ICS Virtual Testbed

Technical Report, FIT VUT

Petr Matoušek, Peter Grofčík

Technical Report no. FIT-TR-2021-02
Faculty of Information Technology, Brno University of Technology

Last modified: June 22, 2021

Contents

Abstract

This paper describes a virtual environment for emulating IEC 104 traffic. Implementation is based on the open framework developed by Peter Maynard, Kieran McLaughlin and Sakir Sezer from Queen's University of Belfast. The framework implements behavior of Human Machine Interface (HMI), Remote Terminal Unit (RTU) and Data Historian devices which are typical components of IEC 104 network.

During our project we extended the framework by additional functionality of HMI and RTU so that we can build a large IEC 104 with more components communicating independently each other. We also implemented several scenarios representing attacks against IEC 104 control protocol. This technical report describes the extended library, configuration of virtual environment and our experiments results.

Chapter 1

Introduction

1.1 Acknowledgement

This work is supported by Brno University of Technology project "Application of AI methods to cyber security and control systems"(2020–2022), no. FIT-S-20-6293.

1.2 About

ICS-TestBed-Framework [1] is a scalable java based framework that consists of three types of virtual devices.

1. HMI (Human-machine interface) – periodically controlling other hosts
2. RTU (Remote Terminal Unit) – represents node for connection between sensors, etc. and HMI
3. Data Historian

The framework itself uses protocol IEC104[2] to communicate between its virtual machines (optionally with physical machines), even throughout the non-virtual network.

The motivation for work with this framework is to create and observe corresponding communication, abnormalities in the common flow of communication or under attacks. Basically for the security analysis of IEC104 communication using virtual devices that correspond to physical hardware.

Chapter 2

Description of the framework packages

The framework consists of three main and two secondary directories:

- Main directories:
    1. *j60870* - The directory containing a structure of source and build files for the main functionality of a framework. Most of the files were not made to be changed in any way, because it may cause a malfunction of the framework, but in some cases, it might be a good way to alter or add some helpful functions for the creation of customized packets or even sequences.
    2. *node* - This directory contains the source and built files that can be used to change the flow of communication mentioned above. A sub-directory *src* contains relevant classes to do so. A sub-directory *target* contains build files as well as a file *node-1.0.jar* that is of executable java type format for framework execution.
    3. *UA-Java* - This directory contains "*Unified Architecture Java Stack and Sample Code to the community*". To my understanding, it was taken from another project repository and can be altered to a user's needs if needed, but will not be necessary for experiments with devices created by this framework.
- Secondary directories:
    1. *attacks* - This directory contains python scripts as well as logs about attack experiments carried by the creators of the framework.
    2. *scripts* - This last directory contains script examples, that can be loaded after the execution of the java executable file to start single virtual devices.

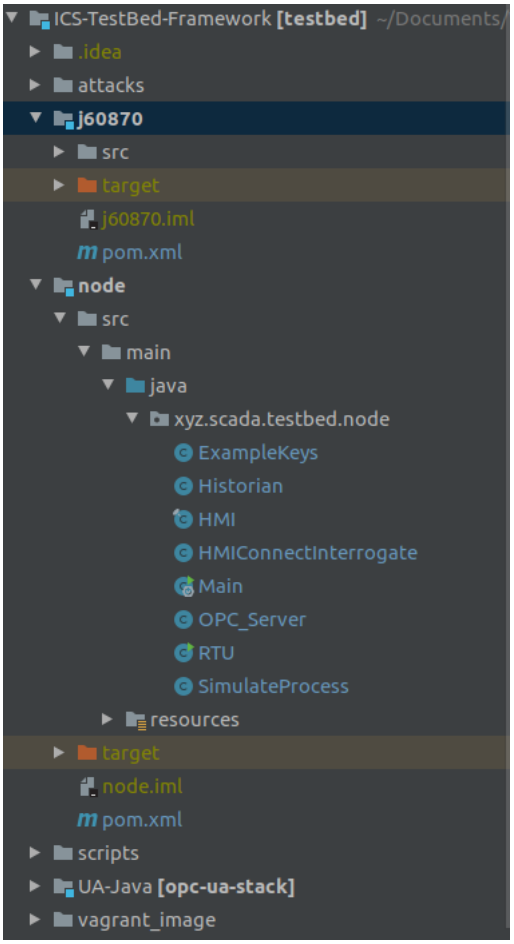

Figure 2.1: Framework structure with relevant classes unrolled

2.1 Running

Each one of the virtual machines needs its shell terminal for deployment using a target node jar created after the successful build (java -jar node/target/node-1.0.jar). As for a default configuration, it is possible to use the default localhost address 127.0.0.1 to deploy more virtual devices that will communicate just over a single physical device. As for the deployment of multiple devices that communicate over physical hardware, there is a need for correct LAN connection with IP address for physical hardware. After that, the virtual machine deployed on it can be started with a correct IP (that is set by commands listed below) that corresponds to physical hardware for the communication to be possible.

At the very beginning, there are commands RTU, HMI that is needed to choose a form of virtual device to be started from the exact terminal. The

default configuration is loaded by framework after a form (HMI, RTU) is chosen.

```
Hosts:    127.0.0.1
IEC104 Common Address: 0
Interval:        10000
IEC104 Port: 2404
```

Figure 2.2: hmi-show command example for default settings

Default settings for the RTU machine are the same for the IP address and port used in communication. The common address is not needed, because it should be declared by the HMI machine in a form of an ASDU number. Interval is as well used only for HMI to determine pause time before the next cycle in communication is invoked.

An example of topology created with this framework might look just like in a picture 2.3. I also used this topology in later steps after my changes in a framework based on captured traffic example from real devices that I used as a model sample of IEC104 communication between RTU and HMI machine

.

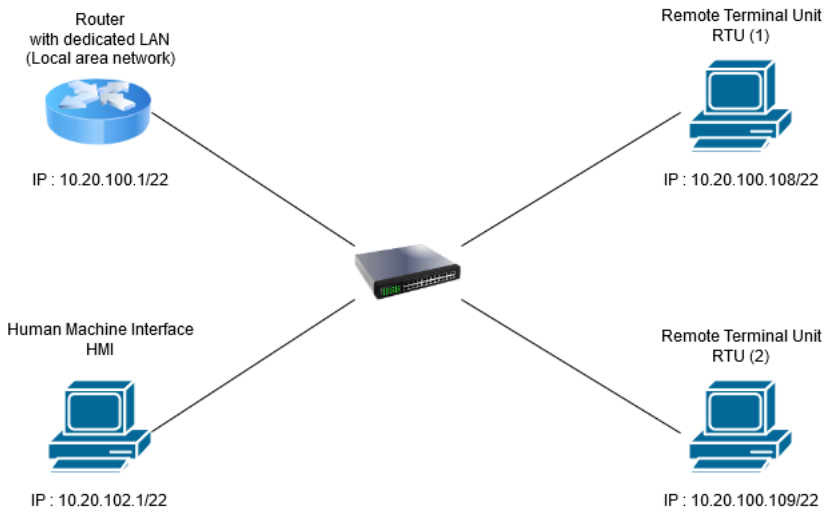

Figure 2.3: Simple example of topology consisting of one HMI and two RTU machines

