognized by the IDE, otherwise something with the include path is wrong.

```
/* This work is licensed under a Creative Commons CCZero 1.0 Universal License. ...

#include <open62541/server_config.h>
#include <open62541/plugin/pki_default.h>
#include <open62541/plugin/log_stdout.h>

#include <hybrid_crypto.h>

#ifdef UA_ENABLE_ENCRYPTION
#include <mbedtls/x509.h>
#include <mbedtls/x509_crt.h>
#include <mbedtls/error.h>
#endif

#define REMOTECERTIFICATETRUSTED 1
#define ISSUERKNOWN              2
#define DUALPARENT               3
#define PARENTFOUND              4

/************/
```

Then add the code for the hybrid certificate verification

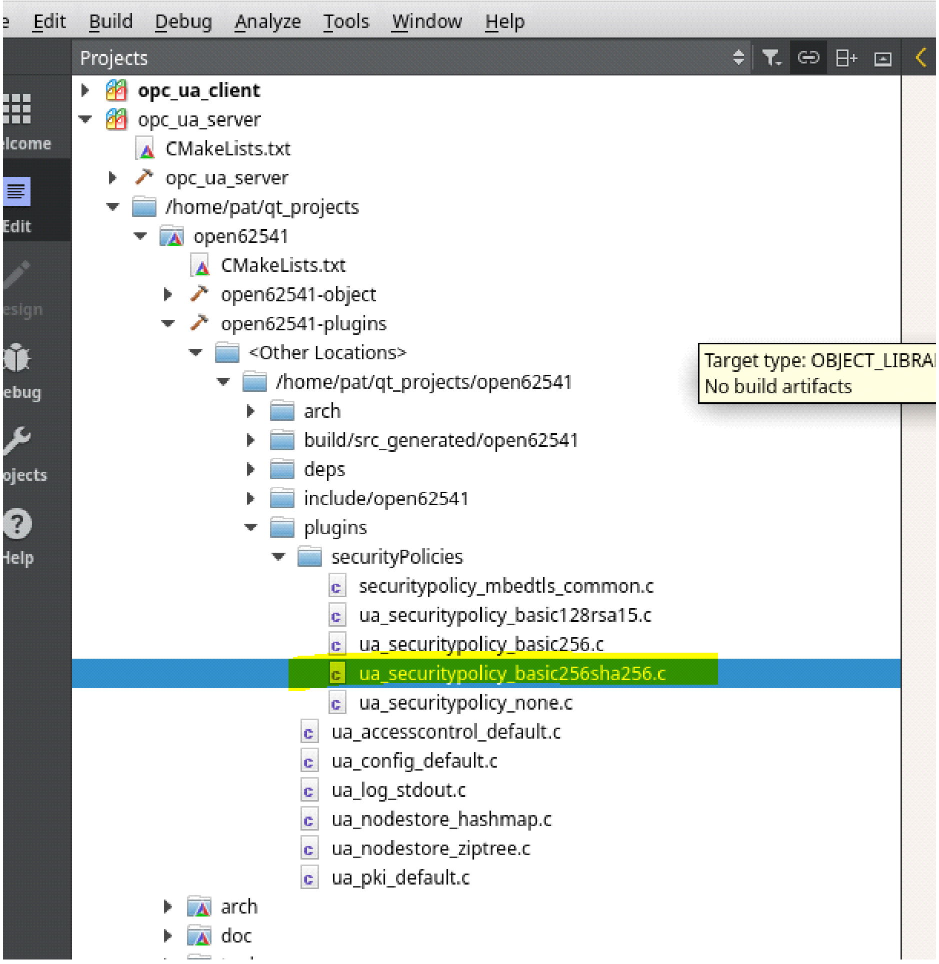


```
#endif

static UA_StatusCode certificateVerification_verify(void *verificationContext, const UA_ByteString *certificate);

static UA_StatusCode
certificateVerification_hybrid(void *verificationContext,
                               const UA_ByteString *certificate) {
    //Verify certificate in the conventional way
    UA_StatusCode retval = certificateVerification_verify(verificationContext, certificate);
    if (retval != UA_STATUSCODE_GOOD)
        return retval;

    //Verify hybrid signatures
    CertInfo *ci = (CertInfo*)verificationContext;
    if(!ci)
        return UA_STATUSCODE_BADINTERNALERROR;

    mbedtls_x509_crt remoteCertificate;
    mbedtls_x509_crt_init(&remoteCertificate);
    int mbedErr = mbedtls_x509_crt_parse(&remoteCertificate, certificate->data,
                                          certificate->length);

    if(mbedErr)
        return UA_STATUSCODE_BADSECURITYCHECKSFAILED;

    int result = verify_hybrid_certificate(&remoteCertificate, &ci->certificateTrustList);

    if (result != 0)
        return UA_STATUSCODE_BADSECURITYCHECKSFAILED;

    return UA_STATUSCODE_GOOD;
}
```

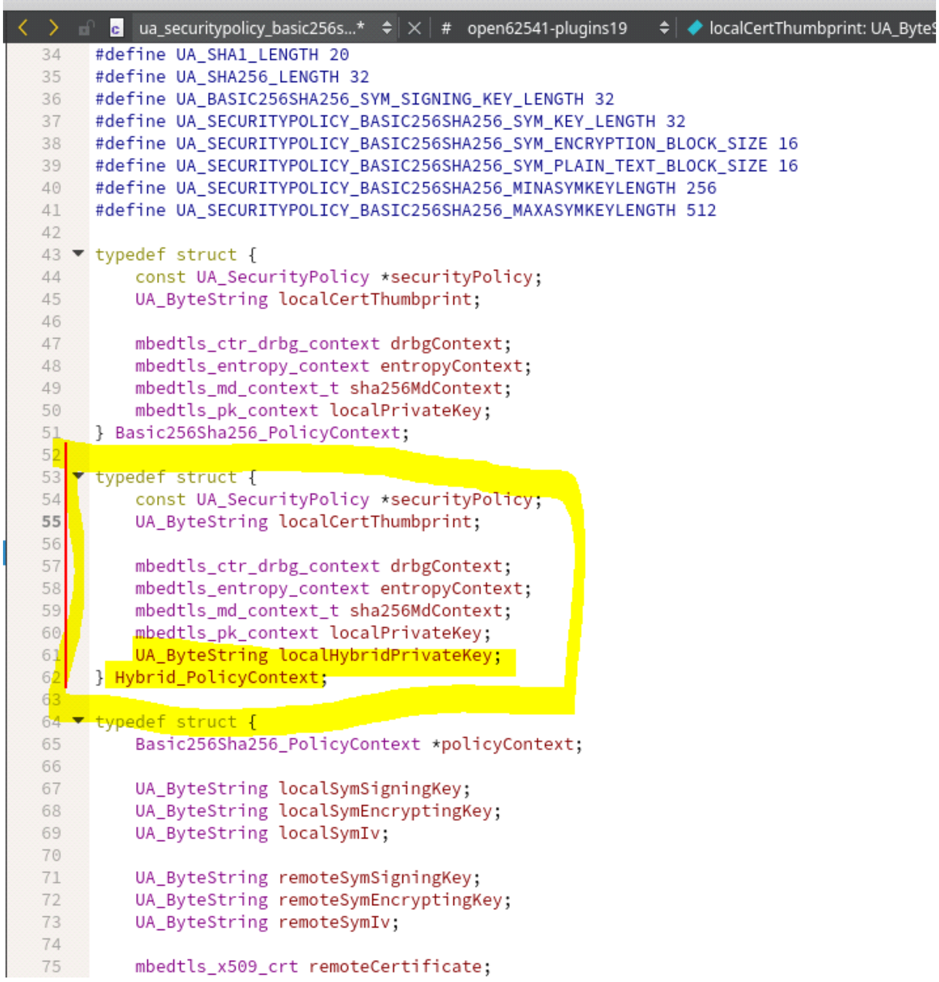
Compile and make sure everything runs fine.

Adding a New Security Policy

Dienstag, 28. Januar 2020 09:52

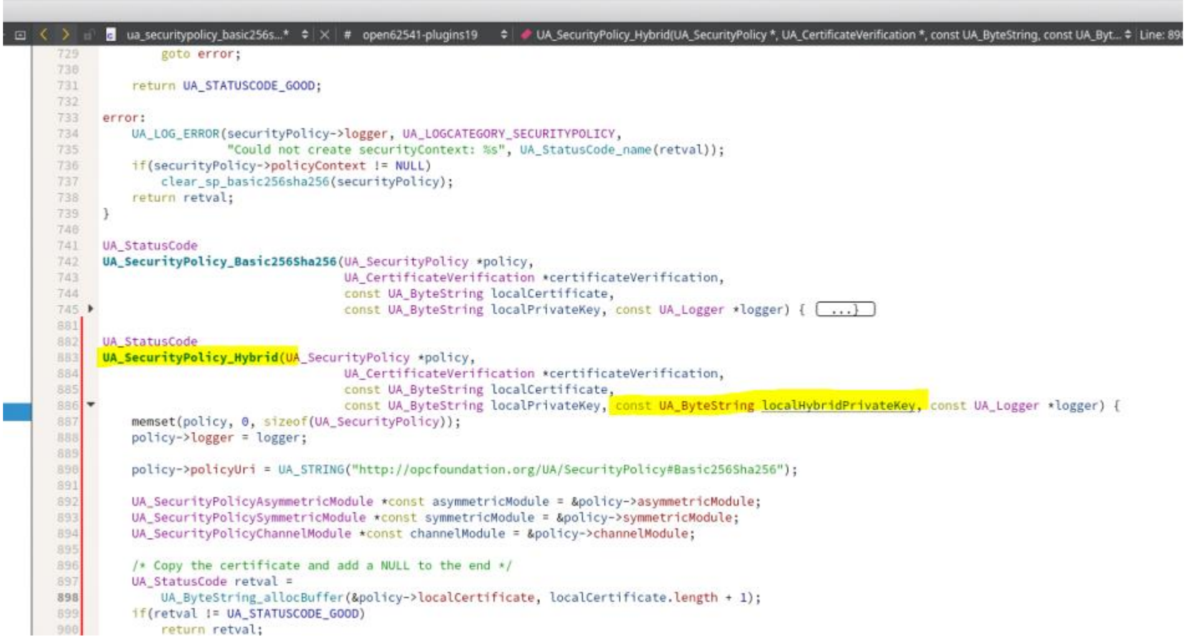


Copy the data structure BAsic256Sha256_PolicyContext and rename it to Hybrid_PolicyContext. Add the localHybridPrivateKey field.



```c
#define UA_SHA1_LENGTH 20
#define UA_SHA256_LENGTH 32
#define UA_BASIC256SHA256_SYM_SIGNING_KEY_LENGTH 32
#define UA_SECURITYPOLICY_BASIC256SHA256_SYM_KEY_LENGTH 32
#define UA_SECURITYPOLICY_BASIC256SHA256_SYM_ENCRYPTION_BLOCK_SIZE 16
#define UA_SECURITYPOLICY_BASIC256SHA256_SYM_PLAIN_TEXT_BLOCK_SIZE 16
#define UA_SECURITYPOLICY_BASIC256SHA256_MINASYMKEYLENGTH 256
#define UA_SECURITYPOLICY_BASIC256SHA256_MAXASYMKEYLENGTH 512

typedef struct {
    const UA_SecurityPolicy *securityPolicy;
    UA_ByteString localCertThumbprint;

    mbedtls_ctr_drbg_context drbgContext;
    mbedtls_entropy_context entropyContext;
    mbedtls_md_context_t sha256MdContext;
    mbedtls_pk_context localPrivateKey;
} Basic256Sha256_PolicyContext;

typedef struct {
    const UA_SecurityPolicy *securityPolicy;
    UA_ByteString localCertThumbprint;

    mbedtls_ctr_drbg_context drbgContext;
    mbedtls_entropy_context entropyContext;
    mbedtls_md_context_t sha256MdContext;
    mbedtls_pk_context localPrivateKey;
    UA_ByteString localHybridPrivateKey;
} Hybrid_PolicyContext;

typedef struct {
    Basic256Sha256_PolicyContext *policyContext;

    UA_ByteString localSymSigningKey;
    UA_ByteString localSymEncryptingKey;
    UA_ByteString localSymIv;

    UA_ByteString remoteSymSigningKey;
    UA_ByteString remoteSymEncryptingKey;
    UA_ByteString remoteSymIv;

    mbedtls_x509_crt remoteCertificate;
```

Copy the function UA_SecurityPolicy_Basic256Sha256 and rename it to UA_SecurityPolicy_Hybrid. Add a parameter for the hybrid private key.

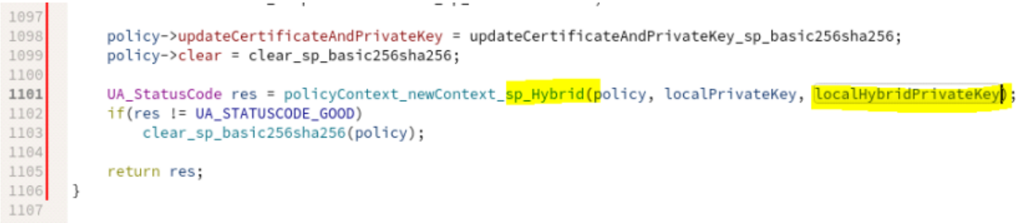


```c
        goto error;

    return UA_STATUSCODE_GOOD;

error:
    UA_LOG_ERROR(securityPolicy->logger, UA_LOGCATEGORY_SECURITYPOLICY,
                 "Could not create securityContext: %s", UA_StatusCode_name(retval));
    if(securityPolicy->policyContext != NULL)
        clear_sp_basic256sha256(securityPolicy);
    return retval;
}

UA_StatusCode
UA_SecurityPolicy_Basic256Sha256(UA_SecurityPolicy *policy,
                                 UA_CertificateVerification *certificateVerification,
                                 const UA_ByteString localCertificate,
                                 const UA_ByteString localPrivateKey, const UA_Logger *logger) { ...}

UA_StatusCode
UA_SecurityPolicy_Hybrid(UA_SecurityPolicy *policy,
                                 UA_CertificateVerification *certificateVerification,
                                 const UA_ByteString localCertificate,
                                 const UA_ByteString localPrivateKey, const UA_ByteString localHybridPrivateKey, const UA_Logger *logger) {
    memset(policy, 0, sizeof(UA_SecurityPolicy));
    policy->logger = logger;

    policy->policyUri = UA_STRING("http://opcfoundation.org/UA/SecurityPolicy#Basic256Sha256");

    UA_SecurityPolicyAsymmetricModule *const asymmetricModule = &policy->asymmetricModule;
    UA_SecurityPolicySymmetricModule *const symmetricModule = &policy->symmetricModule;
    UA_SecurityPolicyChannelModule *const channelModule = &policy->channelModule;

    /* Copy the certificate and add a NULL to the end */
    UA_StatusCode retval =
        UA_ByteString_allocBuffer(&policy->localCertificate, localCertificate.length + 1);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
```

Copy the function policyContext_newContext_sp_basic256sha256 and rename it to policyContext_newContext_sp_Hybrid(). Change the context datatype in the function. Add a parameter for the hybrid private key and assign the key to the data object.

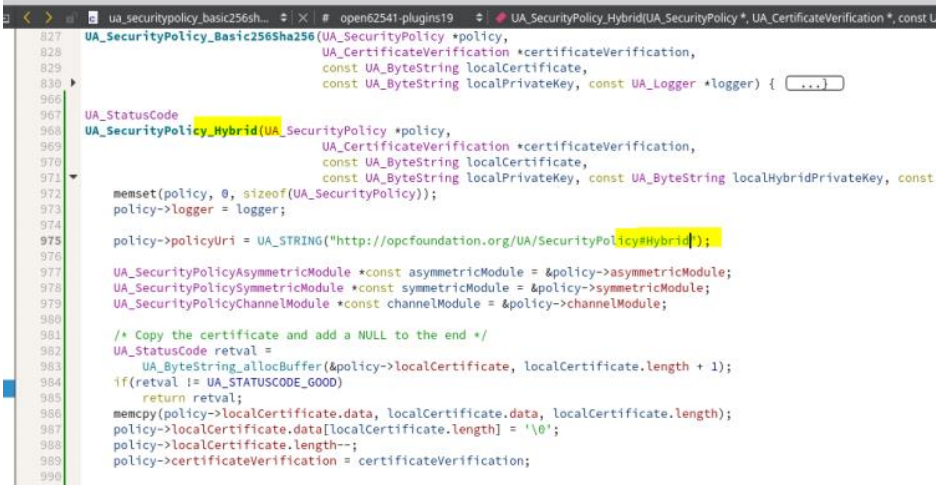
```c
static void
clear_sp_basic256sha256(UA_SecurityPolicy *securityPolicy) { ...}

static UA_StatusCode
updateCertificateAndPrivateKey_sp_basic256sha256(UA_SecurityPolicy *securityPolicy,
                                                 const UA_ByteString newCertificate,
                                                 const UA_ByteString newPrivateKey) { ...}

static UA_StatusCode
policyContext_newContext_sp_basic256sha256(UA_SecurityPolicy *securityPolicy,
                                           const UA_ByteString localPrivateKey) { ...}

static UA_StatusCode
policyContext_newContext_sp_Hybrid(UA_SecurityPolicy *securityPolicy,
                                           const UA_ByteString localPrivateKey, const UA_ByteString localHybridPrivateKey) {
    UA_StatusCode retval = UA_STATUSCODE_GOOD;
    if(securityPolicy == NULL)
        return UA_STATUSCODE_BADINTERNALERROR;

    if (localPrivateKey.length == 0) {
        UA_LOG_ERROR(securityPolicy->logger, UA_LOGCATEGORY_SECURITYPOLICY,
                     "Can not initialize security policy. Private key is empty.");
        return UA_STATUSCODE_BADINVALIDARGUMENT;
    }

    Hybrid_PolicyContext *pc = (Hybrid_PolicyContext *)
        UA_malloc(sizeof(Hybrid_PolicyContext));
    securityPolicy->policyContext = (void *)pc;
    if(!pc) {
        retval = UA_STATUSCODE_BADOUTOFMEMORY;
        goto error;
    }

    /* Initialize the PolicyContext */
    memset(pc, 0, sizeof(Hybrid_PolicyContext));
    mbedtls_ctr_drbg_init(&pc->drbgContext);
    mbedtls_entropy_init(&pc->entropyContext);
    mbedtls_pk_init(&pc->localPrivateKey);
    mbedtls_md_init(&pc->sha256MdContext);
    pc->securityPolicy = securityPolicy;
    pc->localHybridPrivateKey = localHybridPrivateKey;

    /* Initialized the message digest */
    const mbedtls_md_info_t *const mdInfo = mbedtls_md_info_from_type(MBEDTLS_MD_SHA256);
    int mbedErr = mbedtls_md_setup(&pc->sha256MdContext, mdInfo, MBEDTLS_MD_SHA256);
    if(mbedErr) {
        retval = UA_STATUSCODE_BADOUTOFMEMORY;
```

At the end of the function UA_SecurityPolicy_Hybrid, change the function call to call the hybrid policy context. Also pass the hybrid private key.



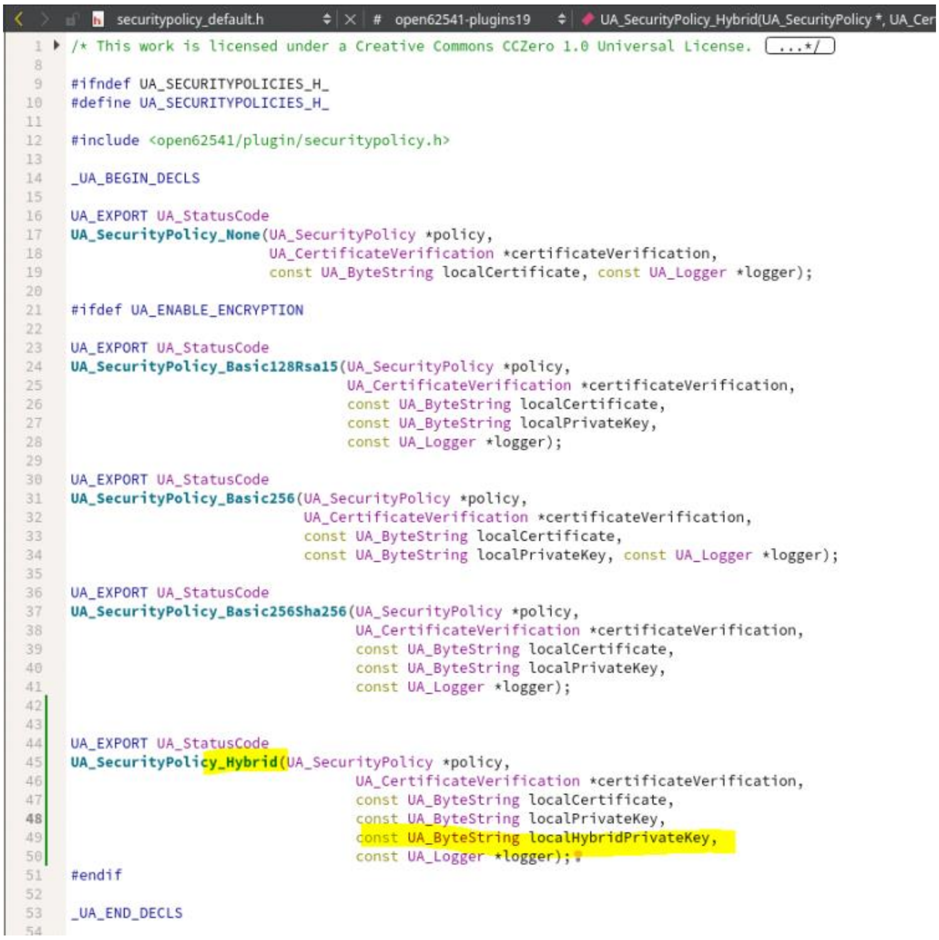
```c
    policy->updateCertificateAndPrivateKey = updateCertificateAndPrivateKey_sp_basic256sha256;
    policy->clear = clear_sp_basic256sha256;

    UA_StatusCode res = policyContext_newContext_sp_Hybrid(policy, localPrivateKey, localHybridPrivateKey);
    if(res != UA_STATUSCODE_GOOD)
        clear_sp_basic256sha256(policy);

    return res;
}
```

Finally change the URI of the new security policy



```c
UA_SecurityPolicy_Basic256Sha256(UA_SecurityPolicy *policy,
                                 UA_CertificateVerification *certificateVerification,
                                 const UA_ByteString localCertificate,
                                 const UA_ByteString localPrivateKey, const UA_Logger *logger) { ...}

UA_StatusCode
UA_SecurityPolicy_Hybrid(UA_SecurityPolicy *policy,
                                 UA_CertificateVerification *certificateVerification,
                                 const UA_ByteString localCertificate,
                                 const UA_ByteString localPrivateKey, const UA_ByteString localHybridPrivateKey, const
    memset(policy, 0, sizeof(UA_SecurityPolicy));
    policy->logger = logger;

    policy->policyUri = UA_STRING("http://opcfoundation.org/UA/SecurityPolicy#Hybrid");

    UA_SecurityPolicyAsymmetricModule *const asymmetricModule = &policy->asymmetricModule;
    UA_SecurityPolicySymmetricModule *const symmetricModule = &policy->symmetricModule;
    UA_SecurityPolicyChannelModule *const channelModule = &policy->channelModule;

    /* Copy the certificate and add a NULL to the end */
    UA_StatusCode retval =
        UA_ByteString_allocBuffer(&policy->localCertificate, localCertificate.length + 1);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    memcpy(policy->localCertificate.data, localCertificate.data, localCertificate.length);
    policy->localCertificate.data[localCertificate.length] = '\0';
    policy->localCertificate.length--;
    policy->certificateVerification = certificateVerification;
```

Go to the header file securitypolicy_default.h and add the function prototype for the hybrid security policy

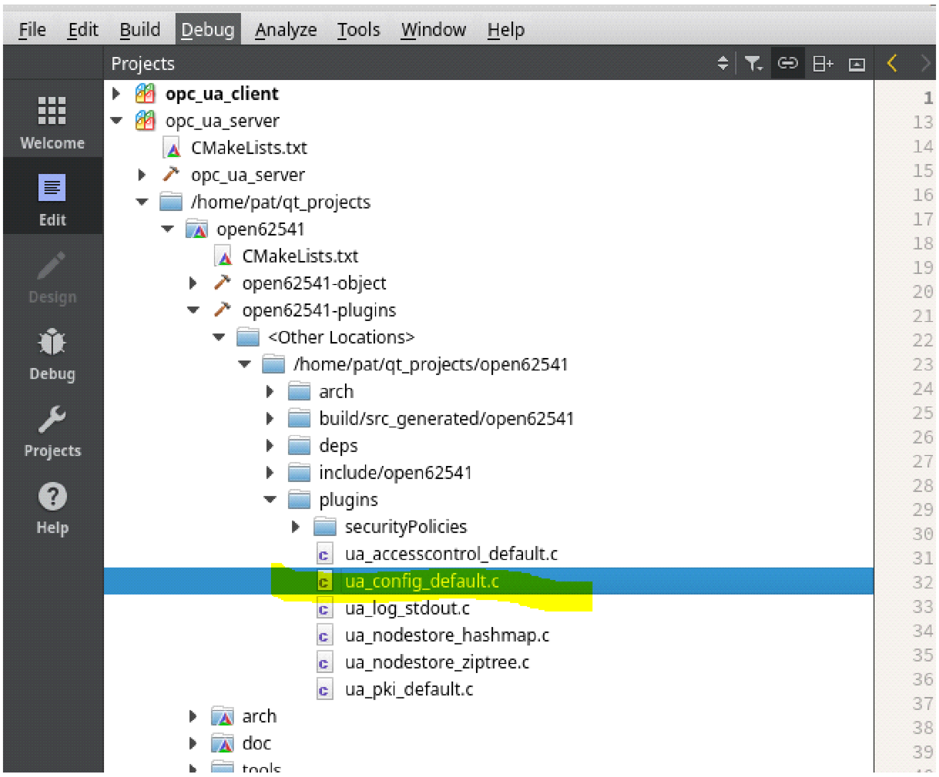


```
/* This work is licensed under a Creative Commons CCZero 1.0 Universal License. ...*/

#ifndef UA_SECURITYPOLICIES_H_
#define UA_SECURITYPOLICIES_H_

#include <open62541/plugin/securitypolicy.h>

_UA_BEGIN_DECLS

UA_EXPORT UA_StatusCode
UA_SecurityPolicy_None(UA_SecurityPolicy *policy,
                       UA_CertificateVerification *certificateVerification,
                       const UA_ByteString localCertificate, const UA_Logger *logger);

#ifdef UA_ENABLE_ENCRYPTION

UA_EXPORT UA_StatusCode
UA_SecurityPolicy_Basic128Rsa15(UA_SecurityPolicy *policy,
                                UA_CertificateVerification *certificateVerification,
                                const UA_ByteString localCertificate,
                                const UA_ByteString localPrivateKey,
                                const UA_Logger *logger);

UA_EXPORT UA_StatusCode
UA_SecurityPolicy_Basic256(UA_SecurityPolicy *policy,
                           UA_CertificateVerification *certificateVerification,
                           const UA_ByteString localCertificate,
                           const UA_ByteString localPrivateKey, const UA_Logger *logger);

UA_EXPORT UA_StatusCode
UA_SecurityPolicy_Basic256Sha256(UA_SecurityPolicy *policy,
                                 UA_CertificateVerification *certificateVerification,
                                 const UA_ByteString localCertificate,
                                 const UA_ByteString localPrivateKey,
                                 const UA_Logger *logger);


UA_EXPORT UA_StatusCode
UA_SecurityPolicy_Hybrid(UA_SecurityPolicy *policy,
                                 UA_CertificateVerification *certificateVerification,
                                 const UA_ByteString localCertificate,
                                 const UA_ByteString localPrivateKey,
                                 const UA_ByteString localHybridPrivateKey,
                                 const UA_Logger *logger);
#endif

_UA_END_DECLS
```

