Design of an Example Network Protocol for Security Tests Targeting Industrial Automation Systems

Steffen Pfrang[a], Mark Giraud, Anne Borcherding, David Meier[b] and Jürgen Beyerer
Fraunhofer IOSB, Karlsruhe, Germany

Keywords: Security Testing, Industrial Automation and Control Systems, IACS, Example Network Protocol, Packet Structure, Protocol Behavior.

Abstract: Emerging concepts like *Industrial Internet of Things (IIOT)* and *Industrie 4.0* require Industrial Automation and Control Systems (IACS) to be connected via networks and even to the Internet. These connections raise the importance of security for those devices enormously. Security testing for IACS aims at searching for vulnerabilities which can be utilized by attackers from the network. Once discovered, those gaps should be closed with patches before they can get exploited. Different tools utilized for this kind of security testing are dealing with network protocols. In practice, they suffer from peculiarities being present in common industrial automation protocols like OPC UA and Profinet IO. This paper tries to improve the situation by providing an extensive overview of network packet structures and network protocol behavior. Based on this analysis, an example protocol has been developed. The idea behind this artificial network protocol is that tools which are able to handle all the specialties of this protocol, are able to handle every imaginable protocol. Finally, those tools can be used to conduct exhaustive security tests for IACS.

1 INTRODUCTION

Security testing for industrial automation and control systems is gaining more and more importance. As emerging concepts and techniques from *Industrial Internet of Things* and *Industrie 4.0* require sensors, actuators and controllers to be connected via networks and the Internet, security becomes a critical factor. In former times, security did not play a major role in industrial devices as they were run in separated networks. But since the adaption of Ethernet, IACS components have to resist attacks originating from other networks, such as office networks or even the whole world.

Security testing aims at discovering vulnerabilities to enable IACS manufacturers to patch their devices before attackers can exploit them. A common approach for security testing is fuzzing. Fuzzing network protocols means sending meaningfully chosen input as payload in network packets to a device under test (DUT). Advanced fuzzing techniques take into account both packet structures and states of the protocol to be fuzzed. This allows for choosing input with a higher probability of being interpreted by the DUT.

Tools dealing with packet structures and states of network protocols are usually developed for and tested with well-known protocols. Examples for such tools are self-learning fuzzers which construct a packet and state model of a specific network protocol by using packet captures. Other tools are meant for crafting network packets or tracing the state of a stateful network protocol. Since especially industrial protocols like Profinet IO or OPC UA use special concepts, many of the tools lack functionality to deal with them. An example for such a missing feature are infinite recursive data types which are used in OPC UA. There is a lack of a common standard protocol which fits all the needs.

The idea to overcome this issue is the specification of an example protocol which makes use of all the concepts discovered in both standard Internet protocols and real world industrial protocols. Tools to be developed or maintained can be tested with this example protocol. If the tool is able to handle all the requirements of the example protocol, it should be able to implement every industrial protocol.

This paper presents a requirement analysis for the needs of common network protocols and the design of such an example protocol. Additionally, a reference

[a] https://orcid.org/0000-0001-7768-7259
[b] https://orcid.org/0000-0003-0660-8087

Pfrang S., Giraud M., Borcherding A., Meier D. and Beyerer J.
Design of an Example Network Protocol for Security Tests Targeting Industrial Automation Systems.
DOI: 10.5220/0007704907270738
In *Proceedings of the 5th International Conference on Information Systems Security and Privacy (ICISSP 3rd International Workshop on FORmal methods for Security Engineering)*, pages 727-738
ISBN: 978-989-758-359-9

implementation is released to the public to enable security researchers to use it.

The paper is structured as follows: Starting with introducing the background in Section 2, the analysis in Section 3 presents three use cases for the example protocol. Two common industrial protocols are examined more closely, before a packet field categorization prepares the collection of the requirements for the example protocol. The example protocol is designed in Section 4 with both protocol behavior and packet structures. Finally, a case study in Section 5 illustrates the usage of the example protocol.

2 BACKGROUND

The research of this paper is located in different domains: Industrial automation, security testing and network protocol design. All domains will be presented shortly in the following.

2.1 Industrial Automation

Physical equipment in industrial environments is monitored and controlled by industrial automation and control systems (IACS). With their reliance on proprietary hardware, software, and networks, these systems have often been considered too complex to be attacked. In addition, IACS have often been located in local networks with no connection to the Internet.

As the systems and network protocols evolve, these points are no longer valid. Systems and networks move towards open standards like Ethernet, TCP/IP and web technologies. In addition, IACS are more and more connected to the Internet. This development makes the IACS more interesting for hackers. The most popular example for a successful attack on IACS is *Stuxnet* (see (Lüders, 2011)). The effects of a successful attack on IACS range from a manipulation or disruption of the production process to personal or environmental damage or loss.

2.2 Security Testing

It is crucial to ensure the security of IACS to avoid the effects of a successful attack. One aspect of this is to identify vulnerabilities and to ensure security functionality. It is called *security testing* and includes many different techniques. These techniques may be classified into four categories which are presented in Table 1. The classification and the corresponding descriptions are adapted from (Felderer et al., 2016) as long as not stated otherwise.

Table 1: Classification of security testing techniques (adapted from (Felderer et al., 2016)).

model-based security testing
... for web applications
... for security policies
risk-based security testing
code-based testing and static analysis
manual code review
static application security testing
penetration testing and dynamic analysis
penetration testing
vulnerability scanning
dynamic taint analysis
fuzzing
security regression testing
test suite minimization
test case prioritization
test case selection

Model-based security testing presumes the existence of architectural and functional models of the system under test. From these models, test cases for the security test are derived systematically. Because of the need of precise models, these techniques require an early and explicit specification. The models may be driven by the concrete system, by security policies, or by risks.