Possible commands for HMI machine:

- hmi-iec104port [port number] – port number that HMI will connect to
- hmi-interval [number (ms)] – interval/timing for cyclic connection from HMI to other devices
- hmi-common-address [number] – common ASDU address of origination for connection
- remote-hosts [IP address] – IP address/addresses of devices that the HMI will connect to (every single input will override the old one → for more devices at the moment needs an input of all desired IP addresses split by comma)
- hmi-show – shows current HMI configuration

Possible commands for RTU machine:

- rtu-iec104port [port number] – port number on which RTU will expect connection
- rtu-listen [IP address] – IP address on which the RTU will expect connection (as said for connection over physical hardware must correspond with IP address of device of which it will be started on)

For both devices, there is a single command "run" when the configuration is finished. HMI machine must be last to start because it requires listeners on the other side, which means that it will fail as soon as there are none.

Already prepared code consist of four default interrogation command communication:

- Act – beginning of interrogation ($C\_CS\_NA\_1$ - Clock synchronization command)
- ActCon – reply to beginning of interrogation ($C\_CS\_NA\_1$ - Clock synchronization command)
- Inrogen – interrogation command ($M\_ME\_NB\_1$ - Measured value, scaled value)
- ActTerm – termination if interrogation ($C\_IC\_NA\_1$ - (General-) Interrogation command)

This sequence is just a simple example of IEC104 interrogation communication between HMI and RTU device, but it is not possible to change it anyhow without the change of a corresponding machine code explained below. As said, the framework does not consist of any sort of commands that can change the flow of communication between virtual (physical) devices.

Change of flow in communication can be achieved by change on code inside some of the relevant classes created for it, which are explained below.

$C\_CS\_NA\_1$, $M\_ME\_NB\_1$ and $C\_IC\_NA\_1$ stands for ASDU (Application Service Data Unit) types that can be used in communication in the right order. Reference about possible combinations can be found for example in a technical report *Description and analysis of IEC 104 Protocol*[1].

2.2 Relevant classes

To alter the flow of communication itself there are few classes in sub-folder */node/src/main/java/xyz/scada/testbed/node*. The main classes for doing so are HMI, HMIConnectInterrogate, and RTU.

HMI class consists of default settings that we can alter yet again just to change IP address, port, etc. for communication, but also a sleep time that separates a sequence of interrogation commands inside of the cycle.

On the other hand, the HMIConnectInterrogate class contains this so-called cycle, inside of which we can add some interrogation command sequences. With the use of some inner logic even for more types of devices to connect to.

RTU class is mostly one big logical switch that will make an RTU response depending on the Act command. It is possible to consider other aspects of incoming activation, but this should always be the most important one because it separates possible responses based on the IEC104 standard.

For most of possible interrogation commands or responses, there are various classes in a sub-folder */j60870/src/main/java/org/openmuc/j60870*. Based on those and with them, I was able to alter the flow of communication to correspond with one that was captured from real devices. A problem in this state is that it will create the corresponding "static" communication that maps only a small-time on communication with static values to be sent between virtual devices, and it's also impossible to make it work with some predefined physical hardware like that.

[1] Https://www.fit.vut.cz/research/publication-file/11570/TR-IEC104.pdf

Chapter 3

Building virtual devices

3.1 Information gathering

Based on already captured traffic of real devices that communicate using IEC104 it might be possible to automatically create corresponding code for virtual devices by filtering important values such as:

- Cause of transmission (Act, ActCon, Init, etc.)
- TypeID of command ($C\_CS\_NA\_1, M\_ME\_NB\_1$, etc.)
- OA – Address of originate
- IOA – objects of interest for command send with all its different values

This approach will only work if it simulates all the devices because it creates specific communication that will require specific responses from the other side. It will not guarantee that if we use some real devices that it will not fail. It might be possible if we map just an HMI side that will dictate RTU devices (a real device used), but only if we use an exact device that was used to capture from and yet it might still fail, because it's not only getting orders to do something but also obtain some other values from sensors and so, in that case, it will respond with no mapped values.

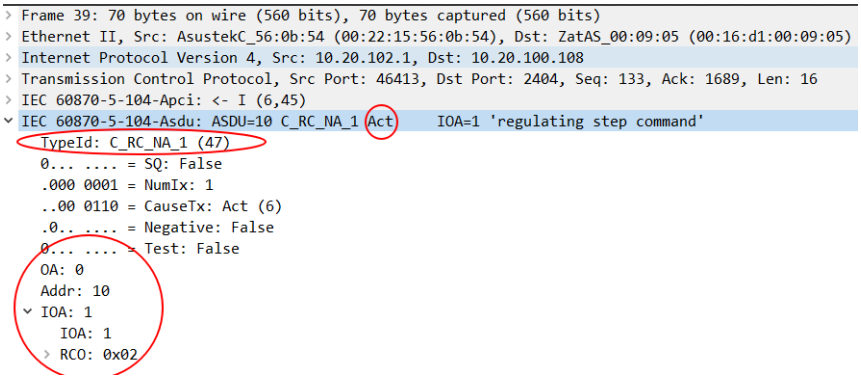

Figure 3.1: Example of captured packet activation from HMI machine

Based on captured traffic we can as a first step determine IP addresses of HMI and RTU in communication, thanks to the type of commands used in it. For example, *Act* command used in figure 3.1 means that this captured packet was sent by the HMI machine because only HMI initiates communication (represented by Act or Init command). Then we can see that it used *OA (ASDU)* of value 10 and TypeID of command $C\_RC\_NA\_1$ as well as all values used in ASDU itself. Also, each ASDU can contain any number of IOA, so we need to parse all of them. These invocations of communication are not conditional and will only be sent in a cycle after a certain time.

