UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Simulating Industrial Control Systems Using Mininet

Dissertation presented by
Bertrand MASSET , Olivier TABURIAUX

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Ramin SADRE

Reader(s)
Gorby KABASELE, Yoann KLEIN , Ramin SADRE

Academic year 2017-2018

Abstract

Industrial Control Systems (ICS) are responsible for controlling, monitoring and managing industrial productions. The critical nature of ICS makes it the target of numerous attacks, thus its security must be well established but this is not always the case. The research on this field are difficult to perform because the devices are expensive and the results are not shareable as they could reveal sensible information about the system used. In this Master Thesis we created an ICS simulator that can support open research with fully virtualized testbeds. The solution uses Mininet to emulate SDN networks with POX as the SDN controller. We modified and used OpenPLC to play the role of PLC and MTU both communicating together using the Modbus/TCP protocol. Furthermore, we used PyScada as a SCADA system and CloudFirewall to act as firewall. In addition we modified the Mininet animator called MiniNAM to support the visualization of our ICS components. Two different ways are proposed to generate ICS examples, one visual by using MiniEdit that we adapted and the other automated by giving parameters to a Python script. Finally we tested attacks that can be encountered in real systems against a running ICS simulation and explained the possible countermeasures.

Keywords: ICS, SIMULATION, NETWORK, PLC, MTU, SCADA, MODBUS, SDN, MININET, SECURITY

Contents

Chapter 1

Introduction

1.1 Introduction to ICS

Industrial Control Systems (ICSs) are responsible for controlling, monitoring and managing industrial productions. Those productions can represent any physical processes occurring in sectors like oil, gas, electric power and many more. With the automation and computerization in today's economy, we can generalize the presence of ICSs in any industrial sectors. ICSs are composed of various systems and the most typical components are the Programmable Logic Controllers (PLCs), small I/O devices that link the production processes and the ICS. They make possible the automation of the industrial equipment which puts them as critical components. A Supervisory Control and Data Acquisition (SCADA) is a well known system in ICS environment that enables a high-level supervision of the industrial process for the operators. Human Machine Interfaces (HMIs) provide a user interface to enable interactions between operators and machines.

As this is only an introduction to ICSs and many more devices exist, we dedicated a chapter for more details.

1.2 Security Issues in ICS

Considering the critical nature of ICSs, their security aspects have to be well established, studied and assessed. This is of course not always the case, this would be of no interest otherwise.

Some examples are presented to get an overview of the current situation. From a 2015 *Dell*'s report [14], the number of reported attacks doubled within a year with more than 600 000 attacks. This is only a lower bound as attacks are only required to be reported in case of involvement in personal information. It is also known that a lot of attacks are not detected as from a 2017 *SANS Institute*'s report [21], 4 out of 10 ICS security practitioners were not able to respond about the possibility that a breach occurred over the past year. From a 2017 *Kaspersky*'s report [28], 54% of the 359 interviewed companies experienced at least one incident over the past year. It also reports the presence of attacks that were not directly targeting ICSs like ransomwares with the example of *WannaCry* which was able to infect automobile manufactures of *Dacia* (*Renault*) and *Nissan*. In 2018, *Radiflow*, a specialized ICS cybersecurity company, detected a crypto-mining malware that infected SCADA servers in a water plant facility [34]. For some serious attacks, in 2015, *Calpine*, one of the biggest power generation and provider in the USA, had a large security breach and credentials to access its systems are suspected of being stolen as well as details about power station infrastructures [20]. During the same period, an electric provider in Ukraine reported a service outage provoked by foreign accesses to its systems which affected more than 200 000 customers [19]. Electricity is one of the most critical sector especially in terms of security of a country.

ICSs face all those attacks nowadays because of the general lack in security in the communication protocols, software, devices and configurations used. Those protocols were designed

for local fieldbus communications (equivalent to a LAN type) but because of their popularity and the spread of Internet, got updated to support Ethernet and IP. Of course, they were never intended to run on such wide open mediums so they lack or even have a complete absence of security mechanisms. Most of them are completely missing any data integrity, authenticity and confidentiality feature in their implementation and relied on air gaps between networks as a protection. On top of that, they now receive all the common issues from Ethernet and IP networks. Also, during those times, proprietary protocols, software and devices used to base their security on obscurity and were thus considered secure by default but only as long as the technology is kept secret. Hardware limitations and bugs in implementations are also added to the problem as well as weak security conceptions like hard coded passwords. Even if, nowadays, manufacturers make security one of their key element by providing support for updates, equipment are still slowly patched because of the cost of downtimes on the production and the difficulties encountered with legacy devices.

ICS meeting with the same threats as IT systems is now a big challenge for companies to avoid loss of money and reputation.

1.3 Research Issues in ICS Security

With all the security issues in ICSs, there are plenty of research activities to perform. The ICS community has gathered a lot of persons from university researcher to manufactures, specialized cybersecurity companies and amateurs. The research are mainly divided in 2 categories.

The first one works on actual equipment which thus ensure realism and accuracy to real world systems. They are able to reproduce behaviors and test vulnerabilities and bugs that are inherent in this environment. This does not come without drawbacks. There are major difficulties to perform security tests in a real industrial environment because of the safety concerns for the production and the equipment itself. It also only represents a specific configuration which is the best way to assess the current solution but will not yield the same value for general purposes. The results are not shareable as they could reveal sensible information about the systems used. Most tests are thus performed on reproduced environments (testbeds) which also come with a lot of drawbacks. The expensive cost of the hardware limits the possibilities. It takes space and time to configure, maintain, modify or add features to at the end get a non portable, hard to reproduce and difficult to share testbed. However testbeds details and results can be shared with the community.

The second works on virtual solutions which greatly enhance the flexibility, the utility via sharing and reduce the cost. The benefits are the easy set up, modifications, portability and with open source projects, the possibility to share results and the source code to avoid duplicate efforts, receive feedbacks and contributions to achieve at the end higher quality. The creation of hybrid testbeds to test the interoperability with real equipment will consolidate exchanges with the first category. Emulation is hardly possible due to proprietary devices, it would require insane amounts of work for each device to support. Simulation is the main focus as it opens the way to multiple concepts. A major drawback is the accuracy to real systems that will need to be stated and compared. From low to high accuracy, every inputs through its own way will help the community to grow.

A section is dedicated to present works related to ICS research but we already want to state that even though a lot of work has already been done there is still a huge lack in tools, access to testbeds details and exchange of results, most of those resources are private with only a few open source projects. The research are still minor compared to the wide variety of protocols, software, devices and configurations used in practice.

1.4 Goals of The Thesis

According to the current state of security issues and research in ICS environments, we propose in this Master Thesis the following objectives:

- Creation of a general ICS simulator that can support different network topologies, execute software to simulate typical ICS components like PLC, MTU and SCADA system communicating between each other through the communication protocol Modbus/TCP.
- Creation of tools and automated configurations to simplify the generation of ICS examples that are ready to run in the simulator.
- Visualization of running ICS simulations to provide a quick overview and facilitate interactions with the components.
- Generation of several ICS examples to present typical topologies found in literatures.
- Presentation of attacks performed on a simulated ICS and establishment of protections against them.

To summarize, we propose an ICS simulator that can be used to discover the most known ICS components, learn their role and how they can be configured, test and create ICS examples and finally discover security challenges that are inherent in ICS environments through attacks and protections.

1.5 Structure of The Thesis

To explain how we have achieved our goals we organized this Master Thesis as follows.

- The second chapter provides more details about the ICS devices and the communications protocols that will be used in following chapters.
- The third chapter presents the state of the art in the field of ICS simulation.
- In the fourth chapter, we explain the decisions we made, the tools used and the modifications performed to simulate an ICS.
- The fifth chapter focuses on explaining how the generation and visualization of ICS work along with some measurements on the limitation of the system.
- In the sixth chapter, we explain attacks performed on a simulated ICS and possible countermeasures.
- Finally, we conclude this Master Thesis in the last chapter with a reminder of our contributions and some ideas to further improve this work.

Chapter 2

Industrial Control System (ICS)

This chapter first provides more details about the ICS devices that were introduced previously then presents some communication protocols found in ICSs with the focus on the Modbus protocol. The rest of the thesis will mostly focus on those devices as they are considered the essentials components of an ICS, they are the ones the most encountered in literatures. Many more exist and nowadays their features often overlap between each others which makes their exact roles difficult to define. Each description will thus be kept general and important roles stated as we considered them in this thesis.

2.1 Programmable Logic Controller (PLC)

A Programmable Logic Controller (PLC) is a specialized device that provides an automatic control over industrial equipment. It is used to start machines, regulate motors speed, open valves, etc and to do so, it communicates with sensors and actuators also called field devices (Figure 2.1). PLCs are ruggedized to survive to harsh environmental conditions like dust, humidity, temperatures and vibrations. They run on custom embedded operating systems often called firmwares as they provide a smaller set of features than commercial OS and are designed with hard real time considerations. They typically communicate with other devices in a master-slave fashion, a master will send requests to a slave PLC to answer.

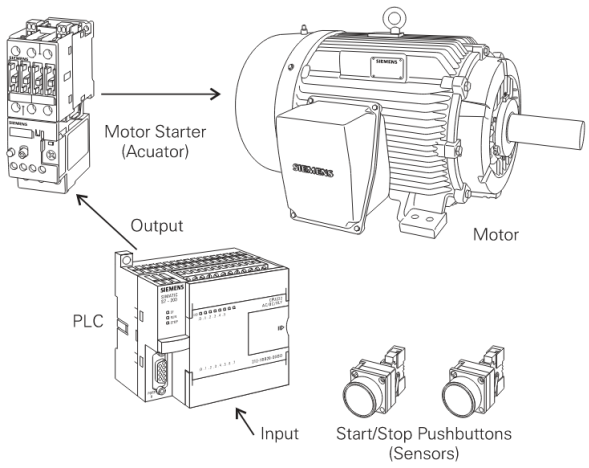

Figure 2.1: PLC connected to sensors and actuators [39]

PLCs are a challenge in terms of security. They are critical components and without them the productions stop. Unfortunately they present a lot of vulnerabilities by themself. They are limited devices for cost and performance which makes them vulnerable to overloads of

processing and memory. Modern PLCs have more and more features like the support for multiple communication protocols or integrated web servers that increase the possibility of flaws from their implementations. They also provide low physical tampering protections as their installations are assumed to be in private environments or in locked closets.

In literatures, you can also find devices like Remote Terminal Units (RTUs) and Intelligent Electronic Devices (IEDs). Those devices and the PLCs now overlap each other in their functionalities making their distinctions not really necessary anymore. In this thesis, we regrouped them all under the PLC category.

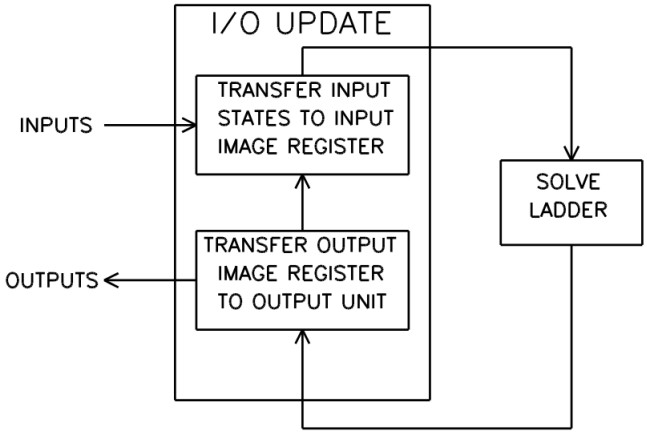

Figure 2.2: PLC scan cycle [16]

PLCs run inside an infinite loop where they read some inputs values, execute a logic and write some outputs. One execution of the loop is called a scan cycle (Figure 2.2). In a hard real time system, such cycle will always be executed within a deterministic time range.

The inputs are separated in 2 categories. The first one which was originally the only available contains the discrete input received through a discrete signal then translated to the value 0 or 1. Pushing a button to start or stop a machine is a typical example of a discrete input. The second category represents the continuous input received through an analog signal and then translated from 0 to the maximum scale with the precision depending on the PLC specifications. Such a signal can carry the current motor speed, water flow, oil pressure, etc.

Outputs can carry the same signals as the inputs and are called discrete and continuous outputs. They are transmitted to the actuators to modify the status of the equipment.

The control logic run by PLCs is standardized by the IEC 61131-3 which presents 5 programming languages (3 graphicals, 2 textuals):

- Ladder Diagram (LD), graphical
- Function Block Diagram (FBD), graphical
- Structured Text (ST), textual
- Instruction List (IL), textual
- Sequential Function Chart (SFC), graphical

We only introduce the Ladder Diagram, also simply called ladder logic, as it is the most commonly used because of its ease to program with graphical editors and its simplicity, it comes from relay logic in electric circuits.

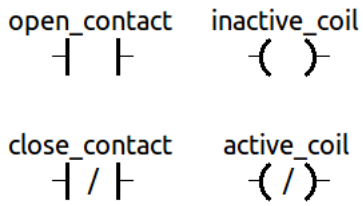

Figure 2.3: Contacts and coils

Shown in Figure 2.3, there are 2 basic elements in ladder logic.

The contacts are the inputs of the logic. They can be physical, received from an external module, or internal from the memory. Like a switch for a light, the open contact stops the electric current by default, its corresponding input is false (0) and said to be de-energized. When the input is true (1) or energized, the current can cross the contact. The close contact works the opposite, it lets the current cross by default and stops it when its corresponding input is energized.

The coils are the outputs of the logic. They can be transmitted to an external module or kept in the internal memory to save their current state for later logic steps. The inactive coil is de-energized by default and energized when current cross it. The active coil works the opposite, it is energized by default and de-energized when current cross it.

A lot of other elements called function blocks provide built-in operations like arithmetic (addition, multiplication,...), selection (min, max,...), comparison (greater-than, equals,...), bitshift, bitwise and time. We will introduce some of them later in the thesis as we use them in examples.

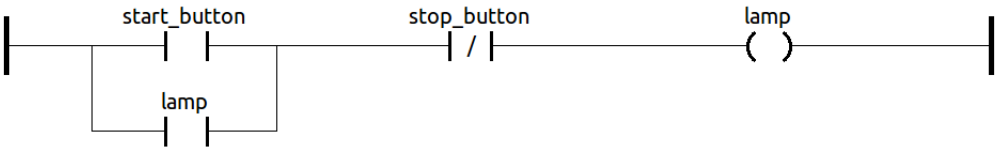

Figure 2.4: Ladder logic example

In ladder logic, each horizontal line is called a rung and can be interpreted as a rule read from left to right. A logic contains a set of rules that are sequentially evaluated from top to bottom which is called a scan. By making the scan time, the time taken to perform a scan, small enough (typically less than 100 milliseconds) the simultaneous and immediate effects found in relay logic can be replicated. However as the execution is still sequential, the order of the rules need to be correctly chosen. It is considered a good practice to put coils at the end of a rung to clearly see what are the produced outputs. Thus if a rung has its evaluation stopped before the end because of an open contact for example, the coils will keep their default value or the value set during a previous rung. A coil will carry its state to the next scan as long as it gets refreshed once per scan otherwise it will return to its default state.

The figure 2.4 presents a simple logic to turn on and off a lamp with a start and stop button. It has a single rung and can be translated in propositional logic as:

$$lamp = (start\_button \vee lamp) \wedge (\neg stop\_button)$$

While looking very simple, ladder logic can easily contain bugs that are difficult to detect and fix. From this simple example, the stop button is dominant over the start button which means that the stop always wins when the 2 are pressed at the same time. If the start button gets stuck on its pressed state, the only way to stop the process is to keep pressing the stop button until the start is finally fixed and released. This can become very dangerous in certain

situations. This is why Safety Instrumented Systems (SISs) should be enforced. Those are only 2 basic considerations but more complex situations will require a lot of precautions. Most vendors provide software with their devices for logic edition and testing to avoid facing bugs in the final production environment that could incur money losses and harm people.

2.2 Supervisory Control And Data Acquisition (SCADA)

In the hierarchy of the industrial control system, the highest level is called SCADA which is an acronym for Supervistory Control and Data Acquisition. SCADA systems are used to monitor and control centralized data acquired from different field sites. This system thus combine data transmission systems with data acquisition systems and HMI (Human Machine Interface) to facilitate the work of the operator.

SCADA systems are composed of several devices like PLC (explained in the previous section) or MTU (Master Terminal Unit) which gather data from RTU or PLC, that will be explained further in detail in the following section. It is also composed by HMI to visualize and control field operation, furthermore, all important data from these fields are kept in a database called Historian.

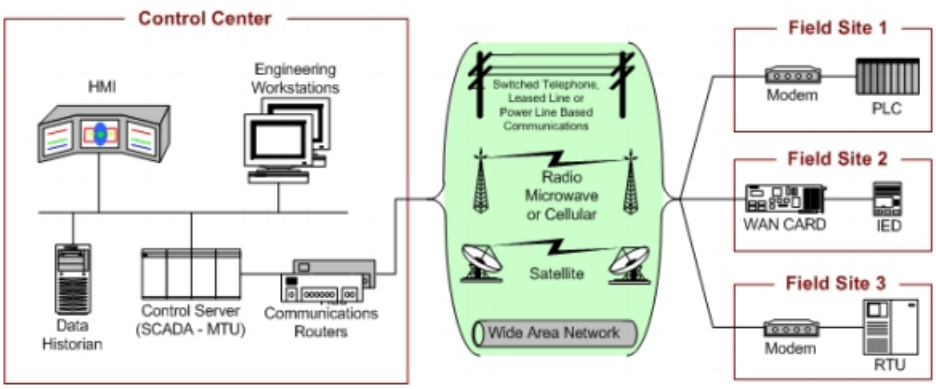

Figure 2.5: Scada Basic Topology [40]

Based on the basic topology of a SCADA system shown in Figure 2.5, its workflow can be resumed as follows, data from PLC and RTU will be gathered and processed by the Control server after which they will be displayed by the HMI to the operator. Then, the operator can take an action or automatic processes have already been set and so actions will be automated. Data gathered and action taken are simultaneously stored by the Historian.

Until recently, the main security of a SCADA systems networks was that these networks could not be accessed by an attacker because they were electronically isolated from other networks. But now, with the wish of increasing the connectivity between the factory floors and the cooperated zone, networks security of these SCADA systems have decreased as there are multiple access point that a determined attacker could exploit. Furthermore, over the year SCADA systems protocols moved from proprietary Standards towards open international standards, resulting in attackers knowing precisely the protocols. That is why there is a gain of interest in reinforcing industrial control systems security. And therefore, three main challenges need to be overcome to ensure SCADA system security. [18]

The first challenge is to reinforce the access controls to the SCADA networks because it is difficult to make proper access control due to the trouble of defining the scope of a SCADA network. The second challenge is to increase the security and its monitoring inside the SCADA

network using firewalls and intrusion detection system. And the third challenge is to enhance the security management. [18]

In some literatures, SCADA is confused with DCS (Distributed Control Systems), it is true that they overlap each other because computer network have become faster but their role are originally different [42] . The former is data-gathering oriented and event driven while the latter is process oriented and process state driven [3].

2.3 Master Terminal Unit (MTU)

The Master Terminal Unit (MTU) can be used as the center of the SCADA system as shown on the Figure 2.5, it initiates the communication with the RTU or the PLC, gather data from the PLCs and process them. The communication between the MTU and the PLCs is bidirectional, however only the MTU can initiate the communication and when the MTU asks data from a PLC, the PLC sends it. Therefore, the MTU uses a master-slave communication where the MTU is the Master and PLCs are the slaves. Messages from the MTU to the PLCs can be triggered by an operator or be automatically triggered. These messages can be either to read memory parts that represent current values like water flow, oil pressure, temperature of a tank, ... Or either to write values in the memory and modify the configuration.

Just like PLC, MTUs are a challenge in term of security because if an attacker successfully gains access to it, he also obtains the right to modify the configuration of the PLCs that communicate with. Beside that, it is also dealing with the same problems as the PLC.

2.4 Human Machine Interface (HMI)

The Human Machine Interface (HMI) is a graphical interface used by the operator to interact with PLCs, RTUs, IEDs or MTUs . HMIs display graphical information about the current production status, values and thus allow the operator to interact with the control process in order to, for example, adjust set points, start or stop the cycle. The Figure 2.6 shows some classic interactions between HMI, controllers and the field plant floor. HMIs can be considered as a bridge between the logic of the PLCs and the human operator. It allows the operator to focus mainly on the process rather than on the complex logic of the PLCs.