Code-based testing and static analysis focuses on the code of a system. No running system is needed to apply these techniques. The review of the code can either be done manually or automatically.

Penetration testing and dynamic analysis can be conducted if a running version of the system is available. These techniques do not need access to the source code of the system and are thus also called *black box testing*. Security tests are performed by interacting with the running system. This includes manual penetration testing, automated scanning for known vulnerabilities, dynamic analysis of data sanitization, and fuzzing. Fuzzing uses random or randomly modified data and feeds it to the system. With this, the reaction of the system to unexpected data can be investigated which may expose vulnerabilities (Pfrang et al., 2018).

Security regression testing aims to ensure that a change of a system does not downgrade the security of the system. To be able to conduct these test efficiently, the test suite can be minimized, the test cases can be prioritized and the test cases can be selected in an optimized way.

Each of the presented techniques is located at dif-

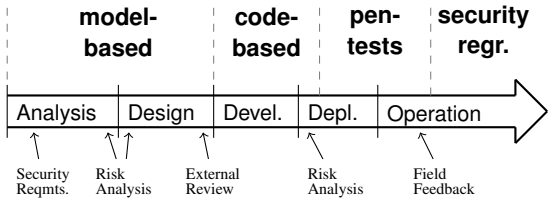

Figure 1: Security testing techniques at different stages of the system development.

ferent stages of the system development. This is shown in Figure 1. Presented are the stages of the system development, adapted from (Felderer et al., 2016): *Analysis*, *Design*, *Development*, *Deployment*, and *Operation (Maintenance)*. On top of the arrow, the security testing techniques which may be used in the respective stages are shown. Additional input which may be used for testing is depicted underneath the arrow (Potter and McGraw, 2004).

Next to the shown classification, a classification using other dimensions is possible. For example, security testing techniques could be classified regarding their accessibility (black-box vs. white box), scope (component vs. integration vs. system), objective (functional vs. non-functional), user interaction (manual vs. automated), and their practice (active vs. passive).

2.3 Network Protocol Design

Communication between programs and services of host systems is commonly achieved through the usage of network protocols.

2.3.1 Protocol Goals and Features

Network protocols provide features to achieve different communication goals. They have varying properties, that connected applications rely on, to achieve these goals. These properties include (Peterson and Davie, 2007):

- **Message Delivery Guarantees:** The protocol makes sure that messages get delivered. If a message is lost during transmission, the protocol can initiate additional delivery attempts.
- **Maintaining Data Order:** A network protocol can make sure that the receiving application gets the data served in its original order, independent from packet order changes that might happen during the transport.
- **Filtering of Duplicates:** During transport packets can get duplicated, e.g. because of retransmissions or multi-path transmission. These duplicates can be automatically identified and dropped by a network protocol.
- **Support of Large Messages:** A sender might want to transmit data that exceeds the frame limit of the carried medium or link technology. A protocol could automatically split the message into multiple smaller ones and reassemble them at the receiver.
- **Flow and Congestion Control:** The maximal data rate between sender and receiver is determined by different factors, like the maximum capacity of the communication channel and the resources of the receiver. Network protocols can automatically adjust the sending rate accordingly.
- **Application Separation:** Protocols can help to ensure that data from different applications stays separated, even when transmitted via the same communication link.

Most of these properties and features are provided transparently to the different layers of the protocol stack. This stack can be organized into layers to achieve a high rate of adaptability. In this layer model, each protocol layer is providing services other layer can utilize (Kurose and Ross, 2013). The *Open Systems Interconnection model* (ISO, 1994), often called ISO/OSI model, or the *internet protocol stack* (Braden, 1989) formalize this approach.

Above the *Physical* layer, which handles the transmission of information over a certain channel medium, the *Link* layer deals with packet frames that are transmitted from one node to another. Data transmission of datagrams over multiple nodes is handled by the *Network* layer. It is also responsible for finding appropriate routes between two nodes. The *Transport* layer handles segments and transmits data between two application instances. On the *Application* layer, transmitted data is referred to as messages.

2.3.2 Protocol Definitions

Individual protocols are defined by their multiple properties. The main properties of a protocol are the structures of its protocol data units (PDU) and its behavioral description.

Because the actual transmission is carried out on binary data, the data needs to be serialized at the sender and parsed at the receiver to enable the proper processing of the PDU. A basic division of a PDU can be done by separating meta data (*header*) and the actual data that needs to be transmitted (*payload*): The packet *header* will include all the information that is needed by the protocol to perform its functions. It can include information on the communication link as

well as information on the appended *payload*. In the protocol definition, the *header* is divided into packet fields. These fields can represent different kinds of information, like integer values or flags for protocol settings. The *payload* can then be further processed by the following layer of the network stack. It is often appended to the end of the PDU, but can also be integrated at different positions of the PDU.

Another part of a protocol definition is the description of the protocol behavior. This includes details on how to establish and maintain a connection. The behavior also includes details on the determination of packet fields, for example, how exactly a checksum needs to be calculated. Part of the behavior description are the states a connection can be in. These states and the transitions between the states can be represented by a state-transition diagram.

An example for a transport layer protocol is the *User Datagram Protocol* (UDP). UDP is a simple, efficient and stateless protocol, meaning there is no connection status established on the transport layer. The *Transmission Control Protocol* (TCP) is a more complex example on the transport layer. It provides functions for flow and congestion control and is stateful. Therefore, a connection establishment process is needed and implemented in a three-way handshake.