On the other hand in figure **??** we can see a reply from the RTU device, generated after HMI invoked communication with TypeID (C_RC_NA_1), which can be seen in *ActCon* reply. Based on that we can determine what kind of reply takes place after what kind of invocation is received from the HMI machine and use that to create a case in class RTU that will correspond to it, also with parsing other information from ASDU as done for HMI. This needs to be done for all ASDU received in a packet as well. *ActTerm* type of command contains the same TypeID as had an invocation so it can be parsed even if it came in a different packet then ASDU with *ActCon* type of command, but *Spont* should and will be only right after *ActTerm* ASDU in the same packet.

```
> Frame 40: 103 bytes on wire (824 bits), 103 bytes captured (824 bits)
> Ethernet II, Src: ZatAS_00:09:05 (00:16:d1:00:09:05), Dst: AsustekC_56:0b:54 (00:22:15:56:0b:54)
> Internet Protocol Version 4, Src: 10.20.100.108, Dst: 10.20.102.1
> Transmission Control Protocol, Src Port: 2404, Dst Port: 46413, Seq: 1689, Ack: 149, Len: 49
> IEC 60870-5-104-Apci: -> I (45,7)
v IEC 60870-5-104-Asdu: ASDU=10 C_RC_NA_1 ActCon  IOA=1 'regulating step command'
    TypeId: C_RC_NA_1 (47)
    0... .... = SQ: False
    .000 0001 = NumIx: 1
    ..00 0111 = CauseTx: ActCon (7)
    .0.. .... = Negative: False
    0... .... = Test: False
    OA: 0
    Addr: 10
  v IOA: 1
      IOA: 1
    > RCO: 0x02
> IEC 60870-5-104-Apci: -> I (46,7)
> IEC 60870-5-104-Asdu: ASDU=10 C_RC_NA_1 ActTerm IOA=1 'regulating step command'
> IEC 60870-5-104-Apci: -> I (47,7)
> IEC 60870-5-104-Asdu: ASDU=10 M_ST_NA_1 Spont   IOA=1 'step position information'
```

Figure 3.2: Example of captured packet response from one RTU

3.2 Creating message itself

Each device does not need to create a whole packet as a response, because packets are automatically generated on ASDUs that were sent to function *send* in class *Connection*. New ASDU can be created as shown in figure 3.2, where:

- TypeId (enum) - stands for TypeID of command
- CauseOfTransmission (enum) - stands for Cause of transmission
- commonAddress (number) - stands for ASDU number
- informationObject (class) - represents IOA with it's address and information elements (figure 3.2)

```
ASdu asdu = new ASdu(
        TypeId.C_IC_NA_1,  isSequenceOfElements: false,
        CauseOfTransmission.ACTIVATION_TERMINATION,  test: false,  negativeConfirm: false,
         originatorAddress: 0,  commonAddress: 10,
        new InformationObject[] {
            new InformationObject( informationObjectAddress: 0,
                new InformationElement[][] { {
                    new IeCauseOfInitialization( value: 0,  initAfterParameterChange: false) } }) }
);
connection.send(asdu);
```

Figure 3.3: Manually created ASDU

```
▼ IOA: 13
    IOA: 13
  ▼ SCO: 0x01
      .... ...1 = ON/OFF: On
      .000 00.. = QU: No pulse defined (0)
      0... .... = S/E: Execute
```

Figure 3.4: IOA and information elements

In some cases (mostly for HMI requests), it's possible to use functions in class *Connection*, that creates and sends ASDUs based on their input values, but all of them got predefined usage. As shown in figure 3.2, it might not be possible to change the information object address (or anything else in others). Using these functions defined in the framework might lessen lines of code if the simulation is fully static, but creating ASDUs manually is more efficient if the same ASDU needs more IOAs or multiple IOAs requests the same ASDU type.

```
public void interrogation(int commonAddress, CauseOfTransmission cot, IeQualifierOfInterrogation qualifier)
        throws IOException {
    ASdu aSdu = new ASdu(TypeId.C_IC_NA_1, isSequenceOfElements: false, cot, test: false, negativeConfirm: false, originatorAddress, commonAddress,
            new InformationObject[] { new InformationObject( informationObjectAddress: 0, new InformationElement[][] { { qualifier } }) });
    send(aSdu);
}
```

Figure 3.5: Example of function that creates and sends ASDU

3.3 Initialization stage

Each communication between RTU and HMI starts with an initial stage (C_IC_NA_1 - interrogation command). As said above, communication starts with the HMI device as initiator issuing interrogation command, on which RTU sends a corresponding response with information about all its known assets (figure 3.3).

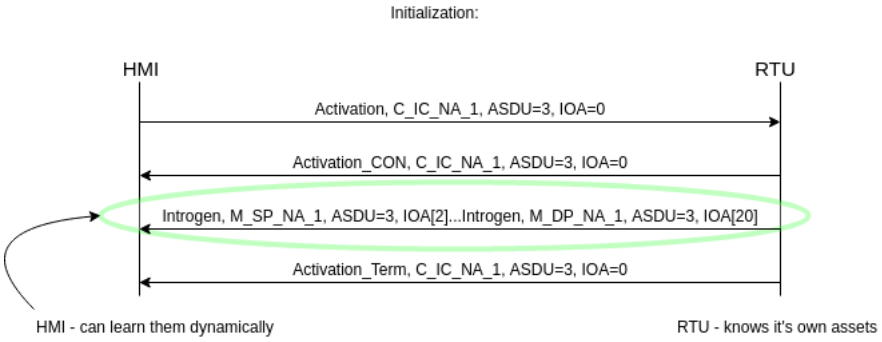

Figure 3.6: Initialization stage of communication

The framework itself does not consist of any way to store assets known

by RTU, so in this stage, I used simple types of arrays that represented them as shown in figure 3.3. These needs to be created in function *newASDU* in class *RTU*, so that our virtual RTU can create responses to *known_ requests* as well as process unknown data described later in section 3.6.

```
Integer[] knownASDU = {10};
CauseOfTransmission[] knownCauses = {CauseOfTransmission.ACTIVATION};
TypeId[] knownTypesASDU_requests = {TypeId.C_IC_NA_1,TypeId.C_SC_NA_1,TypeId.C_DC_NA_1,TypeId.C_RC_NA_1,
                                    TypeId.C_BO_NA_1,TypeId.C_SE_NA_1,TypeId.C_SE_NB_1,TypeId.C_SE_NC_1};
Integer[][] knownIOA = {
        {1,2,3,4},
        {11,12,13,14},
        {0}
};

TypeId[] NormalTypesOfIOA = { TypeId.M_SP_NA_1, TypeId.M_DP_NA_1, TypeId.M_ST_NA_1, TypeId.M_BO_NA_1, TypeId.M_ME_NA_1,
                              TypeId.M_ME_NB_1, TypeId.M_ME_NC_1, TypeId.M_SP_TB_1, TypeId.M_DP_TB_1, TypeId.M_ST_TB_1,
                              TypeId.M_BO_TB_1, TypeId.M_ME_TD_1, TypeId.M_ME_TE_1, TypeId.M_ME_TF_1};
```

Figure 3.7: Simple data usage

As said above, HMI devices can also have statically created arrays that correspond to assets of RTU, but in this case, HMI could not even be used with multiple RTUs of the same type that manages different information objects or corresponding requests. That led me to the creation of dynamic learning on the HMI side. This is not needed for normal communication between RTU and HMI itself, but HMI must be able to correctly process unknown data as well as for us to study anomalies of simple attacks in communication.

