Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Formal verification and risk assessment of an implementation of the OPC-UA Protocol

Candidato:
Enrico Guerra
Matricola VR439666

Relatore:
Prof. Mariano Ceccato

Correlatore:
Marco Rocchetto

Anno Accademico 2021-2022

Contents

List of Figures

1 Introduction

In the context of cybersecurity, companies study the risk of their infrastructures by identifying their assets (e.g. hardware, PLCs, customer data) and by identifying, analysing and evaluating the risks that may affect these assets. In ISO 27000[1], risks are associated with the loss of confidentiality, integrity, and availability for information within the scope of the information security management system". Industrial Control Systems (ICS) are more and more widespread in the world and by now they closely affect our lives. The number of ICS components available over the Internet increases every year, including "manufacturing, transportation, building automation, water treatment, and energy"[2]. In the recent past, industrial Control Systems have been victims of complex cyber attacks, for example:

- Stuxnet [3], a cyber attack that damaged the nuclear program of Iran by attacking the PLC systems of its nuclear centrifuges.
- The Colonial Pipeline ransomware attack[4], where a group of hackers attacked the billing infrastructure of the company, halting the pipeline operation for several days.

In 2016, the authors in [5] reveals that malicious activity targeted one-third of industrial control systems in the first half of 2021, and according to a blog post released by Kaspersky [6] the average cost of a cyber attack rose in 2019 to a sum that is between $108.000 and $1.4 billions, reaching $6 trillion in 2021 [7]. ICS were not connected to the Internet and many ICS standards and protocols were intended to be used only on isolated offline environments. Nowadays, their online availability can allow a malicious user to cause impact on the infrastructure behind the ICS. To meet this need for security in industrial communications, in 2008 the OPC Foundation released the OPC Unified Architecture(OPC-UA), a standard which facilitates communication between industrial components (PLC), Clients, Servers and other machineries. The OPC-UA protocol aims at establishing a secure communication channel between a Server and a Client, for the exchange of commands or industrial messages. We want to ensure that with a relatively high probability the three aforementioned properties (CIA) are not violated on these communications, and estimate with good accuracy the possible impact on the assets of their violation. There are technologies that allow to formally study this impact such as VerifPal [8], an intuitive and open source software for verifying cryptographic protocols. Using VerifPal we have built a simple model that represents the main messages exchange required by an implementation of the protocol, based on a real use case in the ICE laboratory in Verona [9].
Given the criticality of the ICS systems and the extent of the damage in the event of an attack, we consider it useful to further verify some security properties of the protocol. More precisely, it is in our interest that the exchanges of messages preserve the properties of confidentiality and freshness. Knowing that these properties are preserved in the protocol will help the implementation of the Risk Assessment in Section 6. As we will see in Section 4, the exchanges of messages can be abstracted in a model on which those properties can be formally verified. As stated above, in Section 6 we will conduct a simple Threat and Risk Assessment identifying some assets of the ICE laboratory and analyzing logical

threats and vulnerabilities on their OPC-UA protocol implementations. The logical threats have been described, taking into consideration the results of the analysis made on the model built on VerifPal. Infrastructural threats are briefly discussed giving a more complete presentation of the risk assessment carried out on the protocol. Finally, for each threat, the possible mitigations to prevent the described attacks have been described.
The state of the art from which this thesis takes its cue is briefly discussed in the concluding part of the thesis, in Section 7

2 The OPC-UA standard

2.1 Overview

The OPC Unified Architecture (OPC-UA) is a cross-platform, open-source standard developed by the OPC Foundation. With OPC-UA data exchange between similar and manufacturer-independent devices and access to industrial machines are facilitated and standardized. Its particular focus on cybersecurity and its high scalability makes it useful in communication protocols for Industry 4.0 and the IoT.
Industry systems and components like PLCs and sensors are built to exchange data and to use command and control for various industrial processes. Thus, OPC-UA specifies two types of infrastructures to achieve this goal:

1. Sending request and response Messages between Clients and Servers.
2. Sending messages between Publishers, brokers and Subscribers.

This work will focus on the first infrastructure: an OPC-UA Server and a Client create a secure channel for their communication.

Server Application
As in a regular Client-Server context, the OPC-UA Server represents the entity that provides Services to a Client. The Services are defined in detail in the Section 2.2. The structure within which the information displayed by the OPC-UA Server is organized, the *AddressSpace*, can assume both a hierarchical structure and a completely connected network structure of entities. The primary objective of an *AddressSpace* is to provide a standard way for Servers to represents its Objects to Clients. The information modeling is carried out through two basic entities:

- **Nodes**. A node is an entity that contains information, described by a set of attributes accessible by Clients for reading, writing and monitoring. Some examples of an attribute are the *NodeId*, which uniquely identifies a node in the *AdressSpace* of the Server; and *DisplayName*, which contains the name shown in the Server UI. The Client accesses the address space using standardizes methods to read, write and discover the nodes provided by the Server. There are various types of Node called *NodeClass*, we will mention in this paragraph the three main NodeClasses to give a more complete overview of the OPC-UA Client structure:
  - Objects: objects are used to structure the *AddressSpace*. Their main utility is to represent systems, industrial components, real-world objects and software elements.
  - Variables: a variable represents a value about an object that the Client can read, write or subscribe to. A Client can also subscribe to a variable, thus being informed about all its changes.
  - Methods: a method represents a function that must be performed to an object relatively quickly by the Server. A method is generally called by a Client.

- **References**. A reference is a relation between Nodes. Practically, it is a pointer from a Node to another Node.

Figure 1 represents an OPC-UA Object with variables, Methods and References, as described in the manual [10]. The notation used to refer to any entity (objects, variables and methods) represented in this protocol follows object-oriented techniques, assigning *names* that allow the user to capture the semantics of the entities. More precisely, we want to represent a motor with its components in an *AddressSpace.DisplayName*, *NodeId* and *EventNotifier* are attributes describing the motor.
Status, *Emergency Start* and *Reversing Lock-out Time* are variables exposed to the Client. *Configuration* and the referred Object *Object1* are respectively a sub-object containing the two variables and another Object referenced by the motor in the same *AddressSpace*. *Start* and *Stop* are Client-callable methods to modify or read variables. The Client can also receive information from the motor if it is subscribed to the automatic receipt of changes to the motor variables (*EventNotifier* = *Subscribe*).

Client Application
The Client is the part of the Client-Server infrastructure that requests Services from the Server. The Client is implemented with code that sends and receives requests and responses of various kinds. The Client Application is able to:

- Discover OPC-UA Servers, both local and remote
- Create secure communication channels and sessions with OPC-UA Servers
- Browse and modify the *AddressSpace* of any OPC-UA Server
- Monitor and save environmental data and conditions in real-time
- Browse and update historicized data.

2.2 Service sets

Services are procedures used by an OPC-UA Client to access information data made available by an OPC-UA Server. A Service is therefore the structure of the interface communication between two OPC-UA applications. Section 4 of the OPC-UA specification [11] defines a fixed set of Services with their exactly defined parameters and behaviours. All the Services are independent of the data transport protocol and from the environment chosen for programming.
Services are divided in Service sets, defined by the OPC Foundation manual as "logical grouping of Services"[12]. Below, we outline the main sets we will need next.

Discovery Service Set

The Discovery Service is used by the OPC-UA Client to discover Servers that are available in a network and to gather information about their Endpoints. Endpoints are physical addresses (specified by an URL string) available on a network and accessible to Clients, that allows them to use Services exposed by Servers. Endpoints are implemented by individual Servers or sometimes by

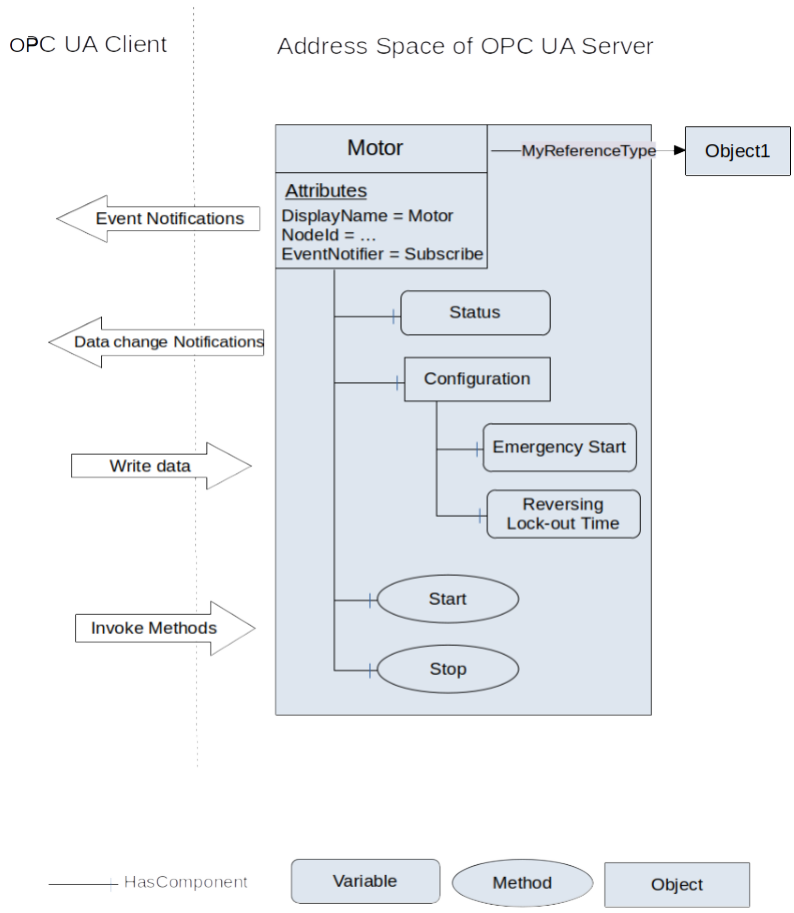

Figure 1: Example of OPC-UA Object

dedicated Local Discovery Servers(LDS).
Every Server shall expose a *DiscoveryEndpoint* that Clients can connect without having established a connection through a Secure Channel. This Service can be implemented in two main ways: with LDS and without LDS. LDS is a feature that allows a Server (the *Discovery Server*) to keep a list of Servers accessible to external Clients. In this way, the Client can have information on all the Servers present without having to query them individually. LDS is beyond the scope of this thesis, so we only illustrate the Discovery Service without it.
The Client must then know the Server TCP/IP address. In our study case, the Discovery endpoint is:

`opc.tcp://localhost:4840`

Figure 2 represents the discovery process without LDS Server. The sequence diagram notation will allow us to show the message exchanges between the various principals in the protocol, keeping only the elements of our interest, shown above the message flow line. Messages are asynchronous.

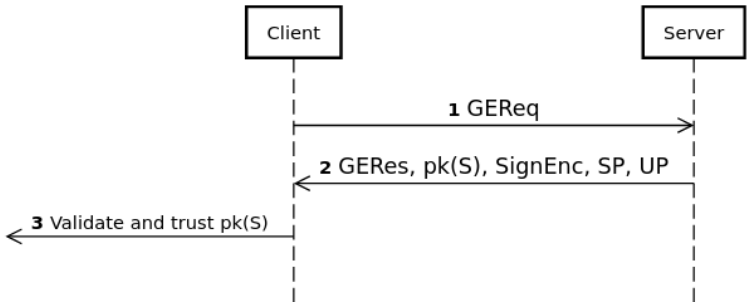

Figure 2: Discovery Process between a Client and a Server

This Service allows the Server to expose its Endpoints and all the configuration information that allows the Client to connect securely with it. This Service does not require encryption. Assuming that `GEReq` in message 1 and `GERes` in message 2 respectively mean Get Endpoint Request and Get Endpoint Response, through this Service four components for an Endpoint may be set, all listed in message 2:

- `Pk(S)`: Server Application Instance Certificate. Shall be in *X.509* format.
- `SignEnc`: Message Security Mode. Can be None, Sign or Sign&Encrypt.
- `SP`: Security Policy. An URI assigned to the exact set of cryptographic algorithms used.
- `UP`: Supported User Identity Tokens: tells the Client how to authenticate in the Activate Session Request. There are three different ways to authenticate: Anonymous, Username-Password or Client *X.509* Certificate.

Having received the Server certificate, the Client will proceed to verify its validity in message 3 as defined in the sub-section 2.4.