3 ANALYSIS

The example protocol has to be an artificial network protocol which makes use of any features observed in existing network protocols. A special focus is laid on industrial communication protocols.

This section starts with presenting use cases which benefit from the example protocol. Then, two popular industrial communication protocols are introduced and examined regarding their characteristics in packet definitions, states and transitions between states. Peculiarities observed in some other network protocols are added. A packet field categorization is introduced which leads to the formulation of the requirements for the example protocol.

3.1 Use Cases

There are multiple use cases which benefit from the existence of an example protocol. Three of them are depicted in this section. The first one, protocol learning tools, will be used in a case study.

3.1.1 Protocol Learning Tools

Security testing in the operation phase of the lifecycle of an IACS component employs fuzzing techniques to generate input to be tested. In the case of network packets, the exact knowledge of the protocol definition helps in crafting input that pays attention to field borders, data types and check sums. This increases the probability that the device under test interprets the fuzzed network packet instead of rejecting it immediately.

Problems arise if the specification of the network protocol is not available. Reverse engineering is performed by guessing both packet definitions and states with a large sample of observed communication. (Duchene et al., 2018) gives a state of the art overview of network reverse engineering tools. The evaluation of those tools is performed by testing with well-known internet protocols like DNS, NTP or DHCP. There is neither a well-defined test set of network protocols nor a differentiation which kinds of packet field definition types are to be used. A well-defined example protocol would overcome this issue and make results more comparable.

3.1.2 Protocol Implementation Frameworks

Software frameworks that are designed to handle arbitrary network protocols need a baseline of features. Scapy (Biondi, 2018), for example, is an interactive packet manipulation program and library used very often in the security testing domain. Amongst others, it is able to decode packets of a wide number of protocols. A drawback is that Scapy is not designed for implementing deep recursion in data types. Additionally, it has problems with the dynamic creation of data types. If there was an example protocol which provided a guideline for features to be implemented, the tools could be designed better.

Implementations of stateful network protocols utilize state machines in order to model states. An automata framework to be developed would profit from an example protocol covering all needed features for states, transitions and transition triggers. Implementing the example protocol in the newly designed automata framework could expose if all needed functionality is available.

3.1.3 Meta Model Design

A meta model for network protocols allows for specifying arbitrary network protocols both formally and technically. Again, if there was a baseline of features to support in packet structures and states, the meta

model could be designed covering all needed functionality. The example protocol aims at fulfilling this requirement.

3.2 Industrial Protocols

Industrial protocols are used to automate the communication of IACS. Two common representatives in the industrial area are OPC UA and Profinet. Both will be introduced with a focus on peculiarities in packet definitions and protocol states. Additionally, special findings from other protocols will be presented.

3.2.1 OPC UA

OPC Unified Architecture (OPC UA) is a machine-to-machine communication protocol used in industrial automation (OPC Foundation, 2017a). The protocol specifies models for information, messages and communication. It offers a built-in address space and object model and follows a service-oriented design. While mainly relying on a *Client/Server* architecture, it also provides functionalities for *Publisher/Subscriber* architectures.

The communication model is divided into three layers, as shown in Figure 2: The transport layer, the secure channel layer and the session layer. The transport layer establishes the network connection between client and server. The secure *SecureChannel* provides a secured, long-running connection between the two communication partners. These security features include encryption and integrity checking, and are defined by a security policy. The *Session* contains the application-layer connection between two OPC UA services and can only be accessed via a single *SecureChannel*. Sessions are stateful and contain, for example, linked authentication information like user credentials. However, sessions are ultimately independent from the underlying communication protocol, making it possible to transfer a session to another *SecureChannel*.

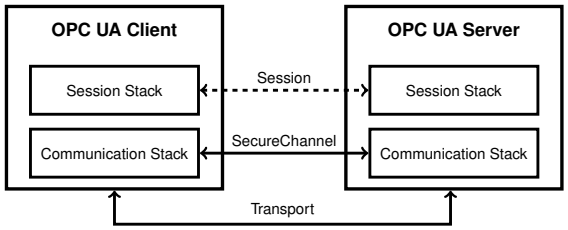

Figure 2: The OPC UA communication stack, adapted from (OPC Foundation, 2017a, 20).

OPC UA has its own type system that is used to send packets over the network. The corresponding standard differentiates between simple and structured data types (OPC Foundation, 2017b, 35).

Some data types that need to be implemented manually are called built-in data types. The simple data types are subtypes of these built-in data types. They cannot be differentiated from another when encoded (OPC Foundation, 2017b, 35). These data types form the base from which all other data types can be generated (OPC Foundation, 2017c, 6). Parsing behavior for built-in data types is specified in the standard (OPC Foundation, 2017c, 6) and is fixed unless the standard is changed. Among other things, the built-in data types contain so called *Variants* and *ExtensionObjects* (OPC Foundation, 2017c, 6). Variants can contain other built-in data types or arrays of other built-in data types. The length of the contained array is represented by a length field in the Variant (OPC Foundation, 2017c, 16). In contrast, ExtensionObjects are used to transmit an arbitrary data type over the network. The sent data type is modeled in the server's address space. In order to achieve this, the ExtensionObject contains the encoded data and the id of the data type that was used to encode the data (OPC Foundation, 2017c, 15).