Dynamic learning itself is done only in the initialization stage after the first initiation *interrogation* command by the HMI device and ends when HMI obtain *Activation_ Term* command. Learning is done in a function *newASDU* in class *HMIConnectInterrogate*. Each ASDU is obtained by the HMI device until the termination command is parsed by the function shown in figure 3.3. All important values (IOA, COT, TID) are simply added to array-lists and can be used to check incoming ASDUs in cyclic communication between RTU and HMI.

```
private void addAsduInfo(ASdu aSdu){
    addToKnown_add(aSdu.getCommonAddress(), StoredInfo.ASDU);
    addToKnown_add(aSdu.getCauseOfTransmission(), StoredInfo.COT);
    addToKnown_add(aSdu.getTypeIdentification(), StoredInfo.TID);
    InformationObject[] objects = aSdu.getInformationObjects();
    ArrayList<Integer> local_IOA = new ArrayList<~>();
    ArrayList<Integer[]> loc = new ArrayList<~>();
    for (InformationObject object : objects){
        local_IOA.add(object.getInformationObjectAddress());
    }
    addToKnown_add(local_IOA.toArray(new Integer[local_IOA.size()]), StoredInfo.IOA);
}
```

Figure 3.8: Dynamic learning

3.4 HMI requests

As initiator HMI creates several requests for RTU after the initial stage as shown in figure 3.4. These requests ask an RTU to change corresponding values for IOAs. This sequence corresponds to the initial configuration by a real person behind the HMI device. In a real device scenario, these commands should be carried by the RTU device on its own connected devices that are represented by IOA (objects of interest - sensors for example).

```
41 10.029175444  192.168.1.10   192.168.1.12   104asdu   82 <- I (2,31) ASDU=10 C_SC_NA_1 Act    IOA=2
47 10.033351119  192.168.1.10   192.168.1.12   104apci   72 <- S (34)
51 20.029494077  192.168.1.10   192.168.1.12   104asdu   82 <- I (3,34) ASDU=10 C_SC_NA_1 Act    IOA=13
57 20.039124781  192.168.1.10   192.168.1.12   104apci   72 <- S (37)
59 30.031989301  192.168.1.10   192.168.1.12   104asdu   82 <- I (4,37) ASDU=10 C_DC_NA_1 Act    IOA=1
70 30.081254621  192.168.1.10   192.168.1.12   104apci   72 <- S (40)
73 40.032341905  192.168.1.10   192.168.1.12   104asdu   82 <- I (5,40) ASDU=10 C_DC_NA_1 Act    IOA=14
```

Figure 3.9: Request sequence

This sequence of commands needs to be created in main function *run* in class *HMIConnectInterrogate*, same as first initialization command. Each of these commands needs to be *"self-learned"* by HMI because RTU will respond with ACT_CON command, which is just a copy of the initiated command with a different cause of transmission. These were not learned in the initialization stage. These can not be carried from RTU when it's stating it's known assets because these are request types of commands that should be used only by HMI with an exception for ACT_CON command that is carried as a confirmation of obtained request. This additional learning can be done by the same function as stated in section 3.3, which should be called after each generated ASDU request.

RTU device is represented by a finite automaton, that can provide information about its assets any time it's asked for. Each response is created based on obtained initiation and should consist of at least:

- Activation confirmation command,
- Activation termination command.

Both of these commands are copies of *Activation* command carried by HMI with different causes of transmission. Activation confirmation command states that corresponding Activation command was obtained by RTU (and it's a valid one 3.6). The activation termination command states that it's been carried on RTU assets, but this is carried by our RTU device immediately because the framework itself does not consist of any data structure for RTU device assets.

In some studied scenarios, these commands were followed by *Spontaneous command*, which stated the actual value of IOA from the last request with the corresponding type of command (same type as in initialization state for appropriate IOA).

3.5 Cyclic communication

In all studied *pcap* files, the cyclic stage of communication mainly stands for HMI issuing interrogation command (*C_IC_NA_1*) asking for all RTU assets over a specified time. RTU should respond the same way as it did in the initialization state, but with up-to-date values obtained from its assets or configured during HMI requests before. As I said before, this is not possible with the framework itself. RTU response for this cycle can be made with correctly changed values of assets, but only using them statically in code, because of a lack of data storage for them. Still, this way is sufficient for the simulation of IEC104 messages between devices.

The creation of data structure for devices is not necessary for simulations itself but might be good in later simulations of attacks. For example, issuing a MITM attack to change some preset values in assets of RTU can be done, but we would be only able to see its consequences in changed incoming single ASDU on RTU device because RTU can not change values in communication with HMI device without data structure that saves actual values from communication. (this stage is still in progress and will be added in chapter 4)

3.6 Processing of unknown data

The framework itself did not consist of any useful functions for processing data that are not corresponding in the scenario. Taken from IEC104 specifications: Each device should be able to check incoming messages for unknown data obtained. This was the reason for creating arrays of data in the first place with addition to class *Connection*. I've created a simple function *checkCorrect* that will check an incoming ASDU against defined arrays created during the initialization stage. This function needs to be called in function *newASDU* for each device (HMI and RTU) with its arrays of assets (static for RTU, learned for HMI).

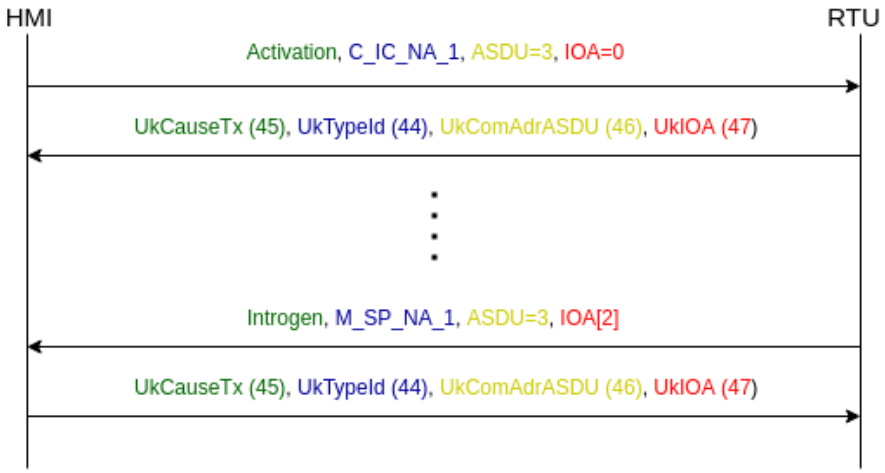

Figure 3.10: Unknown data responses

The function itself returns true if the ASDU request is a valid one, but if it's not it also sends a message corresponding to an unknown type of data obtained based on IEC104 specification:

- UkCauseTx(45) - for unknown cause of transmission (Activation)
- UkTypeid(44) - for unknown type of command issued (C_IC_NA_1)
- UkComAdrASDU(46) - for unknown ASDU value (ASDU=3)
- UkIOA(47) - for unknown Information Object Address (IOA=0)

Chapter 4

Data structure for virtual devices

The framework itself does not consist of any way to store data from various communications. HMI or RTU device can send ASDUs over a network, but all values are expected to be statically included in code commands. This means that if a device was attacked and an attack was successful (eg. MITM attack changed some values), these changes will not be seen in oncoming cyclic communication. This fact led to the creation of a data structure for Information Objects (IOA) so that each device can store either its known assets (RTU) or assets provided by another side of the connection (HMI).