Security Channel Service

This Service is useful for opening or renewing a secure channel between Client and Server, used for a secure message exchange. Practically, this exchange of messages allows the sending of the respective *X.509* certificates, allowing their application authentication and the derivation of two symmetric keys starting from the exchange of two secret nonces (pseudo-random numbers), used to generate the simmetric keys.
As previously said, there are three possible Security profiles:

- **None**: Plain text unauthenticated messages. Used for compatibility or testing.
- **Sign**: Plain text messages, authenticated with a digital signature. Only authentication is guaranteed.
- **Sign&Encrypt**: messages are signed `h(m)sk(X)` and encrypted `mpk(X)`. The message hash (`h`) is signed with the *Private Key* associated with `X` and encrypted with the *Public Key* associated with `X`. This mode offers confidentiality with asymmetric (or symmetric) encryption. Thanks to the digital signature we can also achieve integrity, authentication and non-repudiation.

We are interested in studying the third profile, being the safest and most recommended on an industrial level.
Figure 3 represents this exchange in the sequence diagram notation.

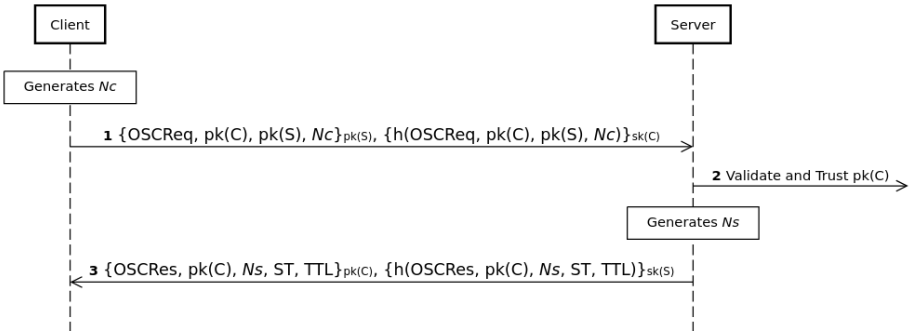

Figure 3: Open Secure Channel in mode Sign&Encrypt

Using the cryptographic primitives negotiated in the Discovery phase, both Client and Server shall generate two unique Nonces, and shall generate new tokens for each time a SecureChannel is renewed. This operation will always be noted as "`Generates Nc`", "`Generates Ns`" within squares in the figure. The Client sends a nonce `Nc` to the Server in message 1. In the message 3 the Server replies with a nonce `Ns` to the Client with `OSCRes` for Open Secure Channel Response, `ST` for SecurityToken and `TTL` for TimeToLive (its lifetime). The `SecurityToken` identifies a specific combination of Client and Server instances. This token is issued by the Server and includes a `channelID` unencrypted field that binds the token to the unique ID of the secure channel created and a lifetime in milliseconds. Time is an important resource in cybersecurity, as "several cryptanalytic attacks become easier as more material encrypted with a specific key is available"([11]). To make these attacks difficult, the token usage time needs to be short, dipending on the expected number of messages and their size. At the end of the lifetime of the token, a new `SecurityToken` needs to be issued. The two fields `OSCReq` and `OSCRes` indicate what those messages are for in the protocol, and they contain also fields like timestamps for diagnostic usage. The presence of the Client's *Public Key* inside the message (representing its own *X.509* certificate) is necessary for the Server to know the Client's identity. The received certificate will then be verified by the Server itself in message 2. The

Public Key of the Client in the first message and that of the Server in the second are instead necessary to avoid a man-in-the-middle attack similar to that of the Needham-Schroeder protocol [13], as specified in [14]. The `OpenSecureChannel` request and response messages shall be encrypted with the receiver's *Public Key* and their hashes shall be signed with the sender's *Private Key*. At the end of this procedure, both Client and Server derive four keys (`KCS, KSigCS , KSC` and `KSigSC`) by hashing the received nonces with a function named `P_hash`, similar to what happens in the TLS protocol [15].
The Service used to terminate a Secure Channel, `CloseSecureChannel`, follows the same cryptographic primitives as the opening of the channel. In order for the channel to close successfully, the Client shall send to the Server his `Authentication Token` acquired during the session and the `channelID` of the closing channel.

Session Service

The open62541 OPC-UA implementation defines the Session Service Set as "Services for an application layer connection establishment in the context of a Session"[16]. A Session uses an already established Secure Channel to allow the Client to send to the Server its credentials, such as an username or a password. A possible modeling of the OPC-UA create and activate session is represented in figure 4.

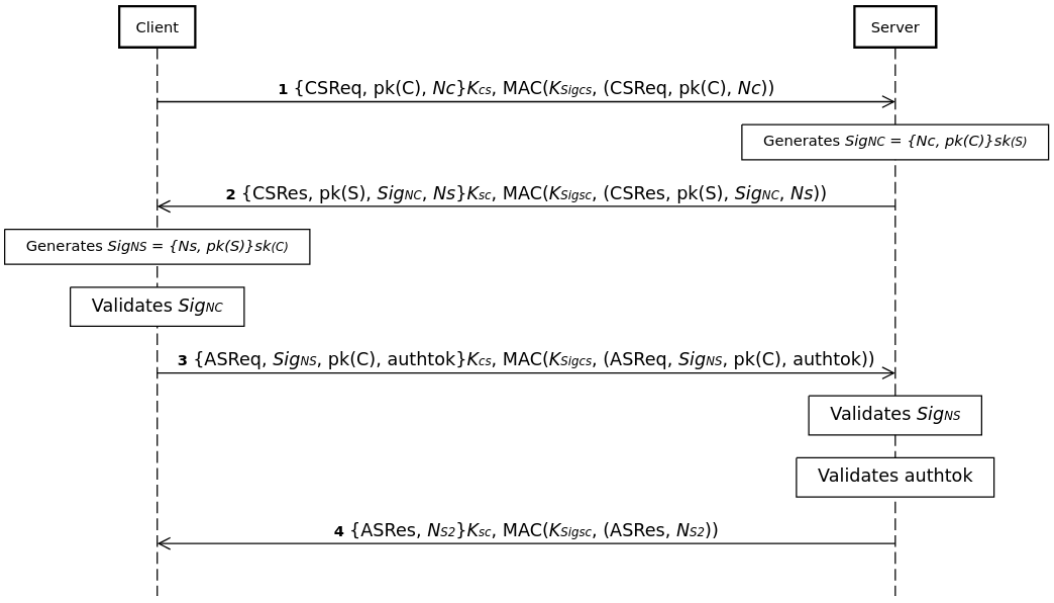

Figure 4: OPC-UA Create Session

The security mode shall be the same of the one that was chosen during the `OpenSecureChannel` phase and uses the symmetric keys derived (`KCS, KSigCS , KSC` and `KSigSC`). Message are encrypted with those symmetric keys `KCS` and signature relies on a Keyed Message Authentication Code (HMAC). The HMAC algorithm uses symmetric signature keys (`KSigSC, KsigCS`) to generate hashes of the plaintexts to append to the encrypted message. The use of the HMAC

also guarantees the integrity of the data. More formally, in the message 1, the Client sends a freshly generated nonce as a challenge to the Server with `CSReq` meaning `CreateSessionRequest`. In the message 2, the Server answers with `SigNC` and `CSRes` for `CreateSessionResponse`. As previously seen in the creation of the Secure Channel, Client and Server exchange their *Public Keys* to avoid the aforementioned man in the middle attack on the Needham-Schroeder protocol. `SigNC` in the first box is the digital signature signed with the Server's *Private Key* of the nonce just received from the Client and the Client's *Public Key*. This signed couple of elements provides proof of authentication for the Server, as it demonstrates that only the Server is able to create the signature of the received nonce. The Client's *Public Key* is inserted into the signature in message 2 to avoid the same man-in-the-middle attack on this signature mentioned above. As can be easily understood, the purpose of the session opening is the mutual authentication of Client and Server applications on an already secure channel, through the Session activation. The same is done by the Client with the creation of the `SigNS` signature in the second box. The Server also provides a maximum session inactivity time (`SessionTimeout`) after which the session shall be closed automatically. It is important to specify that as with the Secure Channel, the Server assigns a `sessionId` to the Client that shall be passed in each request and is used with the `SecureChannelId` to determine whether a Client is authorized to use that session.
The messages 3 and 4 represent the activation of the created Session, Activate Session Request`(ASReq)` Activate Session Response`(ASRes)`. Activation is required for the session to be used, and associates a user's identity with the Session. The exact procedure used to provide proof of the user identity depends on the token `authtok` sent in message 3, provided by the user who is using the Client. If the token is a username/password then the identity proof is the secret password in the token. If the token is an *X.509* certificate, proof is given by the *Private Key* associated with the certificate, which produces a digital signature. By appending the last Server nonce to the X.509 certificate of the Server, the Client gets the two elements to package and sign. The token `Authtok` is validated by the Server separately in the last box, by consulting a database.

Other Service Sets

After the creation of a Secure Channel and the activation of an authenticated Session, the Client can use a series of Services to browse and navigate through the *Address Space* of the Server, modify it or call methods in it. The **View Service Set** is used by the Client to see the AddressSpace or a subset of it, also called View. The Server receives a list of nodes and returns, for each one, the list of nodes connected to it, which includes all the attributes necessary to represent the node in the Address Space. The set **NodeManagementService** defines Services to add/delete Nodes in Address Spaces and add/delete References between them. This set includes `AddNode`, `DeledeNodes`, `AddReference`, `DeleteReferences`. The Sets **Read** and **Write** allow the Client to read and write the Value fields of a variable and to access the metadata of attributes and nodes in the address space. The Read Service is invoked by the Client providing a list of receiving the corresponding values from the Server. The Write Service allows the Client to write to attributes, nodes and subsets of these elements.
A user can also request to be notified when the status of some variables changes,

in contrast to permanently reading information (polling). This operation is called Subscription and is offered by the **Subscription Service Set**. Subscriptions allow a user to always be informed about the modification of some objects to which he has subscribed, called **Monitored Items**. When this happens, the Client automatically receives a notification. This mechanism of "reading" information from a Server reduces the amount of transferred data.
Finally, a very important Service is the **Method Service Set**. Methods represent the function call of an Objects, and return only after their completion (successful or unsuccessful). In particular, methods can be invoked with the Service **Call**. This Service allows the Client to pass input values to a method or return output values from it. Obviously, a method can only be invoked in the context of an active session.

2.3 OPC-UA technology mappings

Choosing the right implementation technology is a major issue when developing an application. In order to be open for future technologies and to provide certain interoperability between OPC-UA products, the OPC-UA standard defines Services explained in this Section and concepts in an abstract and theoretical way, then proceeds to map them to specific technologies. As shown in figure 5, OPC-UA is divided into stacks organized by functionality.

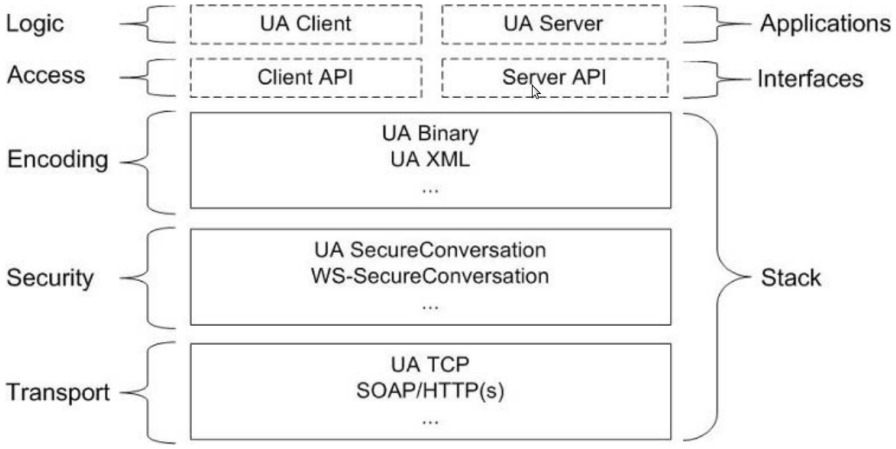

Figure 5: OPC-UA stack and technology mappings

The user has the choice on the implementation of interfaces and APIs, while the stack part is explained in the manual according to the rules decided during the development of this standard.

APIs
Today, commercial SDKs are available in many languages including C and Java, and there are open source solutions written in C, Java, JavaScript, Python, Rust and more.