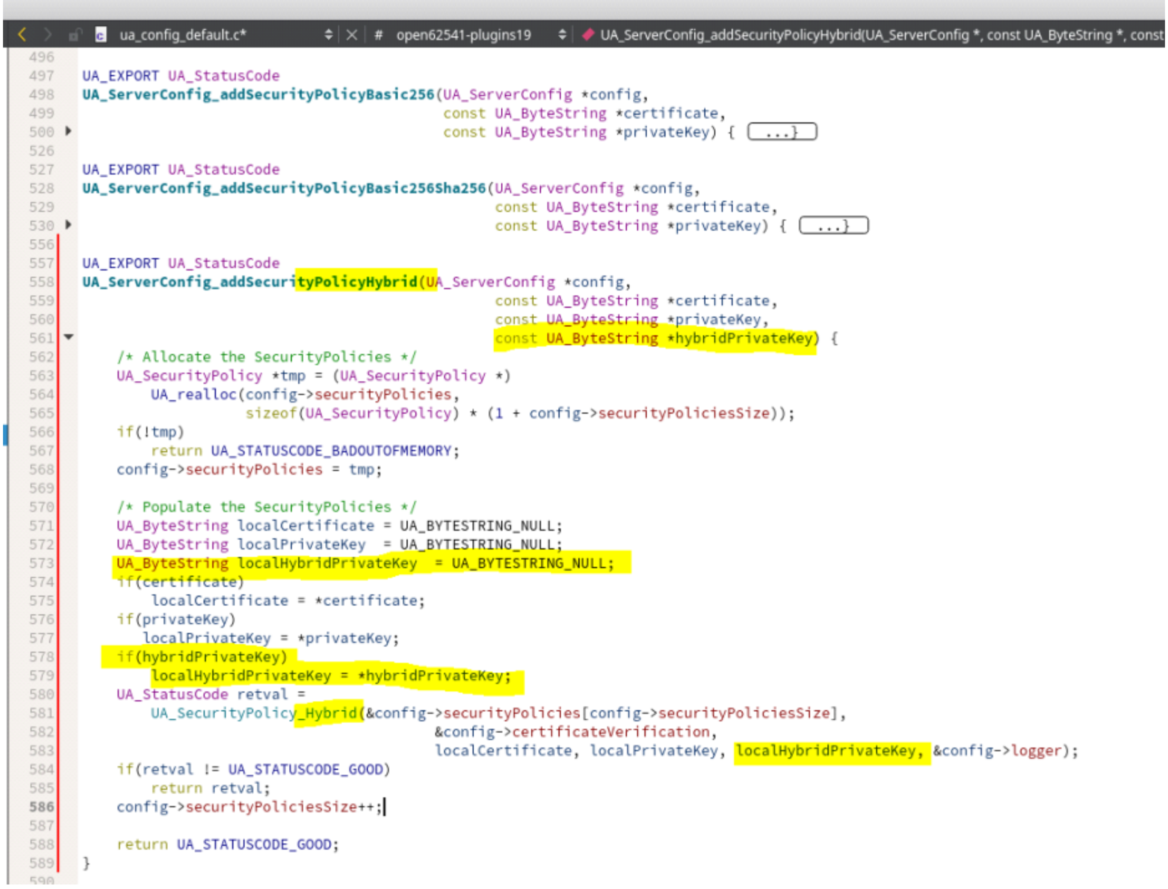
So far we have created a new security policy that is a copy of basic256sha256, but has the name Hybrid and accepts an additional hybrid private key as a parameter.

Adding the security Policy to Server and Client Config

Go to the file ua_config_default.c

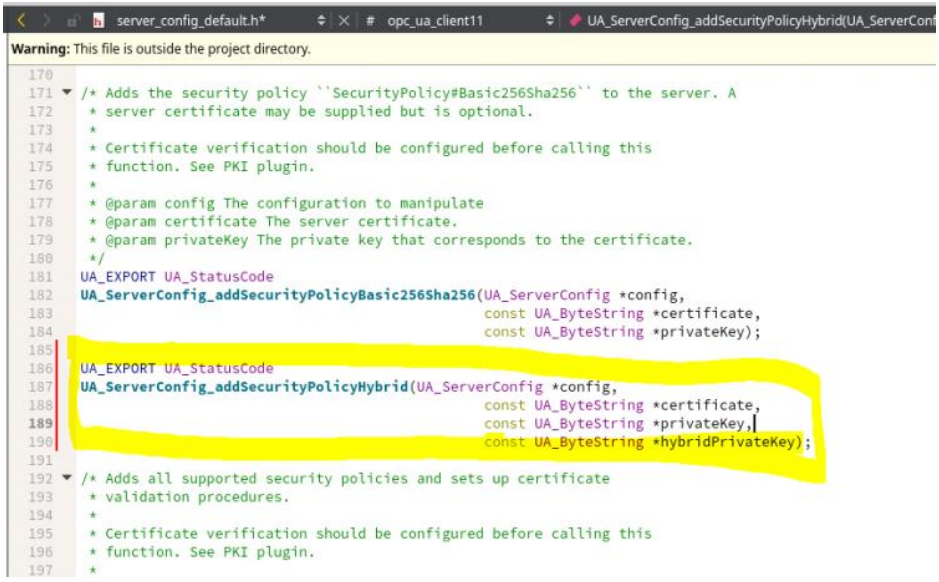


Look for the function UA_ServerConfig_addSecurityPolicyBasic256Sha256, copy it and rename it to UA_ServerConfig_addSecurityPolicyHybrid(). Add a parameter for the hybrid private key. Add a variable for the hybrid private key. Assign it to the variable. Change the function call to a Hybrid policy and add the new parameter to the function call.



```
UA_EXPORT UA_StatusCode
UA_ServerConfig_addSecurityPolicyBasic256(UA_ServerConfig *config,
                                          const UA_ByteString *certificate,
                                          const UA_ByteString *privateKey) { ...}

UA_EXPORT UA_StatusCode
UA_ServerConfig_addSecurityPolicyBasic256Sha256(UA_ServerConfig *config,
                                                const UA_ByteString *certificate,
                                                const UA_ByteString *privateKey) { ...}

UA_EXPORT UA_StatusCode
UA_ServerConfig_addSecurityPolicyHybrid(UA_ServerConfig *config,
                                                const UA_ByteString *certificate,
                                                const UA_ByteString *privateKey,
                                                const UA_ByteString *hybridPrivateKey) {
    /* Allocate the SecurityPolicies */
    UA_SecurityPolicy *tmp = (UA_SecurityPolicy *)
        UA_realloc(config->securityPolicies,
                   sizeof(UA_SecurityPolicy) * (1 + config->securityPoliciesSize));
    if(!tmp)
        return UA_STATUSCODE_BADOUTOFMEMORY;
    config->securityPolicies = tmp;

    /* Populate the SecurityPolicies */
    UA_ByteString localCertificate = UA_BYTESTRING_NULL;
    UA_ByteString localPrivateKey  = UA_BYTESTRING_NULL;
    UA_ByteString localHybridPrivateKey  = UA_BYTESTRING_NULL;
    if(certificate)
        localCertificate = *certificate;
    if(privateKey)
       localPrivateKey = *privateKey;
    if(hybridPrivateKey)
        localHybridPrivateKey = *hybridPrivateKey;
    UA_StatusCode retval =
        UA_SecurityPolicy_Hybrid(&config->securityPolicies[config->securityPoliciesSize],
                                 &config->certificateVerification,
                                 localCertificate, localPrivateKey, localHybridPrivateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    config->securityPoliciesSize++;

    return UA_STATUSCODE_GOOD;
}
```

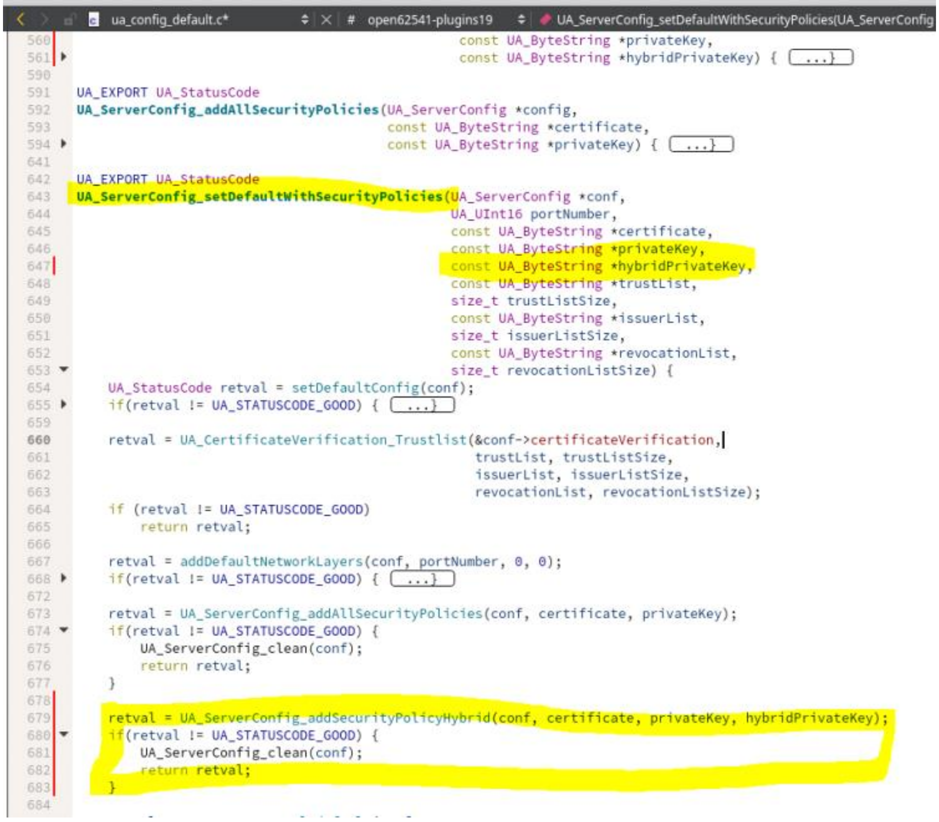
Add the new function to the header file server_config_default.h



Warning: This file is outside the project directory.

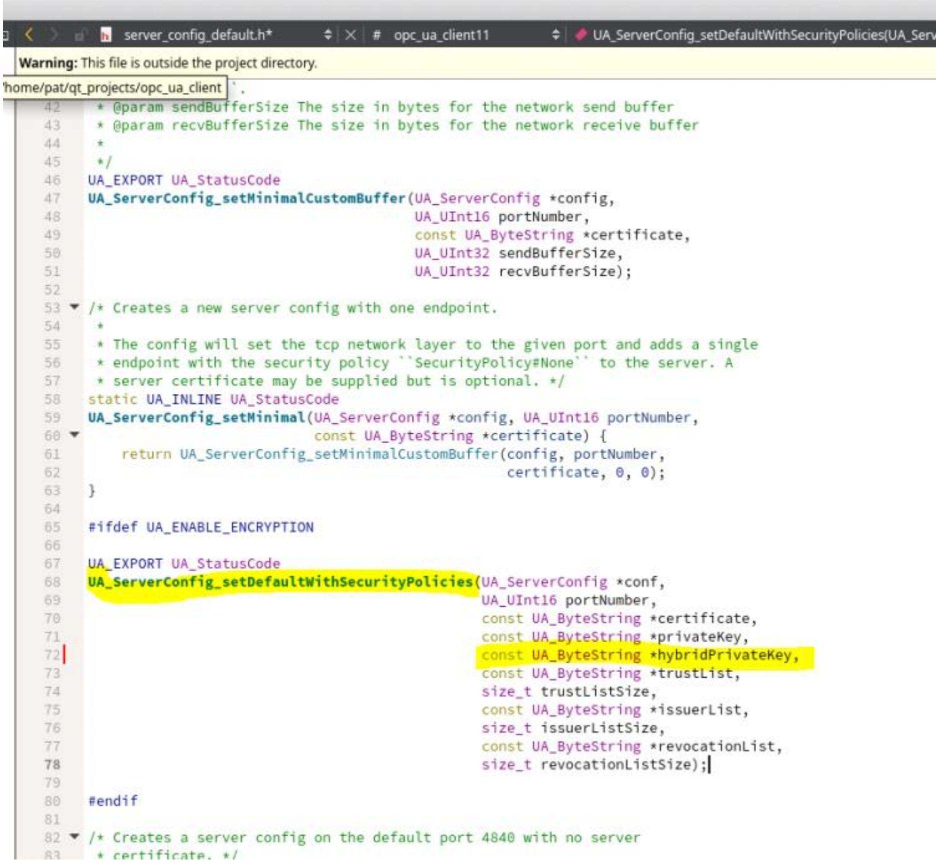
```
/* Adds the security policy ``SecurityPolicy#Basic256Sha256`` to the server. A
 * server certificate may be supplied but is optional.
 *
 * Certificate verification should be configured before calling this
 * function. See PKI plugin.
 *
 * @param config The configuration to manipulate
 * @param certificate The server certificate.
 * @param privateKey The private key that corresponds to the certificate.
 */
UA_EXPORT UA_StatusCode
UA_ServerConfig_addSecurityPolicyBasic256Sha256(UA_ServerConfig *config,
                                                const UA_ByteString *certificate,
                                                const UA_ByteString *privateKey);

UA_EXPORT UA_StatusCode
UA_ServerConfig_addSecurityPolicyHybrid(UA_ServerConfig *config,
                                                const UA_ByteString *certificate,
                                                const UA_ByteString *privateKey,
                                                const UA_ByteString *hybridPrivateKey);

/* Adds all supported security policies and sets up certificate
 * validation procedures.
 *
 * Certificate verification should be configured before calling this
 * function. See PKI plugin.
 *
```

Then edit the function UA_ServerConfig_setDefaultWithSecurityPolicies().
Add a parameter for the hybrid private key.
After the call to UA_serverConfig_addAllSecurityPolicies() call the newly created function



```
                                                const UA_ByteString *privateKey,
                                                const UA_ByteString *hybridPrivateKey) { ...}

UA_EXPORT UA_StatusCode
UA_ServerConfig_addAllSecurityPolicies(UA_ServerConfig *config,
                                       const UA_ByteString *certificate,
                                       const UA_ByteString *privateKey) { ...}

UA_EXPORT UA_StatusCode
UA_ServerConfig_setDefaultWithSecurityPolicies(UA_ServerConfig *conf,
                                               UA_UInt16 portNumber,
                                               const UA_ByteString *certificate,
                                               const UA_ByteString *privateKey,
                                               const UA_ByteString *hybridPrivateKey,
                                               const UA_ByteString *trustList,
                                               size_t trustListSize,
                                               const UA_ByteString *issuerList,
                                               size_t issuerListSize,
                                               const UA_ByteString *revocationList,
                                               size_t revocationListSize) {
    UA_StatusCode retval = setDefaultConfig(conf);
    if(retval != UA_STATUSCODE_GOOD) { ...}

    retval = UA_CertificateVerification_Trustlist(&conf->certificateVerification,
                                                  trustList, trustListSize,
                                                  issuerList, issuerListSize,
                                                  revocationList, revocationListSize);

    if (retval != UA_STATUSCODE_GOOD)
        return retval;

    retval = addDefaultNetworkLayers(conf, portNumber, 0, 0);
    if(retval != UA_STATUSCODE_GOOD) { ...}

    retval = UA_ServerConfig_addAllSecurityPolicies(conf, certificate, privateKey);
    if(retval != UA_STATUSCODE_GOOD) {
        UA_ServerConfig_clean(conf);
        return retval;
    }

    retval = UA_ServerConfig_addSecurityPolicyHybrid(conf, certificate, privateKey, hybridPrivateKey);
    if(retval != UA_STATUSCODE_GOOD) {
        UA_ServerConfig_clean(conf);
        return retval;
    }
```

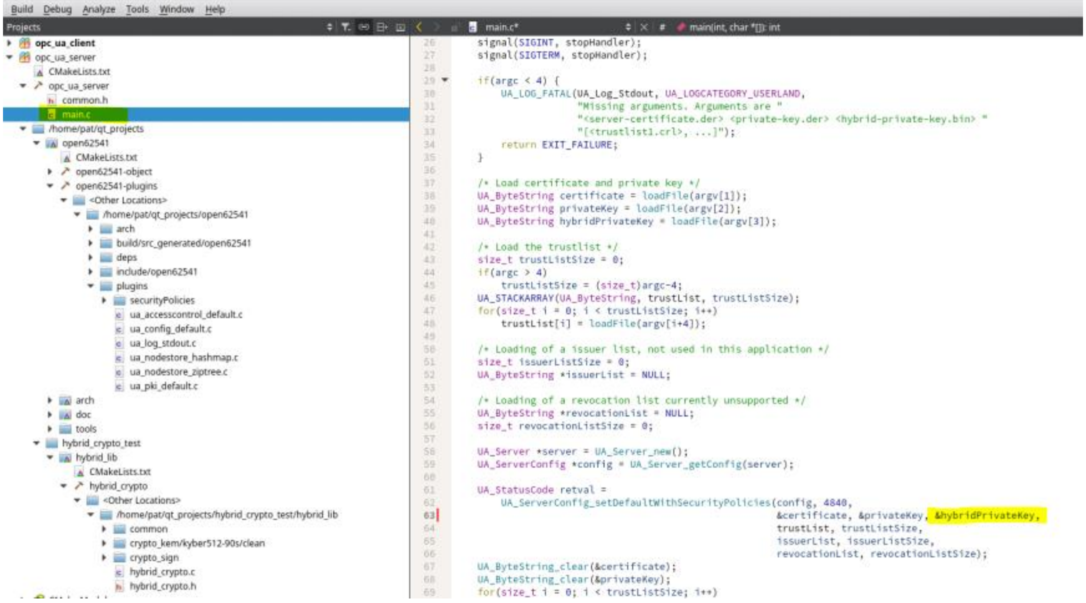
Add the additional parameter to the header file server_config_default.h



Warning: This file is outside the project directory.

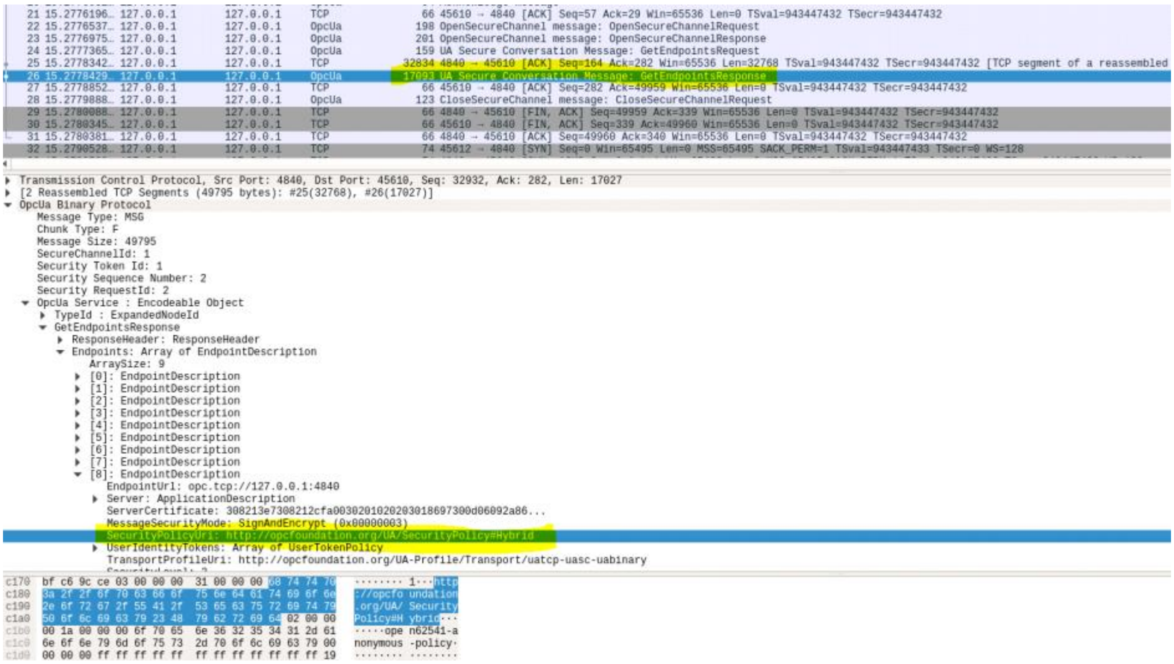
```c
 * @param sendBufferSize The size in bytes for the network send buffer
 * @param recvBufferSize The size in bytes for the network receive buffer
 *
 */
UA_EXPORT UA_StatusCode
UA_ServerConfig_setMinimalCustomBuffer(UA_ServerConfig *config,
                                       UA_UInt16 portNumber,
                                       const UA_ByteString *certificate,
                                       UA_UInt32 sendBufferSize,
                                       UA_UInt32 recvBufferSize);

/* Creates a new server config with one endpoint.
 *
 * The config will set the tcp network layer to the given port and adds a single
 * endpoint with the security policy ``SecurityPolicy#None`` to the server. A
 * server certificate may be supplied but is optional. */
static UA_INLINE UA_StatusCode
UA_ServerConfig_setMinimal(UA_ServerConfig *config, UA_UInt16 portNumber,
                           const UA_ByteString *certificate) {
    return UA_ServerConfig_setMinimalCustomBuffer(config, portNumber,
                                                  certificate, 0, 0);
}

#ifdef UA_ENABLE_ENCRYPTION

UA_EXPORT UA_StatusCode
UA_ServerConfig_setDefaultWithSecurityPolicies(UA_ServerConfig *conf,
                                               UA_UInt16 portNumber,
                                               const UA_ByteString *certificate,
                                               const UA_ByteString *privateKey,
                                               const UA_ByteString *hybridPrivateKey,
                                               const UA_ByteString *trustList,
                                               size_t trustListSize,
                                               const UA_ByteString *issuerList,
                                               size_t issuerListSize,
                                               const UA_ByteString *revocationList,
                                               size_t revocationListSize);

#endif

/* Creates a server config on the default port 4840 with no server
 * certificate. */
```

Finally go to the server main.c file and add the new parameter to the function call



```c
    signal(SIGINT, stopHandler);
    signal(SIGTERM, stopHandler);

    if(argc < 4) {
        UA_LOG_FATAL(UA_Log_Stdout, UA_LOGCATEGORY_USERLAND,
                     "Missing arguments. Arguments are "
                     "<server-certificate.der> <private-key.der> <hybrid-private-key.bin> "
                     "[<trustlist1.crl>, ...]");
        return EXIT_FAILURE;
    }

    /* Load certificate and private key */
    UA_ByteString certificate = loadFile(argv[1]);
    UA_ByteString privateKey = loadFile(argv[2]);
    UA_ByteString hybridPrivateKey = loadFile(argv[3]);

    /* Load the trustlist */
    size_t trustListSize = 0;
    if(argc > 4)
        trustListSize = (size_t)argc-4;
    UA_STACKARRAY(UA_ByteString, trustList, trustListSize);
    for(size_t i = 0; i < trustListSize; i++)
        trustList[i] = loadFile(argv[i+4]);

    /* Loading of a issuer list, not used in this application */
    size_t issuerListSize = 0;
    UA_ByteString *issuerList = NULL;

    /* Loading of a revocation list currently unsupported */
    UA_ByteString *revocationList = NULL;
    size_t revocationListSize = 0;

    UA_Server *server = UA_Server_new();
    UA_ServerConfig *config = UA_Server_getConfig(server);

    UA_StatusCode retval =
        UA_ServerConfig_setDefaultWithSecurityPolicies(config, 4840,
                                                       &certificate, &privateKey, &hybridPrivateKey,
                                                       trustList, trustListSize,
                                                       issuerList, issuerListSize,
                                                       revocationList, revocationListSize);

    UA_ByteString_clear(&certificate);
    UA_ByteString_clear(&privateKey);
    for(size_t i = 0; i < trustListSize; i++)
```