4.1 Data package description

Figure 4.1: Data package classes

Classes shown on figure 4.1 can be divided to two categories. Class *IOA* and *dbWork* are bearing and used to create database of resources in a form of wrapped objects. Other classes, such as *singlePoint*, *doublePoint*, etc. store information about specific values for specific type of command for specified IOA.

4.2 Type of command classes

Each of these classes stores specific information/values for each type of command used in communication and requires enum type *IOA.StoredInfo* (type of timestamp used if any) in the constructor. Based on timestamp enum a response type of command is chosen for the generation of ASDU replies in communication.

All these classes contain two main functions that are meant to be called in during communication itself. Function *singlePointElement* (doublePointElement, stepPositionElement, etc.) generates information element array for specific type of command reply. This helps to generate responses as it's not needed to obtain any stored information manually, but this will generate a whole information array that is used in the creation of ASDU inside of a framework.

Second function *decodeMessage* (same name for each class) has an opposite effect. After we decide what type of command request RTU obtained then *decodeMessage* function for a specific type of command class should be called to decode the information stored in obtained ASDU and store it inside our pseudo DB.

As an example of this process figure 4.2 shows requests obtained on RTU device. From that request, we can see a request with the type of command of Single Point requested on IOA 2. A value used and stored for this type is the only qualifier shown in the figure and we can simply call *decodeMessage* on *singlePoint* class to decode obtained ASDU and store qualifier. For each class, a decodeMessage function is designed to decode and store values from ASDU when called properly (type of command from request must correspond to the proper type of command class on which a *decodeMessage* is called) and it decodes and stores all important values for a specified type.

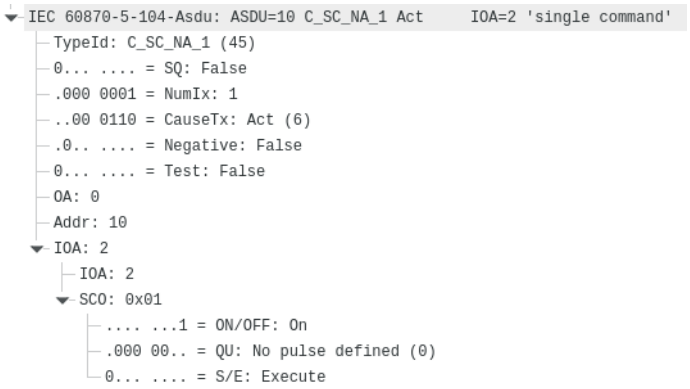

Figure 4.2: Obtained Single point request

4.3 IOA class

Class *IOA* is used to create the information object itself. Its constructor expects a simple number that represents the IOA object inside packets of communication as well as a number representing ASDU. Also, there are specified *Enum* that states for an IOA that uses (or does not use) any format of timestamp.

Each IOA object also wraps all possible objects for each implemented type of command class(singlePoint, doublePoint, etc.) as well as functions to define them when needed. This means that each created IOA object is able to store data for each implemented type of command class, that is needed in cyclic communication. The class itself does not have any specific functions, because it's mainly used to connect all types of command classes with IOA numbers and wrap them under created pseudo DB class dbWork.

4.4 Class dbWork

The class itself serves as a wrapper for other classes from created package and it represents created pseudo DB for values, known information objects, and possible (implemented) responses. In the constructor, it requires *ArrayList* form of IOA objects that together represent known assets of RTU, but can also be empty and each IOA can be added later so that even HMI can have this sort of DB and add them dynamically in the learning phase of communication and *Connection* object that is used inside of a framework so that it can automatically generate responses from its functions.

Once an object of this class is created with all information objects in it, then its main function *rewriteSend* can be used to communicate on a side of RTU device and so it doesn't require any manual creation of responses as long as the communication itself contains only implemented types of command requests and responses. Function contains simple switch that reacts based on incoming type of command from obtained ASDU and separates three types of commands and calls appropriate functions afterwards:

- Interrogation ($C\_IC\_NA\_1$) - cyclic communication invoking the command, called by HMI to obtain the current state of all assets
- Known requests - requests that have been implemented in the data structure and can be searched for and processed if possible (any newly implemented types needs to be added here)
- Unknown requests - simple unknown typeID ASDU is generated as a response

Up to this date there are eight types of requests implemented and so possible to use (created based on communication examples):

- ($C\_IC\_NA\_1$) - interrogation,
- ($C\_SC\_NA\_1$) - single command,
- ($C\_DC\_NA\_1$) - double command,
- ($C\_RC\_NA\_1$) - regulating step command,
- ($C\_BO\_NA\_1$) - bitstring of 32 bits,
- ($C\_SE\_NA\_1$) - set point command (normalized value),
- ($C\_SE\_NB\_1$) - set point command (scaled value),
- ($C\_SE\_NC\_1$) - set point command (short floating point value).

Chapter 5

Man in the middle attack

For this attack, I was forced to create plugins for Ettercap software because it proved to be impossible to use just Ettrcap filters for IEC104 communication. At first, I tried using Ettercap implementation[1] that is linked with the framework itself, but I was not successful with this either. For some reason, there were many issues with dependencies and I was not able to correct them, but the plugin they created inside their implementation proved to be useful as I was able to create my plugins based on structures for IEC104 packets that they have used.

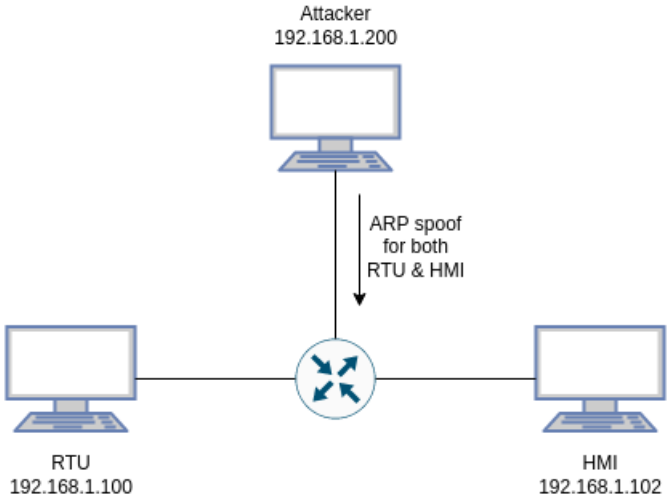

Figure 5.1: MITM attack network example

For this type of attack, I created a simple connection between one RTU and one HMI devices connected via wi-fi router with addresses shown in figure 5.1, and I generated three scenarios of an attacker changing sniffed

[1] https://github.com/PMaynard/ettercap-104-mitm

packets to show what kind of outcome can these changes have on simulated devices (change of):

- vital attribute in communication of HMI request (can be detected on simulated device, change of cause of transmission),
- value (IOA attribute) inside a response generated by RTU device,
- value (IOA attribute) inside a request generated by HMI device.