Structured data types are modeled in a server's address space, where they can be retrieved in order to parse incoming traffic (OPC Foundation, 2017b, 35). These data types represent an aggregation of other structured data types and/or simple data types (OPC Foundation, 2017b, 69). When these data types are sent over the network, the receiving end needs to know how to parse them. This is achieved by encapsulating them in the aforementioned ExtensionObjects. The receiving end can retrieve the parsing rules with the information in the ExtensionObject for the encapsulated type from the server in order to parse the type (OPC Foundation, 2017b, 35).

Data types can be specified via XML schema files, which can be parsed by OPC UA applications. Commonly known data types can be found in a publicly available schema file, so that they do not have to be read from a server each time (OPC Foundation, 2017b, 35).

Due to the way OPC UA enables arbitrary modeling of data types, it is possible to define a type that contains itself directly or indirectly through another type.

3.2.2 Profinet IO

Profinet IO (PNIO) is a real-time enabled industrial Ethernet standard defined in IEC 61158-6-10. According to a study from 2017 (compared to 2016)(HMS Industrial Networks, 2017), industrial Ethernet has a market share of 46% (38%) in IACS, whereas classic field buses have 48% (58%). 6% (4%) of the sold

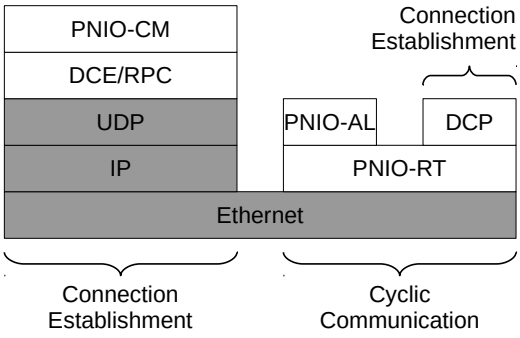

Figure 3: Protocol stack of PNIO.

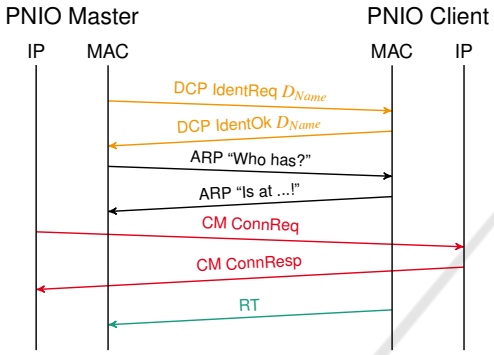

Figure 4: Begin of the regular startup process of a PNIO network with a PNIO master and a PNIO client until the first RT packet, from (Pfrang and Meier, 2018).

communication devices use wireless protocols. Siemens selling the *Profibus family* is the market leader in Europe and China.

As depicted in Figure 3, the PNIO protocol stack is based on Ethernet. For establishing a connection, it uses both DCP (discovery and configuration protocol) and PNIO-CM (configuration management). DCP is based on PNIO-RT (real-time), whereas PNIO-CM makes use of IP, UDP and DCE/RPC. In a nutshell, PNIO communication is set up with different protocols using different protocol stacks.

The interaction between the different protocols is shown as a sequence diagram in Figure 4. A characteristic of PNIO-CM is that the type of a PNIO-CM packet (Read, Write, Connect etc.) is defined by a field within the underlying DCE/RPC protocol. Within the connection establishment, the PNIO master and PNIO client define variables and data types as *blocks* which are exchanged later on as binary data. Once a connection is established, PNIO devices transfer real-time data via PNIO-RT.

While exchanging real-time data, timeouts are reset once a communication partner receives a valid RT message. If the delay of a packet exceeds a configured amount of time, RT communication is stopped and an alarm packet is sent instead.

3.2.3 Other Protocols

DCE/RPC (Distributed Computing Environment / Remote Procedure Calls) is the base of the PNIO-CM protocol. It breaks with the traditional modeling of states and transitions in network protocols as it defines a *maybe* flag. In a client to server communication, a client can issue a maybe request. This means that the client does not expect a response and therefore cannot be sure if the server executes the request or not.

ETSI TS 103 097 V1.2.1 (ETSI, 2015), a protocol for Intelligent Transport Systems, defines a packet field as *variable-length vectors with variable-length length encoding*. This means that the length itself is encoded with a number of *1* bits according to the additional number of octets used to encode the length, followed by a *0* bit and the actual length value.

3.3 Packet Field Categorization

According to the analysis of both common Internet protocols like TCP and UDP and the aforementioned industrial protocols, a characterization of packet fields has been performed. Figure 5 depicts a tree-based schema how packet fields can be categorized. Grey boxes represent leaves. A packet field consists of both a length and a parsing rule.

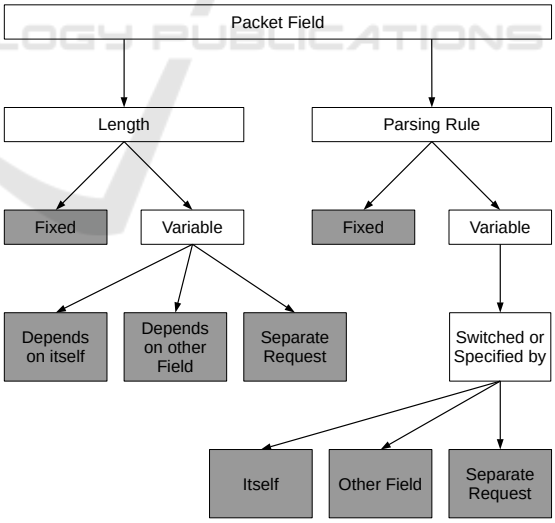

Figure 5: Packet field categorization.