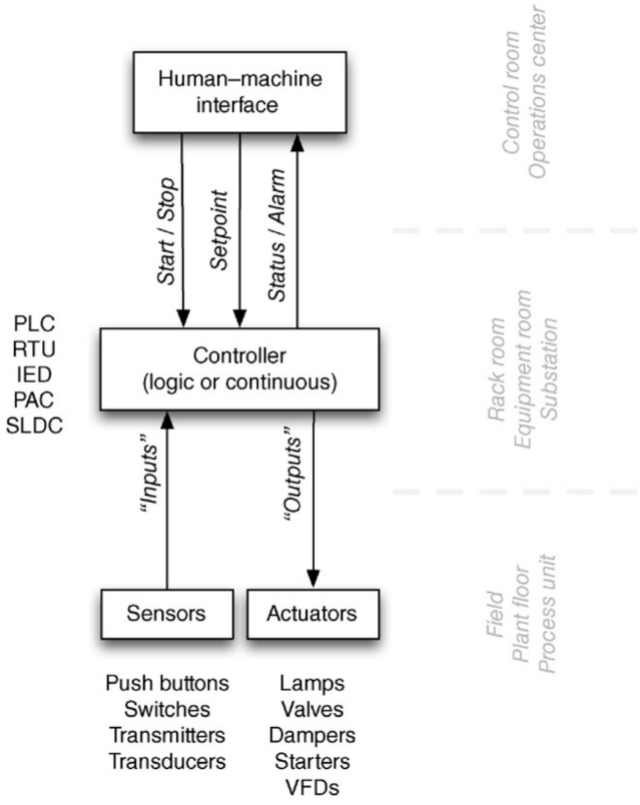

Figure 2.6: HMI interaction [40]

In term of security, the main problem is the access control because to ensure the principle of guaranteed availability even during an abnormal event, for example a password lockout, humans that have access to the HMI do not need to authenticate. Even though this seems to be completely insecure, if we consider that the HMI is generally used by trained and authorized employees and implanted in areas with a strong physical security as shown in Figure 2.5, then it is acceptable but not risk free.

2.5 Communication Protocols

While many communication protocols can be found in industrial environments, we will only cover the Modbus protocol as this thesis specially focus on it. Other protocols will be quickly stated at the end of this section.

Modbus was designed by Modicon in 1979 and integrated in the first PLC. It became the most adopted protocol in industries for its simplicity, low overhead, open standard, free of charge and raw messages that make it usable with low constraints [37]. Since then, Schneider Electric acquired Modicon and in 2004, transferred the rights to the Modbus Organization which is still maintaining it.

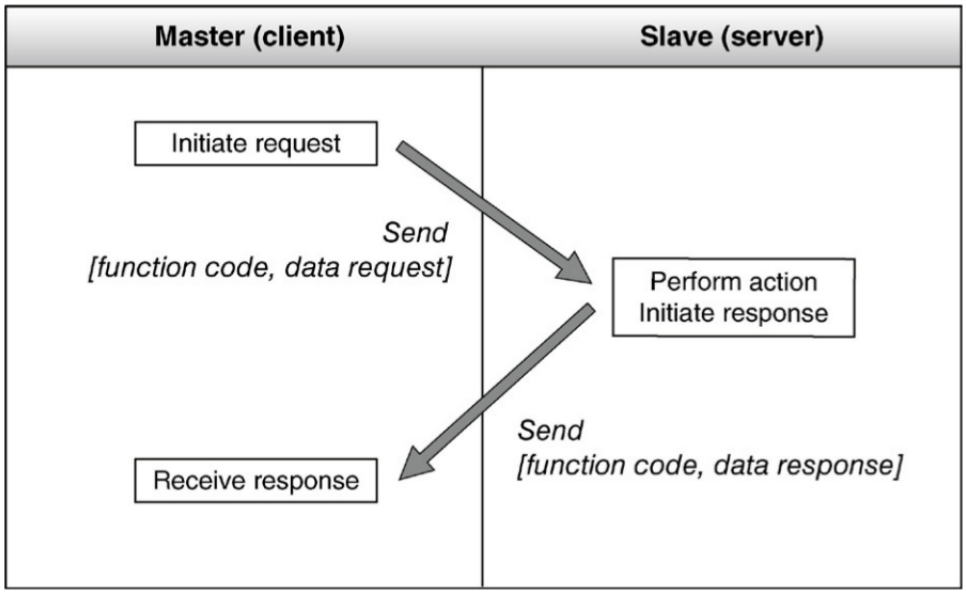

Figure 2.7: Modbus Request-response mechanism [25]

Modbus provides a communication mechanism in a request-response fashion (Figure 2.7) between a client and a server with a simple Protocol Data Unit (PDU). It operates at the Layer 7 of the OSI model, the application layer, which makes it adaptable to different under layers by adjusting the Application Data Unit (ADU).

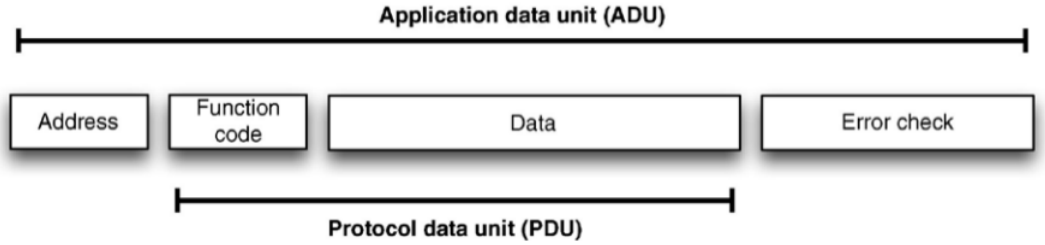

Figure 2.8: General Modbus Frame [25]

As illustrated in Figure 2.8, the PDU is as simple as possible making it independent of its final use. Extras fields are added to it to form the ADU.

Data Type	Size	Access	Address Range
Discrete Input	1 bit	Read-only	00001-09999
Coil	1 bit	Read-write	10001-19999
Input Register	16 bits	Read-only	30001-39999
Holding Register	16 bits	Read-write	40001-49999

Table 2.1: Modbus Data Model

Modbus defines data in 4 categories as shown in Table 2.1. Discrete inputs and coils are single bit values, 0 or 1 (false or true). Input and holding registers range from 0 to 65535, they can represent any kind of data like temperatures, water flow, pressure, etc. Multiple registers can be used if more precision or larger values are needed. Each address range has its first bit set

to a data type and can hold a maximum of 10,000 values. the new Modbus specifications allow up to 65,536 values. Those addresses are only relevant in the data model, the mapping to real memory addresses is left to the manufacturers. All devices do not necessarily support the full address ranges neither all data types. An important point is that Modbus is not able to write on discrete inputs and input registers, those are read-only and reserved for external I/O modules.

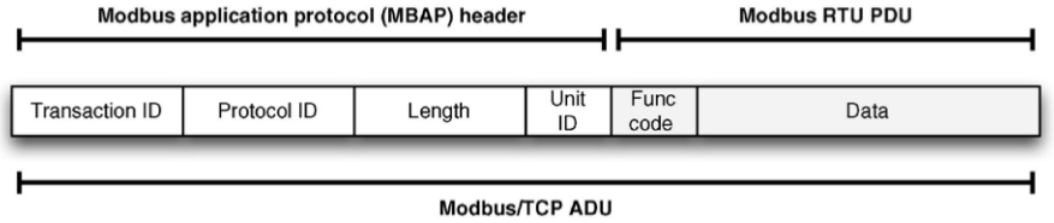

Figure 2.9: Modbus/TCP frame [25]

At its beginnings, Modbus had 2 variants called Modbus RTU and Modbus ASCII and was designed to operate on serial lines like RS-232 and RS-485 or other buses. Because of the growing interest for device communications between networks, Modbus received a new variant called Modbus/TCP that runs over IP, enabling Modbus messages over modern routable networks.

Details of the fields contained in a Modbus/TCP request (Figure 2.9) are as follows:

- Transaction ID (2 bytes): This field is used for pairing the request with the response message. The client chooses an identifier and the server will copy it in its response.
- Protocol ID (2 bytes): It is set to the value 0 by default to identify the Modbus protocol. It can be used for future modifications.
- Length (2 bytes): Count the number of bytes of the following fields, from the Unit ID to the end of the frame.
- Unit ID (1 byte): This field is used to identify a device on a serial line or other buses. After being routed, the message is delivered to its final destination via a gateway. In a TCP/IP only environment, this field is ignored and should be set to 0 or 255.
- Function code (1 byte): The function code represents the actual meaning of the message, it defines the action performed. Function codes are defined in 3 categories. Public function codes are well defined, they include both assigned codes and unassigned codes reserved for future implementations. User-defined function codes are reserved for manufacturers to implement new features. Reserved function codes are not available for public use and used for legacy devices. Only values from 1 to 127 are used, value 0 is illegal and values from 128 to 255 are used for exception function code in response messages.

Function code	Function name	Description
01	Read Coils	Read ON/OFF status
02	Read Discrete Inputs	Read ON/OFF status
03	Read Holding Registers	Read analog output content
04	Read Input Registers	Read analog input content
05	Write Single Coil	Write a single coil to ON/OFF
06	Write Single Holding Register	Write a single analog output register
15	Write Multiple Coils	Write multiple coils to ON/OFF
16	Write Multiple Holding Registers	Write multiple analog output registers

Table 2.2: Basic Modbus function codes and descriptions

- Data (variable length): The data field content depends on the function code used. In a typical read operation, it contains the starting address and the quantity of continuous addresses to read. As some requests-responses contain a single byte value that count the number of bytes, the maximum size of the data field is 255 bytes.

Details of a Modbus/TCP response message triggered by an incoming request are as follows:

- The transaction ID, protocol ID, unit ID are the same as in the corresponding request.
- The same function code is returned if the action was performed successfully. If an exception occurred, the function code is incremented by 128 to fall in the range of the exception function codes.
- The data field depends on the executed function. For a typical read operation, it contains the number of bytes followed by the returned values. If an exception occurred, it contains the corresponding exception code.

Exception code	Exception name	Description
01	Illegal Function Code	Function code not accepted by the server
02	Illegal Data Address	Data address outside server ranges
03	Illegal Data Value	Fault in data field detected by the server
04	Server Failure	Error occurred during server execution
05	Acknowledge	Server is processing, response delayed
06	Server Busy	Retry query later when server is free
07	Negative Acknowledgement	Server couldn't perform requested action
08	Memory Parity Error	Parity error detected in server memory
10	Gateway Path Problem	Misconfiguration of the gateway
11	Gateway Target Problem	No response from the server

Table 2.3: Modbus Exception codes and descriptions

Modbus was not designed with security issues in mind as they were not of any concerns during its creation. Those issues are critical with Modbus/TCP now being transmitted over untrusted networks :

- Confidentiality: Everything is transmitted in clear text, meaning that any captured packets can be read completely.
- Integrity: There is no mechanism to detect possible modifications on the messages.
- Authenticity: No authentication is required to exchange messages. A server always responds to any received messages.

An attacker can easily discover relevant information of the running system with simple scans and monitoring. He can capture, modify, forge or replay messages and no mechanism is present to detect it. Exploiting those issues could harm industrial equipment and productions but could also be dangerous for human lives.

This situation is the reason why a lot of research and work are now being done on ICSs as the importance of critical infrastructures and threats against them keep growing.

Many other industrial communication protocols exist. Some alternatives to Modbus are Profibus and DNP3 which were also originally fiedlbus protocols. Protocols designed with routing capabilities like EtherCAT and EtherNET/IP. They also come with many variants. For example, DNP3 Secure adds challenge-response authentication to DNP3. Each protocol presents its own vulnerabilities rising the difficulties to achieve overall ICS security.

Chapter 3

State of The Art

This chapter presents some works in the field of ICS simulation. Works are presented in a non-exhaustive way as the field possibilities are wide and covering everything that was already conducted is impossible for us. Most of the proposed solutions are private or only available for insiders, only a few are public open source projects. This was a considerable down side for us as some projects sounded promising.

We start with the survey from Holm et al. [17] that presents general statistics over a selection of 30 ICS testbeds. Some important points can be retained from it. Most testbeds are based on real hardware or make use of virtual devices and sometimes use a mix of the two. A large variety of ICS components and protocols are mentioned. The most common devices are RTU, PLC, MTU, HMI and IED. For the protocols, Modbus, DNP3, OPC, IEC 60870 and IEC 61850 are the most discussed. For all the 30 testbeds, the physical process which represents the industrial process controlled by the ICS is simulated via models from Matlab/Simulink, LabView, PowerWorld or other libraries. Only a few testbeds cover the fidelity of the proposed solution against real world systems. From most of the papers we have read ourselves, we also encountered those characteristics. This survey helped us realize what are the possible ways to approach the problem, the difficulties and limitations of simulating an ICS environment.

The following presentation of related works is divided in multiple categories although some projects actually belong to more than a single category.

3.1 Hardware Testbeds

In this Master Thesis, we decided to focus on virtual solutions as buying or accessing real hardware was not an option because of the cost and the lack of dedicated infrastructures. Despite that, we searched for Hardware testbeds in order to have an idea about the state of the art in this domain. And we found that testbeds that use physical devices are generally mixed with simulated components for the sake of cost saving. The testbed from Haihui G., Yong P., Zhonghua D., Ting W., Xuefeng H., Hanjing L. [15] uses this mixed of physical and simulated component. They used real physical PLCs and RTUs that communicate with the emulated corporate network thanks to a special hardware device. They decided to not used any real industrial Boiler or Gas pipeline to reduce cost and reuse input/output layer components. Instead they simulated several control processes using Matlab/Simulink and mathematical models obtained after analyzing an industrial oil-fired boiler.

3.2 Emulations

Already stated in the introduction, emulation is hard and requires way too much work but would be the most accurate solution to virtualize devices. We came across one interesting project.

Awlsim [10] is a Python open source project that emulates the execution of PLC logic like it would be executed on a real Siemens S7 CPU. It can run logic encoded in STL (Statement List) which corresponds to the Instruction List (IL) language from the IEC 61131-3 standard with some vendor specific additional features. For Siemens, STL is closed to the machine code running on their S7 CPUs which is useful for optimizations. Awlsim support a large subset of S7 instructions and try to be as close as possible to real Siemens hardware like S7-300 and S7-400 CPUs. To interact with the running logic, it provides a way to simulate inputs and read outputs values. It can also run on PiXtend V1.3, a raspberry Pi based PLC, and interface with the built-in I/O modules (digital/analogue inputs/outputs and relays). Awlsim also comes with PROFIBUS-DP and LinuxCNC communication modules that enable the exchange of messages.

Babu and Nicol [7] created a network simulation containing Awlsim PLCs. The network part is managed by the discrete network simulator S3F and Awlsim received a modification to support the Modbus protocol. A kernel modification called TimeKeeper is used to keep a tight control over the Awlsim emulators and the network simulator by providing a virtual clock. The system uses this clock instead of the hardware clock to be able to advance synchronously and thus yield a constant behavior under varied configurations. The speed of the virtual clock depends on the chosen Time Dilatation Factor (TDF) which represents the additional time given to the system to compute a step. With this additional time, the system is able to run multiple instances of Awlsim PLC on commodity hardware while keeping a steady behavior. The authors compared the packet delays, throughputs and compute times between a scenario controlled by TimeKeeper and a best-effort scenario. From 7 to 63 PLC nodes, the controlled environment clearly showed constant results while the best-effort environment loses accuracy the more nodes are added. For future works, the authors state that an Intrusion Detection System (IDS) will be created on the testbed. It is possible to implement an IDS that relies on precise network traffic patterns within such accurate environment.

3.3 Simulations

This category is the most interesting for us as the projects are fully virtualized simulations of ICSs. The big advantages of such virtual solutions are the portability, the ease of modification and the possibility to work with the community in open source projects.

While searching for ICS simulations, we encountered a lot of projects specialized on smart grids. Smart grids define electric supply networks that rely on modern digital communications. They differ from typical ICSs as their networks are extremely wide with thousands of devices, remote substations and responsible to supply thousands of customers. Even with those differences, they are still highly relevant for ICSs, ideas can be discussed and maybe adapted between the two.

On the smart grid subject, we found one solution appearing in multiple projects. Mosaik [1] is an open source co-simulation framework written in Python, it is able to combine simulators like power plants, photovoltaic and eolian into a large scenario that incorporates thousands of simulated entities at the same time. Mosaik is directly related to power grids as it already provides simple simulators, scenarios and also an interface to PYPOWER, a power flow solver. The framework uses a synchronization mechanism in discrete time where all simulators advance together by step. This is an important characteristic to orchestrate interactions between connected simulators. The paper from Dede et al. [13] explain that Mosaik assumes perfect links between simulated entities and thus propose to integrate a communication simulator to achieve a more realistic simulation. They present OMNeT++ to simulate communication interactions.

Instead of having Mosaik directly exchanges messages between simulators, the messages are sent to OmNeT++ to be routed through its network simulation. Wermann et al. [44] present ASTORIA, a framework to evaluate the consequences of attacks on smart grids. They use Mosaik with PYPOWER and NS-3 for the network communications. NS-3 was extended to support Modbus/TCP and DNP3 protocols and to simulate the behaviors of MTUs, RTUs, PLCs and field devices like sensors. They simulate Man-in-the-Middle (MitM) and Denial of Service (DoS) attacks on the power grid infrastructures to evaluate the operational and financial losses. Chomik et al. [12] discuss how security can be increased by monitoring the network with an Intrusion Detection System (IDS) that is aware of the current state of the smart grid. They use Mosaik with PYPOWER and simulate RTUs communicating with a SCADA system via Modbus/TCP. Bro, a well known network monitoring tool, is used as an IDS to detect Modbus commands that could harm the power grid in its current state. Koutsandria et al. [26] also worked on a process aware IDS. They use Matlab/Simulink to model the physical process and integrated Modbus to communicate with real and simulated PLCs. Chen et al. [11] present network attacks over a smart grid simulation. They use RTDS for the power simulation, LabView and PXI modules to simulate a SCADA system and IEDs devices and Opnet for the network simulation. They performed real MiTM attacks that were able to sniff all packets from Modbus/TCP communications and modify their contents. In a Dos attack scenario, a Modbus server was flooded by TCP SYN packets to overload its capacity and render it unavailable.

Reaves and Morris [35] propose a Python framework for complete ICS simulation. It contains a physical simulator and virtual devices called VDEVs to simulate PLCs/RTUs that can run control logic code and communicate via a network protocol. They generalize everything via interface making components interchangeable. The communication protocol can be changed without impacting the logic code thanks to the memory abstraction that uses object called points instead of direct memory addresses. VDEVs can also be replaced with real physical devices. This is the only paper that actually compared its virtual testbed against a real one. They obtained satisfying results on a gas pipeline and a water tank testbed with a set of metrics and also in the behaviors under attacks. From the same project, data logs containing 35 cyber attacks on the Modbus protocol were created for IDS research [32] and Modbus/TCP attacks against a virtual testbeds tested [43].

3.4 Honeypots

While searching for ICS security measures, we ran into several papers about honeypots which are more and more used to improve security of an ICS network. An Honeypots is generally a physical or virtual systems that mimic real devices on the network in order to lure attacker in thinking that it is an actual real system. An honeypot can be used to help researchers in understanding new threats and also served as an indicator of malicious activities on the network.

In testbeds we found on Honeypots, these are classified into three categories, low-interaction, high-interaction and hybrid. The first one, runs software that act like the operating system and only basic network services. The attackers or malwares have only limited interaction with the Honeypot, for example an honeypot simulating a Cisco router may act like a real Cisco router but never let an attacker to successfully log in. In contrast, in a high-interaction honeypot, attackers have the possibility to do whatever they want. Therefore, high-interaction honeypots have to act as much as possible like the actual device. Thus no emulation is used, everything is real which makes the implementation a lot harder. But it comes with a great disadvantages as if the honeypot is compromised it will have to be rebuilt and it also increase risks by providing real playgrounds for attackers. The last category is called hybrid because it mixed the two other and try to diminish their downfall.

Honeypots for SCADA networks already exist and can be found in the literature like Conpot or Honeynet.

Conpot is a low-interactive SCADA honeypot which supports simulation of protocols like Modbus, HTTP and SNMP while acting as a PLC device. We found two papers that used and analysed conpot, one is from A. F Jicha III from the university of Arizona [22] and the second is from the *SANS Institute*, written by C. Scott [38]. As Conpot does not proposed any tools for searching into logs or sending alerts other means need to be used, in the paper from C. Scott [38], a Syslog server was set up inside the facilities networks running *Rsyslog* because otherwise the the only way to see the logs was to connect on the honeypot. Furthemore a tool called *Splunk Entreprise* was also used to search through logs and send alerts.

Honeynet can be tought as a network of honeypots, Kuman, Stipe and Groš proposed a solution to design a honeynet using Conpot [27]. To do so, they used *IMUNES* which is a network simulator and emulator to simulate nodes interconnected with links. Moreover, they also used OSSEC Host intruction system that allow a central collection of logs. Each node in *IMUNES* is a Docker image, and so PC nodes are customized to run Conpot and OSSEC.

The Master thesis written by D. Andersson [6] explains how honeyd which is a deamon that creates virtual hosts, can be used alongside the test bed CRATE (Cyber Range and Training Environement) that allow to build large virtual network, in order to simulate a honeynet for an ICS. Ten templates of Honeyd were created to simulate field devices. However Honeyd does not provide interaction with the simulated devices except during the scanning phase, but two scripts emulating Telenet and FTP to provide the attacker with some interactions were added to one of the template.