For each of these scenarios, HMI and RTU device (physical) was set to capture its IEC104 traffic that is captured in files:

- HMI_MITM.pcap
- RTU_MITM.pcap

5.1 Ettercap plugins, brief info

Ettercap plugin is a parser for packets spoofed by Ettercap. Each plugin needs to consist from at least *plugin_load* function and *plugin_ops* structure. Function *plugin_load* is called by Ettercap on plugin load to register created a plugin with its linked structure. Structure *plugin_ops* consists of information strings that describe plugin (name, what it's used for, basically anything author want's to state), but it also need a *init* and *final* function pointer. Anything else inside of a plugin has no other restrictions and can be created to a programmer's desire to parse, change or inject new packets based on packet obtained while spoofing desired communication.

```
struct apci_header {
  u_char start;
  u_int8 length;
  u_int8 control_1;
  u_int8 control_2;
  u_int8 control_3;
  u_int8 control_4;
};

struct asdu_header {
  u_char type_id;
  // Structure Qualifier
  u_char num_objects : 7;
  u_char sq : 1;
  // Cause of transmission
  u_char COT: 6;
  u_char PN : 1;
  u_char T : 1;
  #ifdef ADDR_TWO_OCTECTS
    short originator_addr;
  #else
    u_char originator_addr;
  #endif
  u_int IOA : 16; //8, 16 or 24
  u_char spacer;
  // Specifically for a SIQ information element
  u_char spi : 1;
  u_char blank : 3;
  u_char bl : 1;
  u_char sb : 1;
  u_char nt : 1;
  u_char iv : 1;
};
```

Figure 5.2: IEC104 plugins structure

For plugins used in later described attacks, I've used structures used in already mentioned Ettercap implementation[2] shown in figure 5.1. These structures can be simply used as retype options for a packet in the format received from Ettercap and later based on the scenario used to change desired values inside of the spoofed packet before it gets forwarded.

5.2 Vital attribute scenario

This scenario is based on a change of vital attribute in a transmitted message as an example of a MITM attack that should be detected by the devices themselves if the vital attribute has been changed to generate incorrect/unexpected value in the communication sequence. If the change by an attacker is done properly then it may not be detected at all, but in this example, I am trying to show a change that should end up being detected by the other side of communication and end up as described in section 3.6.

Attack itself can be conducted on messages from each side of communication. For this example, I chose a message generated on the HMI device that is creating cyclic requests to obtain an actual state of assets of the RTU device (figure 5.2).

[2]https://github.com/PMaynard/ettercap-104-mitm

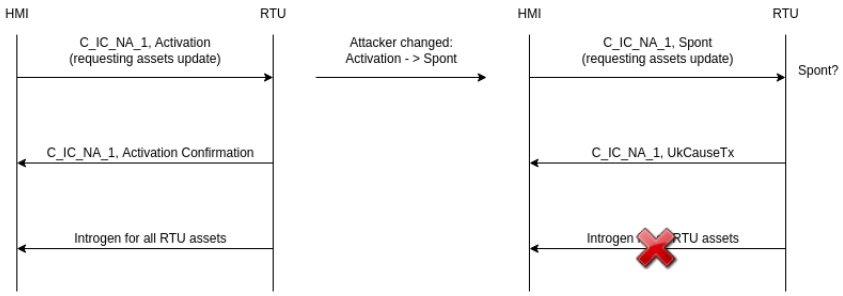

Figure 5.3: Vital attribute change communication example

In this example, I as an attacker changed the cause of transmission from Activation to Spontaneous in a packet sent by an HMI device. The packet shown in figure 5.2 is the one sent by the HMI device as a correct cyclic request sent to the RTU device. Packet in figure 5.2 is a malicious packet injected by an attacker that was received by the RTU device. This malicious packet was marked as incorrect and RTU responded with *UkCauseTx(45)* as shown in figure 5.2.

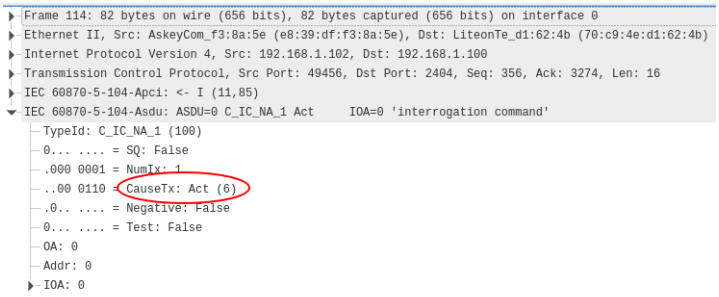

Figure 5.4: HMI generated packet (vital scenario)

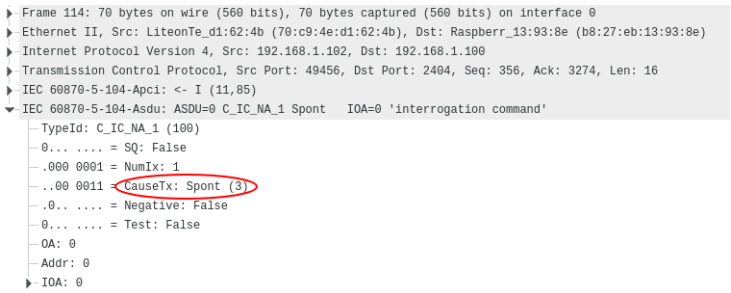

Figure 5.5: RTU received packet (vital scenario)

```
114 110.162022708 192.168.1.102  192.168.1.100  104asdu  70 <- I (11,85) ASDU=0 C_IC_NA_1 Spont   IOA=0
116 110.162849553 192.168.1.100  192.168.1.102  104asdu  82 -> I (85,12) ASDU=0 C_IC_NA_1 UkCauseTx_NEGA IOA=0
```

Figure 5.6: RTU response (vital scenario)

Shown MITM attack was successful, but not a very discrete one. Attack ended up prevented HMI device from getting requested a cyclic update on assets owned by RTU device, but only as long as the attack itself is active and as shown in the above figures, the outcome is an obvious major change inside a cyclic communication that can be easily detected on both devices, unless attacker changes also a response from RTU device.

5.3 Response value change scenario

This scenario is based on change of values inside IOA object in response sent by RTU device. Attack itself is aiming on cyclic communication as it did the previous scenario, but this time it aims on response generated from RTU device. In this scenario attacker is not changing any attribute vital to communication itself, but aims to change a single value inside IOA as shown in figure 5.3.