In this thesis we are interested in FreeOpcUa[17], a project that aims to implement an open-source OPC-UA stack and associated tools. In particular, there is a fork of this project, called opcua-asyncio [18], which is based on the asyncio library [19] and is oriented towards asynchronous and concurrent programming. Opcua-asyncio offers all the important features mentioned above, such as connection between Clients and Servers OPC-UA with creation of Secure Channel and Session, browsing of the Address Space and certificate management.

Data encoding
Data encoding consists of the serialization of the message of a Service, also including input and output parameters, in a format optimized for the network transmission. The technology used in our use case is UA Binary.
UA Binary is a data transfer technology useful when it is important to maintain high performance (fast encoding and decoding) and introduce minimal overhead in communication, for example for applications running on embedded devices. It uses binary serialization for the encoding of the Built-in data types(Int, char, float, ..), which is the most efficient data transfer method between systems.

UA-SecureConversation (UASC)
The security of communication between Client and Server is guaranteed by UA Secure Conversation (UASC), using the previously mentioned binary encoding. The OPC-UA protocol, being technology independent, operates with different transport protocols which can have different characteristics such as different buffer sizes. Messages are therefore divided by UASC into many pieces ('MessageChunks') which have a smaller size than the packet size allowed by the transport protocol. The structure of a message generated with UA-SecureConversation (for unauthenticated messages) is shown in the figure 6.

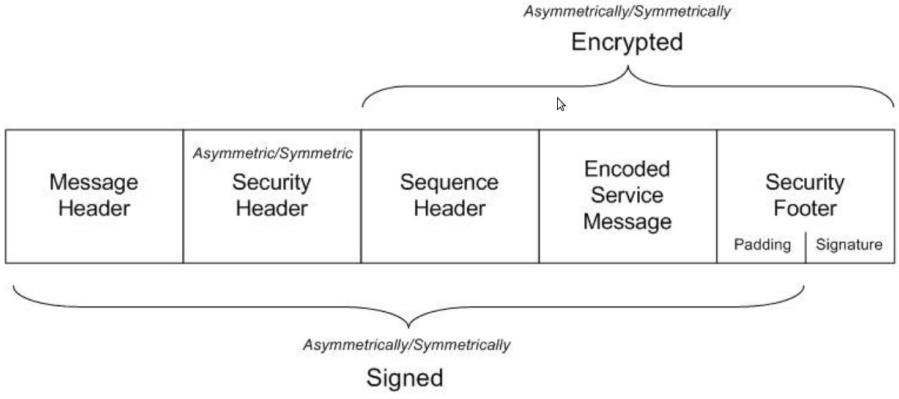

Figure 6: Message chunk generated with UASC

This structure applies to unauthenticated messages, such as those opening a Secure Channel. For authenticated messages(such as the Read Requests) the structure is the same, only the padding is removed and the digital signature is not encrypted.
The **Message Header** contains unencrypted information identifying the type

of the message, for example whether it is an Open Secure Channel request or a Create Session request. It also contains the `SecureChannelId`.
The **Security Header** indicates which cryptographic primitives have been applied to the message. The Security Header may be:

- Asymmetric, only used for Open SecureChannel requests and responses. It contains the security policy, the certificate of the sender used to verify the signature of the message, and a thumbprint identifying the certificate used for encrypting the message.
- Symmetric, applied for all other messages beside the OpenSecureChannel message. In this case the header only contains a unique `TokenId` identifying the group of symmetric keys used to sign and encrypt the exchanged messages.

The **Sequence Header** contains a encrypted "SequenceNumber" identifying a chunk. When a message payload is too large to fit within the buffer size parameters, the message has to split up into multiple chunks with the same `RequestId` and different `SequenceNumbers`, which will then have to be reassembled.
Finally, after the Encoded Service Message the **Security Footer** is placed. It contains the Signature of the message used to verify whether the signed data has been changed after it is sent and whether the message really comes from the sender.

UA Transport Protocol
It is important to say that the network level protocol used in our use case for establishing a secure connection is the TCP protocol. The TCP protocol is mainly useful for the establishment of the communication buffer size, the possibility of share IP address and port number among multiple OPC-UA Endpoints residing on the same device and the addition of error recovery mechanisms, typical of the protocol.
Figure 7 represents the structure of an UA TCP message chunk, which is formed by a Message Header and a Message Body:

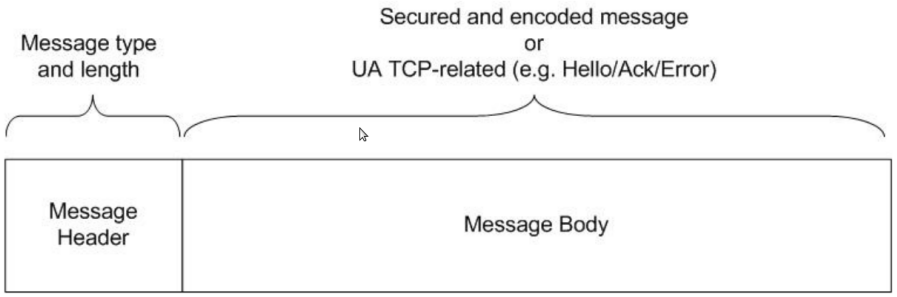

Figure 7: TCP message chunk structure

The Message Header contains general information about the message (type and the length), while the Message Body contains the encoded and secured message

payload.
There are three UA TCP messages types defined:

- Hello message. It is sent by the OPC-UA Client to the Server to establish a socket connection with the Server and request a certain buffer size for exchanged data as well as maximum chunk and total message lengths. This message is useful to protect against an attack called "malformed message attack", which we will see in sub-section 6.2.
- Acknowledge message, in response to the Hello message; confirming or revising the requested buffer sizes as well as chunk and message lengths.
- Error message, in case of general error in communication, send information about the error to the other application. It is useful if, for example, a message cannot be processed since the size is too long.

In the event of a network outage, UA TCP includes recovery mechanisms that allow sessions to be reassigned: when a Client loses the connection, a new Session is created again and assigned to the existing and already authenticated Secure Channel.

2.4 OPC-UA Security Model

Security objectives

As already mentioned, the OPC-UA protocol relies heavily on cybersecurity, in particular to meet seven security objectives, summarized below:

- **Authentication**: Entities such as Clients and Servers should prove their identities, based on something the entity is, has or knows. More precisely, it is the ability to reliably say that whoever sent a message is really who they say they are.
- **Confidentiality**: Confidentiality is the ability to keep secret information to unauthorized parties, be they people, systems or processes. Confidentiality must be maintained through measures that ensure that only authorized parties can have access to the information they are authorized to read, and no one else. One such method is symmetric or asymmetric encryption.
- **Integrity**: The recipient of a message must receive exactly the same information that the sender sent. In other words, the exchanged message must not have been tampered with during the exchange.
- **Availability**: Availability is compromised when the execution of the system that allows communication is blocked or when it is made unavailable, often overwhelmed by too many inputs to process. Systems and applications must be available to users when they need them. In large industrial systems this property is often the most important.
- **Authorization**: When a user has been successfully authenticated, it is necessary to check which resources he can access and what he can do on these resources. These rules are defined by policies, established in an access control mechanism, which is defined according to the system requirements.

- **Non repudiation**: Repudiation is the denial of a valid action. Hence, non-repudiation is the security that a user can reject or deny an action that actually happened. A useful tool for non-repudiation is the digital signature. A digital signature is created using a *Private Key* and verified with the corresponding *Public Key*. It follows that, unless the *Private Key* has been compromised, the user to whom the *Private Key* is associated cannot refuse the event signed with his digital signature.

- **Auditability**: The protocol requires that any activity of all users on the system be logged and saved. This can come in handy in various situations, such as supporting debug activities in case of errors or security accidents. Auditing can be performed directly from the applications, which generate events and store related information in log files or databases.

We note, however, that according to the current implementation of symmetric keys exchange in OPC-UA, if an attacker manages to find the nonces exchanged in the phase of creating the Secure Channel between Client and Server, he can obtain all the keys exchanged in that channel and decrypt all messages[20]. This property, called **Perfect Forward Secrecy**(PFS), was defined in [21] and is preserved if an attacker is not able to derive previously exchanged keys generated from a long-term secret key that the attacker came into possession of. In this protocol, PFS is not respected on the secret nonce.

Security architecture

OPC-UA applications are present at all levels of the automation pyramid[22] and in varying environments, from communication between embedded devices to data transfer between ERP systems and MES. The use of OPC-UA involves dealing with the risk of industrial espionage, sabotage or malwares such as worms, that could result in important financial losses and affect public safety or environmental damage. It is therefore important to define a layered security model that can be adapted to any execution environment. The security architecture is depicted in 8, where each layer has specific responsibilities regarding security.

The application layer with an active Session is responsible for authenticating and authorizing users working with the Client, as well as for authenticating and authorizing certain products. An OPC-UA Session runs "over" a Secure Channel, maintaining its security properties(confidentiality, integrity and application authorization) and adding others.

Certificates in OPC-UA

The OPC-UA standard adopts *X.509v3* type certificates for connection establishment. Certificates are associated to the *Public Keys* needed for Asymmetric Cryptography operations. OPC-UA therefore distinguishes three types of *X.509* certificates:

- **Application Instance Certificates**. Each installation of an OPC-UA product requires a certificate called Application Instance Certificate, which

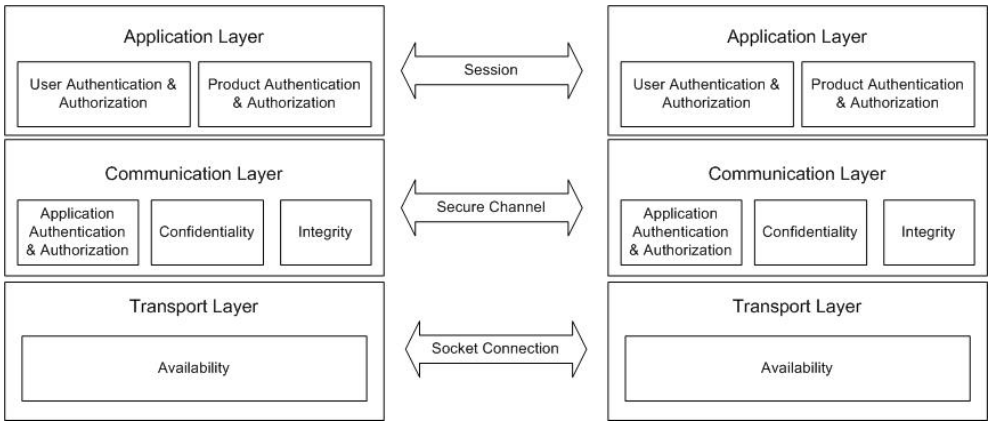

Figure 8: OPC-UA Security architecture [23]

identifies the instance of an OPC-UA application running on a specific host and is obtained from the responsible Certification Authority. The certificate shall be stored in `DER` encoded form. The fields of the certificate shall include information about the holder, including identity information about the application, the public and *Private Key* issued by the CA, their expiry date and a digital signature created by the CA to ensure the validity of the certificate.
In some cases the administrator can opt for the creation of a self-signed certificate, which does not require a CA. Thus, the administrator will use his *Private Key* to sign the certificate. Self-signed certificates are useful in test environments because the speed, ease and cheapness with which they are created, but since they are not issued by any trusted CA they can pose a serious risk if compromised.

- **User Certificates**. User certificates are one of the ways to authenticate and authorize a user in the OPC-UA environment, i.e. to verify that a user is really who he claims to be and that he has the necessary permissions for the operations he intends to perform. The adoption of User Certificates in an OPC-UA environment is not mandatory, as other authentication methods are supported, such as username&password pair or other security tokens (e.g. Kerberos ticket [24]).
- **Software Certificates**. Another type of *X.509*v3 certificate used for OPC-UA is the Software Certificate which identifies a specific version of an OPC-UA product, and can be obtained only by accomplishing the OPC-UA certification process of an accredited test laboratory. By exchanging this information during the connection establishment both applications know whether they can communicate with each other in a proper way and which Services they support.

OPC-UA adopts a Public Key Infrastructure (PKI) for the management of the life cycle of certificates, which includes use cases like requesting, creating, installing, distributing, revoking, renewing and validating certificates. A certificate must be created and signed by a trusted CA (or self-signed), installed in a

Certificate Store (while the *Private Keys* has to be installed in a special secret location accessible by the owner of the key) and distributed. The distribution of certificates is a critical point, which can be carried out either through the same secure channel created by the OPC-UA protocol or through out-of-band methods, such as USB sticks or PGP [23]. A certificate can be marked as revoked if a further usage has to be prohibited, putting it in a Certificate Revocation List (CRL) by the administrator. Finally, Certificates are created to be valid only up to a specific period. When this period expires, the certificate is no longer considered secure(an attacker has less time to compromise a certificate or craft a bogus one) and must be renewed.