Chapter 4

ICS Simulation

This chapter presents an in depth overview about the decisions we made to simulate an ICS. We discuss the design choices taken, the tools used and the modifications performed to support our needs.

4.1 Design

We took several design choices in our approach on ICS simulation.

First, we chose to go for a fully virtualized simulation as we wanted to be as general as possible and thus not tight to some specific hardware. Hardware with Ethernet capability are quite expensive and buying multiple units was not a viable option. A complete software solution is less restrictive on the components and the size of the simulation.

From the related works on virtual solutions, most projects use a time step mechanism to synchronize the whole simulation. Although this is an interesting approach, after some debates between us, we decided to not follow this path. As this project is our first introduction to the ICS field, we wanted to focus on vital components before thinking about a future framework integration and the increase of complexity that comes along with it. A real time simulation was thus decided, it is a much simpler approach that can benefit from tools already available. The drawback is the lost of accuracy to real systems, limited hardware will get overloaded for heavy simulation with many simultaneous running processes.

ICSs are composed of multiple devices that communicate to each others, we chose to use realistic networks that could carry real communications compared to solutions that use fake messages to simulate interactions. This is important in order to support existing tools, variable network topologies and test attacks on communication protocols. We decided to create a simulation that is able to generate ICS instances instead of focusing on a single specific example, create like a workflow that is general enough to support modifications and addition of new features for possible future works.

We considered using existing tools and modified them to support our needs so we could avoid duplicate works and go as far as possible on the subject itself. Open source projects are needed in this case.

Finally, an automatic installation and configuration of the project in a virtual machine is needed for easy setup.

4.2 Tools

In this section we will briefly introduce tools we used to reach our objectives, in order to have a clear overview of all tools before detailing them in the following sections.

First of all, we had to select a tool to be able to create virtual networks and thus we choose Mininet which is a network emulator for Software Defined Networks (SDN). Mininet is easy

to use and allow the user to create custom topologies easily by writing simple Python scripts. Furthermore, Mininet includes another tool called MiniEdit which is a simple graphical editor for Mininet that allow us to create and save networks topologies simply by drag and drop of items. Moreover alongside MiniEdit, there is an open GUI tool written in Python Tinker, called MiniNAM, that provides a visualization of real-time packet flows of any Mininet networks.

As Mininet does not provide ICS components we had to find tools to simulate such devices. After some research, we selected OpenPLC which is an open-source Programmable Logic Controller that supports popular SCADA protocols such as Modbus and DNP3. OpenPLC can play the role of a PLC and MTU which is really convenient in our case as it kills two birds with one stone. Unfortunately, OpenPLC is not enough to simulate a SCADA system, thus we used PyScada which is an open source SCADA system to fulfill that role.

All these tools are used in a Linux environment, Ubuntu 14.04 LTS to be more precise and are mostly programmed in Python or Bash. For portability, everything is automatically installed and configured in a virtual image in VirtualBox via a Vagrant script.

4.3 PLC and MTU Simulation

The first component needed to create an ICS simulation is the PLC. It took quite some time to search and test available solutions and discover their limitations. We were looking for a free Linux compatible software that could simulate a typical PLC, support an actual communication protocol like Modbus/TCP, be general enough to support different behaviors and be part of a bigger simulation to represent an ICS. If possible, find an open source project to support modifications and avoid restrictions of closed sources. First, we found commercial solutions like RSLogix from Rockwell Automation and S7-PLCSIM from Siemens that are most likely not free or with a limited free trial, Windows OS only, closed sources and do not provide any external communications. Second, Matlab/Simulink can be used to directly program PLC for tests and code generations but this solution is too specific and way too heavy to be incorporated in a full ICS simulation. Then we found some smaller projects like ClassicLader, pyModSlave and EasyModbus that did not provide all the features we needed. Finally, we found one project that could fulfill our needs.

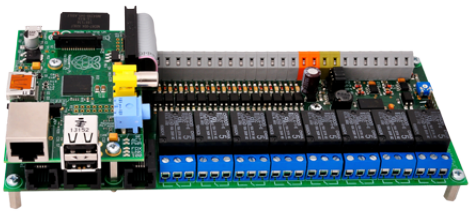

Figure 4.1: UniPi unit

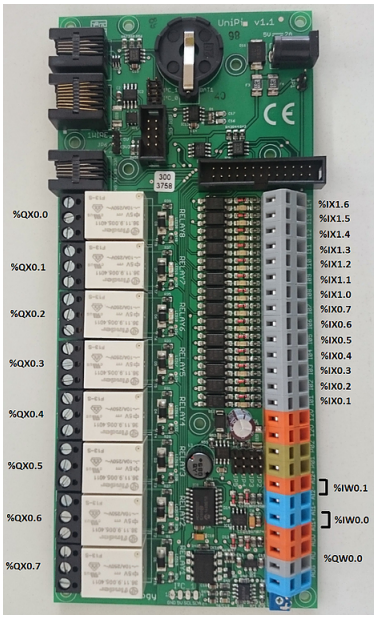

Figure 4.2: OpenPLC I/O pin mapping for UniPi

OpenPLC [4] is a project aimed to provide a standardized open source PLC. Started in

2014 [5], the project is now in its second version (OpenPLC_v2) and is still under development. It can run on open source hardware like Raspberry and Arduino or on more commercial one like UniPi (Figure 4.1). It is also possible to run it as a virtual PLC in complete software mode on Linux and Windows. A virtual PLC is what we were looking for to avoid investing money in hardware and have the possibility to run a high number of instances simultaneously to mimic a typical ICS composition. The core of OpenPLC stays the same between platforms with only minor compilation changes. What needs to be adapted is the code to access the hardware layer, more specifically the code to access the I/O of the platform (example for UniPi in Figure 4.2). It can thus be ported to new platforms without too much effort.

The project is compliant with the standard IEC 61131-3 meaning that it supports the 5 programming languages already presented in section 2.1. PLCopen Editor is proposed as an editor that supports programming in those languages. It is a Python open source software from the Beremiz project [8], a project that aims to create a complete development environment for automation. It is recommended for its ease to use, its option to save projects to their own recognized standard PLCopen XML and option to export any of the languages to the equivalent Structured Text (ST) logic code. The latter is useful as OpenPLC can compile and run ST logic and thus simulate how a real PLC would run with this same logic. The Modbus/TCP and DNP3 communication protocols are supported via the open source libraries libmodbus and opendnp3 respectively. It is thus possible to communicate with a running OpenPLC instance to read the current state of the values from its memory and also send write requests to modify them. If the modified values are also evaluated in the logic, they can influence its execution to perform new actions.

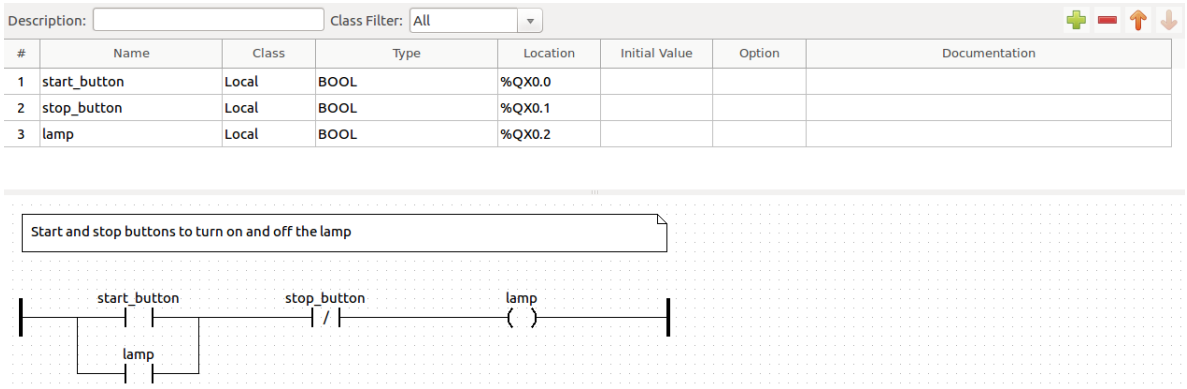

Figure 4.3: Ladder logic example (presented in section 2.1) inside PLCopen Editor

The Figure 4.3 presents an example of how the programming of ladder logic looks like in PLCopen Editor. The variables are declared on the top and the logic is created on the bottom. Each variable has a type and a specific memory location. In this example, the variables are the 3 first coils of the PLC mapping. The memory mappings are presented in Table 4.1, Table 4.2 and the special characters used to encode address ranges in Table 4.3.

Data Type	Size	Access	PLC Address	Modbus Address
Discrete Input	1 bit	Read-only	%IX0.0 - %IX99.7	0 - 799
Coil	1 bit	Read-write	%QX0.0 - %QX99.7	0 - 799
Input Register	16 bits	Read-only	%IW0 - %IW99	0 - 1023
Holding Register	16 bits	Read-write	%QW0 - %QW99	0 - 1023

Table 4.1: OpenPLC I/O address mapping to Modbus address

OpenPLC also provides an extended memory address space for larger registers.

Data Type	Size	Access	PLC Address	Modbus Address
Holding Register	16 bits	Read-write	%MW0 - %MW1023	1024 - 2047
Holding Register	32 bits	Read-write	%MD0 - %MD1023	2048 - 4095
Holding Register	64 bits	Read-write	%ML0 - %ML1023	4096 - 8191

Table 4.2: OpenPLC extended I/O address mapping to Modbus address

Prefix	Description	Prefix	Description
I	Input	Q	Output
M	Memory location	X	Bit
B	Byte (8 bits)	W	Word (16 bits)
D	Double word (32 bits)	L	Long (64 bits)

Table 4.3: Special memory characters

OpenPLC is mostly written in C++ with some Bash scripts for the installation and compilation. It can be started via a NodeJS server that creates an interface accessible from a web browser (Figure 4.4). It enables easy interactions to start and stop the PLC, check logs and upload a new ST logic that will get compiled and run automatically.

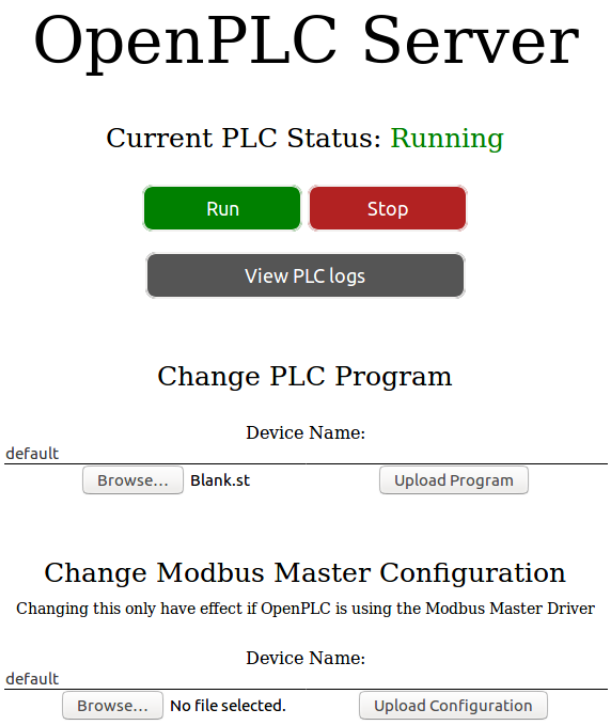

Figure 4.4: OpenPLC web interface

During the installation, the driver for the platform on which OpenPLC will be running is asked. In software mode, the default driver is blank, no I/O access is specified. This, combined with the fact that Modbus cannot send write requests for discrete inputs and input registers (section 2.5) raises problems for the simulation of the physical process controlled by the PLC. The physical process is supposed to receive inputs from the PLC and send its outputs to it, the two entities need to interact between each other. We did not want to use some workarounds with Modbus by using only coils and input registers and hardcode the physical process behavior else where in the simulation and actually, we could not because of problems of compatibility with other components it could have created. Sending uncommon packets and avoiding 2 of the 4 data types used by Modbus would be big losses on the goals of generality and realism. We did not want either to modify the ladder logic running on the PLC to directly incorporate the physical process for the same reasons and the difficulties we would have encountered to achieve that.

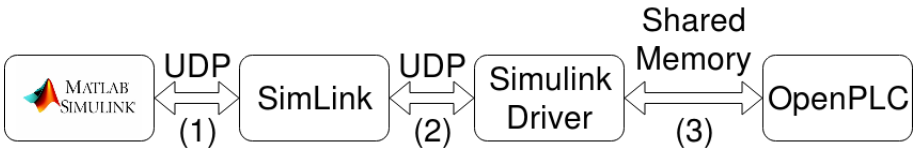

Figure 4.5: Interactions between OpenPLC and Matlab/Simulink

Fortunately, an other driver called Simulink is also provided. Through an other small program called SimLink, the PLC can communicate with the physical process simulated in Matlab/Simulink. Matlab/Simulink provides components that can send and receive data via UDP. As shown on the Figure 4.5, via UDP (1) exchanges, each of those components either send data to SimLink or either receive data from it. The data sent represents discrete inputs and input registers and the data received are coils and holding registers. Via UDP (2) exchanges, SimLink and the Simulink driver transfer the 4 types of data all at once. Via the shared memory (3), the Simulink driver overwrite discrete inputs and input registers on the PLC to apply the values produced by Matlab/Simulink and read the values of the coils and holding registers to send them back and possibly alter the simulated mathematical process. As we do not use Matlab/Simulink, we modified SimLink to be able to program simple physical process simulations that communicate with the PLC and thus do not use the UDP (1) exchanges.

```
#include "simlink_modified.h"
// Update values here
// Can read and write digitalIn (discrete input) and analogIn (input register)
// Can read digitalOut (coil) and analogOut (holding register)
void updateValues() {
    if(stations_data[0].digitalOut[2] == 1){ // If coil 2 is active
        stations_data[0].digitalIn[1] = 1; // Discrete input 1 forced active
        stations_data[0].analogIn[0] += 1; // Increase input register 0 by 1
    }else if(stations_data[0].digitalOut[3] == 1){ // If coil 3 is active
        stations_data[0].digitalIn[1] = 1;  // Discrete input 1 forced active
        stations_data[0].analogIn[0] -= 1; // Decrease input register 0 by 1
    }else{
        stations_data[0].digitalIn[1] = 0; // Discrete input 1 forced inactive
    }
}
```

Figure 4.6: Simple physical process example that increase or decrease a level

#	Name	Class	Type	Location	Initial Value	Option
1	start_button	Local	BOOL	%IX0.0		
2	start_coil	Local	BOOL	%QX0.0		
3	input_state	Local	BOOL	%QX0.1		
4	increase_state	Local	BOOL	%QX0.2		
5	decrease_state	Local	BOOL	%QX0.3		
6	current_level	Local	UINT	%IW0.0		
7	max_level	Local	UINT	%QW0.0	100	

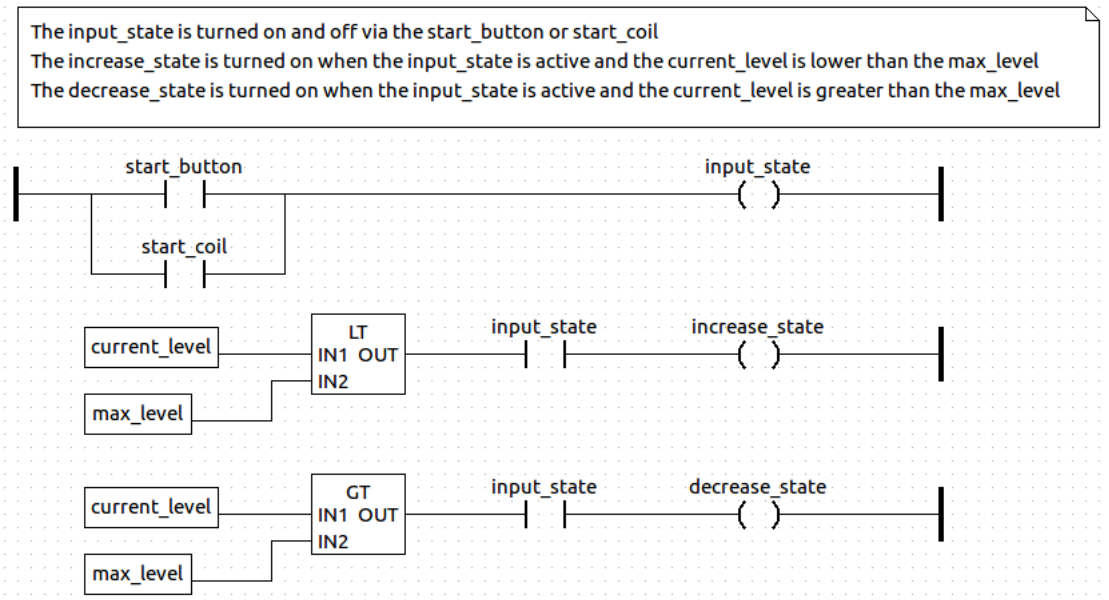

Figure 4.7: Ladder logic example that can control the physical process in Figure 4.6

The Figure 4.6 shows an example of a physical process simulation that can be used thanks to the modifications made to SimLink. The Figure 4.7 shows a ladder logic that can run on a PLC to control the simulated process. This simple simulation can represent the control of a water level. The physical process increases or decreases this level while the PLC controls the start, stop of the simulation and the maximum value that the level can reach.

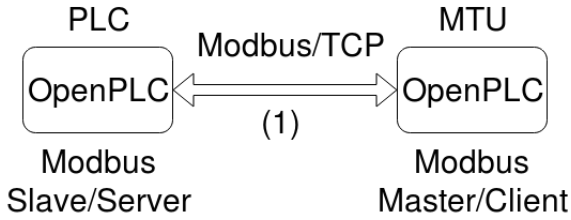

Figure 4.8: Modbus/TCP communication between PLC and MTU

Stated earlier, we also use OpenPLC to simulate MTUs which is very convenient. This is done through another provided driver simply called Modbus. In this mode, an OpenPLC instance will communicate to a Modbus server such as our previous OpenPLC instance to simulate a simple MTU. The MTU plays the role of the master and the PLC plays the role of the slave in the interactions. It can gather data from the PLC like discrete inputs and inputs registers, save them in its own memory and can also overwrite coils and holding registers on the PLC with its own values to propagate new settings. Shown on Figure 4.8, those data are exchange via the Modbus/TCP (1) communication between the MTU and the PLC like in real applications. The

current values in the memory of a running PLC or MTU can be checked via any compatible Modbus/TCP client application. We recommend the open source program called qModMaster to test the Modbus/TCP communication with a PLC. It provides an easy to use user interface to connect, interact and check Modbus messages exchanged with a Modbus server.

OpenPLC was designed to run one instance at a time but for an ICS simulation multiple PLC instances running simultaneously are needed. Each new instance was overwriting the compiled files of the previous one which is a problem for the functionality to start and stop PLCs. Restarting a PLC would make it uses the files from the latest compiled PLC which may differ from its original ones. There is no problem if every PLC in the simulation uses the same logic, the compilation is the same for everyone but otherwise a way to separate every instance is needed. We modified the JavaScript code that starts OpenPLC to keep track of the files, especially the logic of each PLC. Each instance requires an unique name to differentiate it from the others. Every time a PLC is restarted during the simulation, it will now restart with the correct associated logic and uploading a new logic will only overwrite the corresponding files. The same problem was faced with the MTU instances and their configuration files, we added new modifications to the JavaScript code to also keep track of them properly.

We also found and fixed an error in the Modbus implementation of OpenPLC. Some values were getting overwritten when receiving specific requests to write multiple coils at once (Modbus Function Code 15). In those requests, the data that contains the new values to write (new status of the coils) is managed by byte. Sending a request with a number of coils that is not a multiple of 8 creates a byte filled with some zeros at the end of the data field. The implementation was always reading by full bytes and thus overwriting with zeros a few coils that were not supposed to take part in the transaction. Those requests come with a field that states the exact number of coils to write but the value was left unused.

4.4 SCADA and HMI Software

PyScada is an open source SCADA system written in Python. It uses the web framework Django as a backend and MySQL to manage its database. It supports several industrial communication protocols like Modbus/TCP, VISA and 1-Wire.