Now you can compile and run server and client. Note that the new security policy so far is only configure at the server and not at the client. The client will use another default security policy to connect. However when you check with wireshark you can see that the server offers the new hybrid policy.



```
 21 15.2776196… 127.0.0.1   127.0.0.1   TCP      66 45610 → 4840 [ACK] Seq=57 Ack=29 Win=65536 Len=0 TSval=943447432 TSecr=943447432
 22 15.2776537… 127.0.0.1   127.0.0.1   OpcUa   198 OpenSecureChannel message: OpenSecureChannelRequest
 23 15.2776975… 127.0.0.1   127.0.0.1   OpcUa   201 OpenSecureChannel message: OpenSecureChannelResponse
 24 15.2777365… 127.0.0.1   127.0.0.1   OpcUa   159 UA Secure Conversation Message: GetEndpointsRequest
 25 15.2778342… 127.0.0.1   127.0.0.1   TCP   32834 4840 → 45610 [ACK] Seq=164 Ack=282 Win=65536 Len=32768 TSval=943447432 TSecr=943447432 [TCP segment of a reassembled
 26 15.2778429… 127.0.0.1   127.0.0.1   OpcUa 17093 UA Secure Conversation Message: GetEndpointsResponse
 27 15.2778852… 127.0.0.1   127.0.0.1   TCP      66 45610 → 4840 [ACK] Seq=282 Ack=49959 Win=65536 Len=0 TSval=943447432 TSecr=943447432
 28 15.2779888… 127.0.0.1   127.0.0.1   OpcUa   123 CloseSecureChannel message: CloseSecureChannelRequest
 29 15.2780088… 127.0.0.1   127.0.0.1   TCP      66 4840 → 45610 [FIN, ACK] Seq=49959 Ack=339 Win=65536 Len=0 TSval=943447432 TSecr=943447432
 30 15.2780345… 127.0.0.1   127.0.0.1   TCP      66 45610 → 4840 [FIN, ACK] Seq=339 Ack=49960 Win=65536 Len=0 TSval=943447432 TSecr=943447432
 31 15.2780381… 127.0.0.1   127.0.0.1   TCP      66 4840 → 45610 [ACK] Seq=49960 Ack=340 Win=65536 Len=0 TSval=943447432 TSecr=943447432
 32 15.2790528… 127.0.0.1   127.0.0.1   TCP      74 45612 → 4840 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=943447433 TSecr=0 WS=128

▸ Transmission Control Protocol, Src Port: 4840, Dst Port: 45610, Seq: 32932, Ack: 282, Len: 17027
▸ [2 Reassembled TCP Segments (49795 bytes): #25(32768), #26(17027)]
▾ OpcUa Binary Protocol
    Message Type: MSG
    Chunk Type: F
    Message Size: 49795
    SecureChannelId: 1
    Security Token Id: 1
    Security Sequence Number: 2
    Security RequestId: 2
  ▾ OpcUa Service : Encodeable Object
    ▸ TypeId : ExpandedNodeId
    ▾ GetEndpointsResponse
      ▸ ResponseHeader: ResponseHeader
      ▾ Endpoints: Array of EndpointDescription
          ArraySize: 9
        ▸ [0]: EndpointDescription
        ▸ [1]: EndpointDescription
        ▸ [2]: EndpointDescription
        ▸ [3]: EndpointDescription
        ▸ [4]: EndpointDescription
        ▸ [5]: EndpointDescription
        ▸ [6]: EndpointDescription
        ▸ [7]: EndpointDescription
        ▾ [8]: EndpointDescription
            EndpointUrl: opc.tcp://127.0.0.1:4840
          ▸ Server: ApplicationDescription
            ServerCertificate: 308213e7308212cfa003020102020301869730 0d06092a86...
            MessageSecurityMode: SignAndEncrypt (0x00000003)
            SecurityPolicyUri: http://opcfoundation.org/UA/SecurityPolicy#Hybrid
          ▸ UserIdentityTokens: Array of UserTokenPolicy
            TransportProfileUri: http://opcfoundation.org/UA-Profile/Transport/uatcp-uasc-uabinary

c170  bf c6 9c ce 03 00 00 00  31 00 00 00 68 74 74 70   ········ 1···http
c180  3a 2f 2f 6f 70 63 66 6f  75 6e 64 61 74 69 6f 6e   ://opcfo undation
c190  2e 6f 72 67 2f 55 41 2f  53 65 63 75 72 69 74 79   .org/UA/ Security
c1a0  50 6f 6c 69 63 79 23 48  79 62 72 69 64 02 00 00   Policy#H ybrid···
c1b0  00 1a 00 00 00 6f 70 65  6e 36 32 35 34 31 2d 61   ·····ope n62541-a
c1c0  6e 6f 6e 79 6d 6f 75 73  2d 70 6f 6c 69 63 79 00   nonymous -policy·
c1d0  00 00 00 ff ff ff ff ff  ff ff ff ff ff ff ff 19   ········ ········
```

Adding the Policy to the Client

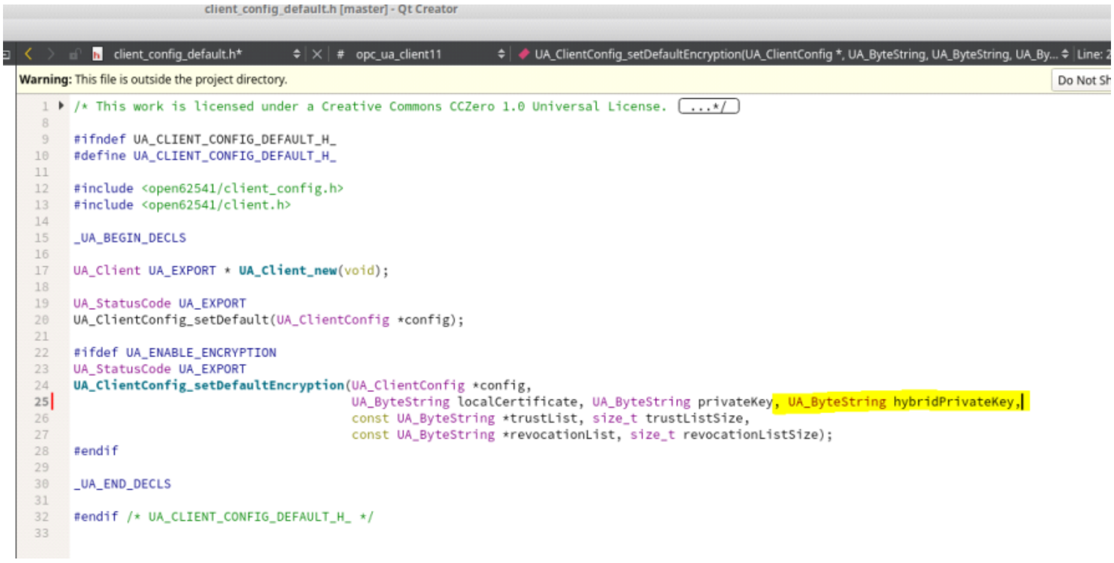
In the file ua_config_default.c go to the function UA_ClientConfig_setDefaultEncryption(). Add the hybrid private key as a parameter. Change the number of allocated security policies from 4 to 5.

```c

#ifdef UA_ENABLE_ENCRYPTION
UA_StatusCode
UA_ClientConfig_setDefaultEncryption(UA_ClientConfig *config,
                                     UA_ByteString localCertificate, UA_ByteString privateKey, UA_ByteString hybridPrivateKey,
                                     const UA_ByteString *trustList, size_t trustListSize,
                                     const UA_ByteString *revocationList, size_t revocationListSize) {
    UA_StatusCode retval = UA_ClientConfig_setDefault(config);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;

    retval = UA_CertificateVerification_Trustlist(&config->certificateVerification,
                                                  trustList, trustListSize,
                                                  NULL, 0,
                                                  revocationList, revocationListSize);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;

    /* Populate SecurityPolicies */
    UA_SecurityPolicy *sp = (UA_SecurityPolicy*)
        UA_realloc(config->securityPolicies, sizeof(UA_SecurityPolicy) * 5);
    if(!sp)
        return UA_STATUSCODE_BADOUTOFMEMORY;
    config->securityPolicies = sp;

    retval = UA_SecurityPolicy_Basic128Rsa15(&config->securityPolicies[1],
                                             &config->certificateVerification,
                                             localCertificate, privateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
```

In the same function, add the new security policy at the end

```c
                                             &config->certificateVerification,
                                             localCertificate, privateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    ++config->securityPoliciesSize;

    retval = UA_SecurityPolicy_Hybrid(&config->securityPolicies[4],
                                             &config->certificateVerification,
                                             localCertificate, privateKey, hybridPrivateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    ++config->securityPoliciesSize;

    return UA_STATUSCODE_GOOD;
}
#endif
```

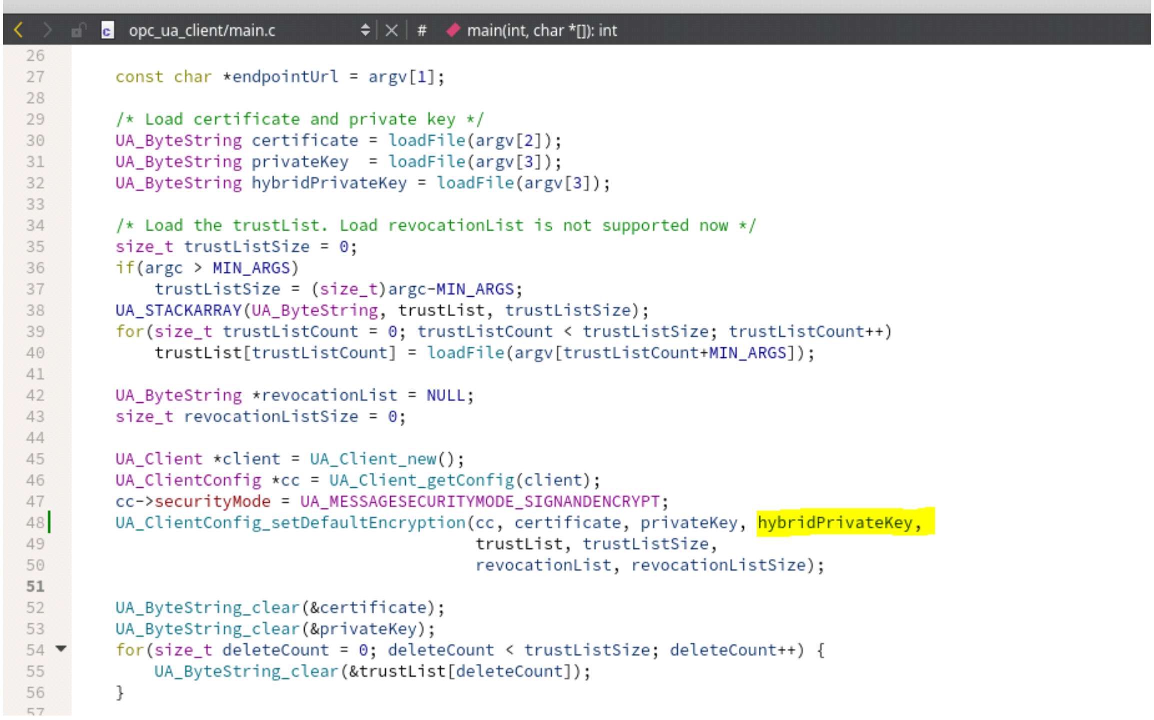
Adopt the header file and add the new parameter



Warning: This file is outside the project directory.

```c
/* This work is licensed under a Creative Commons CCZero 1.0 Universal License. ...*/

#ifndef UA_CLIENT_CONFIG_DEFAULT_H_
#define UA_CLIENT_CONFIG_DEFAULT_H_

#include <open62541/client_config.h>
#include <open62541/client.h>

_UA_BEGIN_DECLS

UA_Client UA_EXPORT * UA_Client_new(void);

UA_StatusCode UA_EXPORT
UA_ClientConfig_setDefault(UA_ClientConfig *config);

#ifdef UA_ENABLE_ENCRYPTION
UA_StatusCode UA_EXPORT
UA_ClientConfig_setDefaultEncryption(UA_ClientConfig *config,
                                     UA_ByteString localCertificate, UA_ByteString privateKey, UA_ByteString hybridPrivateKey,
                                     const UA_ByteString *trustList, size_t trustListSize,
                                     const UA_ByteString *revocationList, size_t revocationListSize);
#endif

_UA_END_DECLS

#endif /* UA_CLIENT_CONFIG_DEFAULT_H_ */
```

In the clients main function add the new parameter to the function call

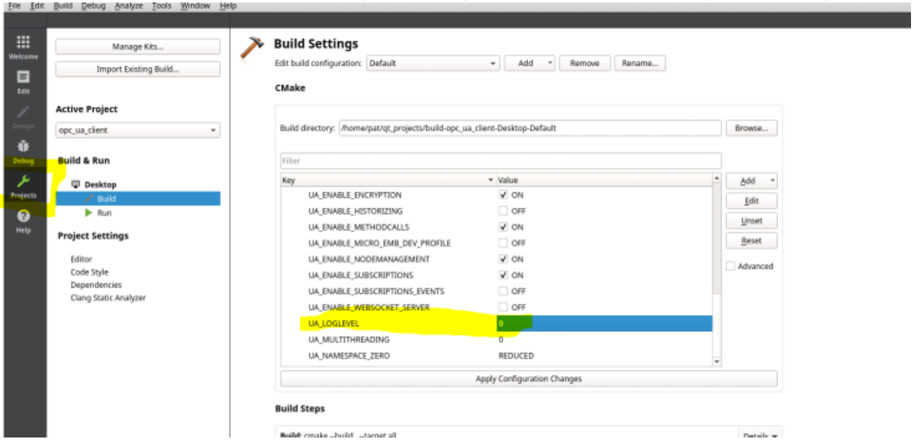


```c

    const char *endpointUrl = argv[1];

    /* Load certificate and private key */
    UA_ByteString certificate = loadFile(argv[2]);
    UA_ByteString privateKey  = loadFile(argv[3]);
    UA_ByteString hybridPrivateKey = loadFile(argv[3]);

    /* Load the trustList. Load revocationList is not supported now */
    size_t trustListSize = 0;
    if(argc > MIN_ARGS)
        trustListSize = (size_t)argc-MIN_ARGS;
    UA_STACKARRAY(UA_ByteString, trustList, trustListSize);
    for(size_t trustListCount = 0; trustListCount < trustListSize; trustListCount++)
        trustList[trustListCount] = loadFile(argv[trustListCount+MIN_ARGS]);

    UA_ByteString *revocationList = NULL;
    size_t revocationListSize = 0;

    UA_Client *client = UA_Client_new();
    UA_ClientConfig *cc = UA_Client_getConfig(client);
    cc->securityMode = UA_MESSAGESECURITYMODE_SIGNANDENCRYPT;
    UA_ClientConfig_setDefaultEncryption(cc, certificate, privateKey, hybridPrivateKey,
                                         trustList, trustListSize,
                                         revocationList, revocationListSize);

    UA_ByteString_clear(&certificate);
    UA_ByteString_clear(&privateKey);
    for(size_t deleteCount = 0; deleteCount < trustListSize; deleteCount++) {
        UA_ByteString_clear(&trustList[deleteCount]);
    }
```

Then add the following two lines to configure the client to select the new security policy

```c
        trustList[trustListCount] = loadFile(argv[trustListCount+MIN_ARGS]);

    UA_ByteString *revocationList = NULL;
    size_t revocationListSize = 0;

    UA_Client *client = UA_Client_new();
    UA_ClientConfig *cc = UA_Client_getConfig(client);
    cc->securityMode = UA_MESSAGESECURITYMODE_SIGNANDENCRYPT;
    UA_String uri = UA_STRING_ALLOC("http://opcfoundation.org/UA/SecurityPolicy#Hybrid");
    cc->securityPolicyUri = uri;
    UA_ClientConfig_setDefaultEncryption(cc, certificate, privateKey, hybridPrivateKey,
                                         trustList, trustListSize,
                                         revocationList, revocationListSize);

    UA_ByteString_clear(&certificate);
    UA_ByteString_clear(&privateKey);
    for(size_t deleteCount = 0; deleteCount < trustListSize; deleteCount++) {
        UA_ByteString_clear(&trustList[deleteCount]);
    }

    /* Secure client connect */
    cc->securityMode = UA_MESSAGESECURITYMODE_SIGNANDENCRYPT; /* require encryption */
    UA_StatusCode retval = UA_Client_connect(client, endpointUrl);
```

To see which policy is used, change the log level to 0 in the project settings



Active Project

opc_ua_client

Build Settings

Edit build configuration: Default

CMake

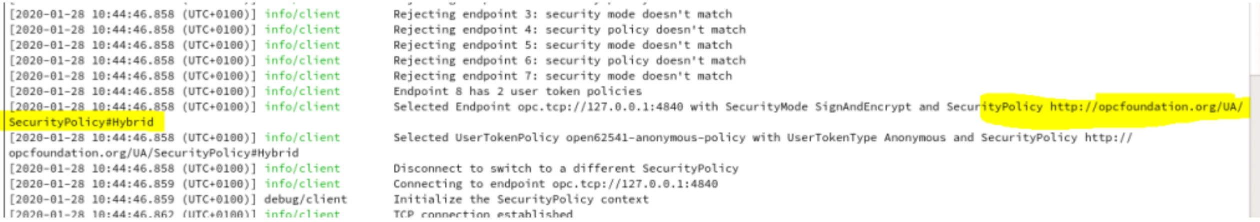
Build directory: /home/pat/qt_projects/build-opc_ua_client-Desktop-Default

Key	Value
UA_ENABLE_ENCRYPTION	☑ ON
UA_ENABLE_HISTORIZING	☐ OFF
UA_ENABLE_METHODCALLS	☑ ON
UA_ENABLE_MICRO_EMB_DEV_PROFILE	☐ OFF
UA_ENABLE_NODEMANAGEMENT	☑ ON
UA_ENABLE_SUBSCRIPTIONS	☑ ON
UA_ENABLE_SUBSCRIPTIONS_EVENTS	☐ OFF
UA_ENABLE_WEBSOCKET_SERVER	☐ OFF
UA_LOGLEVEL	0
UA_MULTITHREADING	0
UA_NAMESPACE_ZERO	REDUCED

Build Steps

Build: cmake --build . --target all

In the clients output we see that the correct security policy is uesed.



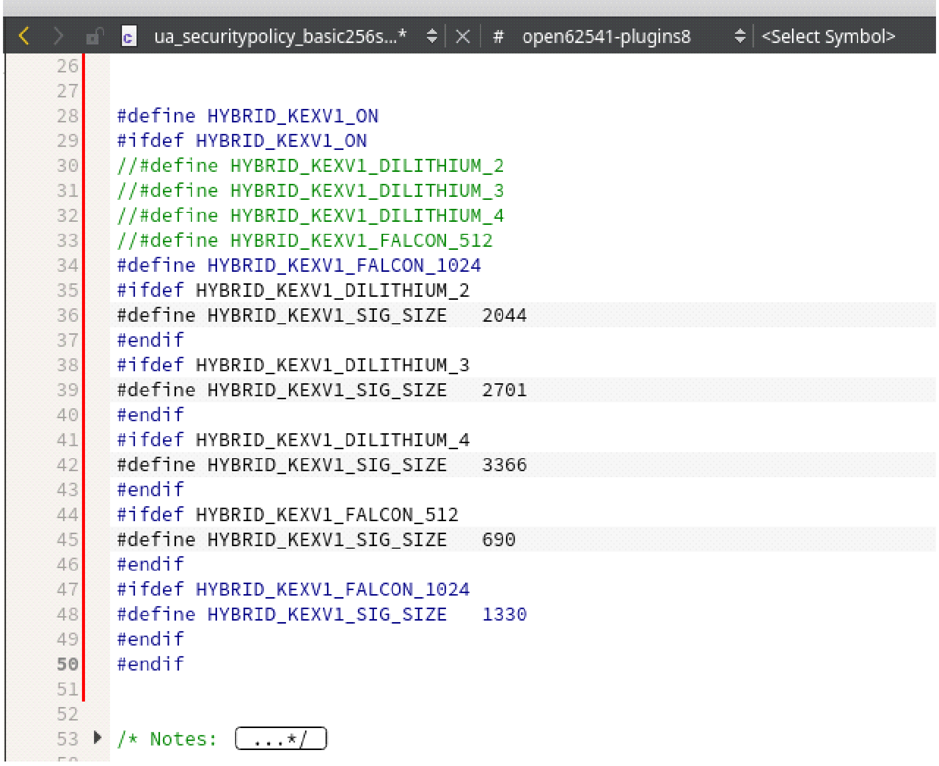
```
[2020-01-28 10:44:46.858 (UTC+0100)] info/client      Rejecting endpoint 3: security mode doesn't match
[2020-01-28 10:44:46.858 (UTC+0100)] info/client      Rejecting endpoint 4: security policy doesn't match
[2020-01-28 10:44:46.858 (UTC+0100)] info/client      Rejecting endpoint 5: security mode doesn't match
[2020-01-28 10:44:46.858 (UTC+0100)] info/client      Rejecting endpoint 6: security policy doesn't match
[2020-01-28 10:44:46.858 (UTC+0100)] info/client      Rejecting endpoint 7: security mode doesn't match
[2020-01-28 10:44:46.858 (UTC+0100)] info/client      Endpoint 8 has 2 user token policies
[2020-01-28 10:44:46.858 (UTC+0100)] info/client      Selected Endpoint opc.tcp://127.0.0.1:4840 with SecurityMode SignAndEncrypt and SecurityPolicy http://opcfoundation.org/UA/
SecurityPolicy#Hybrid
[2020-01-28 10:44:46.858 (UTC+0100)] info/client      Selected UserTokenPolicy open62541-anonymous-policy with UserTokenType Anonymous and SecurityPolicy http://
opcfoundation.org/UA/SecurityPolicy#Hybrid
[2020-01-28 10:44:46.858 (UTC+0100)] info/client      Disconnect to switch to a different SecurityPolicy
[2020-01-28 10:44:46.859 (UTC+0100)] info/client      Connecting to endpoint opc.tcp://127.0.0.1:4840
[2020-01-28 10:44:46.859 (UTC+0100)] debug/client     Initialize the SecurityPolicy context
[2020-01-28 10:44:46.862 (UTC+0100)] info/client      TCP connection established
```

Add a Hybrid Signature to Asymmetric Encrypted Messages

Dienstag, 28. Januar 2020 10:46

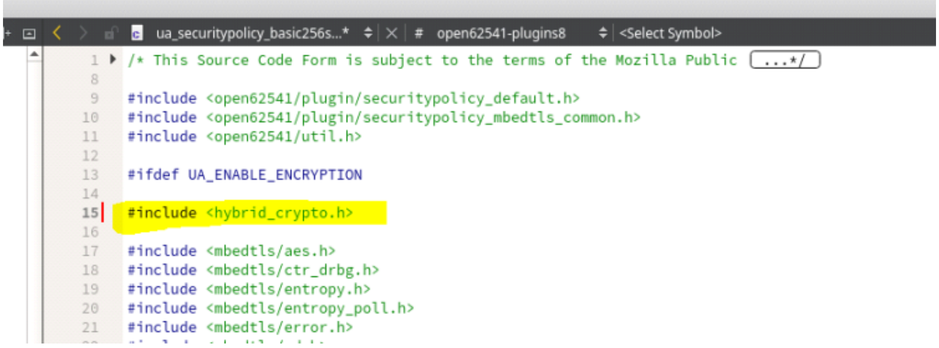
In order to sign messages with an additional hybrid signature, we need to edit the newly created security policy.

First define some that we are going to need later in ua_seucritypolicy_basic256sha256.c



```


#define HYBRID_KEXV1_ON
#ifdef HYBRID_KEXV1_ON
//#define HYBRID_KEXV1_DILITHIUM_2
//#define HYBRID_KEXV1_DILITHIUM_3
//#define HYBRID_KEXV1_DILITHIUM_4
//#define HYBRID_KEXV1_FALCON_512
#define HYBRID_KEXV1_FALCON_1024
#ifdef HYBRID_KEXV1_DILITHIUM_2
#define HYBRID_KEXV1_SIG_SIZE   2044
#endif
#ifdef HYBRID_KEXV1_DILITHIUM_3
#define HYBRID_KEXV1_SIG_SIZE   2701
#endif
#ifdef HYBRID_KEXV1_DILITHIUM_4
#define HYBRID_KEXV1_SIG_SIZE   3366
#endif
#ifdef HYBRID_KEXV1_FALCON_512
#define HYBRID_KEXV1_SIG_SIZE   690
#endif
#ifdef HYBRID_KEXV1_FALCON_1024
#define HYBRID_KEXV1_SIG_SIZE   1330
#endif
#endif


/* Notes: ...*/
```

Then include the hybrid crypto functions



```
/* This Source Code Form is subject to the terms of the Mozilla Public ...*/

#include <open62541/plugin/securitypolicy_default.h>
#include <open62541/plugin/securitypolicy_mbedtls_common.h>
#include <open62541/util.h>

#ifdef UA_ENABLE_ENCRYPTION

#include <hybrid_crypto.h>

#include <mbedtls/aes.h>
#include <mbedtls/ctr_drbg.h>
#include <mbedtls/entropy.h>
#include <mbedtls/entropy_poll.h>
#include <mbedtls/error.h>
```

We need to add functions for signature verification (of messages, not certificates) and signature generation. Then we will use this functions in the security policy callback functions. All the functions are based on the basic256sha256 functions, so it is a good idea to copy and rename these function and then modify them.