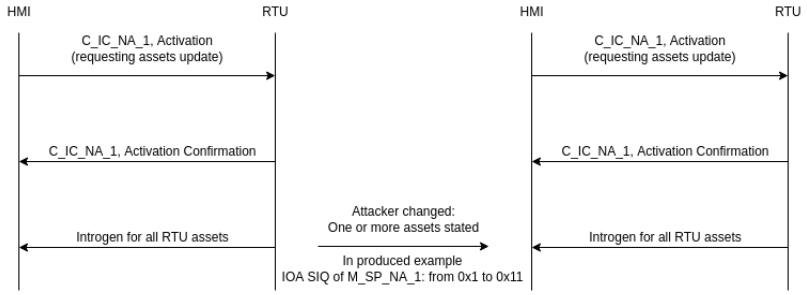

Figure 5.7: Change inside RTU response example

In this example, I chose to change the value of element SIQ for IOA with index 1 under TypeID of command *M_SP_NA_1*. Attack itself modifies only responses sent by RTU device and does not disrupt any other communication, because there is no major change to any vital attributes. The packet shown in figure 5.3 is generated and captured on the RTU device right after its creation. Packet in figure 5.3 is supposed to be the same packet (during not disrupted communication), but it has been changed during delivery.

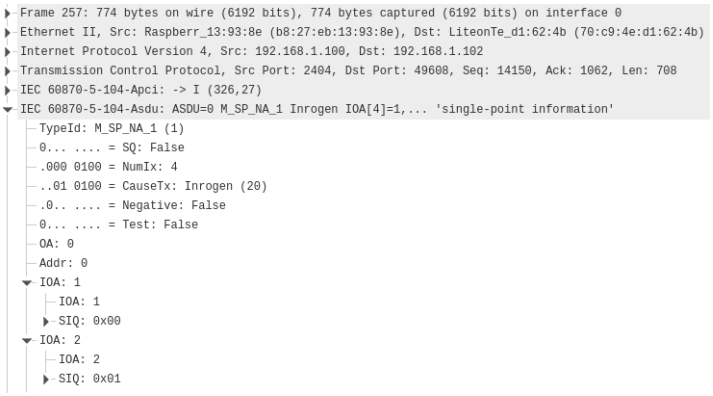

Figure 5.8: RTU generated response

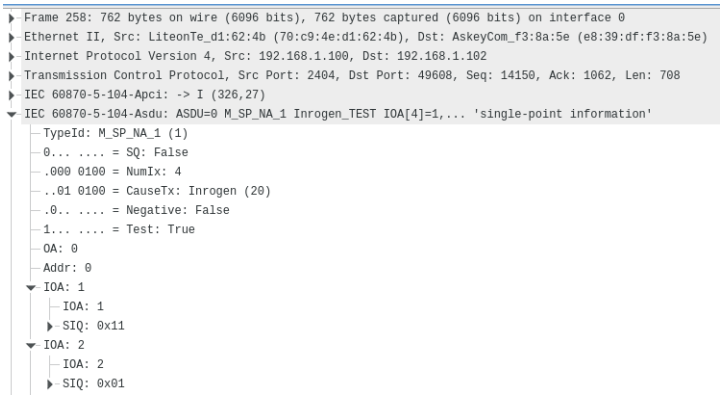

Figure 5.9: HMI received response

Attack itself was successful and is a bit more discrete than the attack on vital attributes, but its effects are taking place only as long as the attack is active. Meaning that the changed value of the SIQ element is present only as long as the attacker is changing it whenever he can spoof a packet containing update from the RTU device and its discretion is valid only as long as no one tries to compare values present on RTU and HMI device.

5.4 Request value change scenario

This scenario is exploiting the database of assets on the RTU device and actual requests of the HMI device to change any values inside of specified assets. It changes the only request of changing specified values sent by HMI device as shown in figure 5.4. This scenario is possible due to data structure implementation described in chapter 4, because without it framework used only static commands to create communication, but with it, it's possible to

simulate a MITM attack that changes values of assets on RTU device.

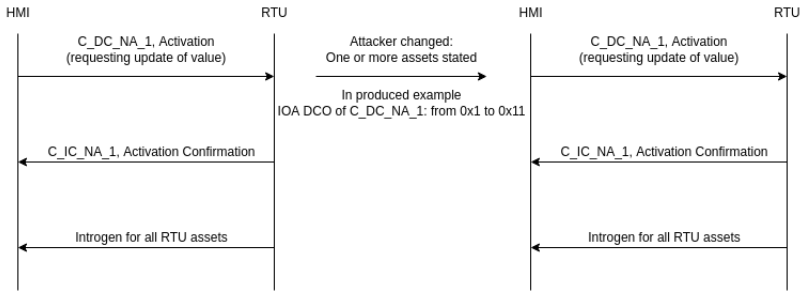

Figure 5.10: Change inside HMI request example

In this example, I chose to change the value of element DCO for IOA with index 1 under TypeID of command *C_DC_NA_1*. This attack aims to change the value requested by the HMI device shown in figure 5.4. The corresponding packet obtained by the RTU device was changed as shown in figure 5.4. This information made RTU change its asset value to the one required by the attacker without triggering any negative response by the RTU device because the request itself was correct.

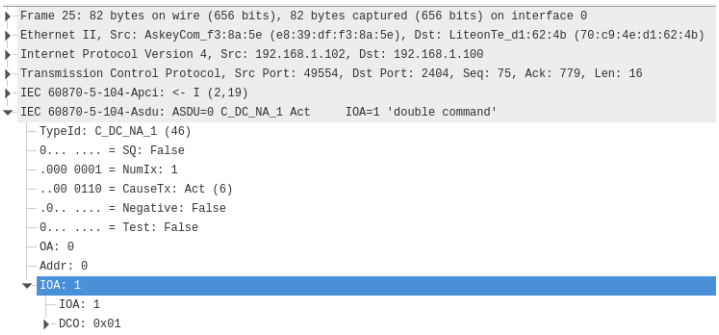

Figure 5.11: HMI generated packet

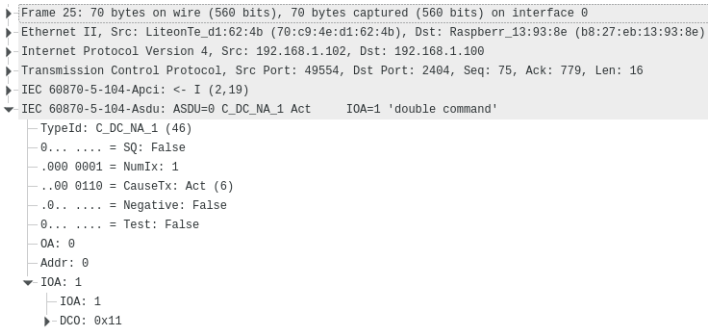

Figure 5.12: RTU received packet

Consequences created by this attack are ongoing even after it ends. HMI device is provided with malicious value each time it requests new updates as shown in figure 5.4 unless a new request to change is sent by the HMI device. This attack requires changing only one transmitted packet which makes it even more discrete as to generate the same consequences as in the second scenario there is no need for the attacker to be constantly spoofing connection between the two devices.