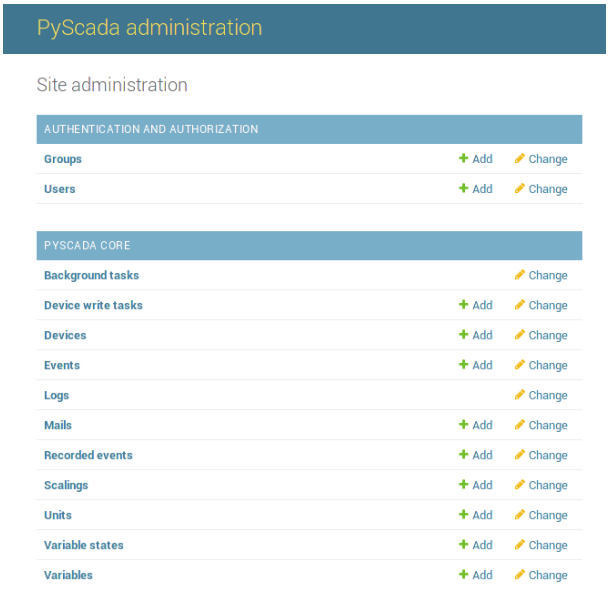

Figure 4.9: PyScada administration page

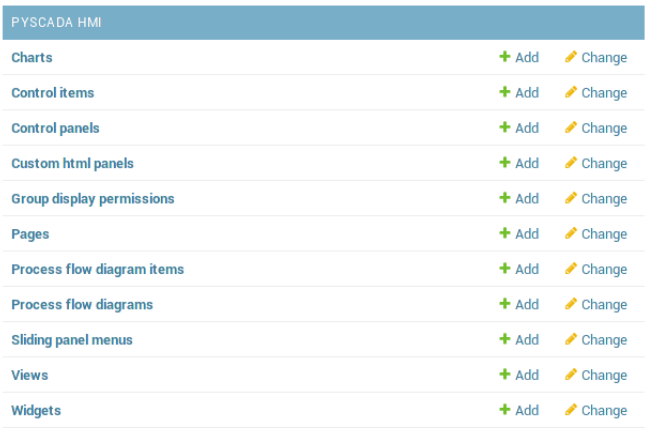

Figure 4.10: PyScada HMI components

The administration backend of PyScada, accessible via a web browser, gives access to the configuration and management of the SCADA system (Figure 4.9). An authorized administrator can add new users and groups of users with their access permissions, add new devices for the system to communicate with, configure tasks that will be performed automatically at the designated time or at the occurrence of an event, and much more.

To add a Modbus/TCP device, a new general device is first created then this device get specialized to use the Modbus protocol. Information like the IP address, the port, the unit ID are required. To have the system collects a specific value from this freshly created device, a new variable is added with information like the type (ex: boolean, integer, float), the unit (ex: second, kilogram, bar), read-only or writable and the device it belongs to. This variable is then specialized to Modbus with a type (coil, discrete input, holding register, input register) and a memory address.

PyScada provides a creation of HMIs. New web pages will automatically be generated from the configuration of simple elements like views, panels and widgets (Figure 4.10). Authorized operators can access the HMIs to get an overview of the current status of the variables in the system and also interact with the connected devices via items that send commands.

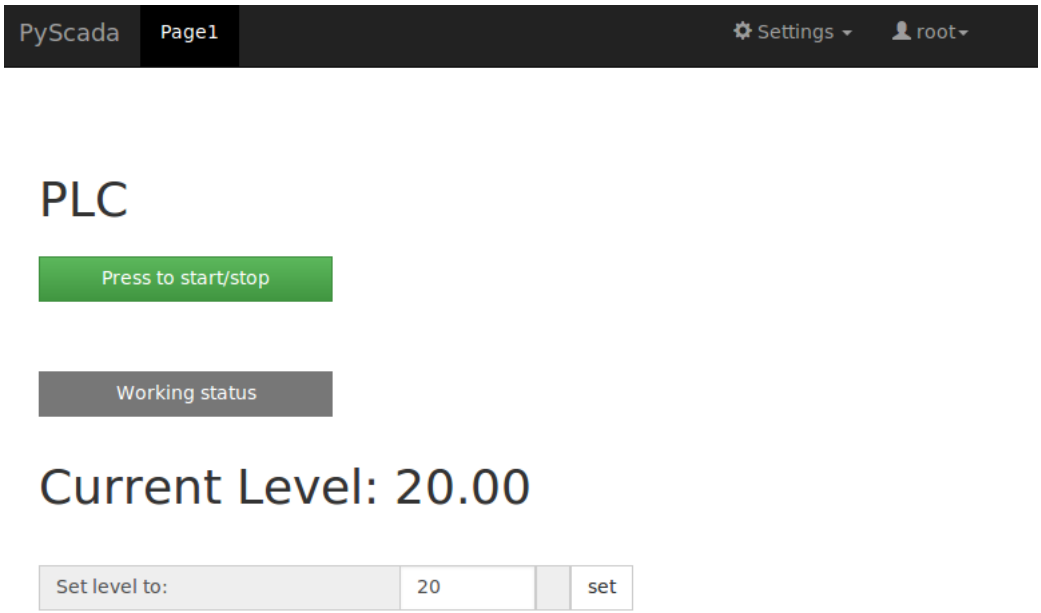

Figure 4.11: Example of a custom HMI in PyScada

Users can also create custom HMIs by providing their own HTML code. To achieve the example shown in Figure 4.11, we first used the automated generation then modified the HTML code to add custom descriptions and change how information is displayed. The first element (in green) is a control item that can start and stop a PLC by sending a write command. The second element (in grey) is a display item. The color grey represents the value 0 and green represents the value 1. The third element shows the value of the level currently sent by the PLC. The fourth element is a control item that can set a new level on the PLC. From the information provided by the HMI, an operator can see that although the PLC is started, it is currently not doing anything because the level has reached the configured value. This example is in relation with the physical process simulation and the PLC ladder logic presented in the previous section.

We can mention other open source SCADA systems like ScadaBR and SCADA-LTS which is based on ScadaBR. We tested them but encountered a lot of issues to install and configure them correctly. We kept PyScada for the web interface it provides, its compatibility with Modbus/TCP and the lightweight HMIs it can create.

4.5 Network Emulation

As stated before, we decided to use Mininet as it allows to create large realistic virtual Software defined networks up to thousand nodes on a simple computer or a virtual machine. Mininet already contains several default topologies but none of them can be used to represent an ICS, fortunately custom topologies can be given to Mininet easily by writing a simple Python script and passing it to Mininet via command line.

4.5.1 Software Defined Network (SDN)

As mentioned above, Mininet allows to experiment with SDN (Software Defined Network). The SDN paradigm is still new but nowadays more and more companies are considering it, unfortunately, paper on ICS using a SDN are hard to find and therefore we thought it could be great to propose a tool capable of generating ICS SDN-based topologies.

As shown in the Figure 4.12, in traditional networks there is a tight coupling between the control plane and the data plane on the same device. In contrast, according to the ONF (Open Network Foundation) definition, SDN is a decoupling of the network control plane and the data plane [9]. Therefore, SDN centralizes the control plane in a place named SDN controller or Network Operating System (NOS) [23], the controller has a global perspective of the entire network and thus ameliorates the management of networks. So the controller is in charge of all the routing decisions and controlling mechanisms. Furthermore the controller sends command to the data plane using the protocol OpenFlow which will be explained further bellow.

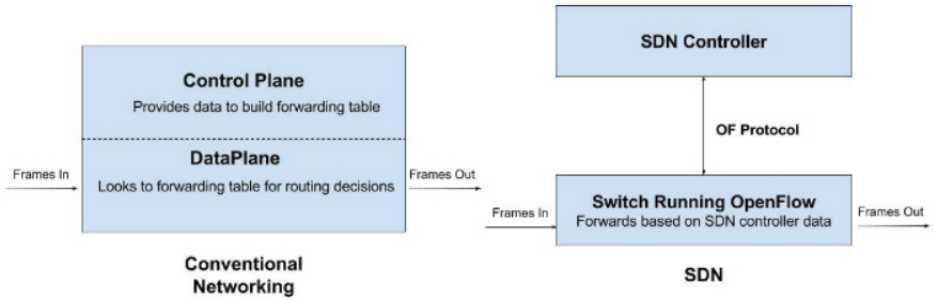

Figure 4.12: Conventional Networking vs SDN [9]

4.5.2 OpenFlow

As stated above and shown in the Figure 4.12 the communication between the controller and the different switches is handled by the protocol OpenFlow (OF) which is a one of the first SDN standards. OpenFlow provides an open protocol to program the flow in different switches and routers by exploiting the fact that most switches or routers contain flow-tables which are used mostly for firewall or QoS. Thus OpenFlow enabled devices are all devices that possess these flow tables and packets that go through these kind of switches have different actions applied to them according to the flow table. Therefore, as displayed in Figure 4.13, a flow table includes several flow entries, each of them are constituted of 3 fields : the rule field also called the match field then the action field and the statistics field. When a packet enter the switch, it will goes through the rules and when a match is found the action from this specific rule will be applied and the statistics field will keep information such as number of Packets or Bytes for each flow. On the other hand, if no match is found then as explained in the Figure 4.14, the switch will send a message to the controller. When the message arrives to the controller, it will decide which action should be taken and enter a new flow entry in the switch.

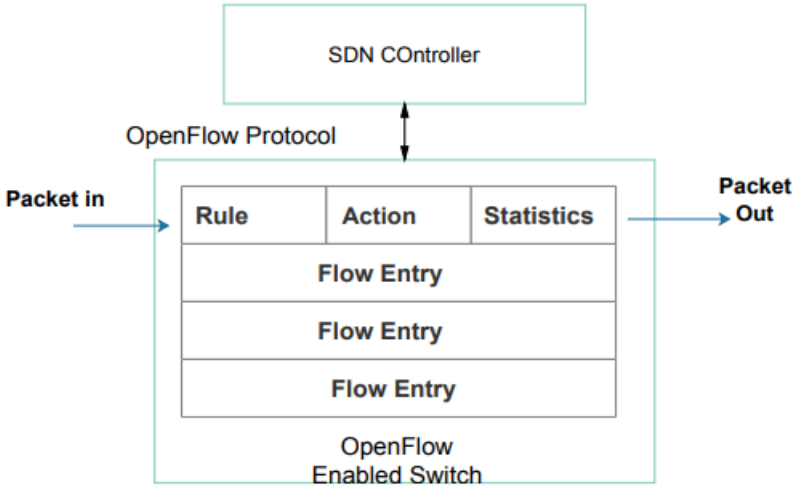

Figure 4.13: An abstract model of an OpenFlow switch [36]

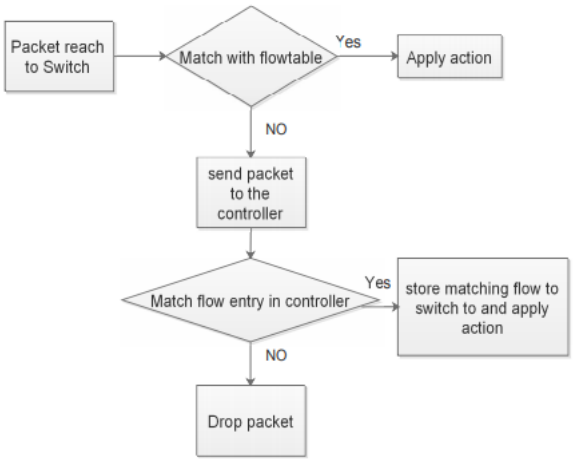

Figure 4.14: Flowchart of execution sequence of a packet in SDN [36]

4.5.3 Controller

As stated previously, in a Software Defined Network, the control plane is no longer coupled with the data plane and is now called Controller. This is what defines the nature of the SDN paradigm as it is responsible of the communication of all programmable elements of the network, moreover the controller has a global view of the network.

It exists several SDN controller with different architectures and implementations. In this Master-Thesis, we used the POX controller which is a Python based open source OpenFlow SDN controller [24]. POX comes pre-installed with Mininet and already have a few components, thus it can be used right away. Furthermore, POX developers may create new components for a more complex SDN controller. Components are additional Python programs that can be invoked through the command line when pox is starting as shown in the Figure 4.15 and each of them implements a network functionality [2].

```
sudo ./pox.py forwarding.l2_learning openflow.discovery openflow.spanning_tree --no-flood --hold-down
```

Figure 4.15: POX command line example

In the Figure 4.15, three components are used, these components are the components we use in our generation and are already implemented by POX :

- **forwarding.l2_learning** : This component makes Openflow switches act like Ethernet learning switches. Ethernet MAC addresses are learned in order to install multiple flows in the networks, for each pair of MAC address. Thus, different flows will be installed for different TCP connections. Furthermore, these flows are created with an idle timeout (which is ten seconds) and flows that have not processed any packet during the timeout period will be deleted.
- **openflow.discovery** : This component uses LLDP messages to discover the network topology. It also notices when a link goes down or up, this information is generally used by other components.
- **openflow.spanning_tree –no-flood –hold-down** : In case of the topology of the network would contain loop the Spanning Tree component is required. It works with the discovery component explained above to create a view of the network and build a spanning tree. In order to ensure that no packets are flooded in the network before the spanning tree is created, the arguments *no-flood* and *hold-down* are used. Thanks to the discovery component, the spanning tree also reacts to changes in the network by creating a new tree.

4.5.4 Hosts in Mininet

In Mininet, hosts are simple processes separated into different Linux's network namespaces. Each host has a virtual console, a virtual interface and a parent shell [29]. To support ICS devices, we add new classes that extend the Mininet Host class as shown in the Figure 4.16. We had to create classes for SCADA, PLC and MTU. Therefore, each ICS component has what a normal host has but runs different processes.

```
class PLC( Node ):

  def config( self, **params ):
    super( PLC, self ).config( **params )
    p=params['params']
    if p['version']=="openplc":
      self.cmd("( cd /home/ubuntu/OpenPLC\_v2/ && sudo nodejs server.js > /dev/
          null ) &")
      if p['logic']:
        self.cmd("python /home/ubuntu/Master-Thesis/src/openplc_upload.py -f "+p[
            'logic'])
      if p['process']:
        self.cmd("( "+p['process']+" > /dev/null ) &")

  def terminate( self ):
    super( PLC, self ).terminate()
```

Figure 4.16: Custom python script for PLC

4.5.5 Firewall

The POX controller does not contain an implemented firewall component and to stay as efficient as possible for the generation, we looked for a SDN firewall on top of POX that was easy to configure for an automated generation. We chose CloudFirewall an open source project created by Matan Ben-Yosef and Nir Parisian [31]. CloudFirewall is written in Python and is built above POX controller, therefore it is a SDN controller programmed to forward or drop certain TCP/UDP flows according to given rules. It also possesses a web user interface for quick visualization and monitoring. However, CloudFirewall came with a few problems. First, only one firewall could be used in the entire network and accordingly there was only one rules file for the network. The Second problem was that CloudFirewall did not tolerate subnetworking which was problematic as ICS network has different zones often with different subnetworks. Therefore, we had to modify the code of CloudFirewall to make it supports several firewalls and subnetworking, but we kept the fact that only one rules file is used for the network because we observed that as the firewall acts as the unique controller it is not problematic and facilitates the generation of firewalls.

Chapter 5

ICS Generation

This chapter presents the details about our generation and visualization of ICS. We discuss the different procedures to generate ICS instances and their visualization.

5.1 Generate ICSs

The most important point in our approach is to provide a way to generate and run ICS simulations as easy and automated as possible. We thought long about it and search tools that could be used, modified or created to achieve this goal.

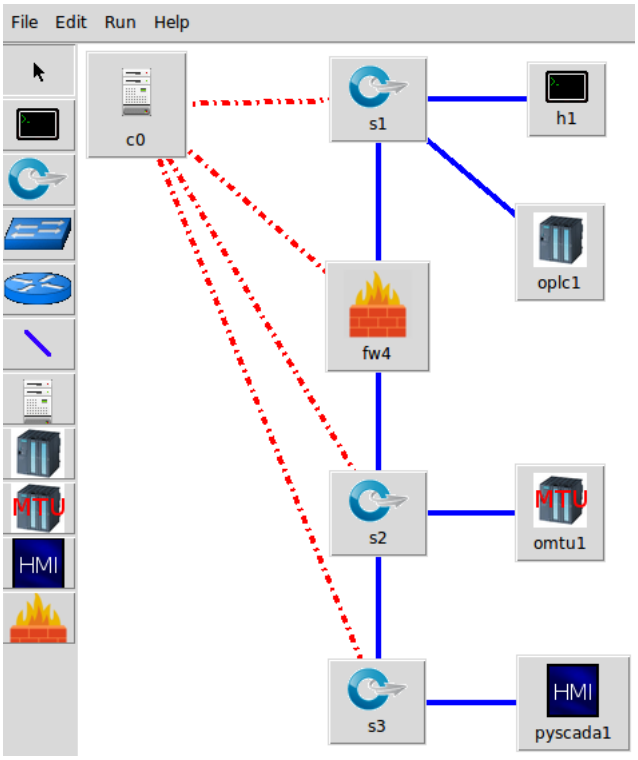

Figure 5.1: Example of an ICS creation in MiniEdit

While learning and testing the network emulator Mininet, we found an interesting tool provided with it. MiniEdit is a tool with a simple user interface to help with the configuration of networks that will then run in Mininet. It provides an easy to use drag and drop functionality to create network topologies with switches, routers, hosts and network links to connect them together. Switches in Mininet run the OpenFlow protocol and need to be connect to a controller

to actually work. Default controllers are provided with Mininet, the OpenFlow reference controller (ref), the Open vSwitch (ovs), and the deprecated NOX controller. External controllers are also supported like the Ryu, Floodlight, OpenDaylight and POX controller. Such configuration creates a SDN environment that is ready to run. A right click with the mouse on an element in the topology will open a new window to setup some properties. On a host, properties like hostname, IP address, default route, virtual LANs (VLANs) and more are accessible. Switches and controllers provide configurations for OpenFlow. Every network link can be configured to simulate bandwidth limit, delay, loss and jitter in the transmission of packets. General properties are also available in the menus like the base IP address used to generate unique IP addresses for every host in the topology. MiniEdit can export the created topology with all the properties to a JSON file and can load it back to make new modifications. It can also export the topology to a Python script that is ready to run with Mininet. The facilities of creation and export is what made our choice for Mininet and MiniEdit. Also, they are written in Python which makes modifications easy to perform, the JSON files are encoded in a human readable format and can be reused in other programs which is convenient. We decided to modify MiniEdit to support ICS components in the same way as the others, PLCs, MTUs, SCADA and firewalls can now be selected in the left tool bar and added to the topology (Figure 5.1). We modified the export to an augmented JSON format and customized the final Python script that will run the ICS with Mininet. An external tool was created to generate the Python script from the JSON because other files containing the MTU and firewall configurations also need to be generated.

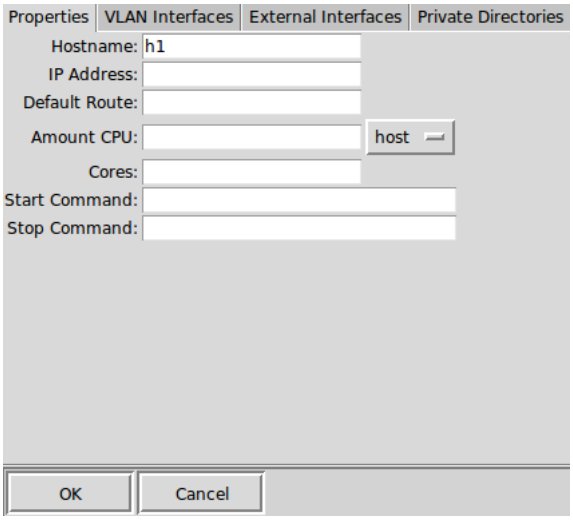

Figure 5.2: Default option menu for a basic host

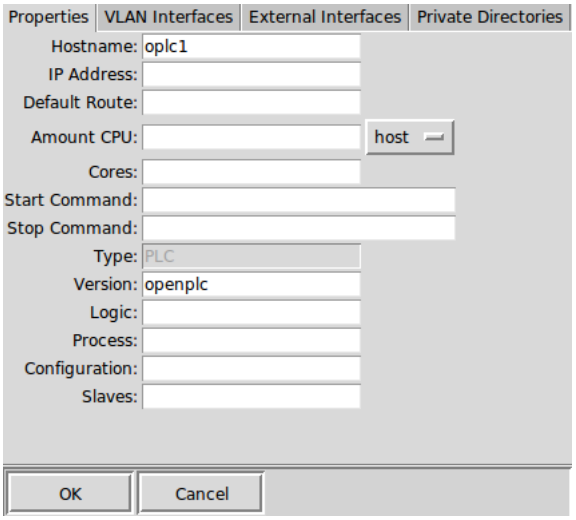

Figure 5.3: Augmented option menu for an ICS host

The Figure 5.2 presents the properties that are accessible for each host present in the topology. Hosts that run ICS components have access to additional properties. The hostname, type and version are filled automatically while the logic, process, configuration and slaves fields can be filled by the user. As some fields are mandatory, the tool that generates the runnable Python script will try to fill the missing information and warn the user.

Based on our work on MiniEdit, we also created a version of our tool to generate with a set of arguments all the files needed to start an ICS with Mininet thus avoiding the manual creation and export to JSON. The strengths of this version are that the creation of large ICS network can be performed just by requesting the numbers of ICS component in the arguments. However, we only propose a few topologies for this generation and therefore to obtain tailor made network some modifications must be done manually.

For the physical process simulation, we modified the program SimLink that communicates with the driver Simulink of OpenPLC (already discussed in section 4.3). A simple template is

provided to program in C++ some kind of physical process behavior that can read the outputs and generate the inputs of an OpenPLC instance. Each OpenPLC instance can get its own or one physical process simulation can communicate with multiple PLCs at once. A Bash script simplifies the compilation and avoid touching the Makefile.