3.3.1 Length

The *length* can be specified with a number of bytes or a number of bits. The latter means that fields might not be aligned to byte borders. Usually, if there are fields that are not a multiple of 8 bits (a byte), they are

laid out (filled up with a so-called padding of zeroes). That results in fields always starting at a byte border if they are a multiple of a byte. It is also possible, that the length specifies a number of fields that are present in the packet. If that is the case, the parsing rule for the field that is repeated, is applied as often as the length field specifies. An example of this would be an array of fields.

The length of a field can be either fixed or variable. An example for a *fixed length field* is a field that contains an integer. The length of a *variable length field* can either *depend on itself* or *depend on other fields* or been agreed on in a *separate request*.

An example for a dependance on itself is if the length field is inline, for example the number of *1* from the beginning until the first *0* depicts the length of the field in bytes (see Section 3.2.3). In case of a length determination from another field, for example a separate integer length field depicts the length of the field in bytes. If the length of a specific field is already agreed on in the connection establishment process, no indication of the length has to be transferred in the particular packet.

3.3.2 Parsing Rule

The *parsing rule* defines the way the field is to be parsed. In case of a *fixed parsing rule*, it is defined in a standard and fixed regardless the circumstances. A *variable parsing rule* however requires a fixed rule that defines how the parsing rule is to be parsed. This fixed rule can either be switched or specified by *itself*, by *another field* or via a *separate request*. If a parsing rule is switched, this can be done using an enumeration. If the parsing rule is specified, for example shell code is transferred which has to be evaluated in order to parse the field.

The parsing rules themselves can be distinguished between two types: *Elementary types* on the one hand usually represent a value that can be stored in a programming languages basic types, like for example ints or floats. *Compound types* on the other hand represent a collection of other fields. These other fields can either be of elementary or compound type.

An example for a compound type is shown in Figure 6. *StringField* is a compound field consisting of a length field and a charlist field. *LengthField* is a an elementary field which is parsed to an integer. It represents the number of char fields that will be parsed in the CharList. *CharList* is a compound field, consisting of multiple char fields. Finally, *CharField* is an elementary field which is parsed to a char.

Network layers according to the ISO/OSI model can be modeled as compound types. The Ethernet layer as a whole, for example, is a compound type

```
StringField   -- Compound field
├─LengthField -- Elementary field (int)
└─CharList    -- Compound field
  └─CharField -- Elementary field (char)
```

Figure 6: String as an example for a compound type.

consisting of the source and destination MAC etc. The payload contains a compound type that is determined by the type field. Usually the payload is parsed according to a fixed parsing rule.

Compound types can be arranged in a way that they allow direct or indirect recursion, which presents a challenge for implementers. In Scapy for example, in order to model direct recursion it is necessary to modify an already created class. The Class has an attribute that contains a list of fields. During the parsing of the class, the class is not yet defined, so it cannot contain a reference to itself. In order to circumvent this restriction the class has to be modified after its creation.

Conditional fields, i.e. fields that are only present depending on other parts of the packet, can be represented by fields with variable parsing rules. The parsing rule changes depending on a switch field. If the field is present, it is parsed according to the fields rule. If it is not present, the parser just parses zero bytes, i.e. skips the field.

3.4 Requirements

According to the analysis of the use cases, the common industrial protocols, and taking into account the packet field categorization, the following requirements for the example protocol are inferred. They are split in two categories: State requirements (prefix `SR`) and packet requirements (prefix `PR`).

`SR_STATE.` The example protocol is a stateful network communication protocol. Its states are connected via transitions.

`SR_TRIG.` Transitions are triggered either by
- `(a)` receiving network packets or
- `(b)` temporal conditions.

`SR_EFFECT.` Transitions trigger one or more of the following effects:
- `(a)` sending network packets,
- `(b)` starting timeouts,
- `(c)` or no effect at all.

`SR_SEND.` In at least one state, network packets are being sent without any transition to another state.

`SR_STACK.` The example protocol is based on existing protocol stacks. It uses more than one stack.

`SR_INFO.` The example protocol makes use of underlain protocol information.

`SR_TIMEOUT_SM.` The example protocol uses more than one timeout in one state.

SR_TIMEOUT_MS. At least one timeout is used in more than one state.

PR_DETERM. The example protocol comprises packets that contain at least one field with each combination of length and parsing rule determination depicted in Figure 5.

4 DESIGN

The example protocol has to be an artificial network-based communication protocol comprising every peculiarity which can be observed in common existing industrial communication protocols. In order to fulfill that demand, the requirements regarding both the protocol behavior and the packet field definitions determined in Section 3.4 have to be met.

Additionally, in order to support the use cases depicted in Section 3.1, the protocol has to be made publicly available. This allows security researchers for implementing and evaluating tools and frameworks dealing with network communication.

4.1 Protocol Behavior

The example protocol uses two layers which are stacked upon TCP: A channel layer and a session layer. Additionally, it provides *regular services* which can be called in different connection states. A *special service* which can be invoked only in the session layer is exchanging RT (real-time) traffic on top of UDP. The layers are depicted in Figure 7. This basic setup of the example protocol fulfills the *SR_STATE.*, the *SR_TRIG. (a)*, the *SR_EFFECT. (a)* and the *SR_STACK.* requirements.

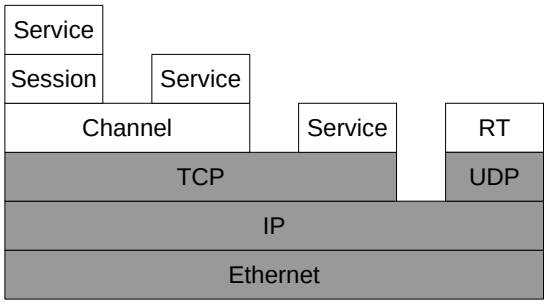

Figure 7: The example protocol stack.