Add the function asym_verify_sp_hybrid() (copy from asym_verify_sp_basic256sha256()). The signature now will be larger because in fact it contains two signatures. The first 256 bytes are the RSA sig. So do not pass the full signature length, but only the first 256 bytes to the mbedtls verify function

```
                              const UA_ByteString *signature) { ...} 

static UA_StatusCode
asym_verify_sp_hybrid(const UA_SecurityPolicy *securityPolicy,
                              Basic256Sha256_ChannelContext *cc,
                              const UA_ByteString *message,
                              const UA_ByteString *signature) {
    if(securityPolicy == NULL || message == NULL || signature == NULL || cc == NULL)
        return UA_STATUSCODE_BADINTERNALERROR;

    unsigned char hash[UA_SHA256_LENGTH];
#if MBEDTLS_VERSION_NUMBER >= 0x02070000
    // TODO check return status
    mbedtls_sha256_ret(message->data, message->length, hash, 0);
#else
    mbedtls_sha256(message->data, message->length, hash, 0);
#endif

    /* Set the RSA settings */
    mbedtls_rsa_context *rsaContext = mbedtls_pk_rsa(cc->remoteCertificate.pk);
    mbedtls_rsa_set_padding(rsaContext, MBEDTLS_RSA_PKCS_V15, MBEDTLS_MD_SHA256);

    /* For RSA keys, the default padding type is PKCS#1 v1.5 in mbedtls_pk_verify() */
    /* Alternatively, use more specific function mbedtls_rsa_rsassa_pkcs1_v15_verify(), i.e. */
    /* int mbedErr = mbedtls_rsa_rsassa_pkcs1_v15_verify(rsaContext, NULL, NULL,
                                                     MBEDTLS_RSA_PUBLIC, MBEDTLS_MD_SHA256,
                                                     UA_SHA256_LENGTH, hash,
                                                     signature->data); */
    int mbedErr = mbedtls_pk_verify(&cc->remoteCertificate.pk,
                                    MBEDTLS_MD_SHA256, hash, UA_SHA256_LENGTH,
                                    signature->data, 256);

    if(mbedErr)
        return UA_STATUSCODE_BADSECURITYCHECKSFAILED;


    // get the public key
    uint8_t *dummy;
```

Then add the code for the PQ signature verification afterwards

```

    // get the public key
    uint8_t *dummy;
    uint8_t *hybridPublicKey;
    char dummy_buf[100];
    get_signature_and_pk(&cc->remoteCertificate, &dummy, &hybridPublicKey, dummy_buf, dummy_buf);

    int verification_result = -1;

#ifdef HYBRID_KEXV1_DILITHIUM_2
    verification_result = dilithium2_verify(signature->data+256, HYBRID_KEXV1_SIG_SIZE, message->data, message->length, hybridPublicKey);
#endif
#ifdef HYBRID_KEXV1_DILITHIUM_3
    verification_result = dilithium3_verify(signature->data+256, HYBRID_KEXV1_SIG_SIZE, message->data, message->length, hybridPublicKey);
#endif
#ifdef HYBRID_KEXV1_DILITHIUM_4
    verification_result = dilithium4_verify(signature->data+256, HYBRID_KEXV1_SIG_SIZE, message->data, message->length, hybridPublicKey);
#endif
#ifdef HYBRID_KEXV1_FALCON_512
    verification_result = falcon512_verify(signature->data+256, HYBRID_KEXV1_SIG_SIZE, message->data, message->length, hybridPublicKey);
#endif
#ifdef HYBRID_KEXV1_FALCON_1024
    verification_result = falcon1024_verify(signature->data+256, HYBRID_KEXV1_SIG_SIZE, message->data, message->length, hybridPublicKey);
#endif


    if (verification_result != 0)
        return UA_STATUSCODE_BADSECURITYCHECKSFAILED;

    return UA_STATUSCODE_GOOD;
}
```

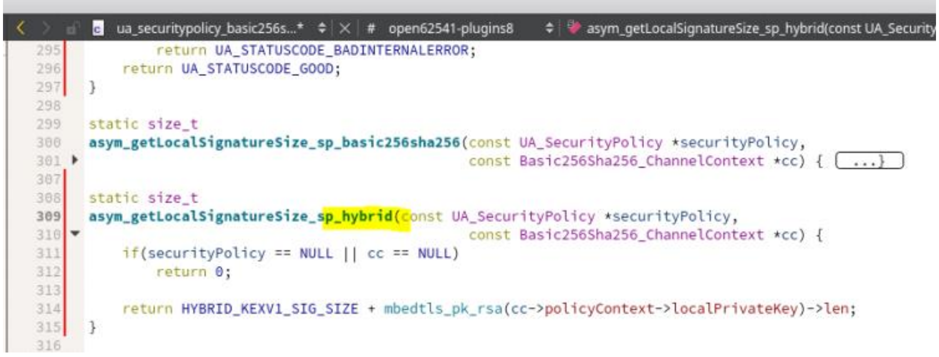
Next add the hybrid signing function. No changes are required here

```
static UA_StatusCode
asym_sign_sp_hybrid(const UA_SecurityPolicy *securityPolicy,
                            Basic256Sha256_ChannelContext *cc,
                            const UA_ByteString *message,
                            UA_ByteString *signature) {
    if(securityPolicy == NULL || message == NULL || signature == NULL || cc == NULL)
        return UA_STATUSCODE_BADINTERNALERROR;

    unsigned char hash[UA_SHA256_LENGTH];
#if MBEDTLS_VERSION_NUMBER >= 0x02070000
    // TODO check return status
    mbedtls_sha256_ret(message->data, message->length, hash, 0);
#else
    mbedtls_sha256(message->data, message->length, hash, 0);
#endif

    Hybrid_PolicyContext *pc = (Hybrid_PolicyContext *)cc->policyContext;
    mbedtls_rsa_context *rsaContext = mbedtls_pk_rsa(pc->localPrivateKey);
    mbedtls_rsa_set_padding(rsaContext, MBEDTLS_RSA_PKCS_V15, MBEDTLS_MD_SHA256);

    size_t sigLen = 0;

    /* For RSA keys, the default padding type is PKCS#1 v1.5 in mbedtls_pk_sign */
    /* Alternatively use more specific function mbedtls_rsa_rsassa_pkcs1_v15_sign() */
    int mbedErr = mbedtls_pk_sign(&pc->localPrivateKey,
                                  MBEDTLS_MD_SHA256, hash,
                                  UA_SHA256_LENGTH, signature->data,
                                  &sigLen, mbedtls_ctr_drbg_random,
                                  &pc->drbgContext);
    if(mbedErr)
        return UA_STATUSCODE_BADINTERNALERROR;

    // sign with hybrid method
```

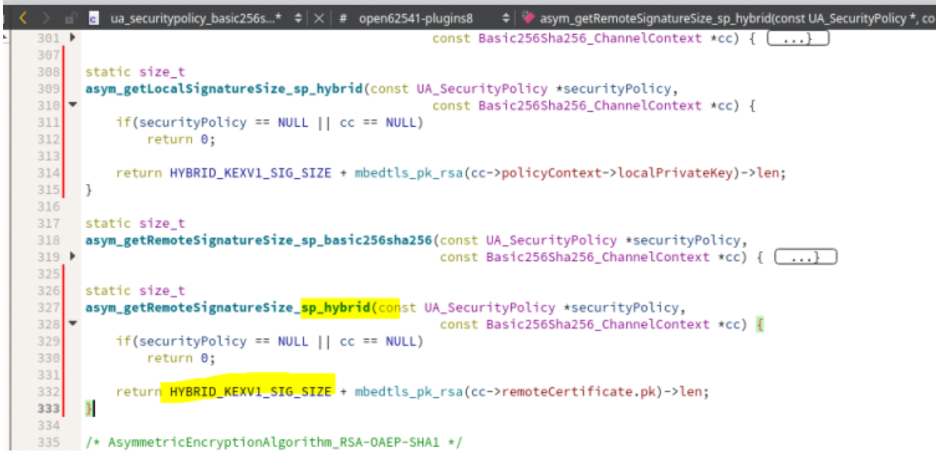
Then add the code for the PQ signature

```
        return UA_STATUSCODE_BADINTERNALERROR;

    // sign with hybrid method
    int error = -1;
#ifdef HYBRID_KEXV1_DILITHIUM_2
    error = dilithium2_sign(signature->data+256, &sigLen, message->data, message->length, pc->localHybridPrivateKey.data);
#endif
#ifdef HYBRID_KEXV1_DILITHIUM_3
    error = dilithium3_sign(signature->data+256, &sigLen, message->data, message->length, pc->localHybridPrivateKey.data);
#endif
#ifdef HYBRID_KEXV1_DILITHIUM_4
    error = dilithium4_sign(signature->data+256, &sigLen, message->data, message->length, pc->localHybridPrivateKey.data);
#endif
#ifdef HYBRID_KEXV1_FALCON_512
    error = falcon512_sign(signature->data+256, &sigLen, message->data, message->length, pc->localHybridPrivateKey.data);
#endif
#ifdef HYBRID_KEXV1_FALCON_1024
    error = falcon1024_sign(signature->data+256, &sigLen, message->data, message->length, pc->localHybridPrivateKey.data);
#endif

    if (error != 0)
        return UA_STATUSCODE_BADINTERNALERROR;
    return UA_STATUSCODE_GOOD;
}
```

Open62541 gets the size of the signatures (that now has changes) also from the security policy. So we have to modify the functions for that as well:

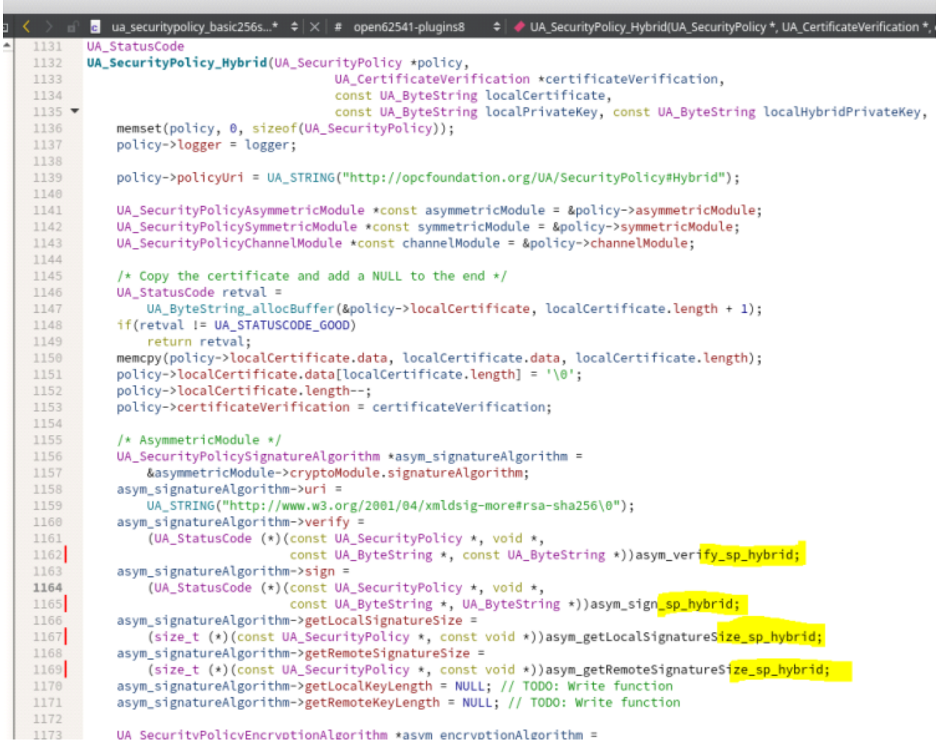


```
        return UA_STATUSCODE_BADINTERNALERROR;
    return UA_STATUSCODE_GOOD;
}

static size_t
asym_getLocalSignatureSize_sp_basic256sha256(const UA_SecurityPolicy *securityPolicy,
                                             const Basic256Sha256_ChannelContext *cc) { ...}

static size_t
asym_getLocalSignatureSize_sp_hybrid(const UA_SecurityPolicy *securityPolicy,
                                             const Basic256Sha256_ChannelContext *cc) {
    if(securityPolicy == NULL || cc == NULL)
        return 0;

    return HYBRID_KEXV1_SIG_SIZE + mbedtls_pk_rsa(cc->policyContext->localPrivateKey)->len;
}
```



```
                                              const Basic256Sha256_ChannelContext *cc) { ...}

static size_t
asym_getLocalSignatureSize_sp_hybrid(const UA_SecurityPolicy *securityPolicy,
                                              const Basic256Sha256_ChannelContext *cc) {
    if(securityPolicy == NULL || cc == NULL)
        return 0;

    return HYBRID_KEXV1_SIG_SIZE + mbedtls_pk_rsa(cc->policyContext->localPrivateKey)->len;
}

static size_t
asym_getRemoteSignatureSize_sp_basic256sha256(const UA_SecurityPolicy *securityPolicy,
                                               const Basic256Sha256_ChannelContext *cc) { ...}

static size_t
asym_getRemoteSignatureSize_sp_hybrid(const UA_SecurityPolicy *securityPolicy,
                                               const Basic256Sha256_ChannelContext *cc) {
    if(securityPolicy == NULL || cc == NULL)
        return 0;

    return HYBRID_KEXV1_SIG_SIZE + mbedtls_pk_rsa(cc->remoteCertificate.pk)->len;
}

/* AsymmetricEncryptionAlgorithm_RSA-OAEP-SHA1 */
```

Finally you need to assign these functions to the hybrid security policy object. Therefore go to the function UA_SecurityPolicy_Hybrid()



```
UA_StatusCode
UA_SecurityPolicy_Hybrid(UA_SecurityPolicy *policy,
                                 UA_CertificateVerification *certificateVerification,
                                 const UA_ByteString localCertificate,
                                 const UA_ByteString localPrivateKey, const UA_ByteString localHybridPrivateKey,
    memset(policy, 0, sizeof(UA_SecurityPolicy));
    policy->logger = logger;

    policy->policyUri = UA_STRING("http://opcfoundation.org/UA/SecurityPolicy#Hybrid");

    UA_SecurityPolicyAsymmetricModule *const asymmetricModule = &policy->asymmetricModule;
    UA_SecurityPolicySymmetricModule *const symmetricModule = &policy->symmetricModule;
    UA_SecurityPolicyChannelModule *const channelModule = &policy->channelModule;

    /* Copy the certificate and add a NULL to the end */
    UA_StatusCode retval =
        UA_ByteString_allocBuffer(&policy->localCertificate, localCertificate.length + 1);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    memcpy(policy->localCertificate.data, localCertificate.data, localCertificate.length);
    policy->localCertificate.data[localCertificate.length] = '\0';
    policy->localCertificate.length--;
    policy->certificateVerification = certificateVerification;

    /* AsymmetricModule */
    UA_SecurityPolicySignatureAlgorithm *asym_signatureAlgorithm =
        &asymmetricModule->cryptoModule.signatureAlgorithm;
    asym_signatureAlgorithm->uri =
        UA_STRING("http://www.w3.org/2001/04/xmldsig-more#rsa-sha256\0");
    asym_signatureAlgorithm->verify =
        (UA_StatusCode (*)(const UA_SecurityPolicy *, void *,
                           const UA_ByteString *, const UA_ByteString *))asym_verify_sp_hybrid;
    asym_signatureAlgorithm->sign =
        (UA_StatusCode (*)(const UA_SecurityPolicy *, void *,
                           const UA_ByteString *, UA_ByteString *))asym_sign_sp_hybrid;
    asym_signatureAlgorithm->getLocalSignatureSize =
        (size_t (*)(const UA_SecurityPolicy *, const void *))asym_getLocalSignatureSize_sp_hybrid;
    asym_signatureAlgorithm->getRemoteSignatureSize =
        (size_t (*)(const UA_SecurityPolicy *, const void *))asym_getRemoteSignatureSize_sp_hybrid;
    asym_signatureAlgorithm->getLocalKeyLength = NULL; // TODO: Write function
    asym_signatureAlgorithm->getRemoteKeyLength = NULL; // TODO: Write function

    UA_SecurityPolicyEncryptionAlgorithm *asym_encryptionAlgorithm =
```

Compile and run server and client.

Summary Modified Files

Dienstag, 28. Januar 2020 12:24

- opc_ua_client/main.c
- opc_ua_server/main.c
- open62541/ua_client_connect.c
    - MAX_DATA_SIZE
- open62541/ua_pki_default.c
    - certificateVerification_verify() (modified)
    - certificateVerification_hybrid() (added)
    - UA_CertificateVerification_Trustlist() (modified)
- Open62541/ua_securitypolicy_basic256sha256.c
    - Include added
    - #defines
    - Hybrid_PolicyContext Struct (added)
    - Asym_verify_sp_hybrid() (added)
    - Asym_sign_sp_hybrid() (added)
    - Asym_getLocalSignatureSize_sp_hybrid() (added)
    - Asym_getRemoteSignatureSize_sp_hybrid() (added)
    - policyContext_NewContext_sp_hybrid() (added)
    - UA_SecurityPolicy_Hybrid() (added)
- Open62541/ua_config_default.c
    - UA_ServerConfig_addSecurityPolicyHybrid() (added)
    - UA_ServerConfig_setDefaultWithSecurityPolicies() (modified)
    - UA_ClientConfig_setDefaultEncryption() (modified)

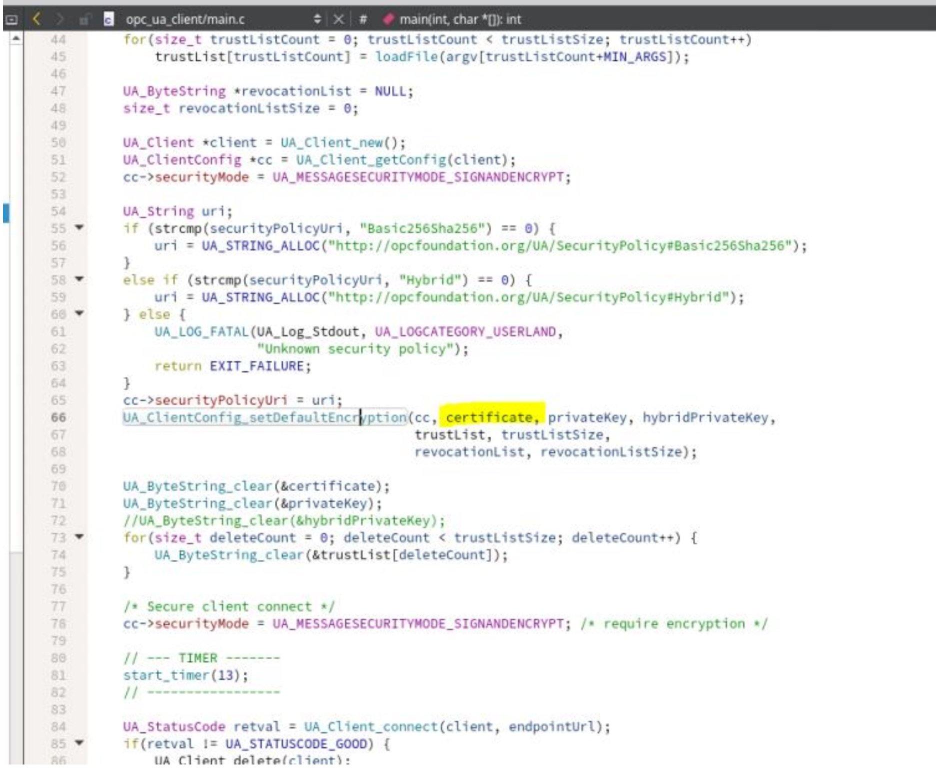
Adding Certificate Chains

Dienstag, 11. Februar 2020 10:05

Scenario: A client uses a certificate that was signed by an intermediate CA. The intermediate CA was signed by the root CA. The server only trusts the root CA.
Therefore the client has to send both, his certificate and the intermediate CA's certificate. The server then has to verify the chain which ends at root CA, which is trusted by the server.

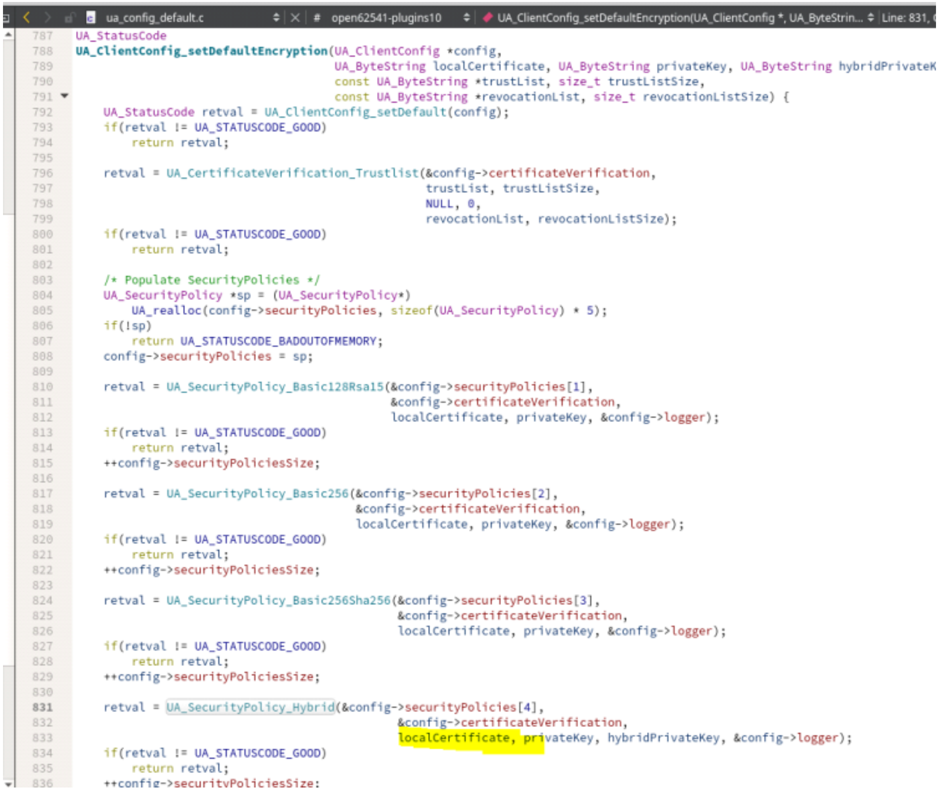
Problem 1: How to add the additional certificate of the intermediate CA to the client's request?

In main() a file with the clients certificate is passed and read binary. We will include two certificates into this file (simple concatenation). Thus we end up with a binary string that includes two certificates.



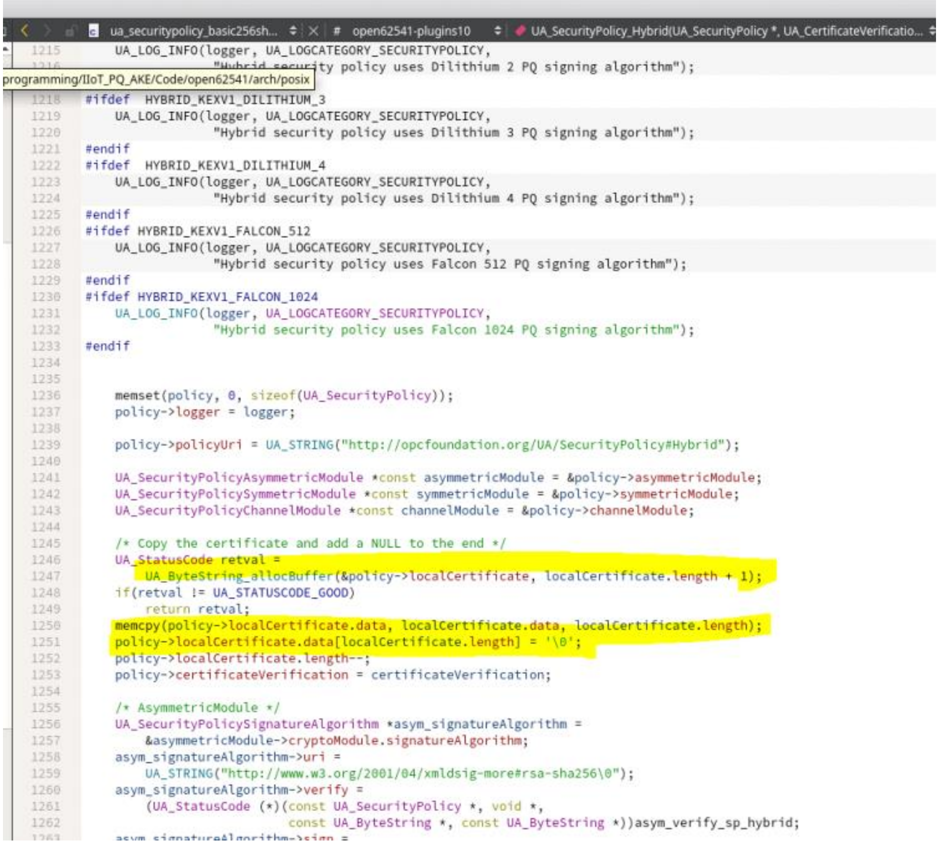
```c
    for(size_t trustListCount = 0; trustListCount < trustListSize; trustListCount++)
        trustList[trustListCount] = loadFile(argv[trustListCount+MIN_ARGS]);

    UA_ByteString *revocationList = NULL;
    size_t revocationListSize = 0;

    UA_Client *client = UA_Client_new();
    UA_ClientConfig *cc = UA_Client_getConfig(client);
    cc->securityMode = UA_MESSAGESECURITYMODE_SIGNANDENCRYPT;

    UA_String uri;
    if (strcmp(securityPolicyUri, "Basic256Sha256") == 0) {
        uri = UA_STRING_ALLOC("http://opcfoundation.org/UA/SecurityPolicy#Basic256Sha256");
    }
    else if (strcmp(securityPolicyUri, "Hybrid") == 0) {
        uri = UA_STRING_ALLOC("http://opcfoundation.org/UA/SecurityPolicy#Hybrid");
    } else {
        UA_LOG_FATAL(UA_Log_Stdout, UA_LOGCATEGORY_USERLAND,
                     "Unknown security policy");
        return EXIT_FAILURE;
    }
    cc->securityPolicyUri = uri;
    UA_ClientConfig_setDefaultEncryption(cc, certificate, privateKey, hybridPrivateKey,
                                         trustList, trustListSize,
                                         revocationList, revocationListSize);

    UA_ByteString_clear(&certificate);
    UA_ByteString_clear(&privateKey);
    //UA_ByteString_clear(&hybridPrivateKey);
    for(size_t deleteCount = 0; deleteCount < trustListSize; deleteCount++) {
        UA_ByteString_clear(&trustList[deleteCount]);
    }

    /* Secure client connect */
    cc->securityMode = UA_MESSAGESECURITYMODE_SIGNANDENCRYPT; /* require encryption */

    // --- TIMER -------
    start_timer(13);
    // -----------------

    UA_StatusCode retval = UA_Client_connect(client, endpointUrl);
    if(retval != UA_STATUSCODE_GOOD) {
        UA_Client_delete(client);
```

From the main() function, UA_ClientConfig_setDefaultEncryption() is called with the certificate byte string.



```c
UA_StatusCode
UA_ClientConfig_setDefaultEncryption(UA_ClientConfig *config,
                                     UA_ByteString localCertificate, UA_ByteString privateKey, UA_ByteString hybridPrivateK
                                     const UA_ByteString *trustList, size_t trustListSize,
                                     const UA_ByteString *revocationList, size_t revocationListSize) {
    UA_StatusCode retval = UA_ClientConfig_setDefault(config);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;

    retval = UA_CertificateVerification_Trustlist(&config->certificateVerification,
                                                  trustList, trustListSize,
                                                  NULL, 0,
                                                  revocationList, revocationListSize);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;

    /* Populate SecurityPolicies */
    UA_SecurityPolicy *sp = (UA_SecurityPolicy*)
        UA_realloc(config->securityPolicies, sizeof(UA_SecurityPolicy) * 5);
    if(!sp)
        return UA_STATUSCODE_BADOUTOFMEMORY;
    config->securityPolicies = sp;

    retval = UA_SecurityPolicy_Basic128Rsa15(&config->securityPolicies[1],
                                             &config->certificateVerification,
                                             localCertificate, privateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    ++config->securityPoliciesSize;

    retval = UA_SecurityPolicy_Basic256(&config->securityPolicies[2],
                                        &config->certificateVerification,
                                        localCertificate, privateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    ++config->securityPoliciesSize;

    retval = UA_SecurityPolicy_Basic256Sha256(&config->securityPolicies[3],
                                              &config->certificateVerification,
                                              localCertificate, privateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    ++config->securityPoliciesSize;

    retval = UA_SecurityPolicy_Hybrid(&config->securityPolicies[4],
                                      &config->certificateVerification,
                                      localCertificate, privateKey, hybridPrivateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    ++config->securityPoliciesSize;
```

In this function, the byte string is directly passed to the function that creates the security policy object.



```c
    UA_LOG_INFO(logger, UA_LOGCATEGORY_SECURITYPOLICY,
                "Hybrid security policy uses Dilithium 2 PQ signing algorithm");
#ifdef  HYBRID_KEXV1_DILITHIUM_3
    UA_LOG_INFO(logger, UA_LOGCATEGORY_SECURITYPOLICY,
                "Hybrid security policy uses Dilithium 3 PQ signing algorithm");
#endif
#ifdef  HYBRID_KEXV1_DILITHIUM_4
    UA_LOG_INFO(logger, UA_LOGCATEGORY_SECURITYPOLICY,
                "Hybrid security policy uses Dilithium 4 PQ signing algorithm");
#endif
#ifdef HYBRID_KEXV1_FALCON_512
    UA_LOG_INFO(logger, UA_LOGCATEGORY_SECURITYPOLICY,
                "Hybrid security policy uses Falcon 512 PQ signing algorithm");
#endif
#ifdef HYBRID_KEXV1_FALCON_1024
    UA_LOG_INFO(logger, UA_LOGCATEGORY_SECURITYPOLICY,
                "Hybrid security policy uses Falcon 1024 PQ signing algorithm");
#endif


    memset(policy, 0, sizeof(UA_SecurityPolicy));
    policy->logger = logger;

    policy->policyUri = UA_STRING("http://opcfoundation.org/UA/SecurityPolicy#Hybrid");

    UA_SecurityPolicyAsymmetricModule *const asymmetricModule = &policy->asymmetricModule;
    UA_SecurityPolicySymmetricModule *const symmetricModule = &policy->symmetricModule;
    UA_SecurityPolicyChannelModule *const channelModule = &policy->channelModule;

    /* Copy the certificate and add a NULL to the end */
    UA_StatusCode retval =
        UA_ByteString_allocBuffer(&policy->localCertificate, localCertificate.length + 1);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    memcpy(policy->localCertificate.data, localCertificate.data, localCertificate.length);
    policy->localCertificate.data[localCertificate.length] = '\0';
    policy->localCertificate.length--;
    policy->certificateVerification = certificateVerification;

    /* AsymmetricModule */
    UA_SecurityPolicySignatureAlgorithm *asym_signatureAlgorithm =
        &asymmetricModule->cryptoModule.signatureAlgorithm;
    asym_signatureAlgorithm->uri =
        UA_STRING("http://www.w3.org/2001/04/xmldsig-more#rsa-sha256\0");
    asym_signatureAlgorithm->verify =
        (UA_StatusCode (*)(const UA_SecurityPolicy *, void *,
                           const UA_ByteString *, const UA_ByteString *))asym_verify_sp_hybrid;
    asym_signatureAlgorithm->sign =
```

Inside, memory is allocated and the byte string is copied to that memory.