For PyScada, we do not have any automated scripts or modifications. It is already a software where everything can be configured via the user interface. We have an external Bash script to dump or load the SQL database of PyScada. This is useful to save databases and to swap them between different ICS simulations.

As OpenPLC needs a logic code to simulate a working PLC, we provide some ladder logic examples created in PLCopen Editor, exported in ST format and also saved in the PLCopen XML format for possible modifications (already discussed in section 4.3). The logic codes can be reused in any of our ICS simulations and also on any compatible devices.

When everything is ready, the generated Python script will start Mininet and every ICS components will launch automatically. First, the network with its switches, basic hosts, links and controller. Switches that integrate a firewall will start monitoring and block prohibited network traffic. Then hosts will run commands to become specialized hosts. The PLCs start OpenPLC with the Simulink driver, receive their logic to run and start their attached physical process. The MTUs start OpenPLC with the Modbus driver, receive their configuration to communicate with the PLCs. The PyScada service is launched and daemonized, it will start communicating with devices setup in its database, start the web service for the HMIs and perform automated tasks like logging activities.

With all this work, we now have a way to generate and simulate ICSs in a friendly manner. ICS components like PLCs, MTUs and SCADA and basic network components like switches and controllers are ready to start working. Everything is general enough so that more can always be added in the future and it also makes possible the simulation of ICSs with different scenarios.

5.2 Visualize ICSs

When running a simulation, we also wanted a way to easily visualize and interact with the elements in the network.

MiniNAM is a network animator for Mininet (hence the letters NAM). It provides an user interface that can display the switches, hosts, controllers that are currently running in Mininet and the network links between them. Its particularity is the fact that it can display packet flows in real-time, each network packet is animated along the link between the sender and the destination. Color codes are used to visually distinguish packets, each of them has the color of the sender or receiver host plus the color of its type like ARP, TCP, UDP, ICMP. This feature helps to quickly debug networks like checking that data is actually being transmitted between hosts on desired links. Filters are also proposed to hide packets of a certain type or those that are originated or destinated to certain IP or MAC addresses. This is useful when interested for only some particular flows in the network. MiniNAM also provides a summary of all the network interfaces created by Mininet. Each interface provides information to which interface it is linked with, the type of the node that own it, the associated IP and MAC addresses and statistics about the number of packets and bytes sent and received. A right click on an element of the topology gives access to more options like opening a new terminal on a host to then run commands or start an installed program. The options to connect and disconnect links are also available to test the network behavior during link failures and repairs in real time. All those functionalities makes MiniNAM a great tool that we decided to use in our project.

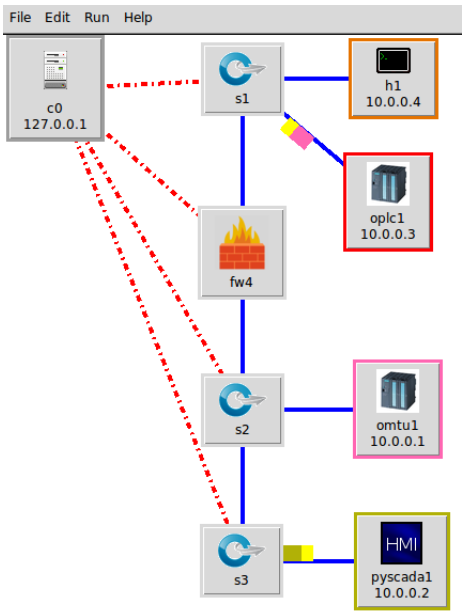

Figure 5.4: Example of an ICS running in Mini-NAM

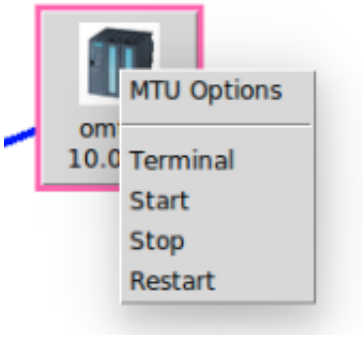

Figure 5.5: New options to start/stop/restart MTUs and PLCs

We applied modifications to MiniNAM to support our ICS components and some new features. As it is an open source Python program which, after looking at the source code, is based on the code of MiniEdit (presented in section 5.1) the difficulties to perform changes to it were no different from those performed on MiniEdit. We added the support for our PLCs, MTUs, SCADA and firewalls on the visualization (Figure 5.4). It is now possible to start MiniNAM with a Python script generated from our tools from the previous section to get an overview of the ICS running in Mininet. The display of the IP address of each host was added to help quickly find devices and detect packet flows senders and receivers. Packets that use the port 502 reserved for Modbus are now colored in yellow and the color can be changed in the preference menu. A right click with the mouse on a PLC or MTU brings new functionalities to start and stop the device, those commands are sent to the OpenPLC instance running on the selected host (Figure 5.5). This is way faster than opening a new terminal on the host, then starting a web browser to access the web interface of OpenPLC. However, to access the SCADA system and HMIs that PyScada provides a web browser is still needed. We installed Midori, a very lightweight web browser, that we can launch on a host inside Mininet.

Interface	Linked To	Node Type	IP Address	MAC Address	TXP	RXP	TXB	RXB
h1-eth0	s1-eth2	Host	10.0.0.4	86:8d:4e:3e:de:d4	0	0	0	0
omtu1-eth0	s2-eth1	MTU	10.0.0.1	22:47:75:32:78:91	38788	34458	3013614	2659541
oplc1-eth0	s1-eth1	PLC	10.0.0.3	ca:fd:f8:73:02:a2	34054	38547	2630609	2995250
pyscada1-eth0	s3-eth2	SCADA	10.0.0.2	c2:ac:b0:30:b3:51	432	217	31088	16470
fw4-eth1	s1-eth3	OVSSwitch	None	6e:4b:2c:f6:94:0f	38559	34042	2996196	2629687
fw4-eth2	s2-eth3	OVSSwitch	None	ba:c8:d0:d8:87:20	34040	38565	2629533	2996666
s1-eth1	oplc1-eth0	OVSSwitch	None	f2:cf:f0:b4:9f:c7	38547	34054	2995250	2630609
s1-eth2	h1-eth0	OVSSwitch	None	c6:5e:68:cc:f8:f5	0	0	0	0
s1-eth3	fw4-eth1	OVSSwitch	None	6e:d6:49:e3:b0:1b	34049	38552	2630224	2995643
s2-eth1	omtu1-eth0	OVSSwitch	None	a6:37:8c:57:34:3d	34458	38788	2659541	3013614
s2-eth2	s3-eth1	OVSSwitch	None	92:8f:05:3c:a2:8e	217	432	16470	31088
s2-eth3	fw4-eth2	OVSSwitch	None	3a:d6:d9:8b:87:c7	38569	34031	2996986	2628837
s3-eth1	s2-eth2	OVSSwitch	None	ee:57:bd:53:47:f4	432	217	31088	16470
s3-eth2	pyscada1-eth0	OVSSwitch	None	5a:67:88:2f:57:bd	217	432	16470	31088

Figure 5.6: Interfaces summary in MiniNAM

```
opennet> dump
<Host h1: h1-eth0:10.0.0.4 pid=5237>
<MTU omtu1: omtu1-eth0:10.0.0.1 pid=5242>
<PLC oplc1: oplc1-eth0:10.0.0.3 pid=5246>
<SCADA pyscada1: pyscada1-eth0:10.0.0.2 pid=5250>
<OVSSwitch fw4: lo:127.0.0.1,fw4-eth1:None,fw4-eth2:None pid=5255>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=5258>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None pid=5261>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None pid=5264>
opennet> net
h1 h1-eth0:s1-eth2
omtu1 omtu1-eth0:s2-eth1
oplc1 oplc1-eth0:s1-eth1
pyscada1 pyscada1-eth0:s3-eth2
fw4 lo:  fw4-eth1:s1-eth3 fw4-eth2:s2-eth3
s1 lo:  s1-eth1:oplc1-eth0 s1-eth2:h1-eth0 s1-eth3:fw4-eth1
s2 lo:  s2-eth1:omtu1-eth0 s2-eth2:s3-eth1 s2-eth3:fw4-eth2
s3 lo:  s3-eth1:s2-eth2 s3-eth2:pyscada1-eth0
opennet>
```

Figure 5.7: Interfaces summary in Mininet

MiniNAM provides a visual of the network (Figure 5.4) and an easy to read interfaces summary (Figure 5.6) which are powerful during tests and troubleshoots. When using the basic Mininet installation, everything is done through command lines and the same information can be retrieved from the results of the 2 commands shown in Figure 5.7. The command *dump* gives the node types, the names and IPs and the command *net* gives the network interface links. There is no doubt that MiniNAM is better at providing information in a user friendly way.

When starting an ICS simulation with MiniNAM, packets should start moving in the network and be visible on the user interface if the communications between PLCs, MTUs and SCADA are correctly configured. All those packets are part of the Modbus/TCP exchanges currently happening between the devices. MiniNAM is not optimized for heavy traffic, we encountered heavy delays in the animations of packets moving in the network but it only affects their visual not their actual transmission that is still fast. We run our project inside a virtual machine which is the main reason for the slowdowns, MiniNAM relies on a lot of threaded operations that cannot access the CPU fast enough. We use a server image with no graphical user interface (GUI) which is fine as most of the project uses command lines. To display MiniNAM and other programs that provide a GUI like the web browser Midori, we use Xterm that forward the GUI information from the guest OS to the host OS. This is the second reason why slowdowns are happening. We do not consider this as a restrictive problem, only the animations are affected and the rest of the program is working as expected. The animations are useful to quickly check if something is actually happening in the network and other tools are still needed for more in depth debug.

5.3 Steps to Generate ICSs

As explain in the section 5.1, we propose two ways of generating an ICS network, one using MiniEdit as a base and the other by passing arguments to the generation script. Both of them have their strengths and weaknesses that will be discussed here alongside the steps to follow in order to use these generations.

5.3.1 Semi-Automated Generation

As stated above, this generation use MiniEdit as a base, it is great for tailor made network but as every ICS components, switches and links have to be placed manually, it does not suit a large network generation. It also needs more steps than the other generation, first a network needs to be created using miniEdit as shown in the Figure 5.8. The network can be easily built by drag

and drop the components and create links between these components. Once the network has been created, it can be saved in the format *.mn* which is actually a JSON file as displayed in Figure 5.9. Then this JSON file is used as a parameter for our generation Script that will create the Python script for Mininet.

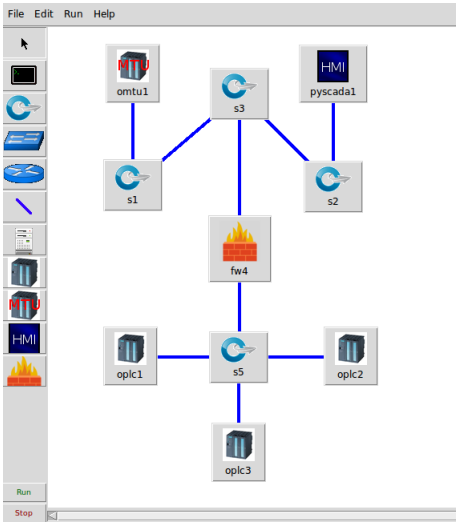

Figure 5.8: MiniEdit Example


```
"hosts": [
    {
        "number": "3",
        "opts": {
            "cls": {
                "logic": "/home/ubuntu/Master-Thesis/extras/Blank.st",
                "process": "",
                "script": "",
                "type": "PLC",
                "version": "openplc"
            },
            "defaultRoute": "None",
            "hostname": "oplc1",
            "ip": "10.0.0.1/8",
            "nodeNum": 3,
            "sched": "host"
        },
        "x": "103.0",
        "y": "411.0"
    },
    {
        "number": "4",
        "opts": {
            "cls": {
                "logic": "/home/ubuntu/Master-Thesis/extras/Blank.st",
                "process": "",
                "script": "",
                "type": "PLC",
                "version": "openplc"
            },
            "defaultRoute": "None",
            "hostname": "oplc2",
            "ip": "10.0.0.2/8",
            "nodeNum": 4,
            "sched": "host"
        },
        "x": "378.0",
        "y": "411.0"
    },
```


Figure 5.9: JSON from MiniEdit

5.3.2 Automated Generation

This generation is the one using the parameters of the generation script, it can create large topologies just by giving some parameters. Therefore, if there is a need to create large specific ICS network, this can be use as a base but will require some manual adjustments in the end to match the requirement. For example, configuration for Firewalls or MTU are already created with a default approach and cannot be modified using parameters. Furthermore, we also propose two different topologies that cannot be adjusted in parameters but hopefully can be easily changed after the generation by modifying the Python file produced. However its ease of use , as it only requires two steps for the generation at most, makes it really convenient for testing.

First, a folder can be created to store the generated configuration files but it is not compulsory. Then the generation script must be run inside this folder with selected parameters as shown in the Figure 5.10. Parameters have a default value depending of the topology chosen for the generation, this will be detailed later in the section 5.4. The generation will create a Python script and configuration files that can be run by Mininet just like the Semi-Automated Generation.

```
$ python /home/ubuntu/Master-Thesis/src/generation.py -a -t default -p 7 -s 6 -
    z 6 -m 4
```

Figure 5.10: Example Generation with Parameters

Here is an exhaustive list of all parameters supported by the script :

- **--auto** or **-a** : Switch the script to the Automated mode
- **--topo** or **-t** [Topology Name] : Which topology will be used for the generation
- **--scada** or **-d** : Scada server in the network
- **--plcs** or **-p** [Number] : Number of PLCs

- **--mtu** or **-m** [Number] : Number of MTUs
- **--switches** or **-s** [Number] : Number of switches
- **--zones** or **-z** [Number] : Number of Zones
- **--hosts** or **-u** [Number] : Number of Hosts

5.3.3 Configurations Generation and Launch

Once the Python script containing the network configuration has been generated, both generations work the same way. Alongside the Python script, at least two configuration files have been created, the first contains the rules and the configuration for the firewall. The other corresponds to the configurations for the Master Terminal Unit, more files may be generated depending of the number of MTUs as each MTU needs a different configuration file. Now that everything is generated, the network can be simulated using two terminals, one for Mininet or MiniNAM as shown in Figure 5.11 and another for POX or CloudFirewall as displayed in the Figure 5.12. When launching Mininet or MiniNAM, the python script containing the generated network is used as a custom network and we need to tell Mininet or MiniNAM that the controller is a remote controller. On the other hand, for launching POX, a bash file has been created because the command line can be quite long and firewalls ID must be given to the POX component *forwarding.l2_learning* to make sure that they will not act as Openflow switches as shown in the Figure 5.13.

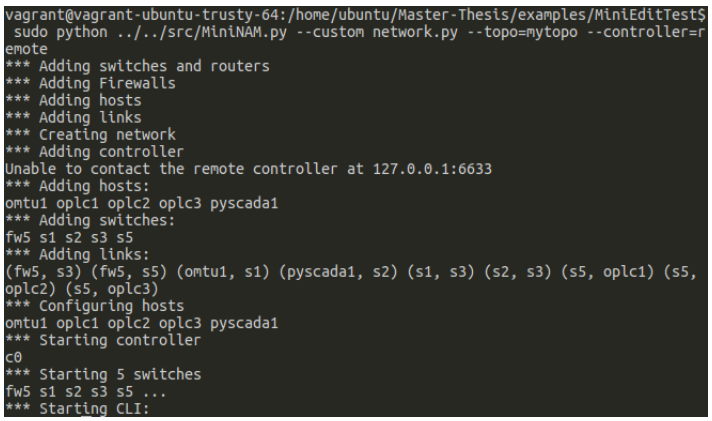

Figure 5.11: Terminal use for MiniNAM and Mininet

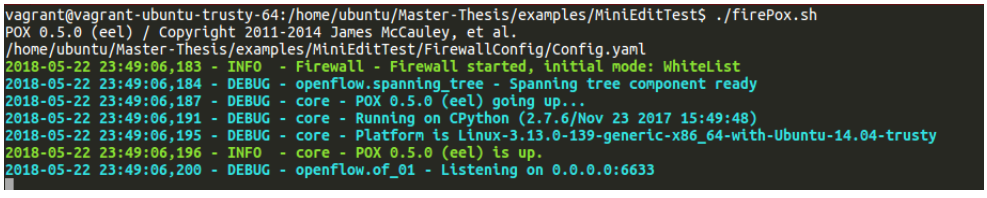

Figure 5.12: Terminal Use for POX and CloudFirewall

```
python ./run_pox.py log.level --DEBUG of_firewall --config=/home/ubuntu/Master-
    Thesis/examples/RingTopo/FirewallConfig/Config.yaml --event=/home/ubuntu/
    Master-Thesis/examples/RingTopo/events.bin forwarding.l2_learning --ignore
    =7,8,9,a,b,c,d openflow.discovery openflow.spanning_tree --no-flood --hold-
    down
```

Figure 5.13: Example of the generated command in the Bash file to run POX and CloudFirewall

5.4 Presentation of Generated ICSs

In this section the three topologies produced by the automated generation will be explained in further detail. As a reminder, these three topologies can be used as a helpful foundation if a large network is needed and modifications can still be done once the Python script has been generated. Moreover, we tried to design these topologies as close as possible to the topologies that can be found in the industry, however they may slightly differ in order to facilitate the generation.

5.4.1 Star Topology

The first topology is shown in the Figure 5.15 and was generated from the command line in Figure 5.14. It is a star topology based on the topology diplayed in the Figure 5.16 where the HMIs, MTUs, PLCs are all connected with one central switch. The main elements of this generation are the central switch that connects each zone and the fact that every zones are in different subnetworks. Every subnetwork possesses a range of IP which allows firewalls to block a certain zone or connections between some zones. In our topology, as we were looking for a generation as general as possible, we splitted all the PLCs and switches evenly between all zones and to ensure some security per zone we added a firewall between the zone and the central switch. In this case , as there is only one MTU, it manages all the PLCs, otherwise PLCs are also equally divided among MTUs. Concerning the firewalls,they use a white-list rule that only accepts TCP messages on port 502 which is the Modbus port.

```
python /home/ubuntu/Master-Thesis/src/generation.py -a -t star -p 8 -s 9 -z 6 -m
    1 -d
```

Figure 5.14: Star topology command line example

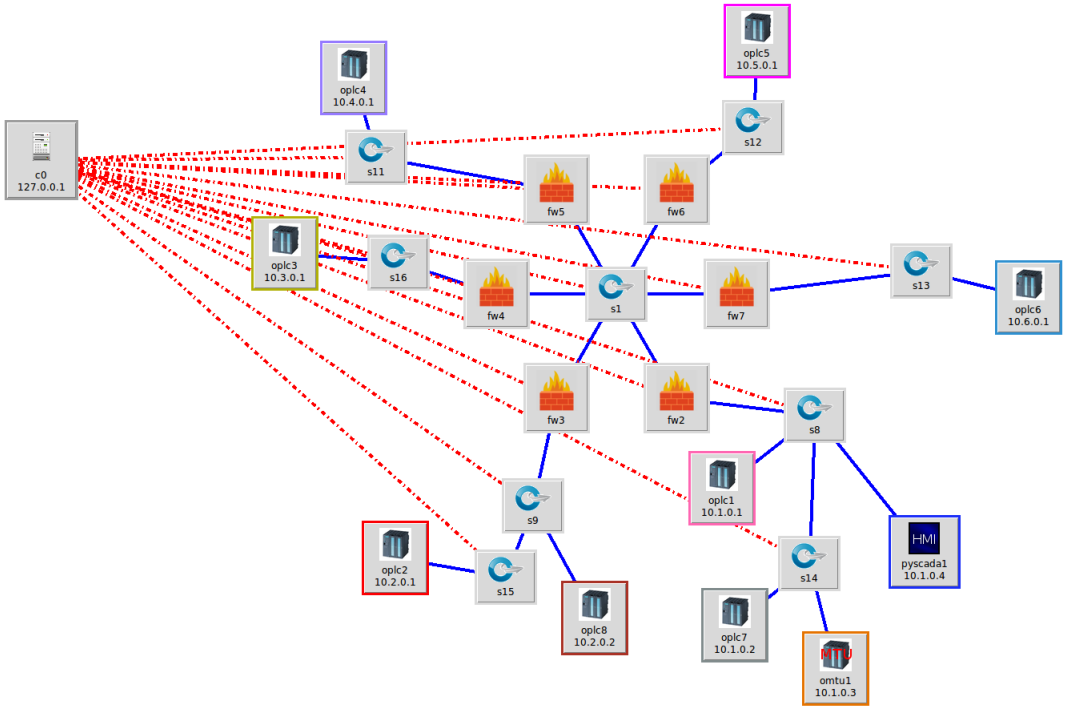

Figure 5.15: Star topology from the generation

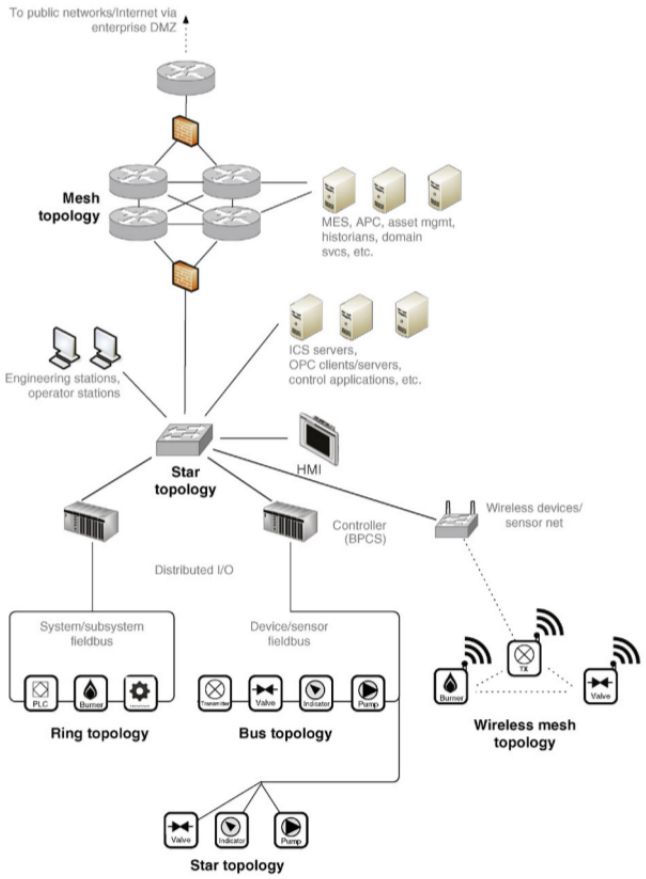

Figure 5.16: Common network topologies as used in industrial network [25]