Certificate Trust Model
Whenever an application intends to use a certificate from another entity it has first to validate its certificate. If its Certificate is not already in the list of trusted applications, the other entity is not directly trusted and the application has to rebuild the chain of certificates trying to reach a trusted CA. Obviously when building a chain each Certificate in the chain shall be available and validated, ending with a self-signed certificate. Figure 9 illustrates the interactions between the Application, the Remote Application which provides the certificate to authenticate and the Certificate Store, a central database in which all the certificates are installed. In message 1 the Certificate is provided to the application responsible for verifying certificates. In messages 2 the application validates the instance Certificate and the whole chain up to the trusted CA. In message 3 it is checked that all the validated Certificates have not been revoked, and in message 4 it checks that at least one certificate in the chain is already trusted.
It is important to note that before its validation a Certificate must also be found. If not all certificates can be retrieved, the trustworthiness of the received certificate cannot be correctly validated and thus it should not be trusted [11].

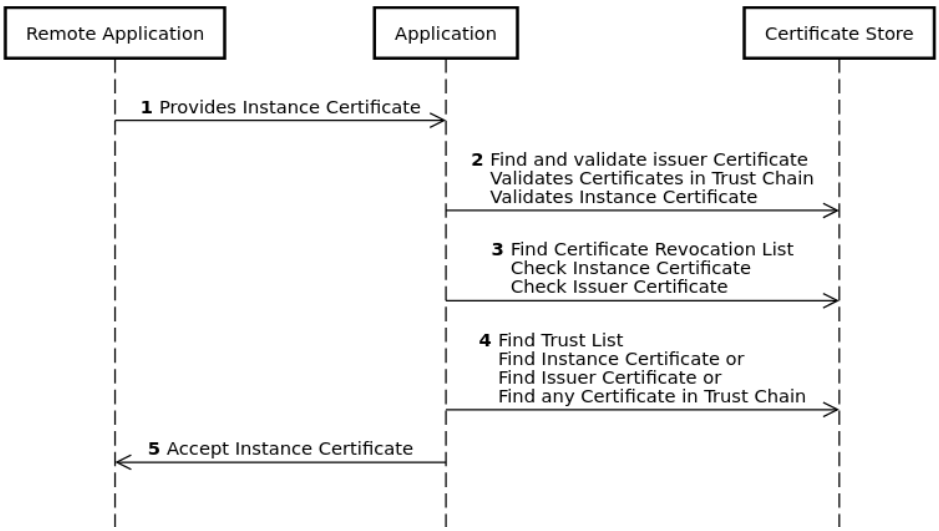

Figure 9: Determining if a Application Instance Certificate is Trusted

3 Use Case: Ice Laboratory

This thesis focuses on the adoption of the OPC-UA protocol by the ICE laboratory based in Verona. The ICE Laboratory is divided into many technological areas, each representative of a type of production, for example assembly and disassembly with robotic arms, 3D printing and storage with a Vertical Automatic Warehouse. All these industrial components will be interconnected by a smart logistic system composed of a conveyor belt integrated with mobile robots and safety is ensured by the presence of a camera tracking system. In addition, IOT sensors collect environmental information such as the temperature or the number of people present in the laboratory. The data collection architecture based on Kubernetes[25] clusters is used to collect, monitor and historicize the data produced by the laboratory infrastructure and by the IOT sensors. As is often seen in industrial environments, all machines(with their own OPC-UA Server) are connected with Ethernet. The transport protocol used is TCP, due to the advantages mentioned in Section 2.3. Figure 10 illustrates the communication channel between the vertical warehouse, its Server and the Client: a Vertical Warehouse whose variables and objects are exposed by a OPC-UA Server communicates with a OPC-UA Client through a VLAN cabinet.

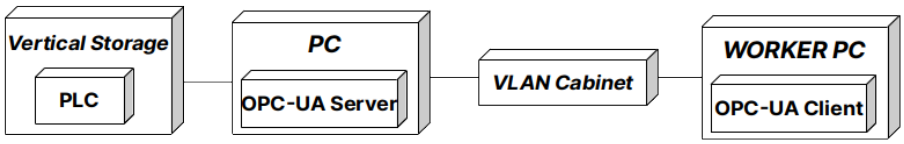

Figure 10: Physical comunication channel

The VLAN cabinet(a physical cabinet with switches to which the VLANs are connected) is a standard use in industrial environments, for a better organization of all internal networks.

OPC-UA Protocol organization
The secure OPC-UA communication is organized in the following way:

- **Servers** are distributed among the various industrial components, such as PLCs, IOT ecosystems for environmental control and control PCs for the industrial machines connected via Ethernet. As previously mentioned, the OPC-UA Server exposes all the variables, objects and methods of the industrial components, allowing them to be viewed and possibly modified by the Client.
- **Clients** are deployed in the Kubernetes clusters, which are hosted on worker PCs in the lab data room. The OPC-UA Clients are accessible to laboratory users with any data monitor or MES.

The exchange of messages takes place through messaging brokers. The broker organizes all the messages in queues for multiple receivers, providing reliable

storage and guaranteed message delivery. The messages are therefore saved and grouped into topics, then they are made available to those authorized to read them, usually OPC-UA Clients.
In this use case, any user may decide to communicate with the Client/Server for two main reasons:

- Call a Remote Procedure Call (RPC) to a Server. The RPC, that is a method, shall be made available by the Server. For example, let's say the Client wants to communicate with the Vertical Automatic Warehouse flanked by a PC with an instance of an OPC-UA Server installed. Figure 11 illustrates illustrates the message flow of an example of a call to an RPC, in the form of a sequence diagram.

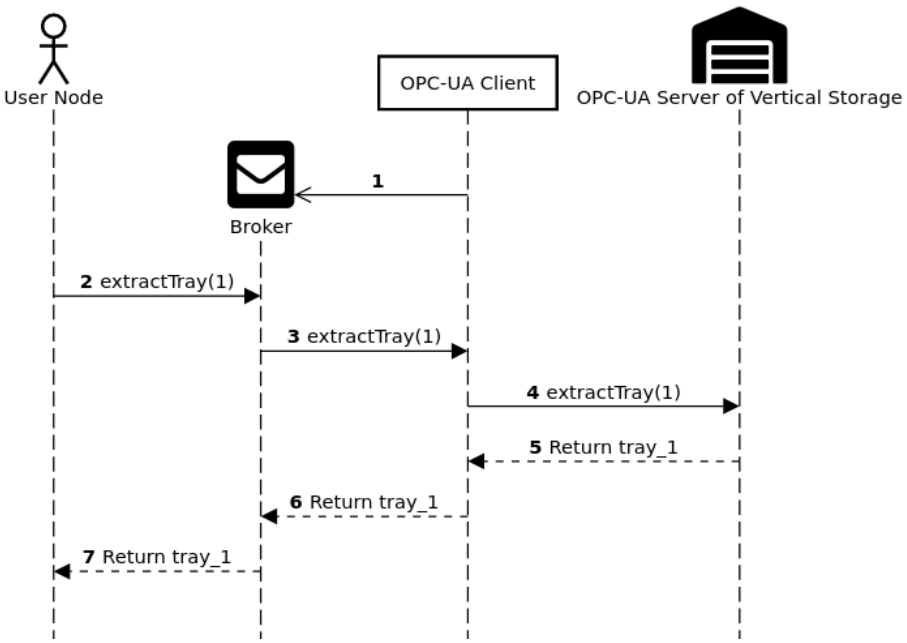

Figure 11: Calling of the RPC extractTray(numTray)

In message 1, the OPC-UA Client creates a buffer instance inside the broker, to allow the receipt and dispatch of inbound/outbound messages. The method `extractTray(1)` extracts the selected cart with the number given in input. In this case the message(RPC) will be sent to a remote call queue in message 2, forwarded to the Client in message 3 and sent via Secure Channel to the OPC-UA Server of the Vertical Storage in message 4.

- Monitor the data(in the form of variables) periodically provided by the machinery/sensors. Figure 12 is an exhaustive representation of how a user can read the variables related to an industrial component exposed by a OPC-UA Server. When the update of a variable occurs (message 1), the update is sent from the Server to the Client in message 2, which will make them available in a broker (message 3). Usually the messages are grouped

into topics within the brokers, and are periodically sent to the Client after having subscribed to the topic (message 4 and 5).

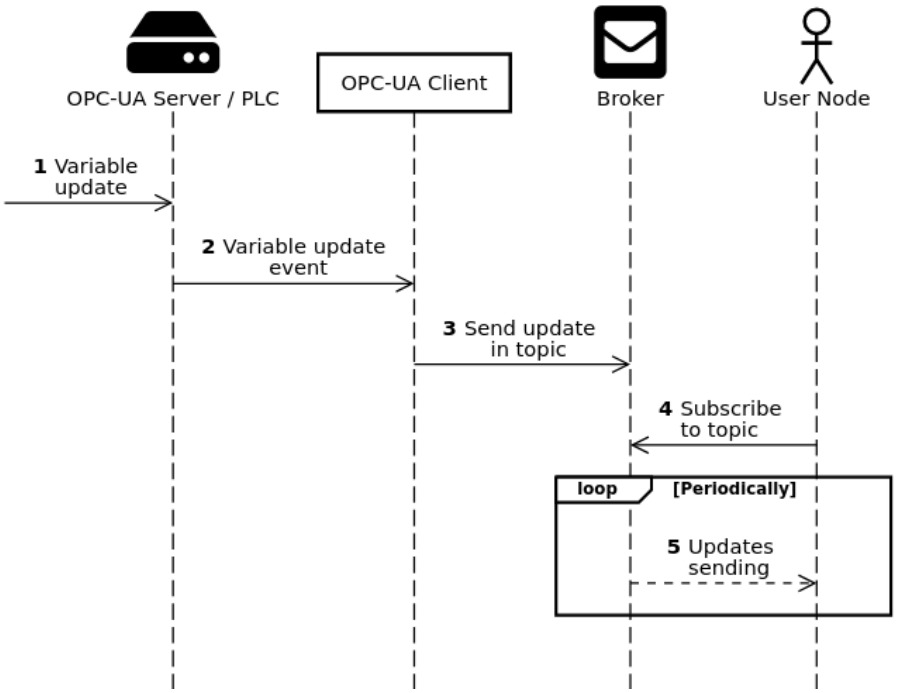

Figure 12: Reading an exposed variable by a user node.

4 The VerifPal formal protocol verifier

Verifpal is a tool able to perform formal analysis of security protocols. VerifPal's modeling language is intuitive, similar to how a protocol might be described in an informal conversation. To achieve this, VerifPal makes sure that the user only has to define the participant agents in the protocol. Agents have independent states and perform operations with cryptographic primitives. The cryptographic primitives are already defined and only need to be called, so the user does not need to implement them, avoiding incorrect implementations. In addition, VerifPal provides an output that is easy to understand: the result is presented in a user-friendly format that associates the attack to a concrete scenario. This tool allows the user to verify three security properties: confidentiality, authentication and freshness. As we will see in Section 6, the integrity property will be implicitly guaranteed through the use of hash functions.

4.1 The Dolev-Yao attacker model

In [26] the authors proposed a model, known as the Dolev-Yao attacker model, for the formalization of the attackers used by the tools for the formal verification of the protocols written on the symbolic model, as we will see in VerifPal.
The attacker in this model can:

- Overhear, intercept and synthesize any message sent within the network;
- As an authorized participant in the network, build new messages starting from the observed ones and send them to any participant;
- Disassemble unencrypted messages and identify the parts that compose it;
- Run multiple runs of the protocol, simulating message exchanges and injecting previously intercepted messages

The Dolev-Yao attacker therefore has almost infinite capabilities and can be identifiable as the network itself. Despite this, there are cryptographic assumptions that limit the Dolev-Yao attacker: it is assumed that cryptographic attacks (e.g. brute force on Private Keys or dictionary attacks on passwords) cannot be carried out by the attacker.