A channel is dependent on a TCP connection. To establish a channel, a TCP connection needs to be established first. If a channel is closed, the TCP connection remains active and can be reused. A session may only be established on an existing channel but can persist even if the channel is closed. To reuse an existing session, it needs to be attached to a new channel.

The following explanation is intended to be an overview over the features of the example protocol and by no means a full protocol description. To show the complexity, Figure 8 pictures a state machine graph of the example protocol. States are depicted as circles, transitions as arrows. While the actual names of the states are not readable, special connection-related states are highlighted: *Connected (CON)*, *Channel opened (OPE)*, *Session created(CRE)*, *Session attached (ATT)*, *Disconnected (DIS)* and *Error (ERR)*. Additionally, accumulations of states responsible for providing the services, are marked using blue boxes.

4.1.1 Channel Layer

As can be seen in Figure 7, the channel layer is built upon TCP. The establishment of a channel is depicted in Figure 9. To establish a new channel, first a TCP connection is established using a three-way handshake.

When the TCP connection is established, the client sends a *CHANNEL_OPEN* request to the server. The client may define a timeout after which the establishment of the channel is canceled. The server sends either a *CHANNEL_OPEN_RESPONSE* or an *ERROR* response. These timeout definition fulfills the requirements *SR_TRIG. (b)* and *SR_EFFECT. (b)*.

Each message sent on a channel has a sequence number which is incremented with each package sent. The sequence number used by the client might be different to the sequence number of the server.

Each channel has an ID. With this, it is possible to manage more than one channel at a time. This ID needs to be transferred with each message sent on the channel. Using the corresponding channel service, it is possible to set the maximum number of channels that can be open in parallel. Channel services can be issued by the client with a *NOACK* flag. If set, the server will perform the action associated with the request but will not send any response packet. This fulfills the *SR_EFFECT. (c)* and *SR_SEND.* requirements.

An active channel may be closed either by the server or the client using *CHANNEL_CLOSE*. With a different request, it is also possible to renew the sequence numbers used. The underlain TCP connection will stay alive allowing for both executing services or creating a new channel.

The example protocol defines a service which provides the possibility to find out the number of currently active channels (*get_active_channels*). It is possible to define a timeout for this request. After this

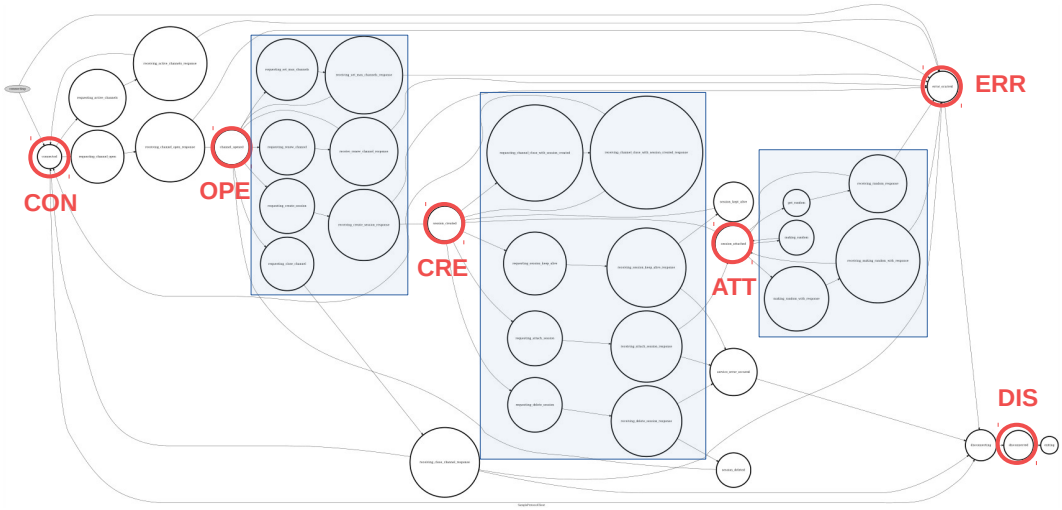

Figure 8: The state machine of the example protocol.

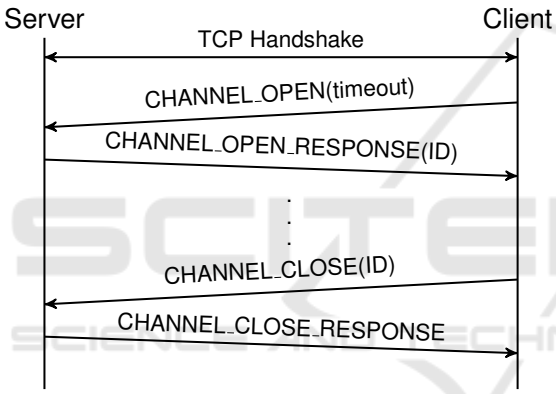

Figure 9: Channel handshake of the example protocol.

time runs out, a response to the request is no longer processed.

4.1.2 Session Layer

As has been described before, a session is built upon a channel but may persist after the channel is closed. The establishment of a session is pictured in Figure 10.

For this, an existing TCP connection and an existing channel is required. First, a session is created using *SESSION_CREATE*. An optional timeout may be sent with this request. The timeout indicates the time-to-live of the session after the last package has been sent. With this, it is possible for the session to persist if a connection is aborted. The persistence of the session timeout fulfills the `SR_TIMEOUT_MS.` requirement. The `SR_TIMEOUT_SM.` requirement is fulfilled as well since there exists additionally the timeout of the TCP connection.