5.4.2 Classic Topology

The command line in Figure 5.17 generates the second topology displayed in Figure 5.19 which is based on the idea shown in the Figure 5.19 where there are three zones each separated by a firewall and with a control zone possessing several smaller zones that contains PLCs. In our topology, since we are not focused on the Business zone, we simply represent it using hosts. The supervisory zone is showed by the MTU and the SCADA server while the control zone contains all the zones where there are PLCs. The PLCs are equally distributed among the different smaller zones inside the control zone, each of these zones have a specific range of IP to allow some zones to communicate with other or not. On the other hand, switches are divided between all the zones with a certainty that each zone contains at least one switch. Just like the previous topology, in this case there is only one MTU but if more MTUs would be requested, PLCs would be splitted evenly between them. Concerning the firewalls, they act the same way as they do in the previous topology.

```
python /home/ubuntu/Master-Thesis/src/generation.py -a -t classic -p 8 -s 9 -z 6
    -m 1 -u 4 -d
```

Figure 5.17: Classic topology command line example

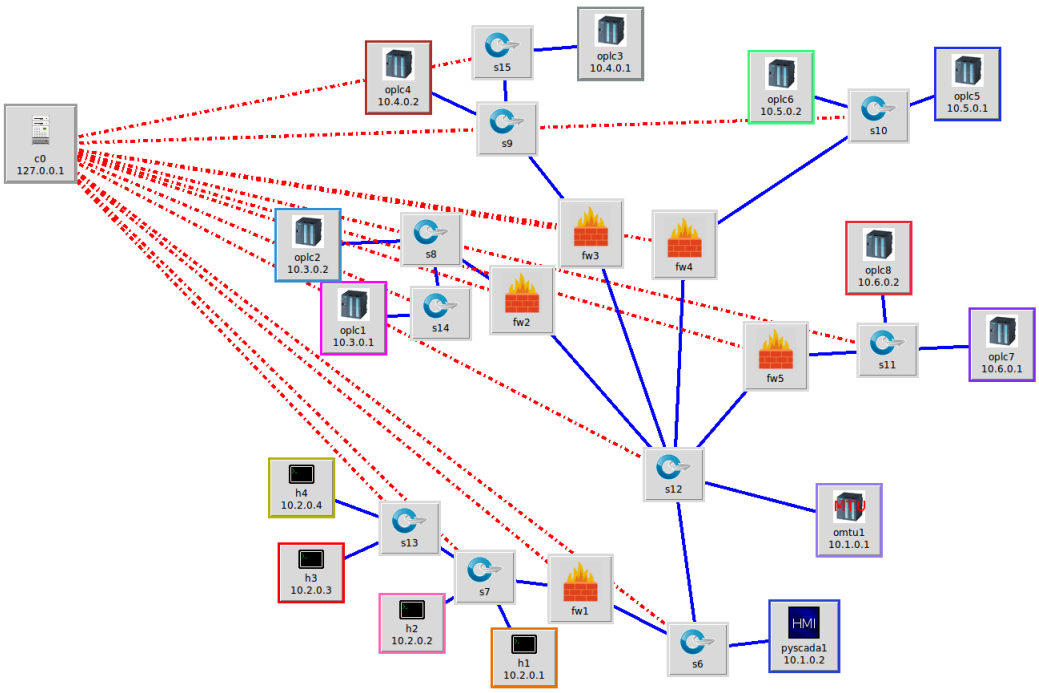

Figure 5.18: Classic topology from our generation

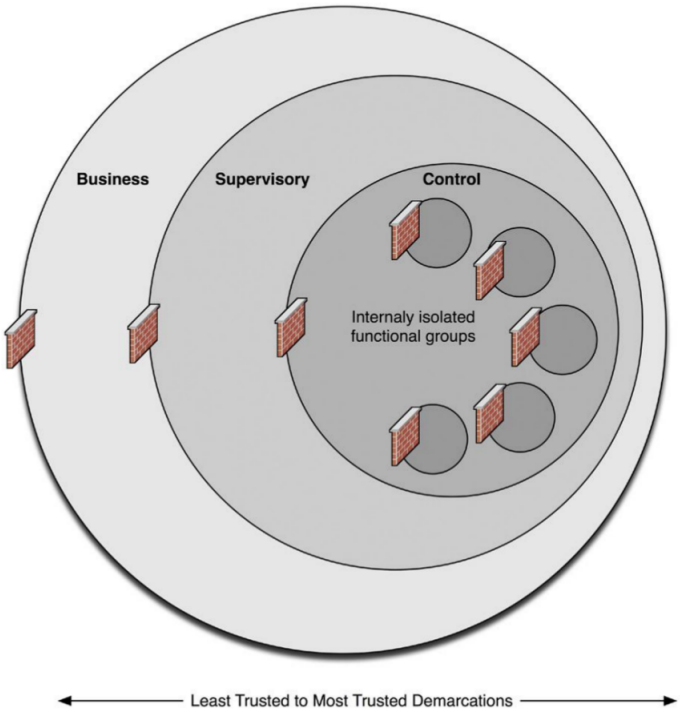

Figure 5.19: Layers of Protection [25]

5.4.3 Ring Topology

The third topology is shown in Figure 5.21, generated by the command in Figure 5.20 and based on the topology in Figure 5.22. This topology resembles and acts like the classic topology but with core switches linked in circle. In the command we only requested nine switches but there are fourteen switches in the example because core switches are created according to zones, one switch per zone.

```
python /home/ubuntu/Master-Thesis/src/generation.py -a -t ring -p 8 -s 9 -z 6 -m
    3 -u 3 -d
```

Figure 5.20: Ring topology command line example

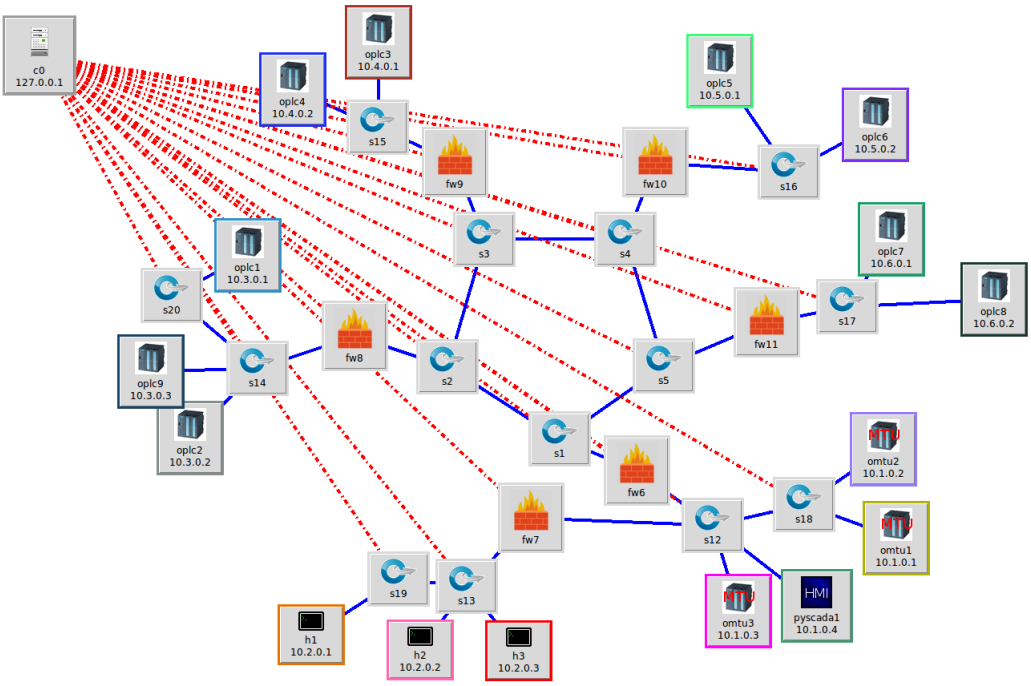

Figure 5.21: Ring topology from our generation

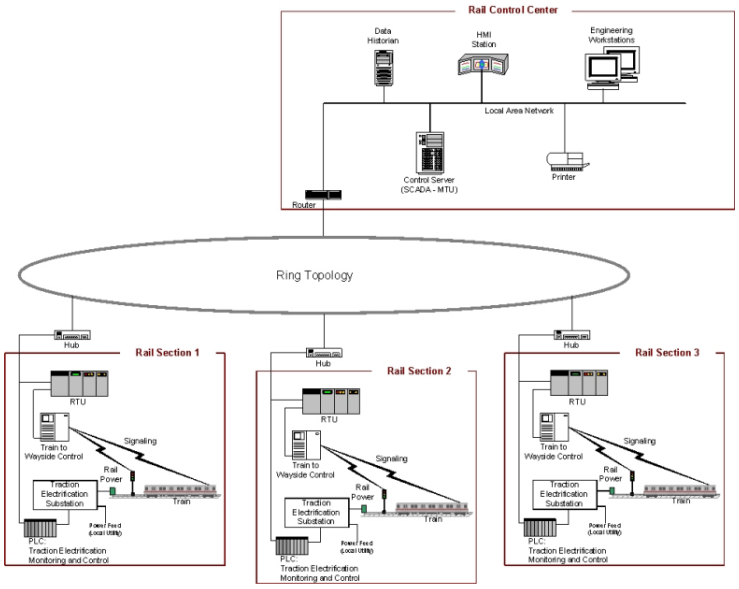

Figure 5.22: Ring Topology [41]

5.5 Measurements

In this section we will explain the procedure for our measurements, the results and the limitations.

5.5.1 Methodology

In order to test our generations, we need to launch at least three concurrent processes. First a processes that run Mininet with our generated network, then a process for the Controller and the Firewall as they are both linked, finally a processes to dump network packets. To do that, we created a Python script which generates several topologies while varying their parameters and runs these three processes on each topology. The network packets are dump into a *.pcap* file using the network protocol analyzer Tshark [45], then handled by the script that computes the mean time delay between every Modbus requests and responses. We made our test on the Ubuntu 14.04 virtual machine with 2 GB of RAM and one CPU.

5.5.2 Results

Our goal with these measurements was to see the impact of the number of PLC on the network. To make our tests we have varied the PLC number from 10 up to 45 and the topology, while fixing the number of MTU to one and divided the network in four zones. The results obtained are shown in the Figure 5.23, where we can see that the delay slightly increases according to the number of PLC for each topology. This delay can be explained by the increasing number of TCP and OpenFlow packets which request the controller to create new flows which obviously takes time. Furthermore, we can also observe that the ring topology does not perform well compared to the two others topologies because it has more links and switches being flooded as it needs more switches and links in its core zone than the others.

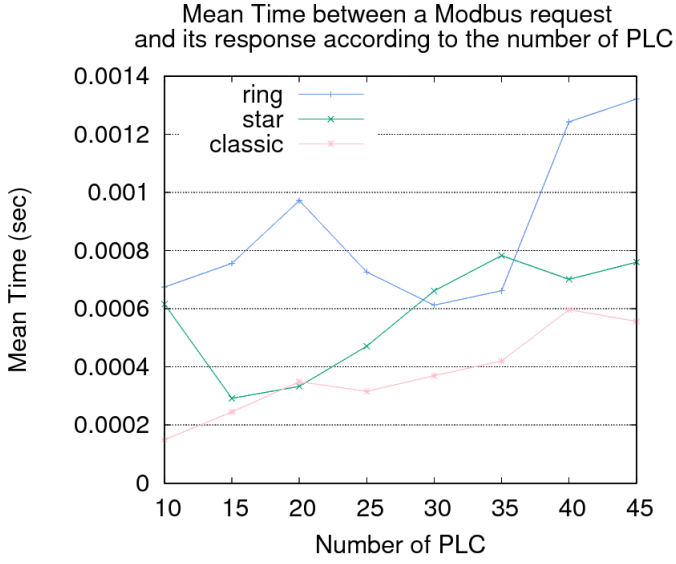

Figure 5.23: Mean Time between Modbus requests and responses according to the number of PLC

5.5.3 Limitations

During our tests we also detect a few limitations of our generation. On this virtual machine, we came to the conclusion that the maximum acceptable number of PLC was 45 because PLC creation passed this value can take several minutes. As explain in the section 4.5, each host is in fact a process in a linux's network namespace and when Mininet launch a PLC, every PLCs created before are already running and consuming the CPU. Thus the more PLCs are created, the more time it will take for each.

We also observed that when we requested more than 4 MTUs, as the MTUs are launching almost at the same time flood the network with thousands of packets which results in too much new flow entries requests that cannot be handled correctly by the Controller.

Chapter 6

ICS Attacks and Protections

In this chapter, we present and perform attacks on a running ICS simulation and test protections against them.

6.1 Setting

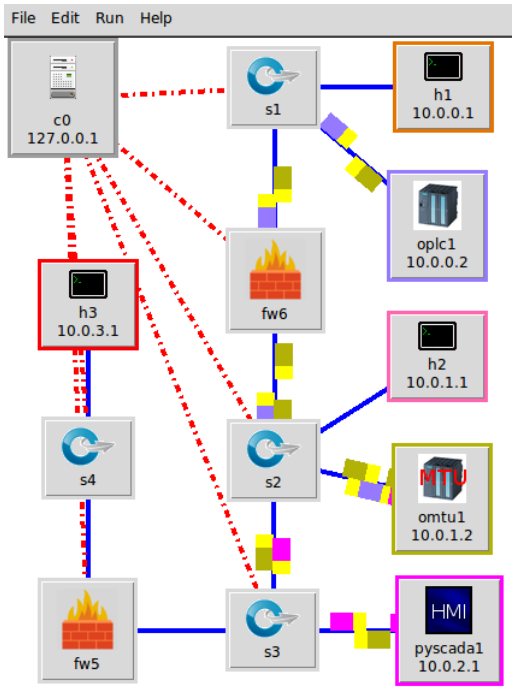

Figure 6.1: Simple ICS used to test attacks and protections

We use the simple ICS instance shown in Figure 6.1. It consists of 3 zones or levels separated by 2 firewalls. The first zone represents the field zone and contains a PLC, the second represents the control zone and contains a MTU and a SCADA system, the third is a more external zone from ICS environment like the corporate zone. Each zone also contains a default host which is useful to test scenarios with infected devices located inside the network or with rogue agents physically connected to the network. This is the assumption we took, we perform the attacks at different levels inside the network, behind the firewalls, to show the limitations of such protection. The firewalls run with the automatically generated rules, they block everything that is not TCP traffic on port 502 reserved for Modbus.

6.2 Goals

The goals of our attacks all converge to a single end, the disruption of Modbus communications. If Modbus compatible devices are discovered in the network, we want to be able to communicate with them by reading data and modify their behaviors by writing data, monitor running Modbus communications like between a PLC and a MTU, block them or modify the exchanged messages to alter their effects. Finally, protections are setup through firewall rules in the intention to stop or mitigate those attacks.

6.3 Discovery Phase

Before conducting any intrusion, the attacker needs to gain some knowledge about the devices connected to the network, specifically the Modbus devices in this scenario.

```
nmap -n -p T:502 -max-rate 1 10.0.0.0/22
```

Figure 6.2: Nmap scan on TCP port 502

Nmap is a well known tool to discover hosts and services available in a network. The command shown in Figure 6.2 is used as a quick way to discover possible Modbus compatible devices. The option -n avoids DNS resolution, -p will try to establish a TCP connection on port 502 on detected alive host, -max-rate limits to 1 packet emission per second mostly used for slow scan to avoid detection, here this option is used because of another reason that will be explained later. At the end of the command, the subnet to scan is specified.

```
root@vagrant-ubuntu-trusty-64:/home/ubuntu/Master-Thesis/examples/TestingAttack
s# nmap -n -p T:502 -max-rate 1 10.0.0.0/22

Starting Nmap 6.40 ( http://nmap.org ) at 2018-06-06 21:40 CST
Nmap scan report for 10.0.0.1
Host is up (0.088s latency).
PORT    STATE  SERVICE
502/tcp closed asa-appl-proto
MAC Address: EE:EB:73:23:B8:0C (Unknown)

Nmap scan report for 10.0.0.2
Host is up (0.063s latency).
PORT    STATE SERVICE
502/tcp open  asa-appl-proto
MAC Address: 16:46:77:50:43:7D (Unknown)

Nmap scan report for 10.0.1.1
Host is up (0.097s latency).
PORT    STATE  SERVICE
502/tcp closed asa-appl-proto
MAC Address: 66:81:5A:58:61:EC (Unknown)

Nmap scan report for 10.0.1.2
Host is up (0.10s latency).
PORT    STATE SERVICE
502/tcp open  asa-appl-proto
MAC Address: 22:78:6F:DE:CB:5E (Unknown)

Nmap scan report for 10.0.2.1
Host is up (0.081s latency).
PORT    STATE  SERVICE
502/tcp closed asa-appl-proto
MAC Address: 9E:31:25:68:95:E9 (Unknown)

Nmap scan report for 10.0.3.1
Host is up (0.000030s latency).
PORT    STATE  SERVICE
502/tcp closed asa-appl-proto

Nmap done: 1024 IP addresses (6 hosts up) scanned in 2148.69 seconds
```

Figure 6.3: Results of running the Nmap scan from host h3

For now, we take the assumption that the attacker is connected in the corporate zone, the highest level. He is represented by the host h3 (IP address 10.0.3.1) that he was able to infect to take full control or was simply able to physically connect his own computer to the network. Thanks to the information from the network configuration, the attacker knows the subnet to scan. He runs the Modbus scan and gets the results shown in Figure 6.3. The scan found 6 alive hosts, including the attacker machine, and returned their IP address, MAC address and the state of the port 502 for TCP. 2 hosts are reported with this port to be open, 10.0.0.2 (oplc1) and 10.0.1.2 (omtu1), the others are reported closed. Those 2 hosts now become the primary targets but the attacker is still not sure if a Modbus service is running on the port and what type of devices he is facing and their capabilities. This will be answered in the next phase by analyzing the devices.