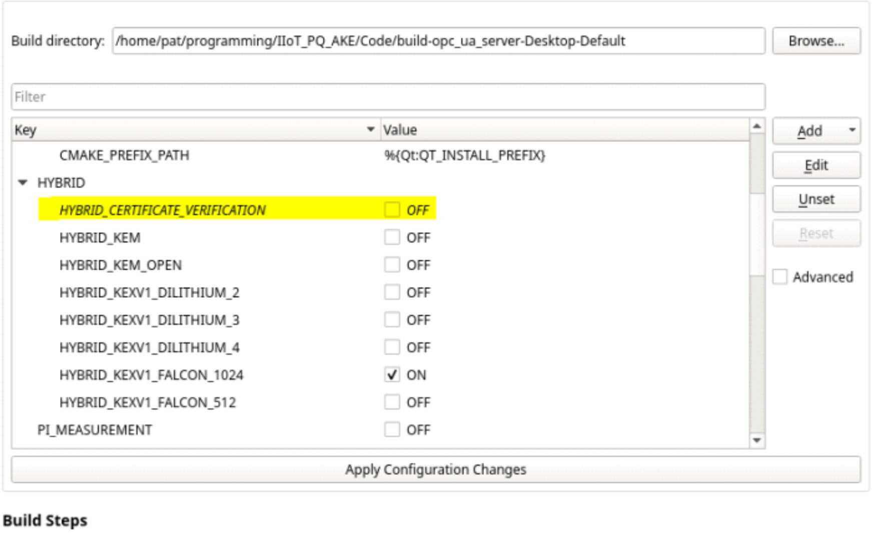
Conclusion: The security policy contains a byte string with two certificates included.

Next we have to look at the function that creates the openSecureChannelRequest. In particular at the function that prepends the asymmetric security header. There the certificate is directly copied to the header, and therefore is sent to the server. Next we have to ensure that the server uses this information properly.

Problem 2: Make the server evaluate certificate chains

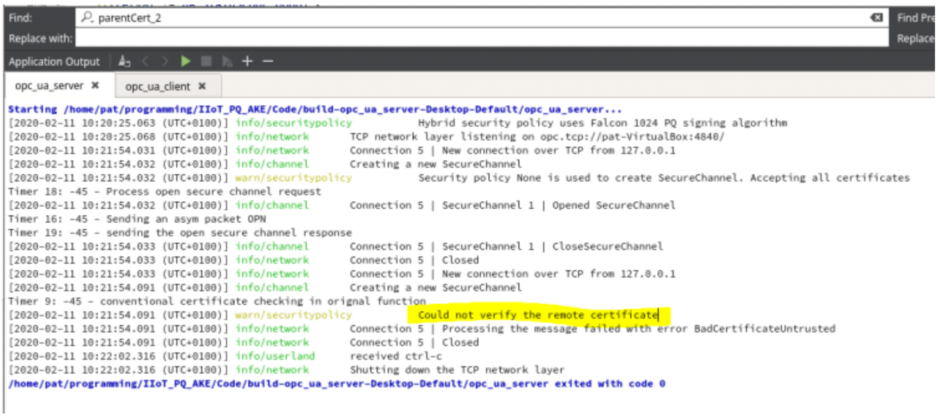
First a quick test: Client uses a certificate chain as described above.

The hybrid certificate verification is off. This ensures that we are using the original function to verify certificates



Build directory: /home/pat/programming/IIoT_PQ_AKE/Code/build-opc_ua_server-Desktop-Default

Key	Value
CMAKE_PREFIX_PATH	%{Qt:QT_INSTALL_PREFIX}
HYBRID	
HYBRID_CERTIFICATE_VERIFICATION	☐ OFF
HYBRID_KEM	☐ OFF
HYBRID_KEM_OPEN	☐ OFF
HYBRID_KEXV1_DILITHIUM_2	☐ OFF
HYBRID_KEXV1_DILITHIUM_3	☐ OFF
HYBRID_KEXV1_DILITHIUM_4	☐ OFF
HYBRID_KEXV1_FALCON_1024	☑ ON
HYBRID_KEXV1_FALCON_512	☐ OFF
PI_MEASUREMENT	☐ OFF

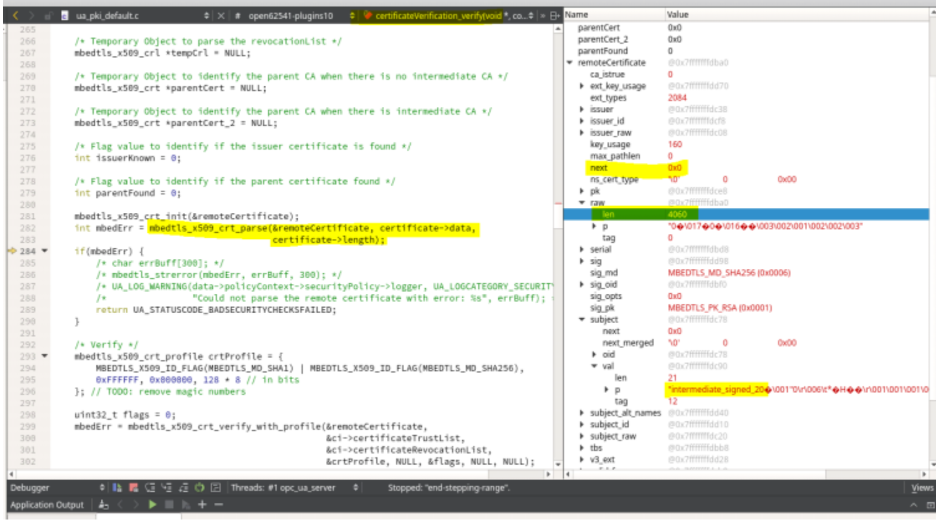


Find: parentCert_2

```
Starting /home/pat/programming/IIoT_PQ_AKE/Code/build-opc_ua_server-Desktop-Default/opc_ua_server...
[2020-02-11 10:20:25.063 (UTC+0100)] info/securitypolicy          Hybrid security policy uses Falcon 1024 PQ signing algorithm
[2020-02-11 10:20:25.068 (UTC+0100)] info/network       TCP network layer listening on opc.tcp://pat-VirtualBox:4840/
[2020-02-11 10:21:54.031 (UTC+0100)] info/network       Connection 5 | New connection over TCP from 127.0.0.1
[2020-02-11 10:21:54.032 (UTC+0100)] info/channel       Creating a new SecureChannel
[2020-02-11 10:21:54.032 (UTC+0100)] warn/securitypolicy          Security policy None is used to create SecureChannel. Accepting all certificates
Timer 18: -45 - Process open secure channel request
[2020-02-11 10:21:54.032 (UTC+0100)] info/channel       Connection 5 | SecureChannel 1 | Opened SecureChannel
Timer 16: -45 - Sending an asym packet OPN
Timer 19: -45 - sending the open secure channel response
[2020-02-11 10:21:54.033 (UTC+0100)] info/channel       Connection 5 | SecureChannel 1 | CloseSecureChannel
[2020-02-11 10:21:54.033 (UTC+0100)] info/network       Connection 5 | Closed
[2020-02-11 10:21:54.033 (UTC+0100)] info/network       Connection 5 | New connection over TCP from 127.0.0.1
[2020-02-11 10:21:54.091 (UTC+0100)] info/channel       Creating a new SecureChannel
Timer 9: -45 - conventional certificate checking in orignal function
[2020-02-11 10:21:54.091 (UTC+0100)] warn/securitypolicy          Could not verify the remote certificate
[2020-02-11 10:21:54.091 (UTC+0100)] info/network       Connection 5 | Processing the message failed with error BadCertificateUntrusted
[2020-02-11 10:21:54.091 (UTC+0100)] info/network       Connection 5 | Closed
[2020-02-11 10:22:02.316 (UTC+0100)] info/userland      received ctrl-c
[2020-02-11 10:22:02.316 (UTC+0100)] info/network       Shutting down the TCP network layer
/home/pat/programming/IIoT_PQ_AKE/Code/build-opc_ua_server-Desktop-Default/opc_ua_server exited with code 0
```

As expected, certificate verification fails.

With a debugger, we check the certificateVerification_verify() function for the server.



```c
    /* Temporary Object to parse the revocationList */
    mbedtls_x509_crl *tempCrl = NULL;

    /* Temporary Object to identify the parent CA when there is no intermediate CA */
    mbedtls_x509_crt *parentCert = NULL;

    /* Temporary Object to identify the parent CA when there is intermediate CA */
    mbedtls_x509_crt *parentCert_2 = NULL;

    /* Flag value to identify if the issuer certificate is found */
    int issuerKnown = 0;

    /* Flag value to identify if the parent certificate found */
    int parentFound = 0;

    mbedtls_x509_crt_init(&remoteCertificate);
    int mbedErr = mbedtls_x509_crt_parse(&remoteCertificate, certificate->data,
                                        certificate->length);

    if(mbedErr) {
        /* char errBuff[300]; */
        /* mbedtls_strerror(mbedErr, errBuff, 300); */
        /* UA_LOG_WARNING(data->policyContext->securityPolicy->logger, UA_LOGCATEGORY_SECURIT
        /*                    "Could not parse the remote certificate with error: %s", errBuff);
        return UA_STATUSCODE_BADSECURITYCHECKSFAILED;
    }

    /* Verify */
    mbedtls_x509_crt_profile crtProfile = {
        MBEDTLS_X509_ID_FLAG(MBEDTLS_MD_SHA1) | MBEDTLS_X509_ID_FLAG(MBEDTLS_MD_SHA256),
        0xFFFFFF, 0x000000, 128 * 8 // in bits
    }; // TODO: remove magic numbers

    uint32_t flags = 0;
    mbedErr = mbedtls_x509_crt_verify_with_profile(&remoteCertificate,
                                                   &ci->certificateTrustList,
                                                   &ci->certificateRevocationList,
                                                   &crtProfile, NULL, &flags, NULL, NULL);
```

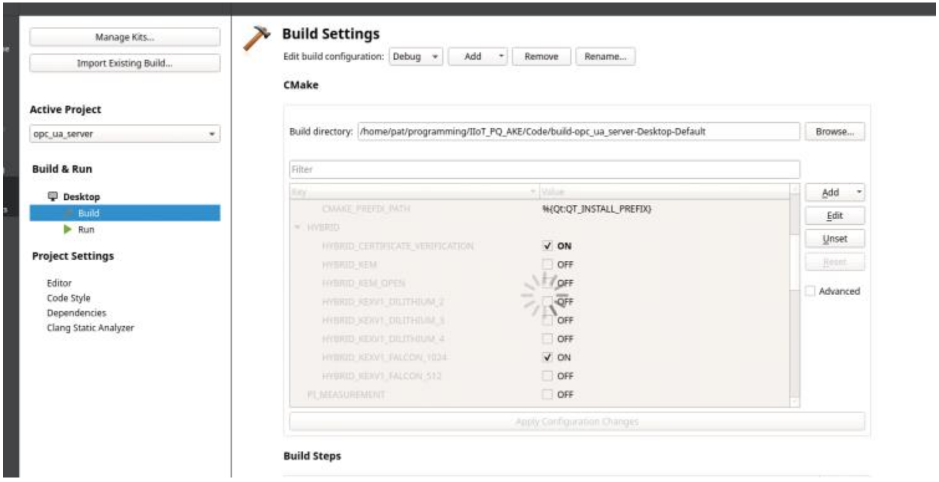
```
Name                 Value
  parentCert         0x0
  parentCert_2       0x0
  parentFound        0
▾ remoteCertificate  @0x7fffffffdba0
    ca_istrue        0
  ▸ ext_key_usage    @0x7fffffffdd70
    ext_types        2084
  ▸ issuer           @0x7fffffffdc38
  ▸ issuer_id        @0x7fffffffdcf8
  ▸ issuer_raw       @0x7fffffffdc08
    key_usage        160
    max_pathlen      0
    next             0x0
    ns_cert_type     '\0'       0       0x00
  ▸ pk               @0x7fffffffdce8
  ▾ raw              @0x7fffffffdba0
      len            4060
    ▸ p              "0�\017�0�\016��\003\002\001\002\002\003"
      tag            0
  ▸ serial           @0x7fffffffdbd8
  ▸ sig              @0x7fffffffdd98
    sig_md           MBEDTLS_MD_SHA256 (0x0006)
  ▸ sig_oid          @0x7fffffffdbf0
    sig_opts         0x0
    sig_pk           MBEDTLS_PK_RSA (0x0001)
  ▾ subject          @0x7fffffffdc78
      next           0x0
      next_merged    '\0'       0       0x00
    ▸ oid            @0x7fffffffdc78
    ▾ val            @0x7fffffffdc90
        len          21
      ▸ p            "intermediate_signed_20�\001"0\r\006\t*�H��\r\001\001\001\0
        tag          12
  ▸ subject_alt_names @0x7fffffffdd40
  ▸ subject_id       @0x7fffffffdd10
  ▸ subject_raw      @0x7fffffffdc20
  ▸ tbs              @0x7fffffffdbb8
  ▸ v3_ext           @0x7fffffffdd28
```

The remote certificate is parsed, it is the end certificated (named intermediate_signed_2), has no next value (therefore this is not a chain) and has a total length of 4060 bytes.

However the certificate received via the network has 8034 bytes. That is 8034 - 4060 = 3974 bytes that have been ignored. That is the exact byte size of the intermediate CA certificate. We can conclude that we have the intermediate CA certificate available but it is ignored by open62541.

Modifying the hybrid certificate verification function in order to allow certificate chain verification

Enable the hybrid certificate verification again



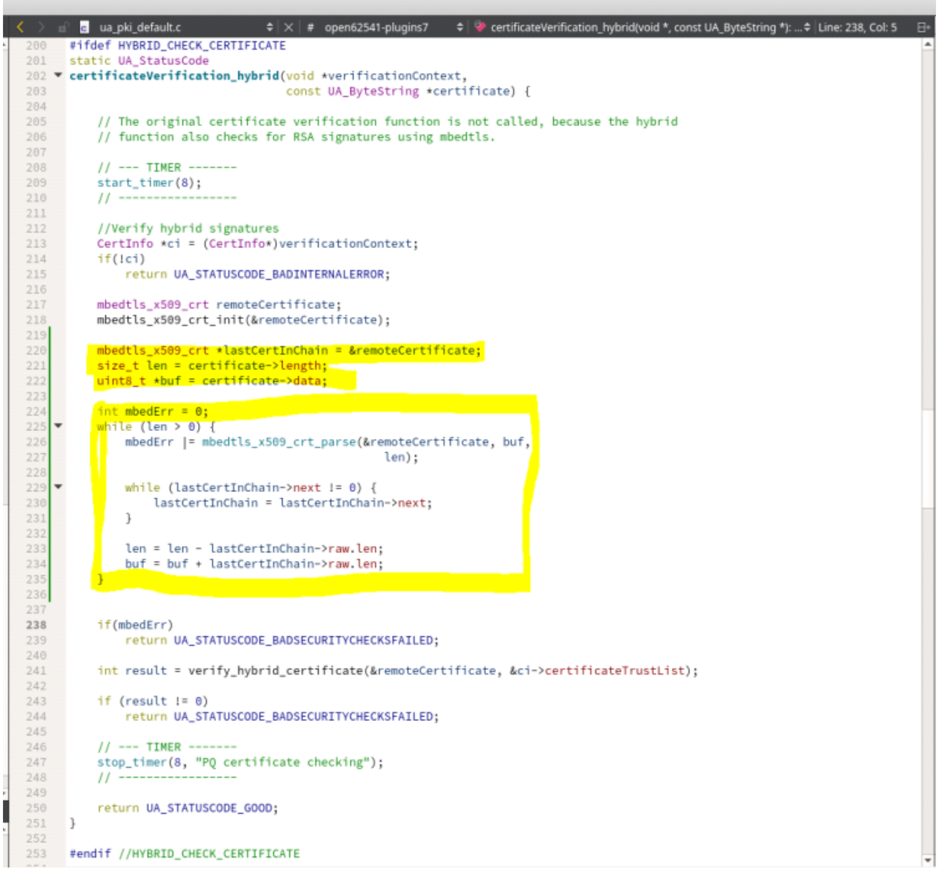
Active Project: opc_ua_server

Edit build configuration: Debug

Build directory: /home/pat/programming/IIoT_PQ_AKE/Code/build-opc_ua_server-Desktop-Default

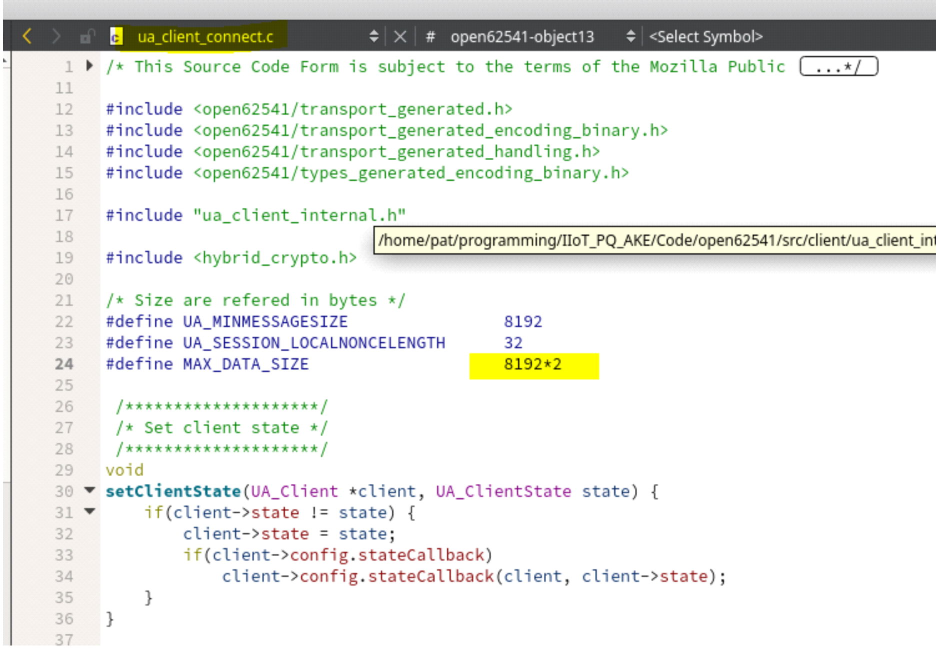
Key	Value
CMAKE_PREFIX_PATH	%{Qt:QT_INSTALL_PREFIX}
▾ HYBRID	
HYBRID_CERTIFICATE_VERIFICATION	☑ ON
HYBRID_KEM	☐ OFF
HYBRID_KEM_OPEN	☐ OFF
HYBRID_KEXV1_DILITHIUM_2	☐ OFF
HYBRID_KEXV1_DILITHIUM_3	☐ OFF
HYBRID_KEXV1_DILITHIUM_4	☐ OFF
HYBRID_KEXV1_FALCON_1024	☑ ON
HYBRID_KEXV1_FALCON_512	☐ OFF
PI_MEASUREMENT	☐ OFF

Then we can change the hybrid certificate verification function to actually parse out all the certificates that are contained in the request.



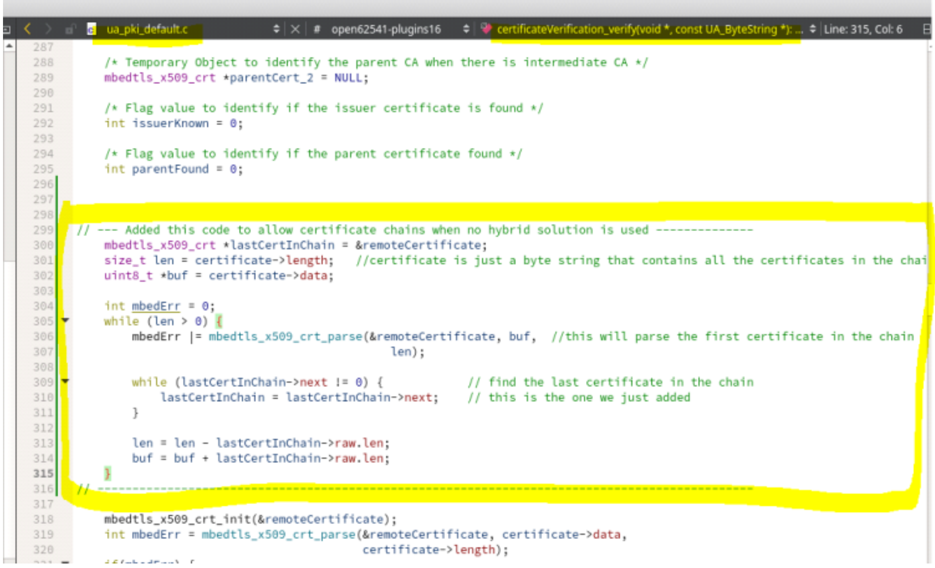
```c
#ifdef HYBRID_CHECK_CERTIFICATE
static UA_StatusCode
certificateVerification_hybrid(void *verificationContext,
                               const UA_ByteString *certificate) {

    // The original certificate verification function is not called, because the hybrid
    // function also checks for RSA signatures using mbedtls.

    // --- TIMER -------
    start_timer(8);
    // -----------------

    //Verify hybrid signatures
    CertInfo *ci = (CertInfo*)verificationContext;
    if(!ci)
        return UA_STATUSCODE_BADINTERNALERROR;

    mbedtls_x509_crt remoteCertificate;
    mbedtls_x509_crt_init(&remoteCertificate);

    mbedtls_x509_crt *lastCertInChain = &remoteCertificate;
    size_t len = certificate->length;
    uint8_t *buf = certificate->data;

    int mbedErr = 0;
    while (len > 0) {
        mbedErr |= mbedtls_x509_crt_parse(&remoteCertificate, buf,
                                          len);

        while (lastCertInChain->next != 0) {
            lastCertInChain = lastCertInChain->next;
        }

        len = len - lastCertInChain->raw.len;
        buf = buf + lastCertInChain->raw.len;
    }

    if(mbedErr)
        return UA_STATUSCODE_BADSECURITYCHECKSFAILED;

    int result = verify_hybrid_certificate(&remoteCertificate, &ci->certificateTrustList);

    if (result != 0)
        return UA_STATUSCODE_BADSECURITYCHECKSFAILED;

    // --- TIMER -------
    stop_timer(8, "PQ certificate checking");
    // -----------------

    return UA_STATUSCODE_GOOD;
}

#endif //HYBRID_CHECK_CERTIFICATE
```

Activate session might fail now because the MAX_DATA_SIZE can be exceeded by the chain. A simple fix is to extent the max size. Here we just double it by adding *2.



```c
/* This Source Code Form is subject to the terms of the Mozilla Public ...*/

#include <open62541/transport_generated.h>
#include <open62541/transport_generated_encoding_binary.h>
#include <open62541/transport_generated_handling.h>
#include <open62541/types_generated_encoding_binary.h>

#include "ua_client_internal.h"

#include <hybrid_crypto.h>

/* Size are refered in bytes */
#define UA_MINMESSAGESIZE                8192
#define UA_SESSION_LOCALNONCELENGTH      32
#define MAX_DATA_SIZE                    8192*2

 /********************/
 /* Set client state */
 /********************/
void
setClientState(UA_Client *client, UA_ClientState state) {
    if(client->state != state) {
        client->state = state;
        if(client->config.stateCallback)
            client->config.stateCallback(client, client->state);
    }
}
```

Now the same fixes are introduced into the original certificate verification function



```c

    /* Temporary Object to identify the parent CA when there is intermediate CA */
    mbedtls_x509_crt *parentCert_2 = NULL;

    /* Flag value to identify if the issuer certificate is found */
    int issuerKnown = 0;

    /* Flag value to identify if the parent certificate found */
    int parentFound = 0;



// --- Added this code to allow certificate chains when no hybrid solution is used --------------
    mbedtls_x509_crt *lastCertInChain = &remoteCertificate;
    size_t len = certificate->length;   //certificate is just a byte string that contains all the certificates in the chai
    uint8_t *buf = certificate->data;

    int mbedErr = 0;
    while (len > 0) {
        mbedErr |= mbedtls_x509_crt_parse(&remoteCertificate, buf,  //this will parse the first certificate in the chain
                                          len);

        while (lastCertInChain->next != 0) {            // find the last certificate in the chain
            lastCertInChain = lastCertInChain->next;    // this is the one we just added
        }

        len = len - lastCertInChain->raw.len;
        buf = buf + lastCertInChain->raw.len;
    }
// ----------------------------------------------------------------------------------------------

    mbedtls_x509_crt_init(&remoteCertificate);
    int mbedErr = mbedtls_x509_crt_parse(&remoteCertificate, certificate->data,
                                         certificate->length);
```

F Implementation Details Variant Two

Variant Two

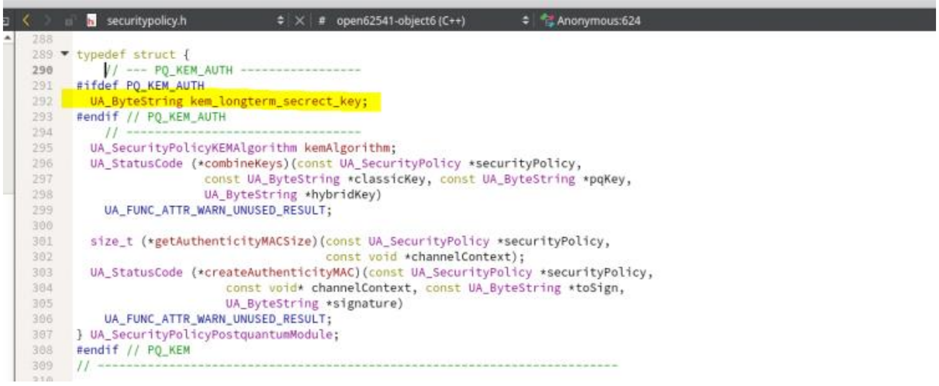
Mittwoch, 18. März 2020 10:59

New Certificates

Variant Two needs new certificates that contain public keys for the KEMs that are used. These are:
Kyber 512 + Dilithium2 / Falcon512
Kyber 768 + Dilithium3
Kyber 1024 + Dilithium4 / Falcon1024