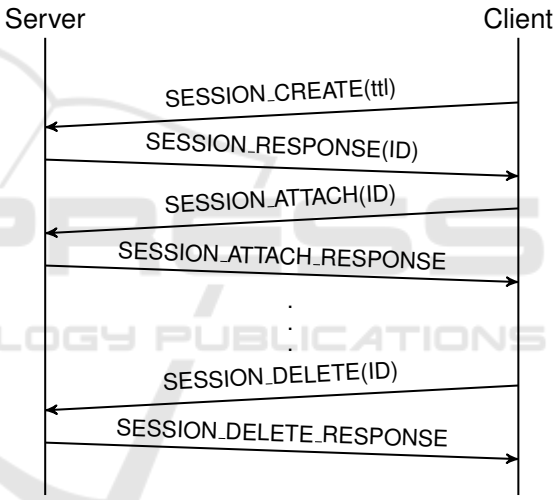

Figure 10: Session handshake of the example protocol.

After the creation of the session, it is necessary to attach it to an existing channel. For this, *SESSION_ATTACH* needs to be requested. This message includes the ID of the session that will be attached to the channel the message is sent over. A session may be deleted using *SESSION_DELETE*.

The session layer defines three services: *make_random*, *get_random* and *send_RT*. These services may only be used if an active session is established. This is only the case if the session is attached to the currently active channel. In short, *make_random* creates a random number and stores it on the server. Similar to the channel service, it provides a *NOACK* bit flag. If this flag is set, no response is sent. The random number may be sent back directly in a response or might be requested

```
TCP
└─IsuProtoHeader
  ├─IsuProtoServices
  └─IsuProtoChannelHeader
    ├─IsuProtoServices
    └─IsuProtoSessionHeader
      └─IsuProtoServices
```

Figure 11: Layering of example protocol packets.

through *get_random*. This decision needs to be sent with the request. *send_RT* starts exchanging RT packets via UDP with a configurable speed.

4.2 Packet Structures

The packet structure of the example protocol follows the layering depicted in Figure 11. An `IsuProtoHeader` can either contain an `IsuProtoService` message or an `IsuProtoChannelHeader`. The same goes for the layers nested deeper with additional headers. The determination which layer follows is done via a type field which fulfills *SR_INFO.*. Each service has a request message and a response message. The services are used to model the protocol behavior.

To fulfill the *PR_DETERM.* requirements, some custom fields were added and included in at least one service packet. These special fields are explained in the following.

The `BitField` is a field with fixed length and a fixed parsing rule. It can have a (fixed) length specified in bits.

The fixed length with a parsing rule switched by another field is modeled implicitly by using the layering in the example protocol. The specified type in the IsuProtoHeader determines whether the following bytes have to be interpreted as `IsuProtoServices` or as `IsuProtoChannelHeader` (see Figure 11).

`FixedLengthVariableRuleFields` contain binary data. The parsing rules that specify how that data is to be parsed and encoded can be sent with the packet, or be supplied separately, because they have been requested beforehand. A `FixedLengthVariableRuleField` needs a parsing and encoding rule, that specifies how the data is parsed and encoded respectively. For the example protocol, any arbitrary Python code is allowed. The length of the data is not specified and the parsing rule needs to decide when to stop parsing.

The `ByteListField` has a fixed parsing rule. The field consists of multiple bytes that comprise a list of bytes. The length of this list is specified in an external field in the packet.

`SwitchedArrays` are arrays that can contain a specified set of other fields that are switched by a type field. The type field determines the type of all the contained fields. The number of fields is determined by a length field.

`DynamicLengthVariableRuleFields` have the same functionality as `FixedLengthVariableRuleFields` except that the length of the data that contains the value is specified by another field, and not fixed.

`VariableLengthLengthFields` are similar to *variable-length vectors with variable-length length encoding fields* from ITS (see Section 3.2). The length of the field in bytes is determined by a sequence of ones at the beginning of the field, followed by a zero. The remaining bits comprise the value of the field.

The `TerminatorStringField` models a byte string that is terminated by a specific byte. The byte that terminates the string is specified in a separate field. Because the string is parsed until the terminator byte is encountered, the field can be categorized as a field whose length depends on its own data.

`VariableRuleFields` contain binary data. The parsing rules that specify how that data is to be parsed and encoded can be sent with the packet, or be supplied separately, because they have been requested beforehand. A `VariableRuleField` needs a parsing and encoding rule, that specify how the data is parsed and encoded respectively. For the example protocol, any arbitrary Python code is allowed. The length of the data is not specified and the parsing rule needs to decide when to stop parsing. It follows, that the length depends on the fields own data, and the parsing rule is variable.

5 CASE STUDY

The example protocol comprises the specification, a prototypical implementation using Python and Scapy as well as several packet captures of the client server communication. It is available online on GitHub.

In order to illustrate the usefulness of the example protocol, the protocol learning use case described in Section 3.1 will be applied in the following.

5.1 Scenario

There exist different tools for network protocol learning. *Netzob* (Bossert and Guihéry, 2012) infers both a *vocabulary* (packet structures) and a *grammar* (protocol behavior) from a given network communication. NEMESYS (Kleber et al., 2018), a newer approach, focuses on inferring packet structures from network messages of binary protocols. All tools accept as input packet captures (PCAP). As output, they provide

different kinds of results which represent the inferred protocol behavior and packet structures.