No.	Time	Source	Destination	Protocol	Length	Info
2648	2136.000160(	de:0d:87:ff:e7:db	Broadcast	ARP	42	Who has 10.0.0.128? Tell 10.0.3.1
2649	2137.000163(	de:0d:87:ff:e7:db	Broadcast	ARP	42	Who has 10.0.0.128? Tell 10.0.3.1
2651	2138.000159(	de:0d:87:ff:e7:db	Broadcast	ARP	42	Who has 10.0.1.8? Tell 10.0.3.1
2652	2139.000160(	de:0d:87:ff:e7:db	Broadcast	ARP	42	Who has 10.0.1.8? Tell 10.0.3.1
2653	2140.000335(	de:0d:87:ff:e7:db	Broadcast	ARP	42	Who has 10.0.0.22? Tell 10.0.3.1
2654	2141.000162(	de:0d:87:ff:e7:db	Broadcast	ARP	42	Who has 10.0.1.1? Tell 10.0.3.1
2655	2141.047791(	66:81:5a:58:61:ec	de:0d:87:ff:e7:db	ARP	42	10.0.1.1 is at 66:81:5a:58:61:ec
2656	2142.000469(	de:0d:87:ff:e7:db	Broadcast	ARP	42	Who has 10.0.0.22? Tell 10.0.3.1
2658	2143.000900(	10.0.3.1	10.0.0.2	TCP	58	48704→502 [SYN] Seq=0 Win=1024 Len=0 MSS=146(
2659	2143.220390(	10.0.0.2	10.0.3.1	TCP	58	502→48704 [SYN, ACK] Seq=0 Ack=1 Win=29200 L(
2660	2143.220412(	10.0.3.1	10.0.0.2	TCP	54	48704→502 [RST] Seq=1 Win=0 Len=0
2661	2144.109579(	10.0.3.1	10.0.1.1	TCP	58	48704→502 [SYN] Seq=0 Win=1024 Len=0 MSS=146(
2662	2144.235971(	10.0.1.1	10.0.3.1	TCP	54	502→48704 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
2663	2145.001009(	de:0d:87:ff:e7:db	Broadcast	ARP	42	Who has 10.0.2.1? Tell 10.0.3.1
2664	2145.144799(	9e:31:25:68:95:e9	de:0d:87:ff:e7:db	ARP	42	10.0.2.1 is at 9e:31:25:68:95:e9
2665	2145.144810(	10.0.3.1	10.0.2.1	TCP	58	48704→502 [SYN] Seq=0 Win=1024 Len=0 MSS=146(
2666	2145.244038(	10.0.2.1	10.0.3.1	TCP	54	502→48704 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
2667	2146.001087(	10.0.3.1	10.0.0.1	TCP	58	48704→502 [SYN] Seq=0 Win=1024 Len=0 MSS=146(
2668	2146.187659(	10.0.0.1	10.0.3.1	TCP	54	502→48704 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
2669	2147.001170(	10.0.3.1	10.0.1.2	TCP	58	48704→502 [SYN] Seq=0 Win=1024 Len=0 MSS=146(
2670	2147.183631(	10.0.1.2	10.0.3.1	TCP	58	502→48704 [SYN, ACK] Seq=0 Ack=1 Win=29200 L(
2671	2147.183653(	10.0.3.1	10.0.1.2	TCP	54	48704→502 [RST] Seq=1 Win=0 Len=0

Figure 6.4: Messages captured with Wireshark during the Nmap scan

The Figure 6.4 shows some messages emitted during the Modbus scan and captured with the packet analyzer Wireshark. The first stage of the scan tries to find alive hosts by broadcasting ARP requests and registering the hosts that respond to them. The use of ARPs is made possible because the scan targets the local Ethernet network. The second stage tries to establish a TCP connection on port 502 with the detected hosts. It sends a SYN packet to start the TCP three-way handshake. If a SYN-ACK is received the port is open and a RST packet is sent to abort the handshake. The port is considered as closed if a RST is first received, the TCP connection could not be established. If nothing is received the port is considered as filtered, the packet got blocked somewhere.

6.4 Analysis Phase

In this phase, the attacker tries to retrieve more in-depth information from the devices discovered previously. This step is helpful to select suitable attacks and increase their efficiency.

In our ICS simulation, PLCs and MTUs are simulated with OpenPLC as presented before (section 4.3). OpenPLC implements all the basic Modbus functionalities but not more specific ones that could be found in commercial hardware. We thus decided to enhance OpenPLC by implementing the Modbus functionality that can request the device identification information. We found documentation about some Modbus/TCP devices that support this feature although it is more common in Modbus over serial line. From a manual of the vendor Littelfuse/Startco [30], 2 devices (MPU-32 and FPU-32) support it and the response they return is also precisely stated

in the document. We implemented this feature to answer the same way (Figure 6.5). This kind of information is useful to make our simulation look more realistic for any outsiders that interact with it. With a more specialized ICS simulation, it is possible to create honeypots that lure attackers in order to monitor their actions and learn from their attempts in disrupting systems.

```
//-------------------------------------------------------------------------
// Implementation of Modbus/TCP Read Device Identification
//-------------------------------------------------------------------------
void ReadDeviceIdentification(unsigned char *buffer, int bufferSize)
{
  //buffer[0] = 0; // Transaction ID: same as the request
  //buffer[1] = 0; // Transaction ID
  //buffer[2] = 0; // Protocol ID: same as the request
  //buffer[3] = 0; // Protocol ID
  buffer[4] = 0; // Message Length: force first byte to 0
  buffer[5] = 3; // Message length: 3 bytes for unit ID, function code and MEI
  //buffer[6] = 1; // Unit ID: same as the request
  buffer[7] = 0x2b; // Function Code: 43
  buffer[8] = 0x0e; // MEI Type: 14

  buffer[9] = 1; // Read Device ID Code
  buffer[10] = 0x52; // Conformity Level: R (Unknown / Illegal value)
  buffer[11] = 0; // More Follows
  buffer[12] = 0; // Next Object ID
  buffer[13] = 3; // Number of Objects

  buffer[14] = 0; // Object ID 0: Vendor name
  buffer[15] = 7; // Object Length
  buffer[16] = 0x53; // S
  buffer[17] = 0x74; // t
  buffer[18] = 0x61; // a
  buffer[19] = 0x72; // r
  buffer[20] = 0x74; // t
  buffer[21] = 0x63; // c
  buffer[22] = 0x6f; // o
  buffer[23] = 1; // Object ID 1: Product code
  buffer[24] = 4; // Object Length
  buffer[25] = 0x50; // P
  buffer[26] = 0x33; // 3
  buffer[27] = 0x30; // 0
  buffer[28] = 0x31; // 1
  buffer[29] = 2; // Object ID 2: Major/Minor Revision
  buffer[30] = 4; // Object Length
  buffer[31] = 0x31; // 1
  buffer[32] = 0x2e; // .
  buffer[33] = 0x34; // 4
  buffer[34] = 0x30; // 0
  MessageLength = 35;
}
```

Figure 6.5: Implementation to respond to Modbus Read Device Identification requests

```
nmap -n -p T:502 - -script=modbus-discover.nse -max-rate 1 10.0.0.2
```

Figure 6.6: Nmap scan on TCP port 502 with Modbus discovery script

```
root@vagrant-ubuntu-trusty-64:/home/ubuntu/Master-Thesis/examples/TestingAttack
s# nmap -n -p T:502 --script=modbus-discover.nse -max-rate 1 10.0.0.2

Starting Nmap 6.40 ( http://nmap.org ) at 2018-06-06 22:39 CST
Nmap scan report for 10.0.0.2
Host is up (0.085s latency).
PORT    STATE SERVICE
502/tcp open  modbus
| modbus-discover:
|   Positive error response for sid = 0x1 (ILLEGAL FUNCTION)
|_    DEVICE IDENTIFICATION: Startco P301 1.40
MAC Address: 16:46:77:50:43:7D (Unknown)

Nmap done: 1 IP address (1 host up) scanned in 1.13 seconds
```

Figure 6.7: Results of running the scan with Modbus discovery script from host h3

The attacker who found 2 hosts (10.0.0.2 and 10.0.1.2) with the port 502 open from the previous phase now runs a more specialized Nmap scan to verify that a Modbus service is running on those hosts. The new scan (Figure 6.6) uses a custom script provided with Nmap that test some particular Modbus features and is targeted to one host at a time to avoid scanning the whole subnet again. The Figure 6.7 presents the results for the first host. The field containing the device identification correctly shows the information we added in the implementation of OpenPLC. It shows the vendor name (Startco), the product code (P301) and the version of the device (1.40). At this point, an attacker can be fooled in thinking that a real hardware responded. Of course, a more experienced or determined attacker that keeps probing the device with more techniques might, at some point, discover anomalies or missing features that could reveal the subterfuge. But the trick already does a great job as most of the network scans are kept simple and general to be fast in probing large ranges of IP addresses.

```
No.   Time          Source      Destination  Protocol  Length  Info
   11 6.943333000   10.0.3.1    10.0.0.2     Modbus/T(    74   Query: Trans:     0; Unit:   1, Func:  17: Report Slave ID
   13 6.947206000   10.0.0.2    10.0.3.1     Modbus/T(    75 Response: Trans:     0; Unit:   1, Func:  17: Report Slave ID. Exception returned
   21 7.088282000   10.0.3.1    10.0.0.2     Modbus/T(    77   Query: Trans:     0; Unit:   1, Func:  43/  1: Read Device Identification
   23 7.092870000   10.0.0.2    10.0.3.1     Modbus/T(   101 Response: Trans:   338; Unit:   7, Func:  83: Unknown function (83)[Packet size l
▷ Frame 23: 101 bytes on wire (808 bits), 101 bytes captured (808 bits) on interface 0
▷ Ethernet II, Src: 16:46:77:50:43:7d (16:46:77:50:43:7d), Dst: de:0d:87:ff:e7:db (de:0d:87:ff:e7:db)
▷ Internet Protocol Version 4, Src: 10.0.0.2 (10.0.0.2), Dst: 10.0.3.1 (10.0.3.1)
▷ Transmission Control Protocol, Src Port: 502 (502), Dst Port: 42471 (42471), Seq: 1, Ack: 12, Len: 35
▽ Modbus/TCP
    Transaction Identifier: 0
    Protocol Identifier: 0
    Length: 3
    Unit Identifier: 1
▽ Modbus
    Function Code: Encapsulated Interface Transport (43)
    MEI type: Read Device Identification (14)
▽ [Malformed Packet: Modbus]
  ▽ [Expert Info (Error/Malformed): Malformed Packet (Exception occurred)]
      [Malformed Packet (Exception occurred)]
      [Severity level: Error]
      [Group: Malformed]
▽ Modbus/TCP
    Transaction Identifier: 338
    Protocol Identifier: 0
    Length: 768
    Unit Identifier: 7
  [Packet size limited during capture: Modbus/TCP truncated]
```

Figure 6.8: Messages captured during the scan with Modbus discovery script

The custom discovery script uses 2 Modbus functions to retrieve specific data. The Figure 6.8 presents a summary of the new packets sent. The first request asks for the slave identifier of the device. The response contains an exception, this is actually a normal behavior as Modbus/TCP devices do not depend on this information. The slave ID is used to distinguish equipment on the same serial line. Modbus/TCP already has the IP address to reach its destination. The second request asks for the device identification which corresponds to the new implementation. In Wireshark, the response is flagged as malformed and the information displayed are wrong because the packet parser is not able to recover correctly after detecting the exception. This

is because of a strange particularity we found in the manual from Littelfuse/Startco that we kept in the implementation to stay close to reality. The value for the conformity level contained in buffer[10] is not part of the Modbus protocol specifications, it is an illegal value. This is an interesting information, if no other product uses this value, an attacker could target his scans to detect with perfect accuracy this particular device. If a vulnerability is known, this device could become an ideal target for attacks.

```
root@vagrant-ubuntu-trusty-64:/home/ubuntu/Master-Thesis/src# python forge.py 1
0.0.0.2
Connected to 10.0.0.2 on port 502
Detecting supported function codes from 1 to 127

Function code 1 supported: Read Coils Response
Function code 2 supported: Read Discrete Inputs Response
Function code 3 supported: Read Holding Registers Response
Function code 4 supported: Read Input Registers Response
Function code 5 supported: Write Single Coil
Function code 6 supported: Write Single Register Response
Function code 15 supported: Write Multiple Coils Response
Function code 16 supported: Write Multiple Registers Response
Function code 43 supported: Read Device Identification
Functions scan finished
```

Figure 6.9: Results of running our Scapy Modbus function scanner from host h3

We also have a custom Python script that uses the libraries from Scapy to produce Modbus packets. With this script, the attacker can detect the function codes supported by a target. From the results shown in Figure 6.9, all the basic Modbus functionalities and the new Read Device Identification function are properly detected.

The attacker has now enough information to start interacting with the devices.

6.5 Attack Phase

With the information gathered, the attacker can now take actions.

The first attack is based on the fact that the Modbus protocol has no authentication mechanism. From the previous phases, the attacker was able to interact with the 2 Modbus devices discovered and knows the features they support. This means that now, he can do whatever he wants with them. He can read any values from the devices and overwrite them to 0 or random values. Depending on the vendor, the equipment can execute more functions like reset its memory, halt or restart and force to listen mode only which prevents anyone to communicate with it. The modifications can have an impact on the physical process controlled by the device, destroy the production or cause a disaster dangerous for the on-site employees.

The second attack is based on a well known TCP vulnerability. When starting a new TCP connection, the three-way handshake is completed first. The connection can stay in a pending state if the final ACK from the client is never sent in response to the SYN-ACK of the server. A Modbus device like a PLC is built on limited hardware, we can thus assume that the capacity to keep those pending connections in memory is very low. From the source code of OpenPLC, this capacity is set to 5 connections but the OS can decide to use a different value. When the capacity is full, no more establishments of new connections are accepted, they are all dropped. The attack thus consists of creating enough pending states on the device to reach the maximum capacity and prevent anyone to start a new connection. This is a Denial-of-Service (DoS) attack as we are able to block the usage of the service. This attack may not be dangerous at first as PLCs are autonomous enough to run by themselves, as long as their logic was well thought out, but if the attack perpetuates the situation can become critical.

```
sysctl net.ipv4.tcp_max_syn_backlog #(default 128 pending connections)
sysctl net.ipv4.tcp_syncookies #(default 1, enabled)

sysctl -w net.ipv4.tcp_max_syn_backlog=5 #(max 5 pending connections)
sysctl -w net.ipv4.tcp_syncookies=0 #(0, disabled)
```

Figure 6.10: Commands used to modify the TCP SYN backlog and SYN cookies

Since the simulation is executing in complete software mode, our Modbus devices run on modern OS, Ubuntu 14.04, and hardware that can support heavy loads. Our installation has a default capacity of 128 pending connections and comes with the SYN cookies mechanism activated which is a protection created to avoid the attack. When the capacity is reached, the new connections are not saved in memory. The information needed to verify a connection are hashed to create the sequence number of the SYN-ACK. When an ACK is received, the hash is recomputed and compared with its sequence number. This technique came later and relies on more sustained computation resources, we can thus assume that PLCs do not incorporate it or did not before long. The Figure 6.10 shows the default values and the new values used to adjust the simulation with those considerations.

```
iptables -A OUTPUT -p tcp -s 10.0.3.1 --tcp-flags RST RST -j DROP #(Drop RST)
hping3 -S --fast -p 502 10.0.0.2
iptables -D OUTPUT -p tcp -s 10.0.3.1 --tcp-flags RST RST -j DROP #(Delete rule)
```

Figure 6.11: Commands used to perform the SYN flood attack

To perform the attack, the target is flooded with SYN packets on port 502. We created a custom script with Scapy but a tool like hping3 can also be used. When receiving a SYN-ACK response, the kernel will send by default a RST packet to teardown the TCP connection. The SYN was sent from the userspace so the kernel does not know anything about this new connection. To block the kernel from sending RST packets which prevent the attack from taking effects, a firewall rule in Iptables is introduced on the attacker host h3 to drop them (Figure 6.11).

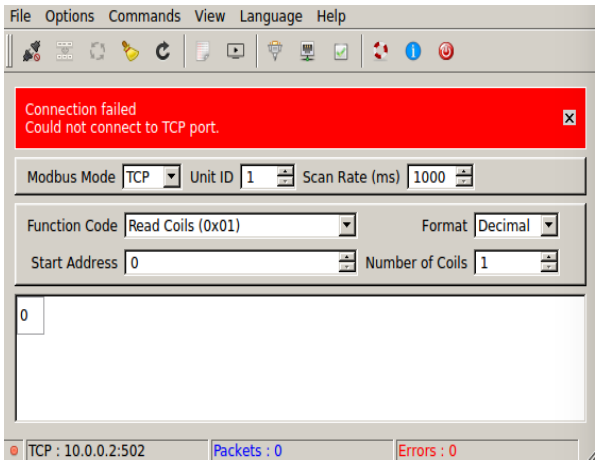

Figure 6.12: Connection to the host oplc1 failed

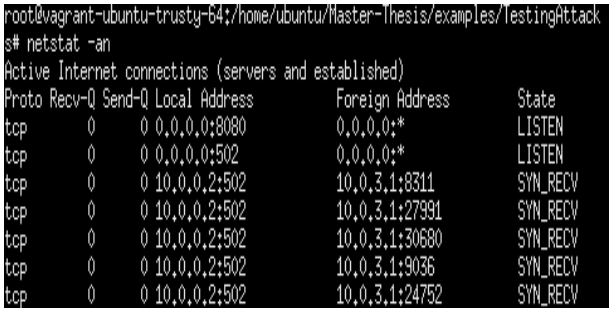

Figure 6.13: Results of Netstat on host oplc1

When enough SYN packets are sent, no one can establish a new connection with the target. The Figure 6.12 shows that the software qModMaster is not able to connect to the host 10.0.0.2. Using Netstat on the host (Figure 6.13), we can see that there are 5 SYN_RECV that represent 5 pending connections which is the limit configured earlier. When the attack stops, a time out will occur and new connections will be accepted.

The third attack is called ARP spoofing or ARP cache poisoning. ARP (Address Resolution Protocol) translates IP addresses to MAC addresses and is used on local area networks (LAN) to be able to send messages to the correct destination. By sending forged ARP messages, this destination can be modified to point to the attacker's machine. He can then eavesdrop packets creating a Man-in-the-Middle (MITM) attack, drop them causing a DoS attack or modify them to inject malicious data.

```
sysctl -w net.ipv4.ip_forward=1 #(Allow forwarding)

etterfilter filter -o filter.ef #(Compile filter)
ettercap -T -F filter.ef -i h3-eth0 -M ARP /10.0.1.2// /10.0.0.2//
```

Figure 6.14: Commands used to perform ARP poisoning

Anyone can perform the ARP spoofing technique within a few commands thanks to the tool Ettercap (Figure 6.14). The attacker can test multiple ranges of IP addresses to find devices that are communicating to each other. It is also possible to reroute all packets on the LAN by spoofing the whole subnet but this can overload the attacker's computer because of the tremendous amount of messages received. If the forwarding option is not enabled on the attacker's machine, the attack becomes a DoS attack as packets no longer reach their destination.

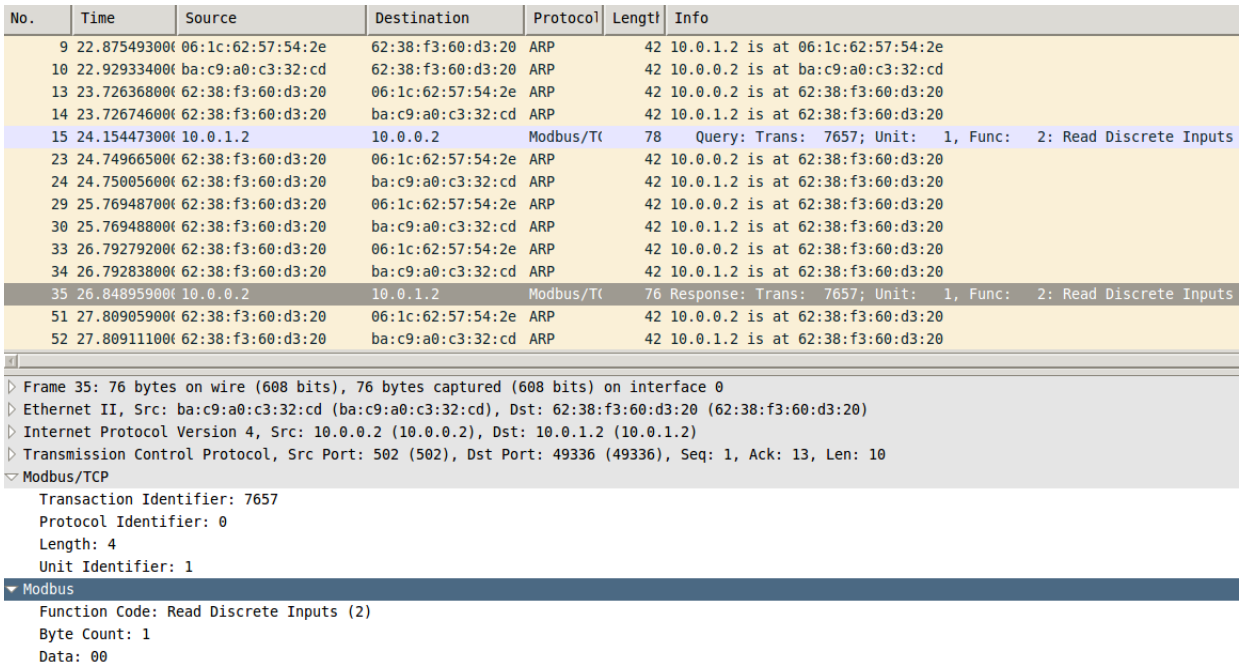

Figure 6.15: Messages captured on host h3 during the ARP poisoning attack

The Figure 6.15 shows how ARP messages containing the IP addresses of the 2 targets (10.0.0.2 and 10.0.1.2) are spoofed and flooded with the same MAC address, the attacker's MAC address. The attacker now receives the packets exchanged between the targets. Because Modbus does not incorporate any confidentiality mechanism, the messages can be read in clear text. The response for a read request is visible on the same figure. By flooding spoofed ARPs, legitimate ARPs get quickly overwritten in caches of the victims and the MITM attack can continue.