1. Make Private Keys Available

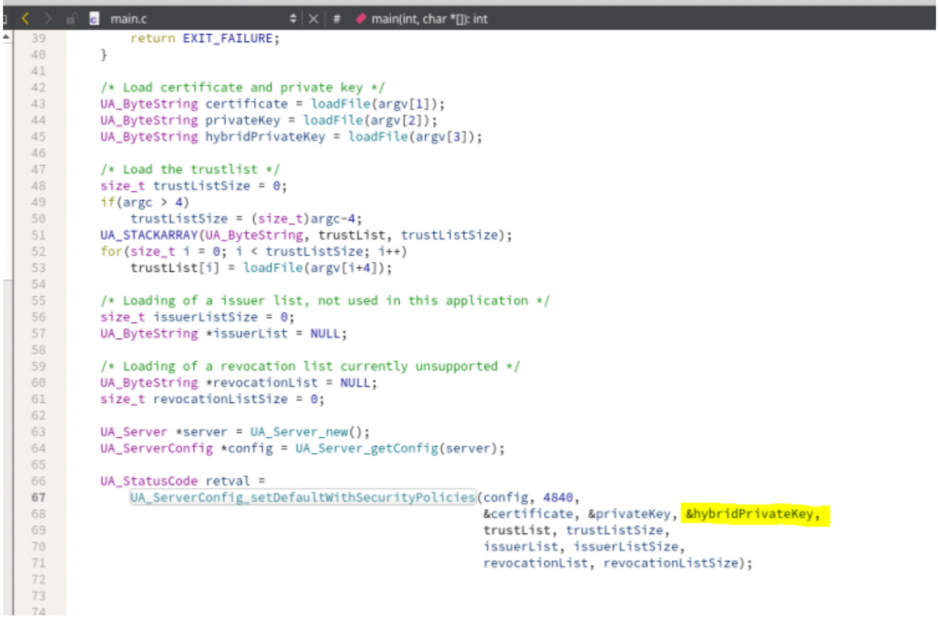
The private keys for the PQ KEMs are passed as a command line argument and then are stored in the security policy object. We just use the Basic256Sha security policies in the tests so the key is added here. For the unauthenticated quantum resistant key exchange, there was already a Post Quantum module added to this security policy. We add the *kem_longterm_secret_key* to this data structure.



```c
typedef struct {
    // --- PQ_KEM_AUTH -----------------
#ifdef PQ_KEM_AUTH
  UA_ByteString kem_longterm_secrect_key;
#endif // PQ_KEM_AUTH
    // --------------------------------
  UA_SecurityPolicyKEMAlgorithm kemAlgorithm;
  UA_StatusCode (*combineKeys)(const UA_SecurityPolicy *securityPolicy,
                const UA_ByteString *classicKey, const UA_ByteString *pqKey,
                UA_ByteString *hybridKey)
    UA_FUNC_ATTR_WARN_UNUSED_RESULT;

  size_t (*getAuthenticityMACSize)(const UA_SecurityPolicy *securityPolicy,
                                   const void *channelContext);
  UA_StatusCode (*createAuthenticityMAC)(const UA_SecurityPolicy *securityPolicy,
                   const void* channelContext, const UA_ByteString *toSign,
                   UA_ByteString *signature)
    UA_FUNC_ATTR_WARN_UNUSED_RESULT;
} UA_SecurityPolicyPostquantumModule;
#endif // PQ_KEM
// ----------------------------------------------------------------------
```

Passing the secret key to the security policy:
Server:

A new parameter is added to the function UA_ServerConfig_setDefaultWithSecurityPolicy().



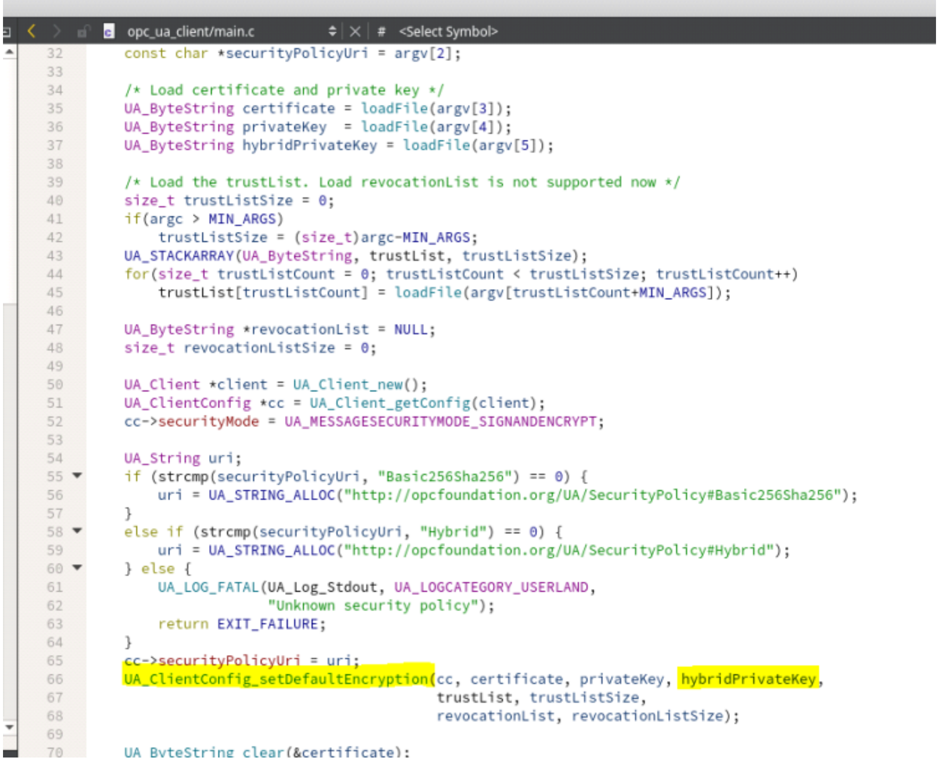
```c
        return EXIT_FAILURE;
    }

    /* Load certificate and private key */
    UA_ByteString certificate = loadFile(argv[1]);
    UA_ByteString privateKey = loadFile(argv[2]);
    UA_ByteString hybridPrivateKey = loadFile(argv[3]);

    /* Load the trustlist */
    size_t trustListSize = 0;
    if(argc > 4)
        trustListSize = (size_t)argc-4;
    UA_STACKARRAY(UA_ByteString, trustList, trustListSize);
    for(size_t i = 0; i < trustListSize; i++)
        trustList[i] = loadFile(argv[i+4]);

    /* Loading of a issuer list, not used in this application */
    size_t issuerListSize = 0;
    UA_ByteString *issuerList = NULL;

    /* Loading of a revocation list currently unsupported */
    UA_ByteString *revocationList = NULL;
    size_t revocationListSize = 0;

    UA_Server *server = UA_Server_new();
    UA_ServerConfig *config = UA_Server_getConfig(server);

    UA_StatusCode retval =
        UA_ServerConfig_setDefaultWithSecurityPolicies(config, 4840,
                                                       &certificate, &privateKey, &hybridPrivateKey,
                                                       trustList, trustListSize,
                                                       issuerList, issuerListSize,
                                                       revocationList, revocationListSize);
```

Inside this function we copy the private key into this field after creating the security policies

```c
        UA_ServerConfig_clean(conf);
        return retval;
    }

    retval = UA_ServerConfig_addAllSecurityPolicies(conf, certificate, privateKey);
    if(retval != UA_STATUSCODE_GOOD) {
        UA_ServerConfig_clean(conf);
        return retval;
    }

    // --- PQ_KEM_AUTH --------
    // add the long term private key to the security policy
#ifdef PQ_KEM_AUTH
    UA_ByteString_copy(hybridPrivateKey, &conf->securityPolicies[3].postquantumModule.kem_longterm_secrect_key);
#endif //PQ_KEM_AUTH
    // -----------------------

    retval = UA_ServerConfig_addSecurityPolicyHybrid(conf, certificate, privateKey, hybridPrivateKey);
```

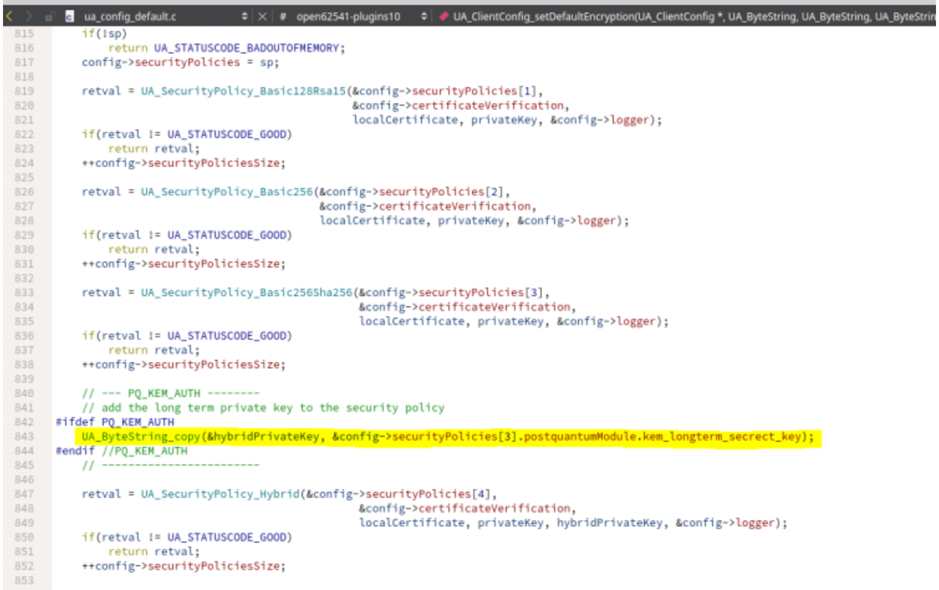
Client:

As for the server, we pass the KEM key to the config creation function



```c
    const char *securityPolicyUri = argv[2];

    /* Load certificate and private key */
    UA_ByteString certificate = loadFile(argv[3]);
    UA_ByteString privateKey  = loadFile(argv[4]);
    UA_ByteString hybridPrivateKey = loadFile(argv[5]);

    /* Load the trustList. Load revocationList is not supported now */
    size_t trustListSize = 0;
    if(argc > MIN_ARGS)
        trustListSize = (size_t)argc-MIN_ARGS;
    UA_STACKARRAY(UA_ByteString, trustList, trustListSize);
    for(size_t trustListCount = 0; trustListCount < trustListSize; trustListCount++)
        trustList[trustListCount] = loadFile(argv[trustListCount+MIN_ARGS]);

    UA_ByteString *revocationList = NULL;
    size_t revocationListSize = 0;

    UA_Client *client = UA_Client_new();
    UA_ClientConfig *cc = UA_Client_getConfig(client);
    cc->securityMode = UA_MESSAGESECURITYMODE_SIGNANDENCRYPT;

    UA_String uri;
    if (strcmp(securityPolicyUri, "Basic256Sha256") == 0) {
        uri = UA_STRING_ALLOC("http://opcfoundation.org/UA/SecurityPolicy#Basic256Sha256");
    }
    else if (strcmp(securityPolicyUri, "Hybrid") == 0) {
        uri = UA_STRING_ALLOC("http://opcfoundation.org/UA/SecurityPolicy#Hybrid");
    } else {
        UA_LOG_FATAL(UA_Log_Stdout, UA_LOGCATEGORY_USERLAND,
                     "Unknown security policy");
        return EXIT_FAILURE;
    }
    cc->securityPolicyUri = uri;
    UA_ClientConfig_setDefaultEncryption(cc, certificate, privateKey, hybridPrivateKey,
                                         trustList, trustListSize,
                                         revocationList, revocationListSize);

    UA_ByteString_clear(&certificate);
```

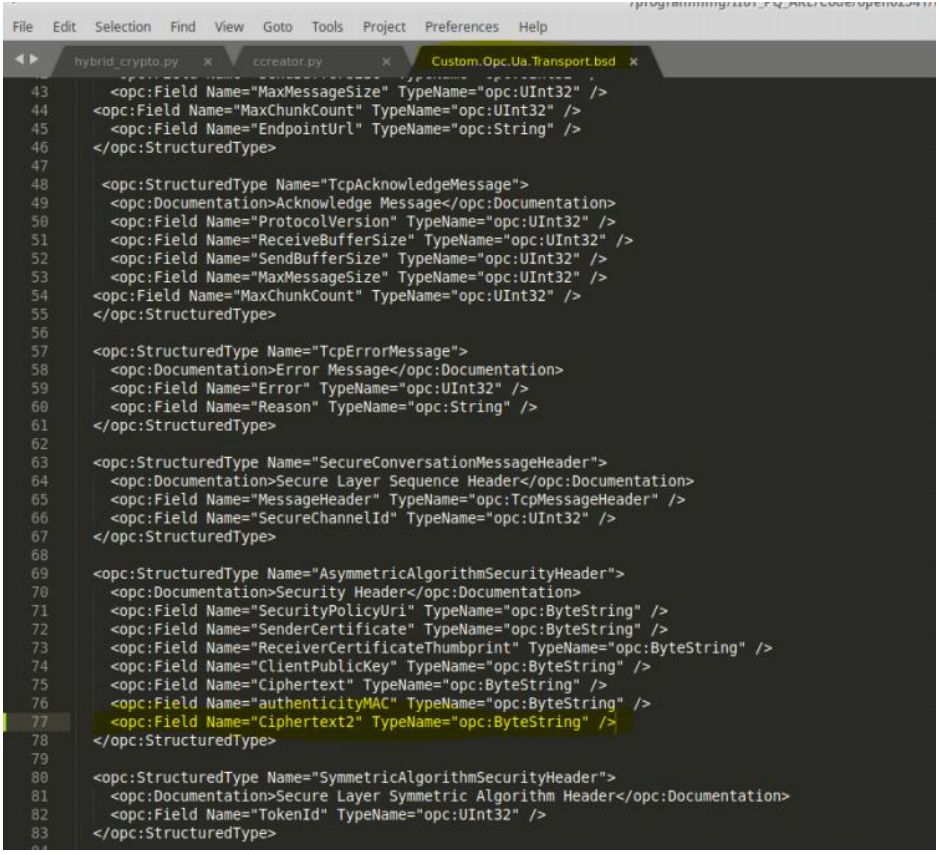
And add the private key to the security policy



```c
    if(!sp)
        return UA_STATUSCODE_BADOUTOFMEMORY;
    config->securityPolicies = sp;

    retval = UA_SecurityPolicy_Basic128Rsa15(&config->securityPolicies[1],
                                             &config->certificateVerification,
                                             localCertificate, privateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    ++config->securityPoliciesSize;

    retval = UA_SecurityPolicy_Basic256(&config->securityPolicies[2],
                                        &config->certificateVerification,
                                        localCertificate, privateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    ++config->securityPoliciesSize;

    retval = UA_SecurityPolicy_Basic256Sha256(&config->securityPolicies[3],
                                              &config->certificateVerification,
                                              localCertificate, privateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    ++config->securityPoliciesSize;

    // --- PQ_KEM_AUTH --------
    // add the long term private key to the security policy
#ifdef PQ_KEM_AUTH
    UA_ByteString_copy(&hybridPrivateKey, &config->securityPolicies[3].postquantumModule.kem_longterm_secrect_key);
#endif //PQ_KEM_AUTH
    // ------------------------

    retval = UA_SecurityPolicy_Hybrid(&config->securityPolicies[4],
                                              &config->certificateVerification,
                                              localCertificate, privateKey, hybridPrivateKey, &config->logger);
    if(retval != UA_STATUSCODE_GOOD)
        return retval;
    ++config->securityPoliciesSize;

```

2. Additional Fields in the openSecureChannel Request/Response

Change the tools/schema/custom.Opc.Ua.Transport.bsd so the open channel request/response (both because it is defined in the header) will have the additional field



```xml
    <opc:Field Name="MaxMessageSize" TypeName="opc:UInt32" />
  <opc:Field Name="MaxChunkCount" TypeName="opc:UInt32" />
    <opc:Field Name="EndpointUrl" TypeName="opc:String" />
  </opc:StructuredType>

   <opc:StructuredType Name="TcpAcknowledgeMessage">
    <opc:Documentation>Acknowledge Message</opc:Documentation>
    <opc:Field Name="ProtocolVersion" TypeName="opc:UInt32" />
    <opc:Field Name="ReceiveBufferSize" TypeName="opc:UInt32" />
    <opc:Field Name="SendBufferSize" TypeName="opc:UInt32" />
    <opc:Field Name="MaxMessageSize" TypeName="opc:UInt32" />
  <opc:Field Name="MaxChunkCount" TypeName="opc:UInt32" />
  </opc:StructuredType>

  <opc:StructuredType Name="TcpErrorMessage">
    <opc:Documentation>Error Message</opc:Documentation>
    <opc:Field Name="Error" TypeName="opc:UInt32" />
    <opc:Field Name="Reason" TypeName="opc:String" />
  </opc:StructuredType>

  <opc:StructuredType Name="SecureConversationMessageHeader">
    <opc:Documentation>Secure Layer Sequence Header</opc:Documentation>
    <opc:Field Name="MessageHeader" TypeName="opc:TcpMessageHeader" />
    <opc:Field Name="SecureChannelId" TypeName="opc:UInt32" />
  </opc:StructuredType>

  <opc:StructuredType Name="AsymmetricAlgorithmSecurityHeader">
    <opc:Documentation>Security Header</opc:Documentation>
    <opc:Field Name="SecurityPolicyUri" TypeName="opc:ByteString" />
    <opc:Field Name="SenderCertificate" TypeName="opc:ByteString" />
    <opc:Field Name="ReceiverCertificateThumbprint" TypeName="opc:ByteString" />
    <opc:Field Name="ClientPublicKey" TypeName="opc:ByteString" />
    <opc:Field Name="Ciphertext" TypeName="opc:ByteString" />
    <opc:Field Name="authenticityMAC" TypeName="opc:ByteString" />
    <opc:Field Name="Ciphertext2" TypeName="opc:ByteString" />
  </opc:StructuredType>

  <opc:StructuredType Name="SymmetricAlgorithmSecurityHeader">
    <opc:Documentation>Secure Layer Symmetric Algorithm Header</opc:Documentation>
    <opc:Field Name="TokenId" TypeName="opc:UInt32" />
  </opc:StructuredType>
```

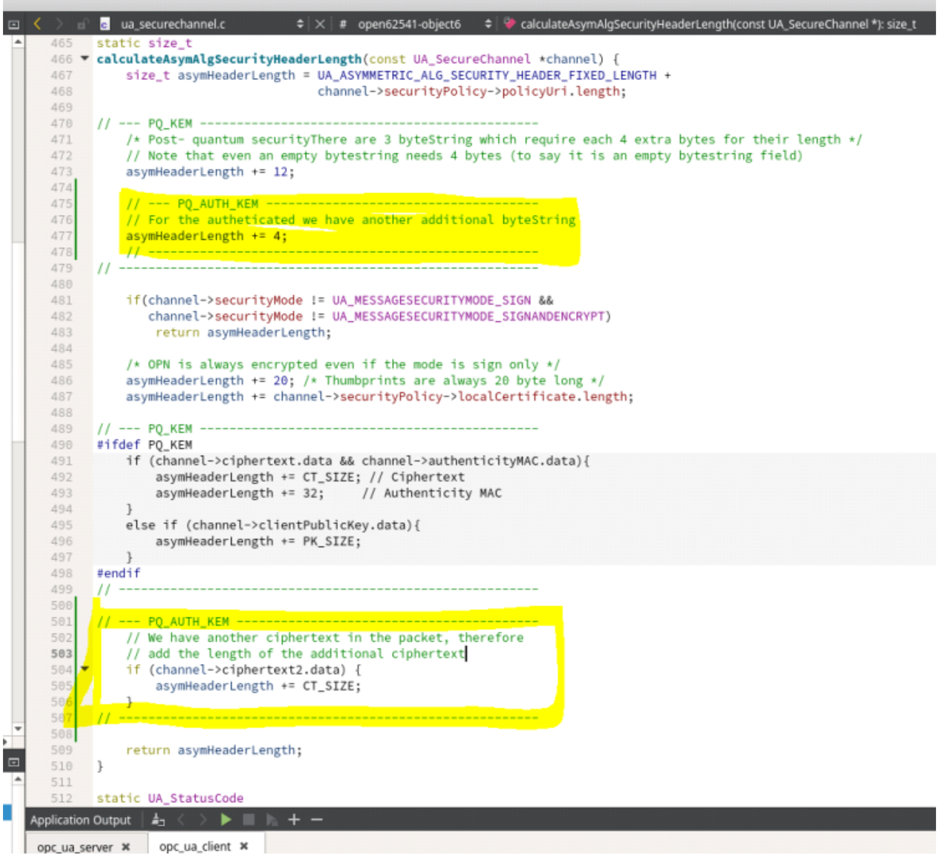
Build.

Check the file transport_generated.h if the structure is changed

```
 * Message Type and whether the message contains an intermediate chunk */
typedef enum {
    UA_MESSAGETYPE_ACK = 0x4B4341,
    UA_MESSAGETYPE_HEL = 0x4C4548,
    UA_MESSAGETYPE_MSG = 0x47534D,
    UA_MESSAGETYPE_OPN = 0x4E504F,
    UA_MESSAGETYPE_CLO = 0x4F4C43,
    UA_MESSAGETYPE_ERR = 0x525245,
    __UA_MESSAGETYPE_FORCE32BIT = 0x7fffffff
} UA_MessageType;
UA_STATIC_ASSERT(sizeof(UA_MessageType) == sizeof(UA_Int32), enum_must_be_32bit);

#define UA_TRANSPORT_MESSAGETYPE 4

/**
 * AsymmetricAlgorithmSecurityHeader
 * ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 * Security Header */
typedef struct {
    UA_ByteString securityPolicyUri;
    UA_ByteString senderCertificate;
    UA_ByteString receiverCertificateThumbprint;
    UA_ByteString clientPublicKey;
    UA_ByteString ciphertext;
    UA_ByteString authenticityMAC;
    UA_ByteString ciphertext2;
} UA_AsymmetricAlgorithmSecurityHeader;

#define UA_TRANSPORT_ASYMMETRICALGORITHMSECURITYHEADER 5

/**
 * TcpAcknowledgeMessage
 * ^^^^^^^^^^^^^^^^^^^^^
 * Acknowledge Message */
typedef struct {
    UA_UInt32 protocolVersion;
    UA_UInt32 receiveBufferSize;
    UA_UInt32 sendBufferSize;
    UA_UInt32 maxMessageSize;
    UA_UInt32 maxChunkCount;
```

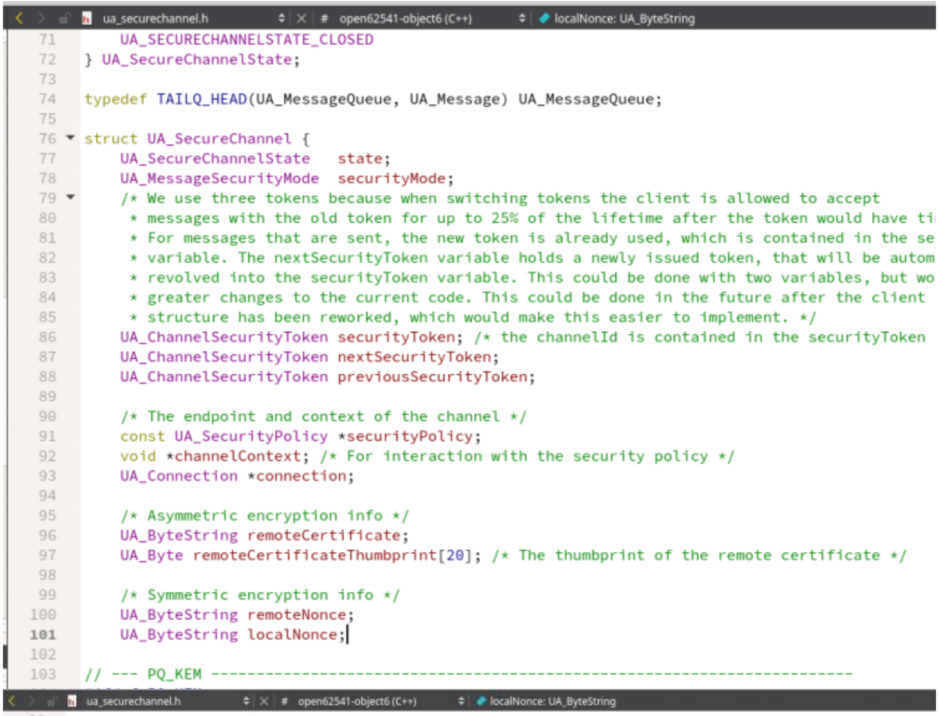
```
Internal "open62541":  transport_generated
/home/pat/programming/IIoT_PQ_AKE/Code/open62541/build/src_generated/open62541/transport_generated.c (1)
/home/pat/programming/IIoT_PQ_AKE/Code/open62541/build/src_generated/open62541/transport_generated.h (3)
    4 #ifndef TRANSPORT_GENERATED_H_
    5 #define TRANSPORT_GENERATED_H_
  181 #endif /* TRANSPORT_GENERATED_H_ */
/home/pat/programming/IIoT_PQ_AKE/Code/open62541/build/src_generated/open62541/transport_generated_encoding_binary.h (
```

Change the function that returns the length of an asymmetric encrypted message. For the empty field we add 4 bytes and then we add another ciphertext length if there is data in the channel object. Channel->ciphertext2.data has to be created in the next step.