4.2 Modeling in VerifPal

The first step in modeling is the attacker's declaration. The syntax is simple: `attacker[passive]` indicates a passive attacker, i.e. a malicious observer on the network that cannot inject or modify messages. An `attacker[active]` is instead a Dolev-Yao attacker, described above.
Then there is the declaration of the participants in the protocol called `Principals`, within which constants can be defined. The principal may already know these constants, in this case the `knows` key word is used, otherwise `generates` is used. In this last case, the constants are randomly(*freshly*) generated at the moment. Furthermore, constants can be defined as `public` if all the participants (even the attacker) are aware of them, and `private` if only the principal who generated them knows them.

The notation for exchanging messages is simple:

```
A → B : m
```

where `m` is the message itself, `A`, `B` are the principals. It is also important to say that VerifPal admits the notion of "guard", that is, it is possible to insert the message in square brackets (`[ ]`) to ensure that the attacker cannot alter the content of the message, even though he can read it. The *guard* is useful when a property is already verified before the protocol, without needing to verify it again, for example in the case of a pre-authenticated *Public Key*. In any case, it is advisable to limit the use of the guard to the cases strictly necessary in order not to limit too much the capability of an attacker. Finally, the keyword `leaks` placed before a constant specifies that the principal will leak that constant to the attacker, rendering the value immediately known to him.

Primitives
In VerifPal, cryptographic primitives are essentially perfect one-way functions, not susceptible to cryptographic attacks. VerifPal defines the following primitives, which also correspond to the primitives of the Dolev-Yao model:

- `CONCAT`(a, b, ...) : c → Concatenates two or more values (up to five) into one value.
- `ASSERT`(value, value) : unused → Checks the equality of two values. Output value is not used.
- `SPLIT`(`CONCAT`(a, b)): a, b → Inverse `CONCAT` operation, returns the values that make up the concatenation. Must contain a `CONCAT` primitive as input.
- `HASH`(a, b): x → Secure hash function. Takes between 1 and 5 inputs and returns one output.
- `MAC`(key, message) : hash → Keyed hash function. Useful for message integrity and authentication.
- `HKDF`(salt, ikm, info) : a, b, ... → Hash-based key derivation function. Used to extract more than one key(up to five) out a single secret value. Salt and info help us by providing other inputs in the creation of keys.
- `AEAD_ENC`(key, plaintext, ad): ciphertext → Authenticated encryption with associated data. Ad represents additional non-encrypted information, which must be reported exactly the same in the decryption of the message, otherwise the message itself will be invalidated.
- `AEAD_DEC`(key, `AEAD_ENC`(key, plaintext, ad), ad): plaintext → Authenticated decryption with associated data.
- `PKE_ENC`(G^key, plaintext): ciphertext → Public key encryption.
- `PKE_DEC`(key, PKE_ENC(G^key, plaintext)): plaintext → Public key decryption.

- `SIGN`(key, message) : signature → Classic signature primitive, where `key` is a *Private Key*.
- `SIGNVERIF`($G^k ey$, message, `SIGN`(key, message)): message → Verifies if signature can be authenticated.

In VerifPal, `ASSERT` , `SPLIT` , `AEAD_DEC` and `SIGNVERIF` are "checkable" primitives: adding a question mark after one(or more) of them will abort the model if the check fails. For example, if `ASSERT` fails to find two provided input equals.

4.3 Queries and goals

A VerifPal model is always concluded with a block called `queries` as seen in Section 5, which contains the properties that we want to be preserved after the model's analysis. In the block of queries we can ask about confidentiality, authentication and freshness of messages using respectively the commands:

- `confidentiality?`: By "confidentiality" VerifPal means the attacker's ability to obtain the message in cleartext despite being encrypted; obviously without using cryptographic attacks.
- `authentication?`: The authentication, as intended by VerifPal, implies that the attacker can successfully convince a principal to validate the decryption of an encrypted message sent(crafted or replayed) by the attacker himself.
- `freshness?`: Freshness queries are useful for preventing replay attacks, where an attacker logs a valid message passed on the network and reenters it later, making the message appear valid. In active attacker mode, since the attacker can run multiple sessions of the protocol, freshness is violated if a value can be used in multiple sessions.

5 OPC-UA verification on VerifPal

In this Section we describe the verification of the confidentiality and freshness properties on OPC-UA protocol on VerifPal. We will also see how we have also indirectly demonstrated the integrity property on each message computing the digital signature on the message hash. The model described here is an abstraction of the real protocol, for reasons of readability and usefulness, since it makes no sense to burden the VerifPal analysis with fields that are not of our interest. The fields kept are exclusively those useful for checking the properties described in sub-section 4.3. Other fields (for example timestamps, present in all the messages of the protocol) are abstracted away. We follow the official OPC-UA standards in our model and checked it against the open-source implementation opcua-asyncio[18]. The protocol has 6 phases preceded by a mandatory pre-phase:

1. Asymmetric keys exchange: Public key exchange phase between the principals.
2. Open channel Request: Client request to open a Secure Channel
3. Open channel Response: Server evaluation and response on opening the Secure Channel
4. Open session Request: Client request on creating a Session
5. Open session Response: Server evaluation and response on creating the Session
6. Activate session Request: Client request on activating the Session and sending its credentials.
7. Activate session Response: Server evaluation on activating the Session and authentication of the user.

The security profile used for most of the OPC-UA communications in the ICE laboratory explained in the Section 3 is Sign&Encrypt, encrypting and signing the data.

5.1 System model

Asymmetric keys exchange

Listing 1: Asymmetric key exchange between Client and Server

```
principal Server[
  knows private s
  gs = G^s
]

principal Client[
  knows private c
  gc = G^c
]
```

```
10
11  Server -> Client: [gs]
12  Client -> Server: [gc]
```

The OPC-UA protocol does not include an initial asymmetric key exchange. It's assumed that a trust list for X.509 certificates is implemented for each Client/Server and each certificate is exchanged in other out-of-band, authenticated and secure ways, explained in sub-section 6.2. Since it is not possible to model this secure exchange, we have decided to exchange the keys directly before the start of the protocol. At line 2 a Server's secret `s` is generated, which will serve as *Private Key*, and a result $gs = G^s$ in line 3 will serve as *Public Key*. This operation reproduces the Diffie-Hellman protocol to securely exchange a symmetric key[27]: obtaining the exponent `s` having an exponential number $gs = G^s$ is in fact considered very difficult, and VerifPal does not allow cryptographic attacks.
The *Public Keys* are guarded in lines 11 and 12, since this part it is not foreseen by the protocol, and we have no interest in an attacker using those keys.
In the original protocol there are also sequence numbers and timestamps to get protection from attacks such as replay attacks or session hijacking, described in sub-section 6.2. Since modeling of these elements is not feasible in VerifPal, we need to introduce elements like sequence numbers and signed nonces(*Number Once*) to achieve freshness in this model.

Open channel Request

Listing 2: Request of a Secure Channel opening

```
1  principal Client[
2    generates cn
3    generates s1
4    m1 = PKE_ENC(gs, CONCAT(s1, gc, gs, cn,
5      SIGN(c, HASH(CONCAT(s1, gc, gs, cn)))))
6  ]
7
8  Client -> Server: m1
```

Having obtained the key gs, the Client generates:

- `cn` in line 2, a unique secret that will be used for generating the symmetric keys. This value is randomly generated by suitable functions that are provided by cryptography libraries.
- `s1` in line 3, an incremental number that identifies a single chunk of message. The Sequence number `s1` concatenated to the previously received sequence number will be added to the message to be sent (in the case of the first message, this field is absent). The principal who creates the nonce will then have to check its presence also in the reply message. These nonces allow you to determine the freshness of the message, according to the sequence diagram represented in figure 13.Let's assume that Alice and

Bob are two principals who want to exchange two fresh nonces. After the mutual generation in messages I and III, and the mutual sending in messages II, IV and VI, each principal checks the nonce he sent previously (messages V and VII). If the message is fresh, the nonce must be the same.

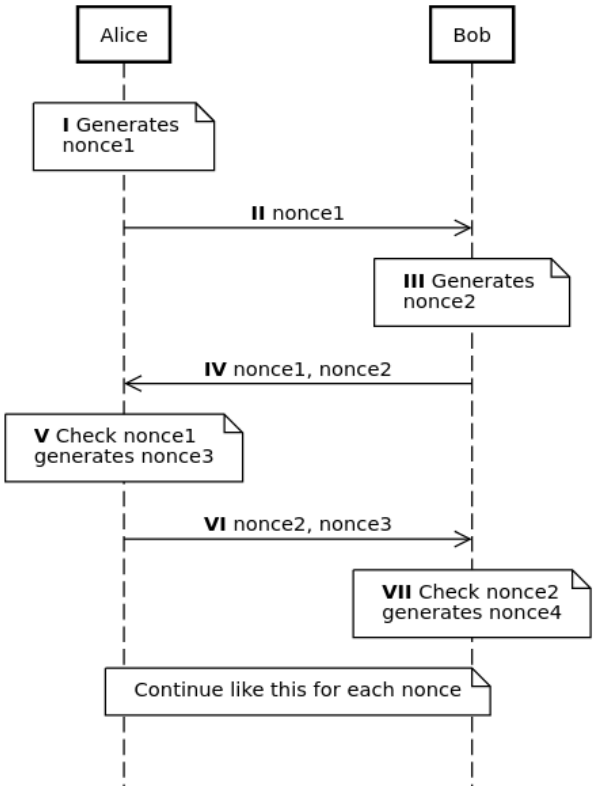

Figure 13: Exchange of nonces for freshness

Obviously, all nonces are exchanged in encrypted and signed messages.
In line 4 and 5, the Client then concatenates five elements:

- The Sequence number `s1`.
- Its certificate `gc` which we identify with the Client's *Public Key*. It will let the Server know who he is talking to. The Server will then have to verify if the certificate is valid, as described in sub-section 2.4.
- The certificate of the Server `gs`, which we identify with the Server's *Public Key*. This field is needed to prevent a **Man-in-The-Middle** attack on the Client, as explained in the above paper[5].
- The secret `cn`.
- The signature of the Client.

This message is encrypted with the Server *Public Key* and its hash is signed with its *Private Key*. Since the hash function is *one way*, finding the value

starting from the hash is is assumed to be impossible. Furthermore, being *collision-resistant*, it is assumed to be impossible to get the same hash starting from two different inputs. For this reason, the integrity, authentication and non repudiation (since they are signed with a *Private Key*) of the hashed values are guaranteed.

Open channel Response

Listing 3: Response to the Open Channel Request by the Server

```
1  principal Server[
2    m1_d = PKE_DEC(s, m1)
3    Seq1_d, gc_d, gs_d, cn_d, m1_s = SPLIT(m1_d)
4    _ = ASSERT(gs_d, gs)?
5    _ = ASSERT(gc, gc_d)?
6    generates s2
7    generates sn
8    valid1 = SIGNVERIF(gc_d,
9      HASH(CONCAT(Seq1_d, gc, gs, cn_d)), m1_s)?
10   s_e_key_s, s_s_key_s = HKDF(cn_d, sn, nil)
11   c_e_key_s, c_s_key_s = HKDF(sn, cn_d, nil)
12   m2 = PKE_ENC(gc_d, CONCAT(s2, Seq1_d, gc_d, sn,
13     SIGN(s, HASH(CONCAT(s2, Seq1_d, gc_d, sn)))))
14 ]
15
16 Server -> Client: m2
```

Having received the request to open the secure channel by the Client, the Server first decomposes and decrypts the message, obtaining the plaintext. The plaintext consists of the following fields present in line 2:

- `m1_d`: the decrypted message, in line.
- `Seq1_d`: the sequence number.
- `gc_d`: Client *Public Key*.
- `gs_d`: Server *Public Key*.
- `cn_d`: the secret of the Client.
- `m1_s`: the signature.

With the primitive `ASSERT`, in line 4 and 5, the Server checks that the decrypted *Public Key* `gs_d` in the message matches its own key, to avoid the MiTM attack mentioned above, and that the public key `gc` is that of the Client with whom he exchanged the key. In line 7 and 8 the Server generates its own secret `sn`, which it will send to the Client for the generation of its symmetric keys; and the sequence number `s2`, which will be sent to the Client for freshness control. With the `SIGNVERIF` function in line 8 and 9 the Server checks that the hash of the signature received and decrypted by the Client matches the hash of the separation of the values that should make it up. If the values do not coincide, the message is not authenticated or intact and shall be discarded.