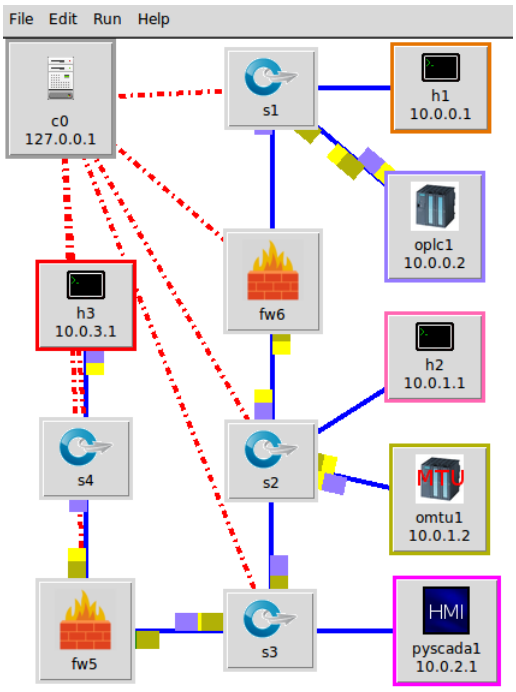

Figure 6.16: Visual confirmation that packets between oplc1 and omtu1 are being rerouted to h3

The Figure 6.16 clearly shows that the attack is a success with packets getting rerouted to the attacker host h3 which was not receiving anything before.

```
if (ip.proto == TCP && tcp.src == 502) { # If TCP and source port 502
    if (DATA.data + 6 == "\x01") { # If Unit ID == 1
        DATA.data + 6 = "\xFF"; # Change to 255
    }
    if (DATA.data + 7 == "\x02") { # If Function Code 2, Read Discrete Inputs
        DATA.data + 7 = "\x82"; # Change to 130, Read Discrete Inputs Exception
        DATA.data + 8 = "\x06"; # Set Exception Code to 6, Slave Device Busy
    }
}
```

Figure 6.17: Ettercap filter used to tamper packets

Ettercap also proposes the execution of user made filters to alter packets. Because Modbus does not contain any integrity check mechanism (TCP does but not the Modbus payload), the attacker is able to modify the content of the messages without being noticed. The simple filter in Figure 6.17 will modify packets coming from a Modbus server. The unit identifier is changed from 1 to 255 and any requests to read discrete inputs will return an exception stating that the device is currently busy and was not able to answer. A tampered response in Figure 6.18 shows the filter correctly working.

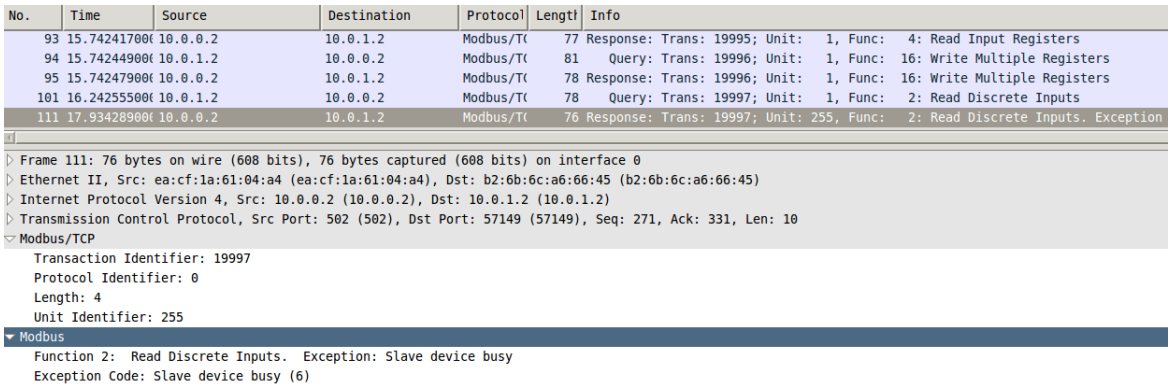

Figure 6.18: Messages captured on host omtu1 during the packet tampering attack

The last attack targets the controller of the SDN network. During our tests, we discovered that the controller was not able to keep up with the number of packets sent in the network. This is caused by the limited computing power provided to the virtual machine. Without doing it on purpose, we were actually creating a DoS attack on the SDN controller. Every time an OpenFlow switch receives a packet unknown from its flow tables, the switch asks the controller what is the action to perform and the controller responds with the creation of a new flow entry on the switch. For example, when scanning the network with Nmap, every packet sent is unknown from every switch in the forwarding path. The controller gets overloaded of requests from the switches and the service slowdowns. This is the reason why a lot of our scripts and commands are limited in the number of packets transmitted per second, like the option max-rate in Nmap that we set to a low value of 1 packet per second.

In this section, we presented and performed attacks at different layers of the ICS simulation to raise awareness about the fact that vulnerabilities can be hidden in every step of the communication between devices. The lowest layer is the device itself where the SYN flood was performed. The next layer is the network with the ARP spoofing and controller DoS attack. Finally, the protocol layer with Modbus/TCP. We showed that direct interactions with a Modbus server are possible because of the lack of authentication. With the MITM attack, the Modbus messages are presented in clear text because of the non-existent confidentiality mechanism and tampering with them is made possible because of the absence of integrity check.

6.6 Protections

After the experimentation of attacks, protections to block them or mitigate them are presented.

While a lot of solutions are available, the first one that is present nowadays in every network is given through firewall rules. Because we are working with SDN networks, the firewall logic needs to be integrated in the SDN controller. We use the project CloudFirewall presented before (section 4.5). The first packet of every new connection is matched with the rules and allowed or blocked. Once the decision is made, a new flow entry is created on the switch and the rest of the connection is directly matched with it.

```
whitelist_rules:
-   direction: Outgoing
    dst_ip: '10.0.1.2'
    dst_port: '*'
    protocol: TCP
    src_ip: '10.0.0.2'
    src_port: 502
-   direction: Incoming
    dst_ip: '10.0.0.2'
    dst_port: 502
    protocol: TCP
    src_ip: '10.0.1.2'
    src_port: '*'
```

Figure 6.19: Firewall rules used in this scenario

The firewall rules in Figure 6.19 only allow the host oplc1 and omtu1 to communicate via Modbus/TCP on port 502. Thanks to the whitelist mode, the rest of the traffic is dropped by default. A firewall has only 2 interfaces, an incoming one and an outgoing one. The direction of a packet depends on the first interface it reaches. The directions ensure that those 2 hosts are connected on the correct side of the firewall, if the devices are moved their communication might get blocked. With such rules, all Modbus based attacks seen earlier and performed from the host h3 are blocked by the first firewall. The ARP spoofing technique is still possible but the rerouted messages will not reach h3 as they are also dropped. This means that it can still be used as a DoS attack. We are able to stop Modbus attacks from outsiders but a big weakness in the protection provided by firewalls is the possibility to perform attacks from inside, behind them. If the attacker is able to connect or infect the host h1, the attacks are working again.

One of the most efficient protection to mitigate attacks is to reduce the information the attacker can find while scanning the network.

```
iptables -A INPUT -p tcp --dport 502 ! -f -m u32 --u32 "0>>22&0x3c @
    12>>26&0x3c @ 7>>24&0xff=0x2b" -j DROP
```

Figure 6.20: Iptables rule to drop Modbus Read Device Identification request [33]

The information we want to hide is the device identification retrieved via the Modbus Read Device Identification request (Function Code 43). It contains the vendor name, the code and the version of the product which are sensible information. Most of the time, such functionality cannot be deactivated and the product information cannot be modified to empty values. If we cannot find a solution from the inside, here because of proprietary hardware, an outside solution is needed.

The Figure 6.20 presents an Iptables rule that will perform a deep packet inspection (DPI). It drops all Modbus/TCP packets on port 502 with a payload that contains the Modbus Function Code 43. We successfully tested it by activating this rule on every Modbus devices. In a more realistic scenario, a host that can forward packets should be placed as a buffer in front of the Modbus devices to execute the firewall rule. Modifying the SDN firewall we use, CloudFirewall, to perform a DPI is not an easy task. Only the first packet of the connection is matched with the rules before creating a new flow entry. This means that the subsequent packets are not verified, they directly match the entry in the flow table of the switches. To verify every packet without overloading the controller, we would need to modify the matching mechanism on the switches themselves, deep in the OpenFlow implementation. This could be performed in a possible future work.

This Iptables rule uses the u32 module to access 32 bits of a packet at a time. The first instruction, $0 >> 22\&0x3c$, finds the length field of the IP header (bits 4 to 7 of the first byte) by shifting 22 bits to the right and applying the mask $0x3c$. The shifting also multiplies the value by 4 which is convenient because the value of the length field represents the number of 4 bytes words before the payload. The @ symbol jumps the number of bytes corresponding to the result of the preceding statement. The first jump thus arrives at the beginning of the TCP header. The second instruction, $12 >> 26\&0x3c$, finds the offset field of the TCP header (bits 0 to 3 of byte 12) by shifting 26 bits to the right and applying the same mask to keep the first 4 bits only. The second jump arrives at the beginning of the Modbus frame. The final instruction, $7 >> 24\&0xff = 0x2b$, finds the function code in the Modbus payload (byte 7) and match it with $0x2b$ which is the hexadecimal representation for the Function Code 43. If the match is positive, the packet is dropped.

Nowadays, firewalls do not provide sufficient protections as attacks are getting more and more inventive and complex. Intrusion Detection System (IDS) and Intrusion Prevention System (IPS) based on attack signatures and abnormal behaviors compatible with SDN networks. The ARP poisoning technique is difficult to block. Fixing IP addresses with MAC addresses ca prevent it but this solution is not feasible in large dynamic networks. A solution could be to create systems that can detect abnormal behaviors in ARP messages to raise alerts or block them. Another solution is to create a system that can verify the authenticity of the ARPs by relying on cryptographic functions, each host could send its own unique ARPs. All those solutions would make interesting future works.

Chapter 7

Conclusions

This chapter concludes this Master Thesis with a reminder of our contributions and ideas that can be explored to further improve this work.

7.1 Contributions

In this section we will first review our goals then use each goal as a base to explain our contributions.

- *Creation of a general ICS simulator that can support different network topologies, execute software to simulate typical ICS components like PLC, MTU and SCADA system communicating between each other through the communication protocol Modbus/TCP.*

  As explained in the chapter 4, we designed an ICS simulator using Mininet as network emulator, we also modified and fixed OpenPLC to play the role of PLCs and MTUs communicating together via the protocol Modbus/TCP. Furthermore, we used the open source PyScada as a SCADA system and also modified the open source SDN firewall, CloudFirewall, to support our needs.

- *Creation of tools and automated configurations to simplify the generation of ICS examples that are ready to run in the simulator.*

  During the chapter 5 about ICS generation, we explained that we created a Python script that can generate ICS network in two different ways. The first way is the semi-automated creation by using our modified MiniEdit to generate a JSON file then used by our Python script as a base for the generation. The second way is by passing parameters to the Python script that will generate a topology ready to use according to these parameters.

- *Visualization of running ICS simulations to provide a quick overview and facilitate interactions with the components.*

  Also in the chapter 5, we talked about MiniNAM, the network animator for Mininet and changes that have been made. MiniNAM can display packet flows in real-time, we modified it by adding the support for ICS components to facilitate the interactions with a running simulation.

- *Generation of several ICS examples to present typical topologies found in literatures.* Currently our generation script supports three topologies based on topologies found in literatures as explain in the section 5.4.

- *Presentation of attacks performed on a simulated ICS and establishment of protections against them.* We dedicated the chapter 6 for testing attacks on our generated network. We explained how the attacks can be performed and their consequences. Furthermore we also present solutions to mitigate or block these attacks.

7.2 Future works

There are several possible works to go further with our ICS simulator.

Find a simple way to attach actual equipment to the network. This will be useful to verify the interoperability between simulated devices and real devices which should not encounter any problems as our network is emulated and the Modbus/TCP messages meet the specifications. It will also make possible the creation of new attack scenarios based on device vulnerabilities.

Evaluate the fidelity of a simulated ICS compared to an equivalent ICS testbed. This can be done by calculating a score from criteria on the network packets and the physical process state.

Adding the support for new communication protocols, ICS components and services is always possible as our simulator is quite general with each components separated into modules.

Finally, testing new attacks and protections for ICS environments over SDN networks could create the foundations of new solutions for tomorrow. The transition from traditional network architecture to the emerging SDN technology will benefit from those new ideas to rethink security.

7.3 Acknowledgements

We would like to thank our supervisor Pr. Ramin Sadre for his help and guidance over the year.

We would also like to thank Mr. Kabasele and Mr. Klein for taking the time to read this Master Thesis.

Finally, we would like to thank the Université catholique de Louvain, all the professors and the administration who made our 5 years of study possible.

Bibliography

[1] Mosaik: A flexible smart grid co-simulation framework. https://mosaik.offis.de/. 27-04-2018.

[2] Using the pox sdn controller. http://www.brianlinkletter.com/using-the-pox-sdn-controller/. 19-05-2018.

[3] What are the two major differences between dcs and scada systems? https://www.researchgate.net/post/What_are_the_two_major_differences_between_DCS_and_SCADA_systems. 27-04-2018.

[4] Thiago Alves. Openplc: An open-source programmable logic controller. http://www.openplcproject.com/. 11-05-2018.

[5] Thiago Rodrigues Alves, Mario Buratto, Flavio Mauricio de Souza, and Thelma Virginia Rodrigues. Openplc: An open source alternative to automation. In *Global Humanitarian Technology Conference (GHTC), 2014 IEEE*, pages 585–589. IEEE, 2014.

[6] Dorothea Andersson. Simulation of industrial control system field devices for cyber security, 2017.

[7] Vignesh Babu and David M Nicol. Emulation/simulation of plc networks with the s3f network simulator. In *Winter Simulation Conference (WSC), 2016*, pages 1475–1486. IEEE, 2016.

[8] Beremiz. Beremiz: Open source framework for automation. https://beremiz.org/. 11-05-2018.

[9] Jitendra Bhatia. Configuring an sdn controller in open source mininet. https://opensourceforu.com/2016/01/configuring-an-sdn-controller-in-open-source-mininet-emulator/. 11-05-2018.

[10] Michael Büsch. Awlsim: S7 compatible soft-plc. https://bues.ch/cms/automation/awlsim.html. 27-04-2018.

[11] Bo Chen, Nishant Pattanaik, Ana Goulart, Karen L Butler-Purry, and Deepa Kundur. Implementing attacks for modbus/tcp protocol in a real-time cyber physical system test bed. In *Communications Quality and Reliability (CQR), 2015 IEEE International Workshop Technical Committee on*, pages 1–6. IEEE, 2015.

[12] Justyna J Chromik, Anne Remke, and Boudewijn R Haverkort. A testbed for locally monitoring scada networks in smart grids. *Energy-Open*, page 14, 2017.

[13] Jens Dede, Koojana Kuladinithi, Anna Förster, Okko Nannen, and Sebastian Lehnhoff. Omnet++ and mosaik: enabling simulation of smart grid communications. *arXiv preprint arXiv:1509.03067*, 2015.

[14] Dell. 2015 dell security annual threat report. http://proconics.co.za/wp-content/uploads/2017/10/2425.pdf, 2015. 13-04-2018.

[15] Haihui Gao, Yong Peng, Zhonghua Dai, Ting Wang, Xuefeng Han, and Hanjing Li. An industrial control system testbed based on emulation, physical devices and simulation. In *International Conference on Critical Infrastructure Protection*, pages 79–91. Springer, 2014.

[16] John R Hackworth and Frederick D Hackworth. *Programmable Logic Controllers: Programming Methods and Applications*. Pearson, 2004.

[17] Hannes Holm, Martin Karresand, Arne Vidström, and Erik Westring. A survey of industrial control system testbeds. In *Secure IT Systems*, pages 11–26. Springer, 2015.

[18] Vinay M Igure, Sean A Laughter, and Ronald D Williams. Security issues in scada networks. *Computers & Security*, 25(7):498–506, 2006.

[19] SANS Institute. Analysis of the cyber attack on the ukrainian power grid. https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf, 2016. 13-04-2018.

[20] SANS Institute. Analysis of the recent reports of attacks on us infrastructure by iranian actors. https://ics.sans.org/media/SANSICS_DUC4_Analysis_of_Attacks_on_US_Infrastructure_V1.1.pdf, 2016. 13-04-2018.

[21] SANS Institute. Securing industrial control systems-2017. https://www.sans.org/reading-room/whitepapers/analyst/securing-industrial-control-systems-2017-37860, 2017. 13-04-2018.

[22] Arthur Jicha, Mark Patton, and Hsinchun Chen. Scada honeypots: An in-depth analysis of conpot. In *Intelligence and Security Informatics (ISI), 2016 IEEE Conference on*, pages 196–198. IEEE, 2016.

[23] Karamjeet Kaur, Japinder Singh, and Navtej Singh Ghumman. Mininet as software defined networking testing platform. In *International Conference on Communication, Computing & Systems (ICCCS)*, pages 139–42, 2014.

[24] Sukhveer Kaur, Japinder Singh, and Navtej Singh Ghumman. Network programmability using pox controller. In *ICCCS International Conference on Communication, Computing & Systems, IEEE*, volume 138, 2014.

[25] Eric D Knapp and Joel Thomas Langill. *Industrial Network Security: Securing critical infrastructure networks for smart grid, SCADA, and other Industrial Control Systems*. Syngress, 2014.

[26] Georgia Koutsandria, Reinhard Gentz, Mahdi Jamei, Anna Scaglione, Sean Peisert, and Chuck McParland. A real-time testbed environment for cyber-physical security on the power grid. In *Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy*, pages 67–78. ACM, 2015.

[27] Stipe Kuman, Stjepan Groš, and Miljenko Mikuc. An experiment in using imunes and conpot to emulate honeypot control networks. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2017 40th International Convention on*, pages 1262–1268. IEEE, 2017.

[28] Kaspersky Lab. The state of industrial cybersecurity 2017. https://go.kaspersky.com/rs/802-IJN-240/images/ICSWHITEPAPER.pdf, 2017. 13-04-2018.

[29] Bob Lantz. Mininet: A simple networking testbed for openflow/sdn! http://multipath-tcp.org/conext2013/mininet/mininet/net.py. 20-05-2018.

[30] Littelfuse. Mpu-32 and fpu-32 modbus tcp manual. http://www.littelfuse.com/products/protection-relays-and-controls/protection-relays/motor-and-pump-protection/mpu-32.aspx. 20-05-2018.

[31] matanby. Cloudfirewall. https://github.com/matanby/CloudFirewall. 20-05-2018.

[32] Thomas H Morris, Zach Thornton, and Ian Turnipseed. Industrial control system simulation and data logging for intrusion detection system research. *7th Annual Southeastern Cyber Security Summit*, 2015.

[33] Jeyasingam Nivethan and Mauricio Papa. On the use of open-source firewalls in ics/scada systems. *Information Security Journal: A Global Perspective*, 25(1-3):83–93, 2016.

[34] Radiflow. Detection of a crypto-mining malware attack at a water utility. http://radiflow.com/detection-of-a-crypto-mining-malware-attack-at-a-water-utility/, 2018. 13-04-2018.

[35] Bradley Reaves and Thomas Morris. An open virtual testbed for industrial control system security research. *International Journal of Information Security*, 11(4):215–229, 2012.

[36] Kshira Sagar Sahoo, Sagarika Mohanty, Mayank Tiwary, Brojo Kishore Mishra, and Bibhudatta Sahoo. A comprehensive tutorial on software defined network: The driving force for the future internet technology. In *Proceedings of the International Conference on Advances in Information Communication Technology & Computing*, page 114. ACM, 2016.

[37] Moses D Schwartz, John Mulder, Jason Trent, and William D Atkins. Control system devices: Architectures and supply channels overview.

[38] Charlie Scott and Richard Carbone. Designing and implementing a honeypot for a scada network. *SANS Institute Reading Room*, 2014.

[39] Siemens. Step 2000 basics of plcs. http://diagramas.diagramasde.com/otros/Siemens%20Basics%20Of%20Plc.pdf. 27-04-2018.

[40] Keith Stouffer and Joe Falco. *Guide to supervisory control and data acquisition (SCADA) and industrial control systems security.* National institute of standards and technology, 2006.

[41] Keith Stouffer, Joe Falco, and Karen Scarfone. Guide to industrial control systems (ics) security. *NIST special publication*, 800(82):16–16, 2011.

[42] DPS Telecom. Dcs vs. scada in modern enterprise environments. http://www.dpstele.com/scada/dcs-vs.php. 27-04-2018.

[43] Zach Thornton, David Mudd, Thomas Morris, and Fei Hu. Virtual scada systems for cyber security.

[44] Alexandre Gustavo Wermann, Marcelo Cardoso Bortolozzo, Eduardo Germano da Silva, Alberto Schaeffer-Filho, Luciano Paschoal Gaspary, and Marinho Barcellos. Astoria: A framework for attack simulation and evaluation in smart grids. In *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*, pages 273–280. IEEE, 2016.

[45] Wireshark. Tshark. https://www.wireshark.org/docs/man-pages/tshark.html. 8-05-2018.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**