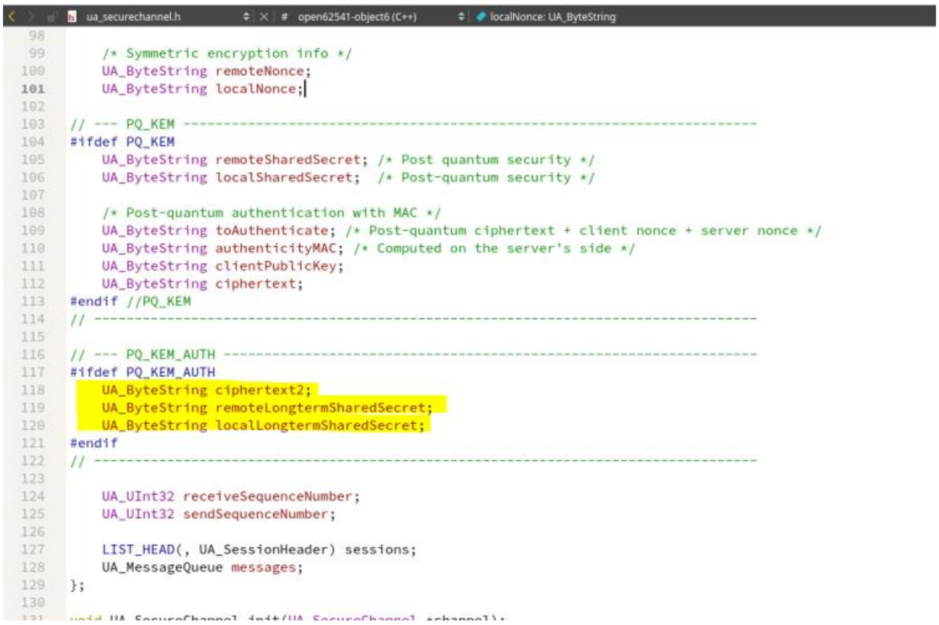
```c
static size_t
calculateAsymAlgSecurityHeaderLength(const UA_SecureChannel *channel) {
    size_t asymHeaderLength = UA_ASYMMETRIC_ALG_SECURITY_HEADER_FIXED_LENGTH +
                              channel->securityPolicy->policyUri.length;

// --- PQ_KEM ---------------------------------------------
    /* Post- quantum securityThere are 3 byteString which require each 4 extra bytes for their length */
    // Note that even an empty bytestring needs 4 bytes (to say it is an empty bytestring field)
    asymHeaderLength += 12;

    // --- PQ_AUTH_KEM ------------------------------------
    // For the autheticated we have another additional byteString
    asymHeaderLength += 4;
    // ----------------------------------------------------
// --------------------------------------------------------

    if(channel->securityMode != UA_MESSAGESECURITYMODE_SIGN &&
       channel->securityMode != UA_MESSAGESECURITYMODE_SIGNANDENCRYPT)
        return asymHeaderLength;

    /* OPN is always encrypted even if the mode is sign only */
    asymHeaderLength += 20; /* Thumbprints are always 20 byte long */
    asymHeaderLength += channel->securityPolicy->localCertificate.length;

// --- PQ_KEM ---------------------------------------------
#ifdef PQ_KEM
    if (channel->ciphertext.data && channel->authenticityMAC.data){
        asymHeaderLength += CT_SIZE; // Ciphertext
        asymHeaderLength += 32;     // Authenticity MAC
    }
    else if (channel->clientPublicKey.data){
        asymHeaderLength += PK_SIZE;
    }
#endif
// --------------------------------------------------------

// --- PQ_AUTH_KEM ----------------------------------------
    // We have another ciphertext in the packet, therefore
    // add the length of the additional ciphertext
    if (channel->ciphertext2.data) {
        asymHeaderLength += CT_SIZE;
    }
// --------------------------------------------------------

    return asymHeaderLength;
}

static UA_StatusCode
```

Adding the ciphertext2 to the channel data structure. Also add variables for the secrets that will be encapsulated and encapsulated from the cipher texts. The local long term shared secret is the one that is return when encapsulation is called and remote long term shared secret is the one that is retrieved from the decapsulation function.



```c
    UA_SECURECHANNELSTATE_CLOSED
} UA_SecureChannelState;

typedef TAILQ_HEAD(UA_MessageQueue, UA_Message) UA_MessageQueue;

struct UA_SecureChannel {
    UA_SecureChannelState   state;
    UA_MessageSecurityMode  securityMode;
    /* We use three tokens because when switching tokens the client is allowed to accept
     * messages with the old token for up to 25% of the lifetime after the token would have ti
     * For messages that are sent, the new token is already used, which is contained in the se
     * variable. The nextSecurityToken variable holds a newly issued token, that will be autom
     * revolved into the securityToken variable. This could be done with two variables, but wo
     * greater changes to the current code. This could be done in the future after the client
     * structure has been reworked, which would make this easier to implement. */
    UA_ChannelSecurityToken securityToken; /* the channelId is contained in the securityToken
    UA_ChannelSecurityToken nextSecurityToken;
    UA_ChannelSecurityToken previousSecurityToken;

    /* The endpoint and context of the channel */
    const UA_SecurityPolicy *securityPolicy;
    void *channelContext; /* For interaction with the security policy */
    UA_Connection *connection;

    /* Asymmetric encryption info */
    UA_ByteString remoteCertificate;
    UA_Byte remoteCertificateThumbprint[20]; /* The thumbprint of the remote certificate */

    /* Symmetric encryption info */
    UA_ByteString remoteNonce;
    UA_ByteString localNonce;

// --- PQ_KEM -----------------------------------------------------------
```

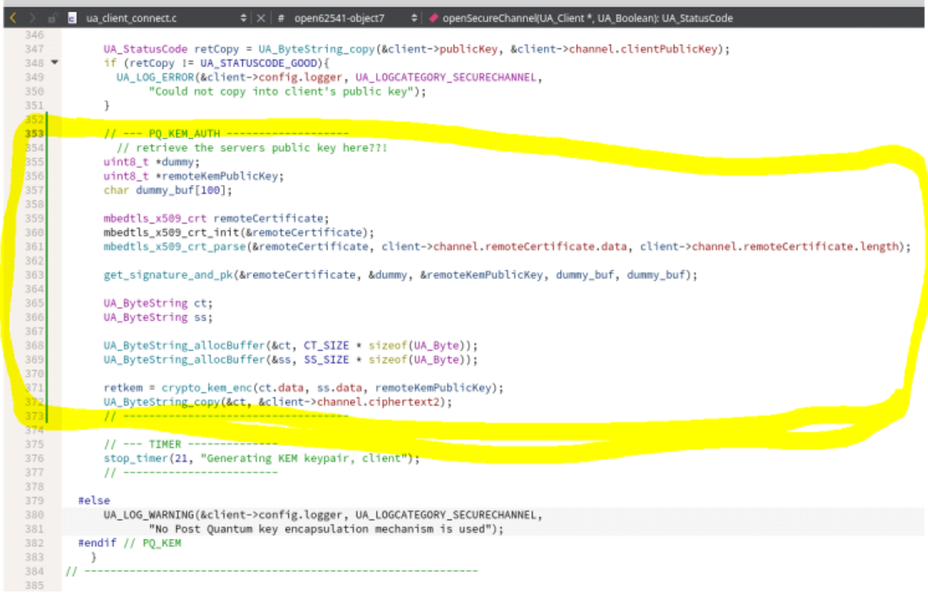
```c
    /* Symmetric encryption info */
    UA_ByteString remoteNonce;
    UA_ByteString localNonce;

// --- PQ_KEM -----------------------------------------------------------
#ifdef PQ_KEM
```



```
    /* Symmetric encryption info */
    UA_ByteString remoteNonce;
    UA_ByteString localNonce;

// --- PQ_KEM ------------------------------------------------------------------------
#ifdef PQ_KEM
    UA_ByteString remoteSharedSecret; /* Post quantum security */
    UA_ByteString localSharedSecret;  /* Post-quantum security */

    /* Post-quantum authentication with MAC */
    UA_ByteString toAuthenticate; /* Post-quantum ciphertext + client nonce + server nonce */
    UA_ByteString authenticityMAC; /* Computed on the server's side */
    UA_ByteString clientPublicKey;
    UA_ByteString ciphertext;
#endif //PQ_KEM
// ---------------------------------------------------------------------------------

// --- PQ_KEM_AUTH -------------------------------------------------------------------
#ifdef PQ_KEM_AUTH
    UA_ByteString ciphertext2;
    UA_ByteString remoteLongtermSharedSecret;
    UA_ByteString localLongtermSharedSecret;
#endif
// ---------------------------------------------------------------------------------

    UA_UInt32 receiveSequenceNumber;
    UA_UInt32 sendSequenceNumber;

    LIST_HEAD(, UA_SessionHeader) sessions;
    UA_MessageQueue messages;
};
```

When the client wants to open a new secure channel, we retrieve the public key from the remote (server) certificate and use the encapsulation function retrieve a new shared secret and a ciphertext. So far we are just transmitting the ciphertext to the server (the actual key generation comes later then we also use shared secret somewhere).

Store the ciphertext into the channel object. When the openSecureChannelRequest-asymmetric security header is created it will look into the channel object and add it.



```
        UA_StatusCode retCopy = UA_ByteString_copy(&client->publicKey, &client->channel.clientPublicKey);
        if (retCopy != UA_STATUSCODE_GOOD){
          UA_LOG_ERROR(&client->config.logger, UA_LOGCATEGORY_SECURECHANNEL,
                 "Could not copy into client's public key");
        }

        // --- PQ_KEM_AUTH ------------------
          // retrieve the servers public key here??!
        uint8_t *dummy;
        uint8_t *remoteKemPublicKey;
        char dummy_buf[100];

        mbedtls_x509_crt remoteCertificate;
        mbedtls_x509_crt_init(&remoteCertificate);
        mbedtls_x509_crt_parse(&remoteCertificate, client->channel.remoteCertificate.data, client->channel.remoteCertificate.length);

        get_signature_and_pk(&remoteCertificate, &dummy, &remoteKemPublicKey, dummy_buf, dummy_buf);

        UA_ByteString ct;
        UA_ByteString ss;

        UA_ByteString_allocBuffer(&ct, CT_SIZE * sizeof(UA_Byte));
        UA_ByteString_allocBuffer(&ss, SS_SIZE * sizeof(UA_Byte));

        retkem = crypto_kem_enc(ct.data, ss.data, remoteKemPublicKey);
        UA_ByteString_copy(&ct, &client->channel.ciphertext2);
        // --------------------------------

        // --- TIMER -------------
        stop_timer(21, "Generating KEM keypair, client");
        // ----------------------

    #else
        UA_LOG_WARNING(&client->config.logger, UA_LOGCATEGORY_SECURECHANNEL,
               "No Post Quantum key encapsulation mechanism is used");
    #endif // PQ_KEM
      }
// ------------------------------------------------------------
```

Add the ciphertext2 from the channel object to the asymmetric security header

```
            channel->securityMode == UA_MESSAGESECURITYMODE_SIGNANDENCRYPT) {
             asymHeader.senderCertificate = channel->securityPolicy->localCertificate;
             asymHeader.receiverCertificateThumbprint.length = 20;
             asymHeader.receiverCertificateThumbprint.data = channel->remoteCertificateThumbprint;
        }

// --- PQ_KEM ---------------------------------------------------------------------
#ifdef PQ_KEM
  if(channel->securityMode != UA_MESSAGESECURITYMODE_SIGN &&
     channel->securityMode != UA_MESSAGESECURITYMODE_SIGNANDENCRYPT){
    asymHeader.clientPublicKey = UA_BYTESTRING_NULL;
    asymHeader.ciphertext = UA_BYTESTRING_NULL;
    asymHeader.authenticityMAC = UA_BYTESTRING_NULL;
    asymHeader.ciphertext2 = UA_BYTESTRING_NULL;
  }
  else {
    // --- PQ_KEM_AUTH ---------------
      if (channel->ciphertext2.length == CT_SIZE) {
          UA_ByteString_copy(&channel->ciphertext2, &asymHeader.ciphertext2);
      }
    // --------------------------------
    if (channel->ciphertext.length == CT_SIZE && channel->authenticityMAC.length == 32){
      UA_ByteString_copy(&channel->ciphertext, &asymHeader.ciphertext);
      UA_ByteString_copy(&channel->authenticityMAC, &asymHeader.authenticityMAC);
      asymHeader.clientPublicKey = UA_BYTESTRING_NULL;
    }
    else if (channel->clientPublicKey.length == PK_SIZE){
      UA_ByteString_copy(&channel->clientPublicKey, &asymHeader.clientPublicKey);
      asymHeader.ciphertext = UA_BYTESTRING_NULL;
      asymHeader.authenticityMAC = UA_BYTESTRING_NULL;
    }
    else {
      UA_LOG_ERROR(channel->securityPolicy->logger, UA_LOGCATEGORY_SECURECHANNEL,
           "Internal error.");
    }
  }
#else
```

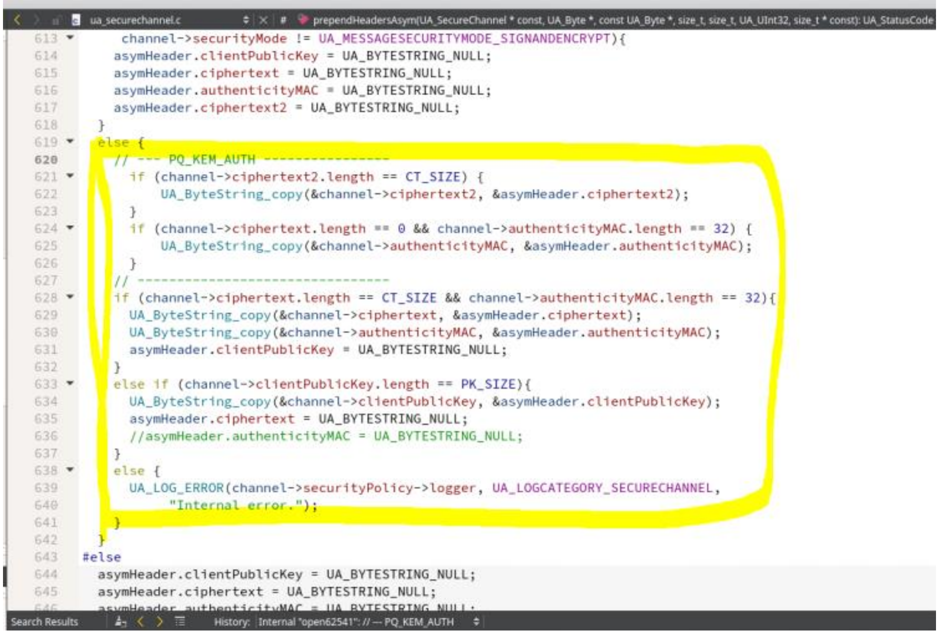
And when the asymmetric security header is evaluated, retrieve the sent data and store it into the channel object. This way the server will have access to the ciphertext.

```
        compareCertificateThumbprint(securityPolicy,
                                     &asymHeader->receiverCertificateThumbprint);
    if(retval != UA_STATUSCODE_GOOD) {
        return retval;
    }

// --- PQ_KEM ------------------------------------------------------------
#ifdef PQ_KEM
  /* Get back whatever I need to put it in channel.*/
  /*The client and the server will check that later with the proper data */
  if (asymHeader->clientPublicKey.data){
    UA_ByteString_copy(&asymHeader->clientPublicKey, &channel->clientPublicKey);
  }
  if (asymHeader->ciphertext.data){
  UA_ByteString_copy(&asymHeader->ciphertext, &channel->ciphertext);
  }
  if (asymHeader->authenticityMAC.data){
  UA_ByteString_copy(&asymHeader->authenticityMAC, &channel->authenticityMAC);
  }
#endif
// ---------------------------------------------------------------------

// --- PQ_KEM_AUTH -----------------------------------------------------
  if (asymHeader->ciphertext2.data) {
      UA_ByteString_copy(&asymHeader->ciphertext2, &channel->ciphertext2);
  }
// ---------------------------------------------------------------------

    return UA_STATUSCODE_GOOD;
}

static UA_StatusCode
checkPreviousToken(UA_SecureChannel *const channel, const UA_UInt32 tokenId) {
    if(tokenId != channel->previousSecurityToken.tokenId)
        return UA_STATUSCODE_BADSECURECHANNELTOKENUNKNOWN;
```

3. Adapt Functions related to Parsing the openSecureChannel Request/Response

Before a message is sent, the asymmetric security header is added. In this function we need to copy the ciphertext etc. data from the channel data structure into the buffer that is actually handed down to the network layer.

Therefore function prependHeaderAsym has to be changed:

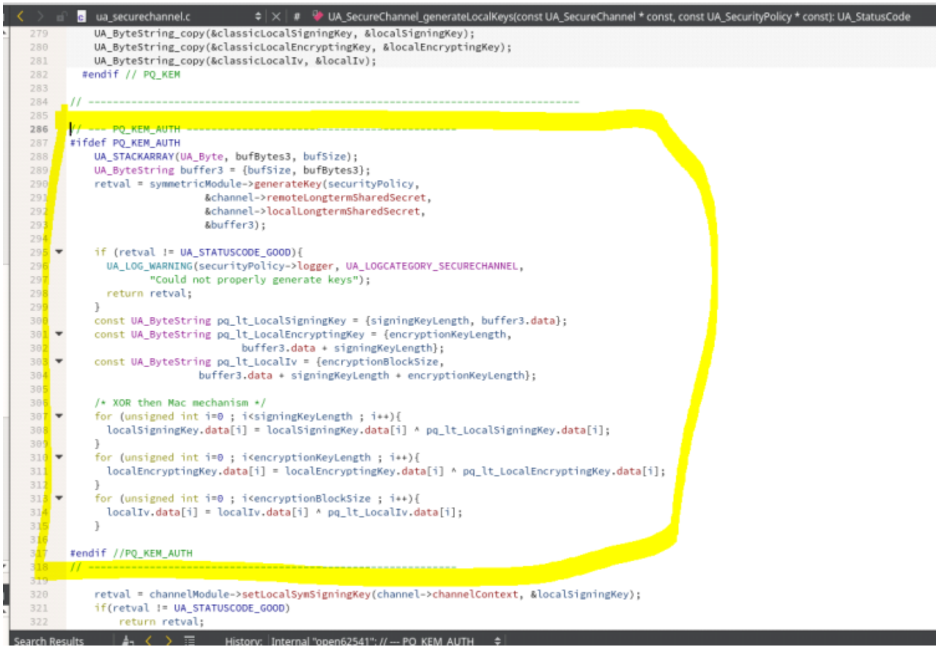


```
     channel->securityMode != UA_MESSAGESECURITYMODE_SIGNANDENCRYPT){
    asymHeader.clientPublicKey = UA_BYTESTRING_NULL;
    asymHeader.ciphertext = UA_BYTESTRING_NULL;
    asymHeader.authenticityMAC = UA_BYTESTRING_NULL;
    asymHeader.ciphertext2 = UA_BYTESTRING_NULL;
  }
  else {
    // --- PQ_KEM_AUTH ----------------
      if (channel->ciphertext2.length == CT_SIZE) {
          UA_ByteString_copy(&channel->ciphertext2, &asymHeader.ciphertext2);
      }
      if (channel->ciphertext.length == 0 && channel->authenticityMAC.length == 32) {
          UA_ByteString_copy(&channel->authenticityMAC, &asymHeader.authenticityMAC);
      }
    // --------------------------------
    if (channel->ciphertext.length == CT_SIZE && channel->authenticityMAC.length == 32){
      UA_ByteString_copy(&channel->ciphertext, &asymHeader.ciphertext);
      UA_ByteString_copy(&channel->authenticityMAC, &asymHeader.authenticityMAC);
      asymHeader.clientPublicKey = UA_BYTESTRING_NULL;
    }
    else if (channel->clientPublicKey.length == PK_SIZE){
      UA_ByteString_copy(&channel->clientPublicKey, &asymHeader.clientPublicKey);
      asymHeader.ciphertext = UA_BYTESTRING_NULL;
      //asymHeader.authenticityMAC = UA_BYTESTRING_NULL;
    }
    else {
      UA_LOG_ERROR(channel->securityPolicy->logger, UA_LOGCATEGORY_SECURECHANNEL,
            "Internal error.");
    }
  }
#else
  asymHeader.clientPublicKey = UA_BYTESTRING_NULL;
  asymHeader.ciphertext = UA_BYTESTRING_NULL;
  asymHeader.authenticityMAC = UA_BYTESTRING_NULL;
```

And when a message is received, when the asymmetric security header is checked, the relevant data has to be copied into the channel data structure so that it is available in the later processing functions.
This is done in the function checkAsymHeader().

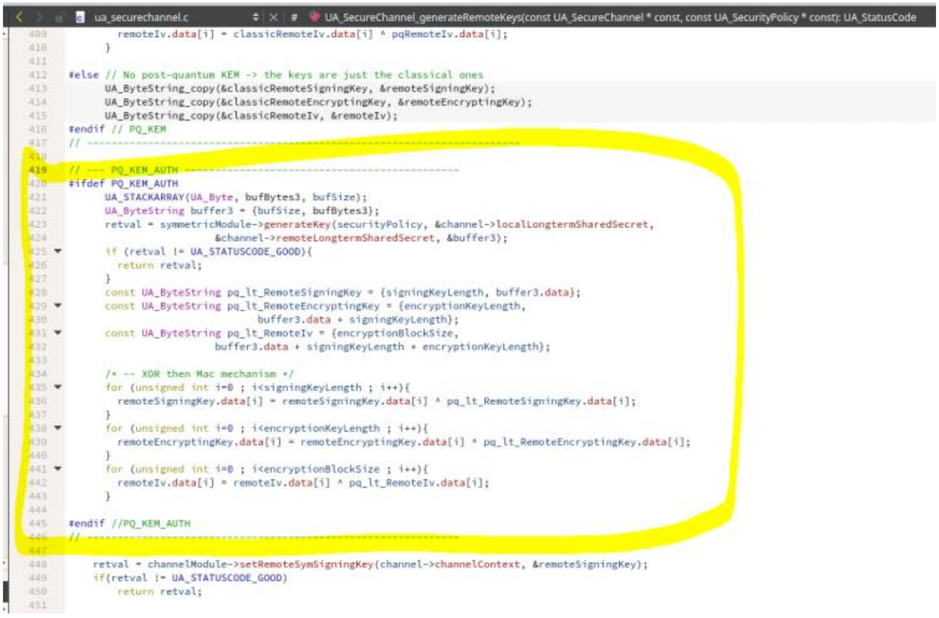
```

    if(!UA_ByteString_equal(&securityPolicy->policyUri,
                            &asymHeader->securityPolicyUri)) {
        return UA_STATUSCODE_BADSECURITYPOLICYREJECTED;
    }

    // TODO: Verify certificate using certificate plugin. This will come with a new PR
    /* Something like this
    retval = certificateManager->verify(certificateStore??, &asymHeader->senderCertificate);
    if(retval != UA_STATUSCODE_GOOD)
    return retval;
    */
    UA_StatusCode retval = securityPolicy->asymmetricModule.
        compareCertificateThumbprint(securityPolicy,
                                     &asymHeader->receiverCertificateThumbprint);
    if(retval != UA_STATUSCODE_GOOD) {
        return retval;
    }

// --- PQ_KEM ------------------------------------------------------------
#ifdef PQ_KEM
  /* Get back whatever I need to put it in channel.*/
  /*The client and the server will check that later with the proper data */
  if (asymHeader->clientPublicKey.data){
    UA_ByteString_copy(&asymHeader->clientPublicKey, &channel->clientPublicKey);
  }
  if (asymHeader->ciphertext.data){
  UA_ByteString_copy(&asymHeader->ciphertext, &channel->ciphertext);
  }
  if (asymHeader->authenticityMAC.data){
  UA_ByteString_copy(&asymHeader->authenticityMAC, &channel->authenticityMAC);
  }
#endif
// ------------------------------------------------------------------------

// --- PQ_KEM_AUTH --------------------------------------------------------
  if (asymHeader->ciphertext2.data) {
      UA_ByteString_copy(&asymHeader->ciphertext2, &channel->ciphertext2);
  }
// ------------------------------------------------------------------------

    return UA_STATUSCODE_GOOD;
}

```

When the local signing keys are generated from the long term shared secrets (which are extracted/created in the next sections) we have to XOR them in. The generateKey() function from the security policy takes the shared secret as inputs and generates a pseudorandom sequence into buffer3 of the required length. Then buffer3 is split up and its parts (IV, localSigningKey, localEncryptionKey) and is XOR with the already existing keys, in order to make it hybrid.



```c
    UA_ByteString_copy(&classicLocalSigningKey, &localSigningKey);
    UA_ByteString_copy(&classicLocalEncryptingKey, &localEncryptingKey);
    UA_ByteString_copy(&classicLocalIv, &localIv);
  #endif // PQ_KEM

// ---------------------------------------------------------------------------

// --- PQ_KEM_AUTH ----------------------------------------------
#ifdef PQ_KEM_AUTH
    UA_STACKARRAY(UA_Byte, bufBytes3, bufSize);
    UA_ByteString buffer3 = {bufSize, bufBytes3};
    retval = symmetricModule->generateKey(securityPolicy,
                      &channel->remoteLongtermSharedSecret,
                      &channel->localLongtermSharedSecret,
                      &buffer3);

    if (retval != UA_STATUSCODE_GOOD){
      UA_LOG_WARNING(securityPolicy->logger, UA_LOGCATEGORY_SECURECHANNEL,
             "Could not properly generate keys");
      return retval;
    }
    const UA_ByteString pq_lt_LocalSigningKey = {signingKeyLength, buffer3.data};
    const UA_ByteString pq_lt_LocalEncryptingKey = {encryptionKeyLength,
                           buffer3.data + signingKeyLength};
    const UA_ByteString pq_lt_LocalIv = {encryptionBlockSize,
                      buffer3.data + signingKeyLength + encryptionKeyLength};

    /* XOR then Mac mechanism */
    for (unsigned int i=0 ; i<signingKeyLength ; i++){
      localSigningKey.data[i] = localSigningKey.data[i] ^ pq_lt_LocalSigningKey.data[i];
    }
    for (unsigned int i=0 ; i<encryptionKeyLength ; i++){
      localEncryptingKey.data[i] = localEncryptingKey.data[i] ^ pq_lt_LocalEncryptingKey.data[i];
    }
    for (unsigned int i=0 ; i<encryptionBlockSize ; i++){
      localIv.data[i] = localIv.data[i] ^ pq_lt_LocalIv.data[i];
    }

#endif //PQ_KEM_AUTH
// ------------------------------------------------------------

    retval = channelModule->setLocalSymSigningKey(channel->channelContext, &localSigningKey);
    if(retval != UA_STATUSCODE_GOOD)
       return retval;
```

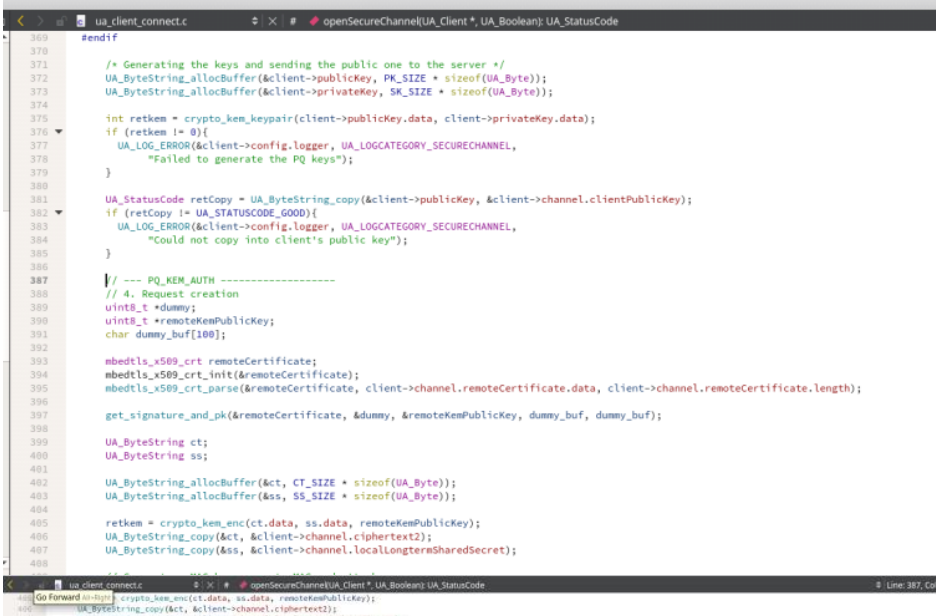
The same is done for the remote local key.