Symmetric keys derivation

In [28], the OPC Foundation explains the method of deriving symmetric keys starting from two numbers, which are used as input parameters for a pseudo-random function called `P_HASH`. The function is iterated until it produces enough data for all of the required keys. However, this complex calculation was abstracted to the `HKDF` function shown in the VerifPal manual and explained in Section 4.2, which allows extracting up to five keys starting from a seed. In our case, 4 keys will be needed for each principal, in total 8 keys on VerifPal:

- `s_e_key_s`: Server encryption key, owned by the Server.
- `s_s_key_s`: Server signing key, owned by the Server.
- `c_e_key_s`: Client encryption key owned by the Server.
- `c_s_key_s`: Client signing key owned by the Server.
- `s_e_key_c`: Server encryption key owned by the Client.
- `s_s_key_c`: Server signing key owned by the Client.
- `c_e_key_c`: Client encryption key owned by the Client.
- `c_s_key_c`: Client signing key owned by the Client.

The keys are available to all principals, but since VerifPal does not allow the redefinition of the same variable, 4 mutually identical keys must be created for each principal. Since the pseudo-random function for key derivation takes both nonces for each key as input, we decided to use the *salt* field of the `HKDF` function for one of the two secrets, alternatively. The *info* field of the `HKDF` function is instead unused and set `nil`.

Open session Request

Listing 4: Open Session Request by the Client

```
principal Client[
  m2_d = PKE_DEC(c, m2)
  Seq2_d, Seq1_dcoda, gc_2d, sn_d, m2_s = SPLIT(m2_d)
  _ = ASSERT(gc_2d, gc)?
  _ = ASSERT(Seq1_dcoda, s1)?
  valid2 = SIGNVERIF(gs,
    HASH(CONCAT(Seq2_d, Seq1_dcoda, gc, sn_d)), m2_s)?
  generates ch_nc
  c_e_key_c, c_s_key_c = HKDF(sn_d, cn, nil)
  s_e_key_c, s_s_key_c = HKDF(cn, sn_d, nil)
  m3_pay = AEAD_ENC(c_e_key_c, CONCAT(Seq2_d, gc,
      ch_nc), nil)
  m3_mac = MAC(c_s_key_c, m3_pay)
  m3 = CONCAT(m3_pay, m3_mac)
]

Client -> Server: m3
```

Now that the secure channel is established and the nonces have been exchanged, we proceed with the creation of the session and the exchange of the challenge nonces, indicated as `ch_nc, ch_ns`. The operations made by the Client are similar to the previous exchanges. `Seq1_dcoda` in line 3 is the nonce that the Client created at the beginning, its verification confirms that this message is exactly the reply to the message he sent earlier, proving its freshness. Then the Client derives its symmetric keys, in line 9 and 10. From this point on, symmetric keys will be used almost exclusively. At this stage, in line 12, it was decided to use the MAC in *Encrypt-then-MAC* mode to guarantee the authentication and integrity of the message.
The fields of the `m3` message, shown in the line 3, are:

- `Seq2_d`: Server's nonce received.
- `gc`: Client *Public Key*, to prevent the MiTM attack previously seen.
- `ch_nc`: new challenge nonce randomly generated, that will be used by the Server to create a "fresh signature" with its *Private Key*, to prove that it is the same Server who created the Secure Channel. This method also allows to check the freshness of the next message, without the need to use sequence numbers.

Open session Response

Listing 5: Open Session Response by the Server

```
1  principal Server[
2    m3_e, m3_s = SPLIT(m3)
3    m3_d = AEAD_DEC(c_e_key_s, m3_e, nil)
4    Seq2_dcoda, gc_3d, ch_nc_d = SPLIT(m3_d)
5    _ = ASSERT(Seq2_dcoda, s2)?
6    _ = ASSERT(MAC(c_s_key_s, m3_e), m3_s)?
7    _ = ASSERT(gc_3d, gc)?
8    generates ch_ns
9    sign1 = SIGN(s, CONCAT(ch_nc_d, gc_d))
10   m4_pay = AEAD_ENC(s_e_key_s, CONCAT(gs, sign1, ch_ns
         ), nil)
11   m4_mac = MAC(s_s_key_s, m4_pay)
12   m4 = CONCAT(m4_pay, m4_mac)
13 ]
14
15 Server -> Client: m4
```

After decrypting the message components and verifying in line 7 that the `gc_3d` key is the same key as the Client that requested the Secure Channel, the Server verifies the authenticity of the MAC, as shown in line 6. Then it creates its own challenge nonce `ch_ns` in line 8 which must be signed by the Client in the next

step.
Finally, it creates a fresh signature `Sign1` in line 9, containing:

- `ch_nc_d`: the challenge nonce received by the Client.
- `gc_3d`: The Client *Public Key*. This element is essential to protect the Server from the MiTM attack on this fresh signature mentioned above.

The Server signs the packet `Sign1` with its *Private Key*. By doing so, only he can have signed the nonce just received, so the message is fresh, intact(noone can forge a digital signature without having the *Private Key* of the Server) and authenticated.

Activate Session Request

Listing 6: Activate Session Request and credentials sending by the Client

```
1  principal Client[
2    m4_e, m4_s = SPLIT(m4)
3    m4_d = AEAD_DEC(s_e_key_c, m4_e, nil)
4    gs_2d, sign1_d, ch_ns_d = SPLIT(m4_d)
5    _ = ASSERT(MAC(s_s_key_c, m4_e), m4_s)?
6    _ = ASSERT(gs_2d, gs)?
7    validsign = SIGNVERIF(gs, CONCAT(ch_nc, gc), sign1_d
         )?
8    generates authtok
9    sign2 = SIGN(c, CONCAT(ch_ns_d, gs))
10   m5_pay = AEAD_ENC(c_e_key_c, CONCAT(sign2, gc,
         authtok), nil)
11   m5_mac = MAC(c_s_key_c, m5_pay)
12   m5 = CONCAT(m5_pay, m5_mac)
13 ]
14
15 Client -> Server: m5
```

With the activation of the session, the user authentication phase of the Client begins. Once the checks similar to those carried out by the Server have been completed, the Client creates its fresh signature in line 9, `sign2`, using the received nonce `ch_ns_d`. When activating the session, the user authentication and authorization tokens (credentials) of the Client are sent at the application level (usually username and password). The password is usually the asset of which we are interested in keeping the secrecy at this stage, while the username can be made public. For simplicity, the Client generates a secret `authtok` in line 8, similar to a password, which will include in the encrypted message to access the service offered.

Activate session Response

Listing 7: Activate Session Response by the Server

```
1  principal Server[
2    m5_e, m5_s = SPLIT(m5)
3    m5_d = AEAD_DEC(c_e_key_s, m5_e, nil)
4    sign2_d, gc_5d, authtok_d = SPLIT(m5_d)
5    _ = ASSERT(gc_5d, gc)?
6    validsign2 = SIGNVERIF(gc_d, CONCAT(ch_ns, gs),
         sign2_d)?
7    _ = ASSERT(MAC(c_s_key_s, m5_e), m5_s)?
8    generates ns2
9    m6_pay = AEAD_ENC(s_e_key_s, ns2, nil)
10   m6_mac = MAC(s_s_key_s, ns2)
11   m6 = CONCAT(m6_pay, m6_mac)
12 ]
13
14 Server -> Client: m6
15
16 principal Client[
17   m6_e, m6_s = SPLIT(m6)
18   m6_d = AEAD_DEC(s_e_key_c, m6_e, nil)
19   ns2_d, Seq5_dcoda = SPLIT(m6_d)
20   _ = ASSERT(MAC(s_s_key_c, m6_d), m6_s)?
21 ]
```

Final couple of message of the protocol handshake. The Server receives the Client's `authtok` in line 4 and checks its validity in line 6. This check will take place in a specific database, and does not concern the logic of the protocol. The OPC Foundation has not been officially established what the credentials should be like. However, the paper above mentioned [5] suggests that a possible pair for credentials could be:

- username: (*Client Public Key*, host) public, identifies an user.
- password: (*Client Private Key*, *Server Public Key*) private, which identifies the secret password of the user and the service with which to access, identified by the *Server Public Key*.

The Client then generates a second nonce `ns2`, in line 8. The Client shall use this value to prove possession of its application certificate in the next call to `Activate Session` request. From here on, all requests forwarded by the Client to the Server are encrypted and signed with the symmetric keys generated, following the model of `Open Session` and `Activate Session`.

5.2 Queries and results

Listing 8: Queries to verify

```
1  queries[
2      confidentiality? c_e_key_c
3      confidentiality? c_e_key_s
4      confidentiality? s_e_key_c
5      confidentiality? s_e_key_s
6      confidentiality? authtok
7      freshness? m1
8      freshness? m2
9      freshness? m3
10     freshness? m4
11 ]
```

When opening a Secure Channel it is in our interest to ensure that a possible attacker does not learn about the symmetric keys. This is why we are interested in their confidentiality. As for the Create Session part, we are interested in the authentication token `authtok` being confidential. The authentication of OPC-UA is based on the validation of certificates, which cannot be modeled in the input language of VerifPal. For this reason we have focused on confidentiality and freshness. In any case, we know that integrity is also guaranteed in the model, excluding cryptographic attacks.

6 Risk assessment on the OPC-UA protocol

Now we will give a brief but exhaustive analysis on the possible threats of the OPC-UA protocol, giving the possible impacts and providing mitigations.

6.1 Assets

As described in Section 2, an OPC-UA Client installed on a PC may need to connect to an OPC-UA Server installed in the same way on a PC to exchange messages relating to industrial machinery (PLCs, conveyors, sensors, robotic arms, etc.). All the physical connections are made via Ethernet. Our use case is exactly the one described in Figure 10: an OPC-UA Client establishes a Secure Channel with an OPC-UA Server (in this case, located in a PC that controls a Vertical Storage) to view its status, monitor its variables or call its methods. The connections are made through an ethernet channel, with the TCP transport protocol. As for immaterial assets, as already explained in sub-section 5.1, we have the following elements:

- The secret nonces exchanged for the creation of symmetric keys.
- The authentication token `authtok` exchanged to authenticate the user.

We expect these assets to be **confidential**, **intact** and **fresh**.

6.2 Protocol threats

We shall first take into account that the security profile we use is Sign&Encrypt with the `Basic256Sha256` protocol, considered by the OPC Foundation a security policy for configurations with high security needs [29].
We are interested in keeping the symmetric keys of the principals and the Client authentication token confidential. Furthermore, the messages must all be authenticated and fresh. Following a possible list of threats:

Rogue Server
There are many ways in which the "Rogue Server" attack can be understood. If an attacker succeeds in taking over an authorized Server, this attack is called "*Unauthorized access to the OPC-UA infrastructure*", described in detail below. By "Rogue Server" we will refer to a malicious Server that is injected without the ability to pose as another Authorized Server. The authorization of a Server takes place with the secure exchange and validation of its certificate. Upon receiving a certificate the Client decides whether the received certificate can be trusted. Secure trust lists should be implemented and populated with trusted, untrusted or revoked certificates, and whoever is responsible for managing the certificates must carry out the check explained in Section 2.4. The goal of this attack could be the stealing of credentials. This attack can easily be mitigated by:

- Disabling the security profile None. In this case, disabling means that anyone who wants to open a Secure Channel using this security profile (which does not require certificate validation) is immediately rejected.

- Relying on out-of-band Secure Channels for certificates distribution, using SSH, PGP or physical solutions such as USB sticks.
- Populate the trust lists with trusted certificates before use of the OPC-UA Secure Channels and not through the establishment of the channel. Obviously, all the trust list must not be disabled by default.

The Rogue Server attack impacts **Confidentiality**, **Authentication**, **Authorization** and **Availability**.