5.2 Example Protocol Usage

Network communication comprising the example protocol can be generated by starting both a server and a client running the prototypical implementation. Packet captures can be recorded locally on either the server or the client running *tcpdump*.

The listing in Figure 12 presents the example communication of a client which first opens a connection to the server, retrieves 100 random numbers and closes the connection finally. In line 2, the TCP connection is set up. Line 3 creates a channel, line 4 a session. The session id is extracted in line 5 from the response packet which is used for attaching the session in line 6. In the lines 7 and 8, the get_random() service is being used for 100 times. Line 9 deletes the session, line 10 closes the channel and line 11 disconnects from the server.

```
 1  c = Client()
 2  c.connect()
 3  c.create_channel()
 4  p = c.create_session("My Session")
 5  session_id = p[SessionCreateResp].id
 6  c.attach_session(session_id)
 7  for i in range(0, 100):
 8    c.get_random(session_id=session_id)
 9  c.delete_session(session_id)
10  c.close_channel()
11  c.disconnect()
```

Figure 12: Example client source code for connecting to a server using the example protocol.

The resulting PCAP file consists of 218 network packets. While 2 times 100 contain the random number requests and responses, 11 are used to setup the communication and 7 for their termination.

5.3 Discussion

NEMESYS compares the results of the inferred packet structure with the real packet structure extracted via *tshark* from Wireshark dissectors. As long as no Wireshark dissector for the example protocol exists, the needed structures for this comparison have to be written manually. Besides that issue, the prototypical implementation of the example protocol provides a useful basis for training and the development of protocol learning tools dealing with both packet structures and protocol behavior.

6 CONCLUSION

The design and publication of an example protocol is a big advantage for the security testing domain in IACS. Recent work in the creation of tools for dealing with arbitrary network protocols suffered from missing complete knowledge about possibilities for both packet structures and protocol behavior. With the example protocol in mind, tool developers can improve their tools accordingly.

In the case of protocol learning, tools can be improved regarding their ability to recognize packet fields. The variety of length and parsing rule determination described in the analysis is realized in the example protocol prototypical client server implementation.

Future work includes implementing a Wireshark dissector for the example protocol in order to allow for automatic success verification of protocol learning tools. Additionally, the design of a meta model for the description of network protocols which covers all the observed peculiarities in the example protocol is a great challenge. Applying this meta model approach to the example protocol, a machine-readable definition of the example protocol can be given.

Last but not least, packet crafting and processing tools have to be improved or re-implemented by scratch taking into account the results of this research.

ACKNOWLEDGEMENTS

We thank Andreas Fleig for ideas on requirements for the example protocol and Mario Kaufmann for assistance at implementing the example protocol.

This work was supported by the German Federal Ministry of Education and Research within the framework of the project *KASTEL_SKI* in the Competence Center for Applied Security Technology (KASTEL).

REFERENCES

Biondi, P. (2018). Scapy: Packet crafting for python2 and python3. https://scapy.net/. [Online; accessed 2018-12-23].

Bossert, G. and Guihéry, F. (2012). Security evaluation of communication protocols in common criteria. In *Proc of IEEE International Conference on Communications*.

Braden, R. (1989). Rfc-1122: Requirements for internet hosts. *Request for Comments*, pages 356–363.

Duchene, J., Le Guernic, C., Alata, E., Nicomette, V., and Kaâniche, M. (2018). State of the art of network pro-

tocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques*, 14(1):53–68.

ETSI (2015). *ETSI TS 103 097: Intelligent Transport Systems (ITS)*. ETSI, v 1.2.1 edition.

Felderer, M., Büchler, M., Johns, M., Brucker, A. D., Breu, R., and Pretschner, A. (2016). Security testing: A survey. In *Advances in Computers*, volume 101, pages 1–51. Elsevier.

HMS Industrial Networks (2017). Variantenvielfalt bei Kommunikationssystemen. https://www.feldbusse.de/Trends/trends.shtml. [Online; accessed 17-December-2018].

ISO (1994). *ISO/IEC 7498-1:1994 - Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. International Organization for Standardization (ISO), Geneva, Switzerland.

Kleber, S., Kopp, H., and Kargl, F. (2018). NEMESYS: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD. USENIX Association.

Kurose, J. F. and Ross, K. W. (2013). *Computer networking: a top-down approach: international edition*. Pearson Higher Ed.

Lüders, S. (2011). Stuxnet and the impact on accelerator control systems. *Proceedings of ICALEPCS2011, Grenoble, France*, pages 1285–1288.

OPC Foundation (2017a). *OPC Unified Architecture Specification Part 1: Overview and Concepts*. OPC Foundation, version 1.04 edition.

OPC Foundation (2017b). *OPC Unified Architecture Specification Part 3: Address Space Model*. OPC Foundation, version 1.04 edition.

OPC Foundation (2017c). *OPC Unified Architecture Specification Part 6: Mappings*. OPC Foundation, version 1.04 edition.

Peterson, L. L. and Davie, B. S. (2007). *Computer networks: a systems approach*. Elsevier.

Pfrang, S. and Meier, D. (2018). Detecting and preventing replay attacks in industrial automation networks operated with profinet io. *Journal of Computer Virology and Hacking Techniques*, 14(4):253–268.

Pfrang, S., Meier, D., Friedrich, M., and Beyerer, J. (2018). Advancing protocol fuzzing for industrial automation and control systems. *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*, pages 570–580.

Potter, B. and McGraw, G. (2004). Software security testing. *IEEE Security & Privacy*, 2(5):81–85.