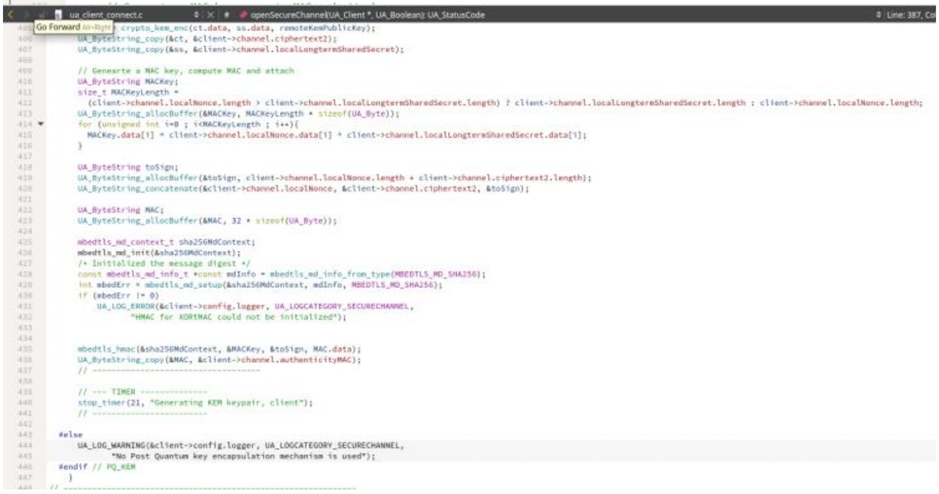
```c
        remoteIv.data[i] = classicRemoteIv.data[i] ^ pqRemoteIv.data[i];
      }

#else // No post-quantum KEM -> the keys are just the classical ones
      UA_ByteString_copy(&classicRemoteSigningKey, &remoteSigningKey);
      UA_ByteString_copy(&classicRemoteEncryptingKey, &remoteEncryptingKey);
      UA_ByteString_copy(&classicRemoteIv, &remoteIv);
#endif // PQ_KEM
// ----------------------------------------------------------------------

// --- PQ_KEM_AUTH ----------------------------------------------
#ifdef PQ_KEM_AUTH
      UA_STACKARRAY(UA_Byte, bufBytes3, bufSize);
      UA_ByteString buffer3 = {bufSize, bufBytes3};
      retval = symmetricModule->generateKey(securityPolicy, &channel->localLongtermSharedSecret,
                        &channel->remoteLongtermSharedSecret, &buffer3);
      if (retval != UA_STATUSCODE_GOOD){
        return retval;
      }
      const UA_ByteString pq_lt_RemoteSigningKey = {signingKeyLength, buffer3.data};
      const UA_ByteString pq_lt_RemoteEncryptingKey = {encryptionKeyLength,
                             buffer3.data + signingKeyLength};
      const UA_ByteString pq_lt_RemoteIv = {encryptionBlockSize,
                        buffer3.data + signingKeyLength + encryptionKeyLength};

      /* -- XOR then Mac mechanism */
      for (unsigned int i=0 ; i<signingKeyLength ; i++){
        remoteSigningKey.data[i] = remoteSigningKey.data[i] ^ pq_lt_RemoteSigningKey.data[i];
      }
      for (unsigned int i=0 ; i<encryptionKeyLength ; i++){
        remoteEncryptingKey.data[i] = remoteEncryptingKey.data[i] ^ pq_lt_RemoteEncryptingKey.data[i];
      }
      for (unsigned int i=0 ; i<encryptionBlockSize ; i++){
        remoteIv.data[i] = remoteIv.data[i] ^ pq_lt_RemoteIv.data[i];
      }

#endif //PQ_KEM_AUTH
// ------------------------------------------------------------

    retval = channelModule->setRemoteSymSigningKey(channel->channelContext, &remoteSigningKey);
    if(retval != UA_STATUSCODE_GOOD)
       return retval;
```

4. Request Creation on Client Side

The remote KEM public key is retrieved from the remote certificate. Then the encapsulation function of the KEM is used (crypto_kem_enc()) and cipher text and shared secret are stored in the channel data structure. Later when the message is sent, that information is automatically embedded into the message. After that a MAC is calculated. The MAC key is the XOR of the local nonce (standard OPC UA) and the local shared secret (from the KEM).



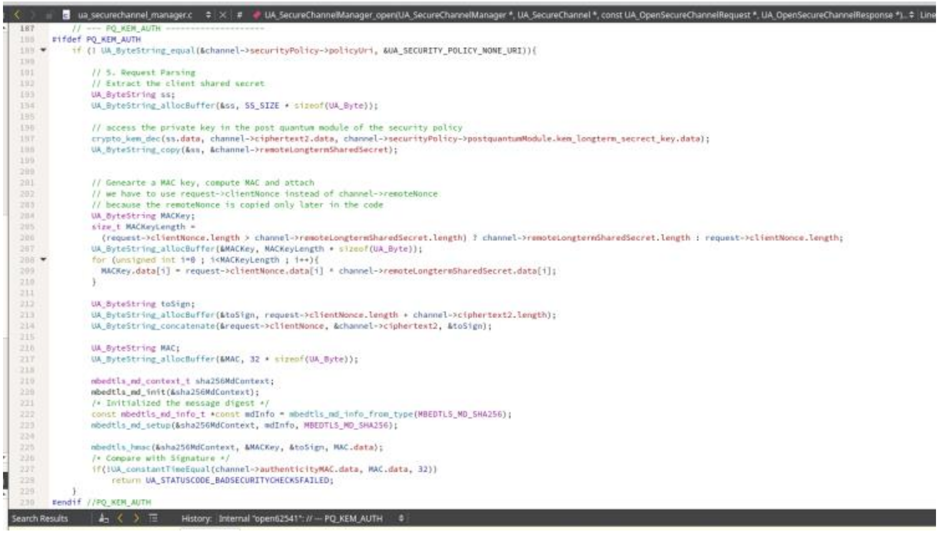
```c
#endif

    /* Generating the keys and sending the public one to the server */
    UA_ByteString_allocBuffer(&client->publicKey, PK_SIZE * sizeof(UA_Byte));
    UA_ByteString_allocBuffer(&client->privateKey, SK_SIZE * sizeof(UA_Byte));

    int retkem = crypto_kem_keypair(client->publicKey.data, client->privateKey.data);
    if (retkem != 0){
      UA_LOG_ERROR(&client->config.logger, UA_LOGCATEGORY_SECURECHANNEL,
             "Failed to generate the PQ keys");
    }

    UA_StatusCode retCopy = UA_ByteString_copy(&client->publicKey, &client->channel.clientPublicKey);
    if (retCopy != UA_STATUSCODE_GOOD){
      UA_LOG_ERROR(&client->config.logger, UA_LOGCATEGORY_SECURECHANNEL,
             "Could not copy into client's public key");
    }

    // --- PQ_KEM_AUTH -------------------
    // 4. Request creation
    uint8_t *dummy;
    uint8_t *remoteKemPublicKey;
    char dummy_buf[100];

    mbedtls_x509_crt remoteCertificate;
    mbedtls_x509_crt_init(&remoteCertificate);
    mbedtls_x509_crt_parse(&remoteCertificate, client->channel.remoteCertificate.data, client->channel.remoteCertificate.length);

    get_signature_and_pk(&remoteCertificate, &dummy, &remoteKemPublicKey, dummy_buf, dummy_buf);

    UA_ByteString ct;
    UA_ByteString ss;

    UA_ByteString_allocBuffer(&ct, CT_SIZE * sizeof(UA_Byte));
    UA_ByteString_allocBuffer(&ss, SS_SIZE * sizeof(UA_Byte));

    retkem = crypto_kem_enc(ct.data, ss.data, remoteKemPublicKey);
    UA_ByteString_copy(&ct, &client->channel.ciphertext2);
    UA_ByteString_copy(&ss, &client->channel.localLongtermSharedSecret);
```



```c
    crypto_kem_enc(ct.data, ss.data, remoteKemPublicKey);
    UA_ByteString_copy(&ct, &client->channel.ciphertext2);
    UA_ByteString_copy(&ss, &client->channel.localLongtermSharedSecret);

    // Genearte a MAC key, compute MAC and attach
    UA_ByteString MACKey;
    size_t MACKeyLength =
      (client->channel.localNonce.length > client->channel.localLongtermSharedSecret.length) ? client->channel.localLongtermSharedSecret.length : client->channel.localNonce.length;
    UA_ByteString_allocBuffer(&MACKey, MACKeyLength * sizeof(UA_Byte));
    for (unsigned int i=0 ; i<MACKeyLength ; i++){
      MACKey.data[i] = client->channel.localNonce.data[i] ^ client->channel.localLongtermSharedSecret.data[i];
    }

    UA_ByteString toSign;
    UA_ByteString_allocBuffer(&toSign, client->channel.localNonce.length + client->channel.ciphertext2.length);
    UA_ByteString_concatenate(&client->channel.localNonce, &client->channel.ciphertext2, &toSign);

    UA_ByteString MAC;
    UA_ByteString_allocBuffer(&MAC, 32 * sizeof(UA_Byte));

    mbedtls_md_context_t sha256MdContext;
    mbedtls_md_init(&sha256MdContext);
    /* Initialized the message digest */
    const mbedtls_md_info_t *const mdInfo = mbedtls_md_info_from_type(MBEDTLS_MD_SHA256);
    int mbedErr = mbedtls_md_setup(&sha256MdContext, mdInfo, MBEDTLS_MD_SHA256);
    if (mbedErr != 0)
        UA_LOG_ERROR(&client->config.logger, UA_LOGCATEGORY_SECURECHANNEL,
               "HMAC for XORtMAC could not be initialized");


    mbedtls_hmac(&sha256MdContext, &MACKey, &toSign, MAC.data);
    UA_ByteString_copy(&MAC, &client->channel.authenticityMAC);
    // -----------------------------------

    // --- TIMER -------------
    stop_timer(21, "Generating KEM keypair, client");
    // ----------------------

  #else
    UA_LOG_WARNING(&client->config.logger, UA_LOGCATEGORY_SECURECHANNEL,
           "No Post Quantum key encapsulation mechanism is used");
  #endif // PQ_KEM
  }
// ----------------------------------------------------------
```

5. Request Parsing on Server Side

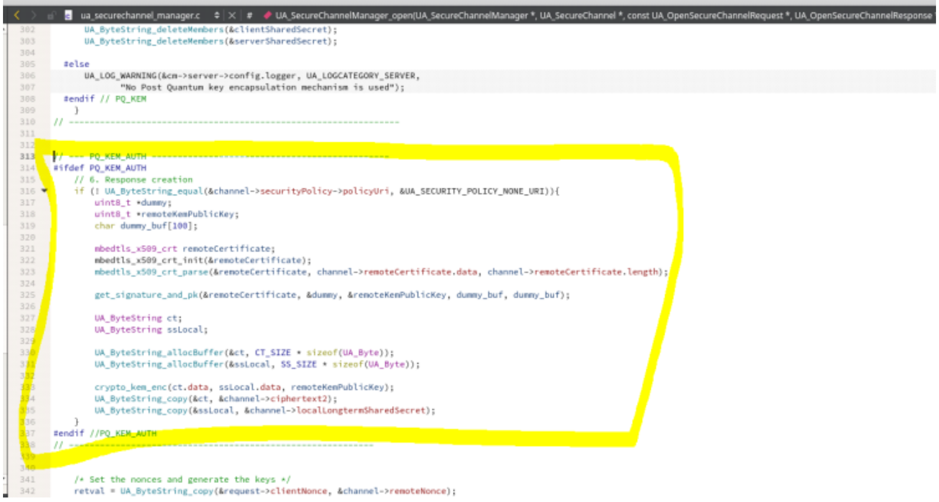
The shared secret is extracted from the received ciphertext. The ciphertext is already available in the channel struct. The extracted secret is also stored as the remote long term shared secret in the channel struct. Then also the MAC is verified.



```c
    // --- PQ_KEM_AUTH --------------------
#ifdef PQ_KEM_AUTH
    if (! UA_ByteString_equal(&channel->securityPolicy->policyUri, &UA_SECURITY_POLICY_NONE_URI)){

        // 5. Request Parsing
        // Extract the client shared secret
        UA_ByteString ss;
        UA_ByteString_allocBuffer(&ss, SS_SIZE * sizeof(UA_Byte));

        // access the private key in the post quantum module of the security policy
        crypto_kem_dec(ss.data, channel->ciphertext2.data, channel->securityPolicy->postquantumModule.kem_longterm_secrect_key.data);
        UA_ByteString_copy(&ss, &channel->remoteLongtermSharedSecret);


        // Genearte a MAC key, compute MAC and attach
        // we have to use request->clientNonce instead of channel->remoteNonce
        // because the remoteNonce is copied only later in the code
        UA_ByteString MACKey;
        size_t MACKeyLength =
          (request->clientNonce.length > channel->remoteLongtermSharedSecret.length) ? channel->remoteLongtermSharedSecret.length : request->clientNonce.length;
        UA_ByteString_allocBuffer(&MACKey, MACKeyLength * sizeof(UA_Byte));
        for (unsigned int i=0 ; i<MACKeyLength ; i++){
          MACKey.data[i] = request->clientNonce.data[i] ^ channel->remoteLongtermSharedSecret.data[i];
        }

        UA_ByteString toSign;
        UA_ByteString_allocBuffer(&toSign, request->clientNonce.length + channel->ciphertext2.length);
        UA_ByteString_concatenate(&request->clientNonce, &channel->ciphertext2, &toSign);

        UA_ByteString MAC;
        UA_ByteString_allocBuffer(&MAC, 32 * sizeof(UA_Byte));

        mbedtls_md_context_t sha256MdContext;
        mbedtls_md_init(&sha256MdContext);
        /* Initialized the message digest */
        const mbedtls_md_info_t *const mdInfo = mbedtls_md_info_from_type(MBEDTLS_MD_SHA256);
        mbedtls_md_setup(&sha256MdContext, mdInfo, MBEDTLS_MD_SHA256);

        mbedtls_hmac(&sha256MdContext, &MACKey, &toSign, MAC.data);
        /* Compare with Signature */
        if(!UA_constantTimeEqual(channel->authenticityMAC.data, MAC.data, 32))
            return UA_STATUSCODE_BADSECURITYCHECKSFAILED;
    }
#endif //PQ_KEM_AUTH
```

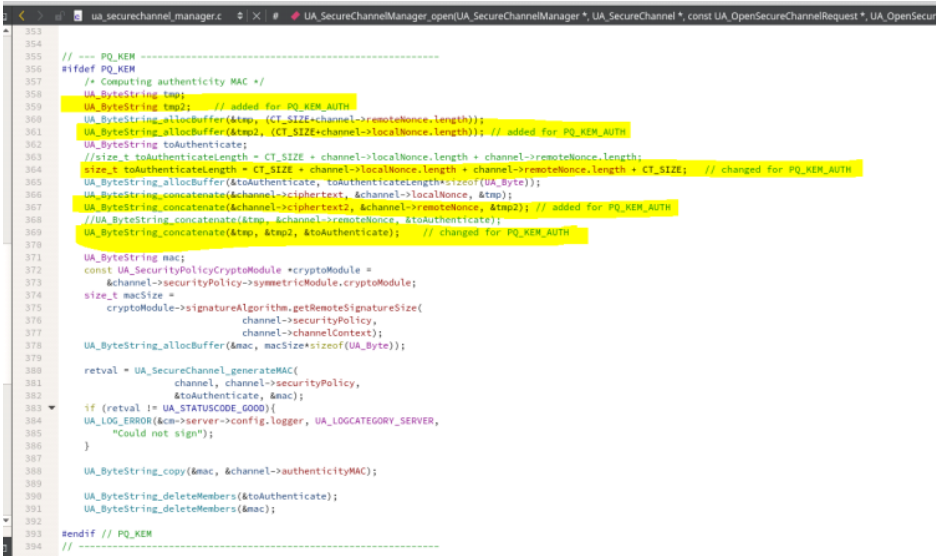
6. Response Creation on Server Side

Then the server also encapsulates two secrets: The long term secret and the ephemeral secret



```
        UA_ByteString_deleteMembers(&clientSharedSecret);
        UA_ByteString_deleteMembers(&serverSharedSecret);

    #else
        UA_LOG_WARNING(&cm->server->config.logger, UA_LOGCATEGORY_SERVER,
                   "No Post Quantum key encapsulation mechanism is used");
    #endif // PQ_KEM
      }
  // ------------------------------------------------------------


  // --- PQ_KEM_AUTH ------------------------------------------
  #ifdef PQ_KEM_AUTH
      // 6. Response creation
      if (! UA_ByteString_equal(&channel->securityPolicy->policyUri, &UA_SECURITY_POLICY_NONE_URI)){
          uint8_t *dummy;
          uint8_t *remoteKemPublicKey;
          char dummy_buf[100];

          mbedtls_x509_crt remoteCertificate;
          mbedtls_x509_crt_init(&remoteCertificate);
          mbedtls_x509_crt_parse(&remoteCertificate, channel->remoteCertificate.data, channel->remoteCertificate.length);

          get_signature_and_pk(&remoteCertificate, &dummy, &remoteKemPublicKey, dummy_buf, dummy_buf);

          UA_ByteString ct;
          UA_ByteString ssLocal;

          UA_ByteString_allocBuffer(&ct, CT_SIZE * sizeof(UA_Byte));
          UA_ByteString_allocBuffer(&ssLocal, SS_SIZE * sizeof(UA_Byte));

          crypto_kem_enc(ct.data, ssLocal.data, remoteKemPublicKey);
          UA_ByteString_copy(&ct, &channel->ciphertext2);
          UA_ByteString_copy(&ssLocal, &channel->localLongtermSharedSecret);
      }
  #endif //PQ_KEM_AUTH
  // ---------------------------------------------------------


      /* Set the nonces and generate the keys */
      retval = UA_ByteString_copy(&request->clientNonce, &channel->remoteNonce);
```

When the MAC is calculated (which was already done with the unauthenticated PQ KEM), we need to include the new ciphertext2 into the data that is signed. The new long term shared secret is already part of the derived local keys, which are used as the key for the MAC.



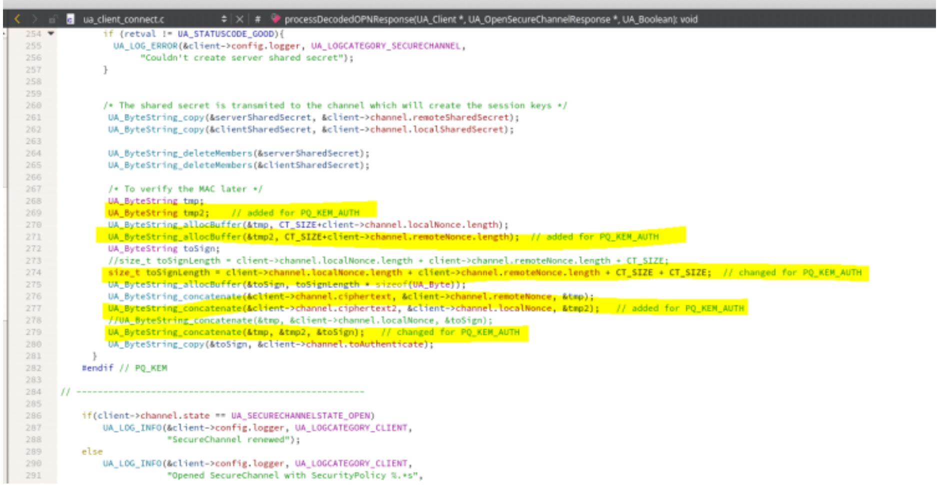
```


// --- PQ_KEM ------------------------------------------------------
#ifdef PQ_KEM
    /* Computing authenticity MAC */
    UA_ByteString tmp;
    UA_ByteString tmp2;    // added for PQ_KEM_AUTH
    UA_ByteString_allocBuffer(&tmp, (CT_SIZE+channel->remoteNonce.length));
    UA_ByteString_allocBuffer(&tmp2, (CT_SIZE+channel->localNonce.length)); // added for PQ_KEM_AUTH
    UA_ByteString toAuthenticate;
    //size_t toAuthenticateLength = CT_SIZE + channel->localNonce.length + channel->remoteNonce.length;
    size_t toAuthenticateLength = CT_SIZE + channel->localNonce.length + channel->remoteNonce.length + CT_SIZE;   // changed for PQ_KEM_AUTH
    UA_ByteString_allocBuffer(&toAuthenticate, toAuthenticateLength*sizeof(UA_Byte));
    UA_ByteString_concatenate(&channel->ciphertext, &channel->localNonce, &tmp);
    UA_ByteString_concatenate(&channel->ciphertext2, &channel->remoteNonce, &tmp2); // added for PQ_KEM_AUTH
    //UA_ByteString_concatenate(&tmp, &channel->remoteNonce, &toAuthenticate);
    UA_ByteString_concatenate(&tmp, &tmp2, &toAuthenticate);    // changed for PQ_KEM_AUTH

    UA_ByteString mac;
    const UA_SecurityPolicyCryptoModule *cryptoModule =
        &channel->securityPolicy->symmetricModule.cryptoModule;
    size_t macSize =
        cryptoModule->signatureAlgorithm.getRemoteSignatureSize(
                                 channel->securityPolicy,
                                 channel->channelContext);
    UA_ByteString_allocBuffer(&mac, macSize*sizeof(UA_Byte));

    retval = UA_SecureChannel_generateMAC(
                    channel, channel->securityPolicy,
                    &toAuthenticate, &mac);
    if (retval != UA_STATUSCODE_GOOD){
    UA_LOG_ERROR(&cm->server->config.logger, UA_LOGCATEGORY_SERVER,
        "Could not sign");
    }

    UA_ByteString_copy(&mac, &channel->authenticityMAC);

    UA_ByteString_deleteMembers(&toAuthenticate);
    UA_ByteString_deleteMembers(&mac);

#endif // PQ_KEM
// ------------------------------------------------------------
```

7. Response Parsing at Client Side

Finally the client needs to extract the long term shared secret that the server sent.

```
edOPNResponse(UA_Client *client, UA_OpenSecureChannelResponse *response,
                                UA_Boolean renew) {
    /* Replace the token */
    if(renew)
        client->channel.nextSecurityToken = response->securityToken;
    else
        client->channel.securityToken = response->securityToken;

    /* Replace the nonce */
    UA_ByteString_deleteMembers(&client->channel.remoteNonce);
    client->channel.remoteNonce = response->serverNonce;
    UA_ByteString_init(&response->serverNonce);


// --- PQ_KEM_AUTH --------------------------------------
// 7. Response Parsing
#ifdef PQ_KEM_AUTH
    if (! UA_ByteString_equal(&client->channel.securityPolicy->policyUri, &UA_SECURITY_POLICY_NONE_URI)){
        UA_ByteString ss;
        UA_ByteString_allocBuffer(&ss, SS_SIZE * sizeof(UA_Byte));

        // access the private key in the post quantum module of the security policy
        crypto_kem_dec(ss.data, client->channel.ciphertext2.data, client->channel.securityPolicy->postquantumModule.kem_longterm_secrect_key.data);
        UA_ByteString_copy(&ss, &client->channel.remoteLongtermSharedSecret);
    }
#endif //PQ_KEM_AUTH
// ------------------------------------------------------

// --- PQ_KEM -------------------------------------------
    /* Post Quantum KEM */
    #ifdef PQ_KEM
```

The verification of the MAC has to be adjusted, because the ciphertext2 has become part of the message that was signed.



```c
        if (retval != UA_STATUSCODE_GOOD){
          UA_LOG_ERROR(&client->config.logger, UA_LOGCATEGORY_SECURECHANNEL,
                "Couldn't create server shared secret");
        }


        /* The shared secret is transmited to the channel which will create the session keys */
         UA_ByteString_copy(&serverSharedSecret, &client->channel.remoteSharedSecret);
         UA_ByteString_copy(&clientSharedSecret, &client->channel.localSharedSecret);

         UA_ByteString_deleteMembers(&serverSharedSecret);
         UA_ByteString_deleteMembers(&clientSharedSecret);

         /* To verify the MAC later */
         UA_ByteString tmp;
         UA_ByteString tmp2;    // added for PQ_KEM_AUTH
         UA_ByteString_allocBuffer(&tmp, CT_SIZE+client->channel.localNonce.length);
         UA_ByteString_allocBuffer(&tmp2, CT_SIZE+client->channel.remoteNonce.length);  // added for PQ_KEM_AUTH
         UA_ByteString toSign;
         //size_t toSignLength = client->channel.localNonce.length + client->channel.remoteNonce.length + CT_SIZE;
         size_t toSignLength = client->channel.localNonce.length + client->channel.remoteNonce.length + CT_SIZE + CT_SIZE;  // changed for PQ_KEM_AUTH
         UA_ByteString_allocBuffer(&toSign, toSignLength * sizeof(UA_Byte));
         UA_ByteString_concatenate(&client->channel.ciphertext, &client->channel.remoteNonce, &tmp);
         UA_ByteString_concatenate(&client->channel.ciphertext2, &client->channel.localNonce, &tmp2);   // added for PQ_KEM_AUTH
         //UA_ByteString_concatenate(&tmp, &client->channel.localNonce, &toSign);
         UA_ByteString_concatenate(&tmp, &tmp2, &toSign);   // changed for PQ_KEM_AUTH
         UA_ByteString_copy(&toSign, &client->channel.toAuthenticate);
      }
    #endif // PQ_KEM

// ---------------------------------------------------------

    if(client->channel.state == UA_SECURECHANNELSTATE_OPEN)
        UA_LOG_INFO(&client->config.logger, UA_LOGCATEGORY_CLIENT,
                    "SecureChannel renewed");
    else
        UA_LOG_INFO(&client->config.logger, UA_LOGCATEGORY_CLIENT,
                    "Opened SecureChannel with SecurityPolicy %.*s",
```