Denial of Service of Untrusted Clients
This threat includes an untrusted Client who send a large volume of messages with the goal of overwhelming the OPC-UA Server or other components, such as CPU or operating systems. This attack may lead to slowdown of processes, application crashes or resource exhaustion, also causing the suspension of the service given to users. Flooding attack may include both well-formed and malformed messages(protocol messages with wrong syntax) and can be done by both trusted or untrusted Clients. Such DoS scenarios are summarized below:

- `HEL/ACK/ERR/CLO`**/incorrect messages flooding**: The Client floods the Server by sending `HEL`(Hello), `ACK`(Acknowledge), `ERR`(Error), `CLO`(close) or incorrect (malformed) messages, overloading the network. The Server will always reply with ACK and/or ERR messages. This attack overloads the network with reply messages, but will not significantly impact on Server power consumption, since Server processing is not burdensome for processing these type of messages.
- **FindServer** or **getEndpoints flooding**: The Client can establish a Secure Channel using the security mode None and then flood the network with `FindServers()` and `getEndpoints()` requests. The Server will always reply with a `FindServers()` and `getEndpoints()` response, still with low impact on CPU. Nevertheless, this attack can be mitigated disabling the security mode None.
- **HEL + OPN request flooding**: The Client can send continuously `HEL` and `OPN`(open Secure Channel request) messages to the Server, which replies with `ACK` and `ERR` messages. Every request should be examined by the Server validating the certificate, check the signature and encrypting the response message. Thus, this attack overloads the Server CPU and the network, and becomes more powerful when the CA is located in a different system, as more time and resources are required to validate all certificates. This attack may lead to the exhaustion of all the resources, since a malicious Client may obtain all the sessions available excluding all other Clients.

The DoS attack can be mitigated by:

- Define patterns of normal behaviour on Clients by building acceptance thresholds on the number of messages received. For example, 3 `HEL` messages from the same source could still considered normal behaviour, while

more than 3 `HEL` messages could mean a possible DoS attack by the malicious Client

- Instruct the Server to delay the processing of Open Secure Channel Request once it receives more than some minimum number of bad Secure Channel Requests to avoid resource exhaustion.
- Instruct the Server to reply with an error response without signature and encryption when a certain number of concurrent channels are open in parallel to avoid resource exhaustion. The resources and time used to sign and encrypt are thus saved.

The DoS attack impacts **Availability**.

Eavesdropping
An attacker may learn sensitive information that might result in a critical security breach or be used in other attacks. As we could see in the sub-section 5.1, VerifPal allowed to formally verify the confidentiality of the messages in the protocol handshake phase using the using Symmetric and Asymmetric encryption. **Asymmetric Encryption** is used for session-key agreement and **Symmetric encryption** is used for securing all other messages sent between Client and Server with ephemeral session keys derived from the key derivation function (Session opening, Session activating, Read/Write requests, ...). Furthermore, OPC-UA relies upon the PKI (in the case of a certificate issuance by a PKI) to manage the key used for Asymmetric Encryption, and upon the Cyber Security Risk Management(CSMS) to protect the confidentiality on the system infrastructure.
Eavesdropping impacts directly **Confidentiality**, **Authentication** and **Authorization**.

Message spoofing
With a message spoofing attack an attacker may forge messages to appear to be from an application other than the sending application.
The OPC-UA protocol counters this threat by providing the ability to sign messages. The signature is made with the sender's private key (assuming that it has not been violated) or with a symmetric secret signing key. As we have seen in the sub-section 5.1, messages encrypted with an asymmetric key will have a digital signature of the plaintext inside. Messages encrypted with a ephemeral symmetric key will instead have a plaintext MAC concatenated (MAC-And-Encrypt). Additionally, messages will always contain random numbers like `RequestID`, `SessionID` or monotonically increasing Sequence numbers which make spoofing very difficult.
Message spoofing impacts **Authorization** and **Integrity**.

Message Alteration
If an attacker is able to intercept unauthenticated and unencrypted messages between Client and Server, he can manipulate or suppress them before he forward it to the recipient. OPC-UA counters this threat by signing messages with

digital signature and using MAC.
This attack impacts **Integrity**, **Authorization** and **Authentication** if done during a Session/Secure Channel establishment.

Malformed Message
An attacker may craft messages with invalid and malformed structure, data type or syntax and send them to OPC-UA Clients or Servers. This may lead to a slowdown of service (for Server processing of the message) including unexpected termination of the application.
OPC-UA counters this threat by defining a proper form for the message and its parameters, defined in the `HEL/ACK` messages. If the rules defined in the HEL message are not strictly respected, the message is immediately rejected. Message authentication is also helpful to discard incorrect unauthenticated messages. The Server may specify information like `MaxMessageSize`, `BufferSize` and `MaxChunkCount` to put a limit on the size of message and buffer, and on the maximum count of chunk for each message. Furthermore, every parameter has its own Data type, and incorrect data types are rejected.
Message Alteration impacts **Integrity** and **Availability**.

Session hijacking
An attacker may use information retrieved by sniffing communication or guessing about a running *Session* to inject manipulated but valid Messages that allow him to take over the Session without authorization. However, there are so many encrypted parameters to guess to successfully take over a session, some of which are seen in the model in sub-section 5.1, others are explained in the documentation of the OPC-UA protocol:

- **SequenceNumber**: a monotonically increasing sequence number assigned by the sender (ISO/OSI transport layer) to each `MessageChunk` sent over the Secure Channel. It is important to say that sequence numbers that have already been used should not be reused. In this case an error should be thrown. The number of total sequence numbers must therefore be extremely large to minimize this risk. However, an experiment by Dreier et al. [30] has shown that not all implementations of this protocol respect this property, leading to a (theoretical, but a little less practical) possible replay attack with already used sequence numbers.
- **RequestId**: an identifier assigned by the Client to OPC-UA request Message (transport layer). All `MessageChunks` for a request and the associated response use the same identifier.
- **SecurityToken**: a unique secret token issued by the Server in the Open Secure Channel Response, and reused by the Client for each subsequent request. This token is composed of:
  - The Secure Channel Id.
  - `TokenId`, an unique Id for the token.
  - `CreatedAt`, the creation date of the token.
  - `Lifetime`, the expected life of the token, after which it will no longer be valid. The lifetime is requested by the Client and revised or accepted by the Server.

If the attacker is performing a session hijacking attack simulating a Client, all these parameters need to be guessed correctly, and it's expected to be hard to (except the `SecureChannelId`, that is unencrypted).

- **Authentication Token**: a unique Id assigned by the Server to the Session in the *Create Session Response*. This identifier shall be used to see if a Client has the rights to access to the Session.
- **UserIdentityToken**: the credentials of the user associated with the Client application in the Activate Session Request. Since the Server will check whether a Client is authorized to activate the Session, the attacker must have learned the Client's secret credentials.

Summarizing, an attacker must have compromised a Secure Channel by guessing the Security token issued by the Server, a Session Creation guessing the Authentication token and the Client's credentials to successful activate a Session. Obviously, the attacker must guess the Sequence number and the RequestId too to successful avoid the countermeasures against the replay attack.
In case the credentials are username and password, an attacker could try to do a brute force or dictionary attack on the password. Such a way to prevent this attack of Session hijacking is to delay the Server response when the validation of an user identity fails repeatedly.
Session hijacking impacts **Confidentiality**, **Integrity**, **Authentication**, **Authorization** and **Availability**.

Replay Attack
An attacker may capture and resend valid messages to OPC-UA Clients or Servers without modification, causing loss of authentication. Theoretically, all messages can be recorded by the attacker and sent at a later time, but since they cannot be read by the attacker because they are encrypted, the attack becomes very impractical. Furthermore, verification with VerifPal allowed us to formally verify the freshness of all messages in the handshake phase. Hence assuming that the encryption has not been breached, an effective Replay Attack is possible only on unauthenticated messages. When the Open Secure Channel phase is completed, the following parameters are entered for the preservation of *freshness*:

- `SecureChannelId`, although unencrypted.
- `SessionId`.
- `RequestId`.
- `SequenceNumber`.
- `AuthenticationToken`.
- `Timestamps`, encrypted, present in every authenticated message to ensure freshness.

Another way to carry out a replay attack could be the sending of an unauthenticated and recorded `HEL/ACK` message again, interrupting the connection that must be re-established.
Replay Attack impacts **Authentication** and **Authorization**, since an attacker can send old known credentials to successfully activate a Session.

Server profiling
An attacker may try to indentify type, software version or vendor of the Server or Client in order to exploit known vulnerabilities and mount an existing more intrusive attack. This indirect attack may be carried out in any part of the protocol, although in authenticated parts there is less possibility of drawing conclusion about a Server. In unauthenticated messages, the Server Profiling can be done drawing conclusions about `HEL/ACK` messages, more specifically:

- The **Connection Protocol(UACP) Version.**
- **Buffer size** for sending/receiving messages and `MaxMessageSize`, useful for future attacks.
- The **EndpointUrl**.

Another part of the protocol used in an unsigned and unencrypted form is the Discovery Service:`FindServers()` and `GetEndpoints()`.
In `FindServers()`, an attacker could learn:

- `ServerUri`, `productUri`, `ServerName`, `applicationType` and other informations for each Server. The field `gatewayServerUri` is specified only if a gateway Server is present on the network.
- *Timestamps* and `ServiceResult` to test the responsiveness and the behaviour of the Server.

Instead, from an `GetEndpoint()` response an attacker could learn:

- All the informations about the Servers mentioned above, like the Server Description (including the type: Server, Client, ClientAndServer, DiscoveryServer) and its public certificate.
- Again, the response time of the Server, useful to plan further DoS attacks.
- The Security Profile to apply to the messages. For example, if the secure mode is None, an attacker can proceed with a direct attack, like the session hijacking.
- The `ServiceResult`, the result of the Service invocation.

In authenticated exchanges all the unauthenticated requests are rejected, hence this kind of attack should not work if the attacker has not authenticated first. If an attacker is able to violate authentication, he could retrieve further informations from the Server through the Secure Channel.
An attacker could send incorrect messages too, to see how the Server will react and draw possible conclusions about its normal pattern or catch possible vulnerabilities on bad managed situations. For example, the OPC Foundation believes that attention must be paid to the error codes returned in the event of a malformed message. The error code should be generic in unauthenticated messages, so as not to give clues about the error. Response times for error messages should also be randomly generated.
Server profiling impacts indirectly all our security goals.

6.3 Infrastructural threats

It is also important to specify infrastructural threats, which do not depend on the logic of the protocol but on other factors, such as the security of the system on which the OPC-UA protocol is used.

Unauthorized access to OPC-UA infrastructure
If an attacker gains control of the infrastructures in which an OPC-UA application runs, it can obtain sensitive information such as certificates issuer lists, certificate revocation lists, configuration files, audit data, Certificate Store and system time. If the attacker gains control the Operating Systems it can also read the memory or terminate the applications.
OPC-UA depends on Cyber Security Management System (CSMS) to protect the infrastructure from these kind of attacks.

Compromising user credentials
An attacker may obtain user credentials such as username, passwords, Certificates or keys using automated tools like password crackers or using social engineering. Once compromised credentials are used, subsequent malicious activities may appear legitimate. OPC-UA protects user credentials sent over the network using symmetric/asymmetric encryption, and depends upon the CSMS again other attacks to gain user credentials.
Compromised user credentials impact **Authentication**, **Authorization** and **Confidentiality**[31].

Attacks on implementation of cryptographic algorithms
If there are some known vulnerabilities on the cryptographic functions or the entropy for the generation of random number is insufficient, an attacker may obtain access to protected data.

6.4 Risk Assessment standard table

All the threats seen so far can be summarized in a standard Risk Assessment table, depicted in Figure 14.

In the lines we will have the logical and infrastructural threats mentioned in this Section. In the columns instead we will have, for each threat:

- The **likelihood**, in column 2. It is a measure of the real likelihood of a threat occurring.
- The **impact**, in column 3. It indicates the damage and consequences that the threat has on the assets involved.
- The **risk**, in column 4. It is the measure of the total risk of that threat, taking as input its likelihood and its impact.
- **Confidentiality**, **Integrity** and **Availability** in columns 5, 6 and 7 are the three properties we want preserved in the affected assets. Their value will indicate how much they have been compromised by the current threat on the chosen assets.

THREAT	LIKELIHOOD	IMPACT	RISK	C	I	A	MITIGATION	ATTACK COST
HEL/ACK/ERR/CLO flooding	2.19	1.5	3	0	0	1	Partial	Easy
FindServer()/GetEndpoints() flooding	2.06	1.5	3	0	0	1	Fixed	Easy
OPN+HEL flooding	1.75	2.2	4	0	0	2	Partial	Medium
Rogue Server	2.06	2.9	6	1	0	1	Partial	Easy
Eavesdropping	1.5	2.9	4	2	0	0	Partial	Medium
Message spoofing	0.94	1.9	2	0	0	0	Fixed	Hard
Message alteration	1.25	1.9	2	0	2	0	Fixed	Hard
Malformed message	1.93	1.9	4	0	2	0	Fixed	Hard
Message replay	1.94	1.7	3	0	0	0	Fixed	Easy
Session hijacking	1.5	4.6	7	2	1	1	Fixed	Medium
Server profiling	2.07	0.9	0	0	0	0	Partial	Easy
Unauthorized access of the OS	1.38	4.9	7	2	2	2	Fixed	Hard
Attack on cryptographic algorithms	1.5	2.9	4	2	0	0	Fixed	Hard

Figure 14: Standard Risk Assessment table

- The **mitigation**, in column 8. It indicates whether the solution to the threat exists and whether it is successfully applied.
- The **total cost**, in column 9. It indicates the difficulty in implementing the attack, either in financial terms or in procuring the tools necessary for an effective attack.