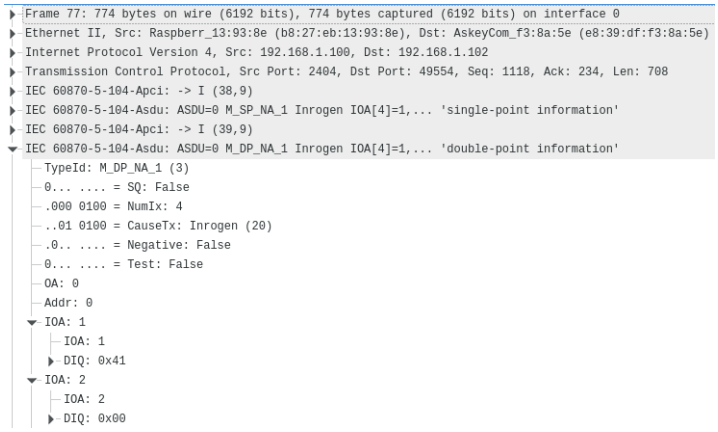

Figure 5.13: RTU cyclic responses (with malicious value)

Even tho constant spoofing is not necessary for this scenario, it still might be useful, because it would make the HMI device to be unable to do any changes to specified assets via requests as long as the attacker keeps changing them to his desired value.

Bibliography

[1] P. Maynard, K. McLaughlin, and S. Sezer, "An open framework for deploying experimental scada testbed networks," 2018.

[2] "INTERNATIONAL STANDARD IEC60870-5-104," tech. rep., Jun 2006. [Online; 25.03.2020].

Appendix A

Installation of framework (based on Github repository[1])

1. git clone –recurse-submodules Https://github.com/PMaynard/ICS-TestBed-Framework.git
2. $JAVA\_HOME$ variable is needed to be openjdk 1.8
   - sudo apt install openjdk-8-jdk maven
   - export $JAVA\_HOME = /usr/lib/jvm/jre{-}1.8.0{-}openjdk.x86\_64/$
3. add three dependencies to a pom.xml file in a root directory, because in a development state they were propably included in maven standard libraries, but got depreciated overtime
   - ```
     <dependency>
         <groupId>com.sun.xml.bind</groupId>
         <artifactId>jaxb-core</artifactId>
         <version>2.2.11</version>
     </dependency>
     ```
   - ```
     <dependency>
         <groupId>com.sun.xml.bind</groupId>
         <artifactId>jaxb-impl</artifactId>
         <version>2.2.11</version>
     </dependency>
     ```
   - ```
     <dependency>
         <groupId>javax.xml.bind</groupId>
         <artifactId>jaxb-api</artifactId>
         <version>2.2.11</version>
     </dependency>
     ```

[1] Https://github.com/PMaynard/ICS-TestBed-Framework
     ```

4. mvn package -DskipTests (-DskipTests just skips some internal framework tests that end up failing in a wrong timezone)

5. Firewall - some problems migH occur with a firewall blocking virtual machines and might need to be disabled

Appendix B

Installation of Ettercap with created plugins

Command to easily install required dependencies on debian or debian based distro:

- apt-get install build-essential debhelper bison check cmake flex groff libbsd-dev libcurl4-openssl-dev libgeoip-dev libgtk-3-dev libltdl-dev libluajit-5.1-dev libncurses5-dev libnet1-dev libpcap-dev libpcre3-dev libssl-dev

B.1 Required programs

- c compiler
- flex - lex-compatible parser generator for *.l files
- bison - yacc-compatible parser generator for *.y files
- cmake

B.2 Required libraries

- libpcap >= 0.8.1
- libpcap >= 0.8.1
- libnet >= 1.1.2.1 (>= 1.1.5 for IPv6 support)
- openssl >= 0.9.7
- libpthread
- zlib

- libgeoip
- CMake 2.8
- Curl >= 7.26.0 to build SSLStrip plugin

B.3 Installation

1. git clone https://github.com/Ettercap/ettercap.git
2. cd ettercap
3. copy desired (created) plugins to plug-ins folder
4. mkdir build
5. cmake ..
6. sudo make install

Ettercap application should appear in system with pre-loaded basic as well as inserted plugins once the installation is done. Then they can be used either via Ettercap GUI environment or from console using in code specified names to be linked to Ettercap (names also appear in GUI).

For example, command:

- *sudo ettercap -T -q -i wlp3s0 -P COT_change_104 -M ARP /192.168.1.100/ /192.168.1.102/"*

starts MITM attack between two hosts in LAN with a usage of plugin *COT_change_104*.

Appendix C

Captured PCAP files

File name	Packet c.	IEC 104 packet c.	Length of communication	No. devices
IEC104	135298	67828	46 h	2
Scada_To_Sub	60126	30838	46 h	2
HMI_IEC104&Sub	195424	98666	46 h	3
HMI_MITM	258593	129943	75 h	2
RTU_MITM	258582	129958	75 h	2

Files *IEC104*, *Scada_To_Sub* and *HMI_IEC104&Sub* were created in earlier stages using only framework capabilities. Communication captured inside them is from topology shown in figure 2.3. File *IEC104* was captured on first RTU device that was programmed to communicate based on provided *iec104* file. The other file *Scada_To_Sub* was captured on the second RTU device based on communication from provided *SCADA_to_substation_normal* file. The last one is the file captured on an HMI device that was an initiator for both RTU devices. Communication in these files is standard with no interruptions as an example of longer IEC104 communication between created virtual devices.

C.1 MITM attacks

Next table coresponds to files *HMI_MITM* and *RTU_MITM*. Table shows occurances of attacks based on MITM scenarios described in chapter 5 and captured on topology shown in figure 5.1. Each of them was created with a predefined filter to capture only traffic over port 2404 to filter out any unnecessary traffic related to a physical device running virtualized HMI or RTU device.

Scenario	Start		End		Duration
	Packet	Time	Packet	Time	
Vital attribute	34188	35632 s	35466	39742 s	68.5 min
Response value	63182	68432 s	67828	73373 s	82.35 min
Request value	201908	212597 s	208587	219547 s	115.83 min

Used values in the table correspond to the HMI_MITM file but are almost the same for the RTU_MITM file they both were started just with a difference of a couple of seconds on opposite HMI/RTU devices.

Request value scenario is a bit different in attack consequences as it requires specific HMI request that occurs in communication only once an hour (C_DC_NA_1). The attack happened to be effective 42 minutes after execution of attack at time 215167 on packet 204452 and lasts until time 222787 on packet 211827 (54 minutes after the end of attack). There are only two packets directly affected during the ongoing attack (packet no. 204452 and 208154), but the effects of those packets can be seen in a window of 127 minutes after packet 204452 in cyclic updates in each *C_IC_NA_1 Introgen* responses by RTU device.