Likelihood

The **likelihood** indicates the actual probability that the threat could realistically occur, and is represented as a decimal value that can range from **0** (no risk) to **4** (critical risk). This value is calculated taking into account:

- If the threat is currently exploitable. The *likelihood* value will tend to be high even if the necessary tests have not been carried out to find out if the threat is exploitable.
- How many times has the threat been exploited in the past, and whether it is frequently exploited today, as we will see in the penultimate point.
- Whether a full or partial mitigation of this threat currently exists.
- The total cost of the attack, seen as its economic cost, the resources needed to implement it, and the physical obstacles to its implementation. The cost of the attack is discussed in the last point.

Impact
The **impact** is represented by a decimal value ranging from **0** (no consequences) up to **5** (catastrophic consequences), and indicates the consequences of a possible impact of an attack on an asset, regarding the CIA triad and other economic or social consequences like the company reputation. It is calculated taking into account:

- The consequences on *Confidentiality* of the main asset. The more information the attack can capture, the higher the value.
- The consequences on *Integrity* of the main asset. The more violations to the integrity of the assets are extensive and easily controlled by the attacker, the higher the value.
- The consequences on *Availability* on the main asset. If the violation leads only to a decrease in system performance or to non-significant service interruptions, the value rises slightly. If, on the other hand, the attack leads to a total interruption of access to resources by the attacker, the value is greatly increased
- Possible impacts on immaterial assets like the violation of intellectual property, the hindrance to business continuity due to a Denial of Service attack or the damage to the company's reputation.

Risk
The **Risk** is the final value that takes likelihood and impact as inputs and returns an integer value from **0** (no risk) to **10** (maximum risk).

Confidentiality, integrity, availability
C (Confidentiality), **I**(Integrity) and **A**(Availability), in columns 5, 6 and 7. They are represented by three numbers that represents how severely these three security properties were violated after the attack, and are fundamental contributions for the calculation of the *impact* value.These value can be:

- **0**: The threat does not affect this property, or it does affect it but it produces no consequences.
- **1**: There is some loss of integrity, confidentiality or availability, but the attacker does not have full control over what information he may have violated, the consequences of his modifications, or what resources it may have denied to the user.
- **2**: There is a total loss of confidentiality, integrity or availability. The attacker is able to know, modify or fully deny access to all resources protected by the attacked component. There are serious consequences on the assets involved.

Mitigation
The **mitigation** field indicates whether effective measures have been taken to prevent the attack. It is a fundamental element for the calculation of the *likelihood*, and can have two values:

- **Partial mitigation**(already implemented), that almost entirely prevents the attack patterns, excluding some of significant importance.
- **Fully mitigation**, successfully prevents all attack patterns.

There are currently no threats in this protocol whose mitigation has not yet been implemented or is not possible to by design.

Total cost
The **total cost** of the attack is an input element for calculating *likelihood*. It is calculated taking into account several values:

- How important are the motivations and objectives that lead an attacker to exploit the threat. If the attack is "less attractive" to state-owned industrial espionage organizations or large cyber-criminal groups it is more likely to be launched, and the value of *likelihood* will drop.
- What is the financial budget available to the attacker, and how much manpower is needed. If the attack is financially expensive, the *likelihood* of it happening decreases.
- How complex and advanced are the tools used to conduct the attack. Some technologically advanced tools can be difficult to find by an attacker, making it difficult to implement the threat.
- What is the physical distance between the attacker and the System. If physical proximity of the attacker to the system is required, it will be more difficult for an attacker to exploit the threat. Conversely, if an Internet connection is sufficient, the *likelihood* of the attack increases.
- How difficult it is to access the information useful for conducting the attack. For example, if knowledge of private informations in the system is required, the attacker will have to compromise them or be an insider, making the attack more difficult to implement.

Having made these assessments, the total difficulty of the attack can be *Easy*, *Medium* or *Hard*.

7 Conclusions

We have seen how the formal verification of the OPC-UA protocol is important for the security of the data and information exchanged on industrial systems. The VerifPal tool gave us an intuitive and fast way to model the protocol, while making us realize the limitations of this ease of modeling. Thanks to the work previously carried out on this protocol, we were able to conduct a risk assessment of the logical and infrastructural risks of the asyncio implementation of OPC-UA, improving the documentation available to the ICE laboratory. This has allowed us to demonstrate that formal verification is not enough without the user's commitment to protect their industrial infrastructure; given the criticality of industrial systems and the fact that new OPC-UA vulnerabilities are discovered on a monthly basis.
Of particular importance in the development of this thesis are the following papers:

- Formal Analysis of Security Properties on the OPC-UA SCADA Protocol [5], where Puys et al. verified the confidentiality and authentication of an implementation of the OPC-UA protocol (FreeOpcUa[17]) with ProVerif[32], providing sequence diagrams of the message flow and possible countermeasures.
- The Federal Office for Information Security (BSI) proposes every 5 years an article [20] containing a very thorough security analysis of some implementations of the OPC-UA protocol.
- Varlei Neu et al. in [33] focused on the Denial Of Service attack by an unauthenticated Client to a Server on the OPC-UA protocol and on its possible mitigations.

In future works it could be useful to study and formalize the interoperability of the protocol with other components with which it collaborates, such as messaging brokers in the Publisher/Subscriber mode.

References

[1] Technical Committee ISO/IEC JTC 1. *ISO/IEC 27000:2018 - Information technology - Security techniques - Information security management systems - Overview and vocabulary*. en. Tech. rep. Version 5. International Organization for Standardization, Feb. 2018. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:27000:ed-5:v1:en.

[2] *Over One-third of Industrial Control Systems Were Attacked in Q1 2021*. [Online; accessed 04-July-2022]. Oct. 2021. URL: https://www.packetlabs.net/posts/industrial-control-systems-attacked/.

[3] Eric Chien Nicolas Falliere Liam O Murchu. "W32.Stuxnet Dossier". In: (2010). URL: https://www.wired.com/images_blogs/threatlevel/2010/11/w32_stuxnet_dossier.pdf.

[4] Wikipedia. *Colonial Pipeline ransomware attack — Wikipedia, The Free Encyclopedia*. [Online; accessed 05-May-2022]. 2022. URL: http://en.wikipedia.org/w/index.php?title=Colonial%5C%20Pipeline%5C%20ransomware%5C%20attack%5C&oldid=1074669069.

[5] Pascal Lafourcade Maxime Puys Marie-Laure Potet. "Formal Analysis of Security Properties on the OPC-UA SCADA Protocol". In: (2016). DOI: 10.1007/978-3-319-45477-1_6. URL: http://dx.doi.org/10.1007/978-3-319-45477-1_6.

[6] *A True Cost of Cyberattacks*. [Online; accessed 05-May-2022]. URL: https://www.kaspersky.com/blog/cost-cyberattack-enterprise/5195/.

[7] *Cybercrime To Cost The World $10.5 Trillion Annually By 2025*. [Online; accessed 05-May-2022]. May 2021. URL: https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/.

[8] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. *Verifpal: Cryptographic Protocol Analysis for the Real World*. Cryptology ePrint Archive, Paper 2019/971. https://eprint.iacr.org/2019/971. 2019. URL: https://eprint.iacr.org/2019/971.

[9] *IceLab Laboratory*. URL: https://www.icelab.di.univr.it/laboratory/ (visited on 2022).

[10] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. Berlin: Springer, 2009. ISBN: 978-3-540-68898-3. DOI: 10.1007/978-3-540-68899-0.

[11] OPC Foundation. *OPC 10000-4 Unified Architecture Part 4 Services*. Version 1.05.00. [Online; accessed 06-June-2022]. Nov. 2021. URL: https://reference.opcfoundation.org/v104/Core/docs/Part4/.

[12] OPC Foundation. *OPC 10000-1 Unified Architecture Part 1 Overview and Concepts General*. [Online; accessed 02-June-2022]. 2022. URL: https://reference.opcfoundation.org/v104/Core/docs/Part1/7.1/.

[13] Gavin Lowe. "Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 147–166. ISBN: 978-3-540-49874-2. DOI: 10.1007/3-540-61042-1_43.

[14] Mart N Abadi and Roger Needham. "Prudent engineering practice for cryptographic protocols". In: *IEEE Transactions on Software Engineering* 22 (1996), pp. 122–136.

[15] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[16] *open62541*. URL: https://github.com/open62541/open62541 (visited on 2022).

[17] *FreeOpcUa*. URL: https://freeopcua.github.io/ (visited on 2022).

[18] *opcua-asyncio*. Version 0.9.92. Nov. 2021. URL: https://github.com/FreeOpcUa/opcua-asyncio (visited on 2022).

[19] *Python Documentation - asyncio - Asynchronous I/O*. URL: https://docs.python.org/3/library/asyncio.html (visited on 2022).

[20] Federal Office for Information Security(BSI). "OPC UA Security Analysis - OPC Foundation". In: (). [Online; accessed 25-June-2022]. URL: https://opcfoundation.org/wp-content/uploads/2017/04/OPC_UA_security_analysis-OPC-F-Responses-2017_04_21.pdf.

[21] Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. "Authentication and authenticated key exchanges". In: vol. 2. June 1992, pp. 107–125. DOI: 10.1007/BF00124891. URL: https://doi.org/10.1007/BF00124891.

[22] Wikipedia. *Automation pyramid*. [Online; accessed 02-June-2022]. Nov. 2020. URL: https://second.wiki/wiki/automatisierungspyramide.

[23] Clemens Heinrich. "Pretty Good Privacy (PGP)". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg. Boston, MA: Springer US, 2005, pp. 466–470. ISBN: 978-0-387-23483-0. DOI: 10.1007/0-387-23483-7_310. URL: https://doi.org/10.1007/0-387-23483-7_310.

[24] Carlisle Adams. "Kerberos Authentication Protocol". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg. Boston, MA: Springer US, 2005, pp. 323–323. ISBN: 978-0-387-23483-0. DOI: 10.1007/0-387-23483-7_216. URL: https://doi.org/10.1007/0-387-23483-7_216.

[25] *Kubernetes*. URL: https://kubernetes.io/ (visited on 2022).

[26] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: 10.1109/TIT.1983.1056650.

[27] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654.

[28] OPC Foundation. *OPC 10000-6 Unified Architecture Part 6 Mappings Deriving keys*. Version 1.05.01. [Online; accessed 06-June-2022]. Feb. 2022. URL: https://reference.opcfoundation.org/v104/Core/docs/Part6/6.7.5/.

[29] OPC Foundation. *OPC 10000-7 Unified Architecture Part 7 Profiles Security Policy [B] – Basic256Sha256*. Version 1.05.00. [Online; accessed 06-June-2022]. Feb. 2022. URL: https://reference.opcfoundation.org/v104/Core/docs/Part7/6.6.165/.

[30] Jannik Dreier et al. "Formally and Practically Verifying Flow Integrity Properties in Industrial Systems". In: *Computers and Security* 86 (Dec. 2018), pp. 453–470. DOI: 10.1016/j.cose.2018.09.018. URL: https://hal.archives-ouvertes.fr/hal-01959766.

[31] OPC Foundation. *OPC 10000-2 Unified Architecture Part 2 Security Model Compromising user credentials*. Version 1.05.00. [Online; accessed 24-June-2022]. Nov. 2021. URL: https://reference.opcfoundation.org/v104/Core/docs/Part2/4.3.12/.

[32] Bruno Blanchet et al. "ProVerif 2.04: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial". In: (Nov. 2021).

[33] Charles Varlei Neu, Ina Schiering, and Avelino Zorzo. "Simulating and Detecting Attacks of Untrusted Clients in OPC UA Networks". In: *Proceedings of the Third Central European Cybersecurity Conference*. CECC 2019. Munich, Germany: Association for Computing Machinery, 2019. ISBN: 9781450372961. DOI: 10.1145/3360664.3360675. URL: https://doi.org/10.1145/3360664.3360675.