AMON: an Automaton MONitor for Industrial Cyber-Physical Security

Giuseppe Bernieri
bernieri@math.unipd.it
Department of Mathematics
University of Padua
Padua, Italy

Mauro Conti
conti@math.unipd.it
Department of Mathematics
University of Padua
Padua, Italy

Gabriele Pozzan
gabriele.pozzan@studenti.unipd.it
Department of Mathematics
University of Padua
Padua, Italy
CyBrain Srl
Vicenza, Italy

ABSTRACT

The rapid evolution towards the *Industry 4.0* improves the performances of Industrial Control Systems (ICSs). However, due to the unmanageable re-engineering cost of pre-existing industrial devices, insecure protocols continue to be used to manage these systems. In this scenario, legacy protocols, such as the Modbus/TCP, are still largely used to control a range of industrial processes alongside with modern technologies. Consequently, hybrid industrial infrastructures with both legacy and innovative devices require novel security and prevention methodologies.

In this work, we present AMON (Automaton MONitor): an Intrusion Detection System (IDS) based on Deterministic Finite Automata (DFA) for Modbus/TCP traffic monitoring. AMON combines DFA with the Longest Repeating Subsequence (LRS) algorithm, commonly used in bioinformatics, to model the traffic and identify anomalies. In order to address the challenges presented in hybrid scenarios, we extend AMON to work with the Constrained Application Protocol (CoAP), used for the Industrial Internet of Things (IIoT). We show preliminary results in a simulated industrial network and discuss possible implementation of the developed detection system to secure hybrid industrial infrastructures.

CCS CONCEPTS

• **Security and privacy → Intrusion detection systems**; • **Computer systems organization → Sensors and actuators**.

KEYWORDS

Intrusion Detection System, Anomaly Detection, Cyber-Physical System, Industrial Security

1 INTRODUCTION

Supervisory Control and Data Acquisition (SCADA) systems allow the management of a wide variety of industrial facility processes. Threats to ICS infrastructures can cause serious economic and human damages. *Stuxnet* [10] represents the most famous malicious worm targeting ICSs. Discovered in 2010, it was designed to target Programmable Logic Controllers (PLCs) in nuclear plants in order to modify the behavior of the centrifuges and damage them. Another example of threat to ICS networks is the *Industroyer* worm [4], which caused an outage in the Ukraine's power grid in 2016. This malware installed a backdoor to communicate to a Command & Control server and exploited the trusting nature of the protocol to execute its payload by simply issuing commands. Last but not least, the *Triton* worm [9], discovered in 2017, targeted Safety Instrumented Systems (SISs) of a petrochemical processing plant. SISs are special PLCs designed to keep the physical processes in a safe state preventing incidents.

Nowadays, the *Industry 4.0* paradigm created an hybrid industrial scenario where legacy and novel systems coexist. This situation opens up a series of challenges for the security of such systems. The Modbus/TCP protocol [16] is a clear example of this kind of challenges: designed with the idea of an isolated and trusted network, it lacks basic security controls and is still largely used by modern ICS operators. In order to protect the industrial control networks, vendors provide specific solutions for intrusion detection. However, industries need novel security solutions addressing monitoring of hybrid scenarios where legacy protocols work coupled with IIoT ones, such as the CoAP [17]. Therefore, it is important to develop novel detection techniques because the importance of industrial systems makes them likely targets of *Advanced Persistent Threats (APTs)*: well funded, prolonged attacks which cover all the attack surface of their targets, from social engineering to zero-day exploits.

As contribution, in this work we:

- conceive a novel methodology based on DFA and the LRS algorithm to model multi-periodic traffic;
- present AMON: a novel security framework able to recognize normal patterns of industrial Modbus/TCP and CoAP communications, predicting data exchanges, and alerting in case of suspicious and potentially dangerous deviations;
- build a hybrid ICS network simulation scenario with Modbus/TCP and CoAP protocols using *Mininet*[1];

[1] http://mininet.org

- evaluate AMON with experiments on different threat scenarios, such as *Denial of Service* (*DoS*), *Buffer Overflow*, and *Man In The Middle* (*MITM*) attacks.

The remainder of this paper is organized as follows. Section 2 describes the background with the protocols used in this work. Section 3 analyzes related work considering Modbus/TCP and DFA in security contexts. In Section 4, we describe our proposed methodology which we evaluate in Section 5. Section 6 concludes the paper.

2 BACKGROUND

In this section we give an overview of the industrial protocols used in this work.

2.1 Modbus and Modbus/TCP

Modbus [15] is an application layer protocol largely used in industrial networks. Conceived in 1979, it became the *de facto* standard for industrial communication. The protocol was designed for serial networks but is nowadays available on the TCP/IP stack with reserved port 502. The basic unit of a Modbus message is the Protocol Data Unit (PDU) which is independent from the layer over which it is communicated and consists of a **Function Code (FC)** which is a 1 *byte* identifier for the operation requested and an optional **Data** field used by the server to perform the operation requested by the client. The Modbus/TCP implementation includes a Modbus Application Protocol Header (MBAP) which adds some information:

- **Transaction Identifier (TI)**: a unique number which defines a particular query/response couple;
- **Protocol Identifier**: used to identify Modbus/TCP packets for intra-system multiplexing;
- **Unit Identifier**: used to identify entities when the network consists of both serial and TCP/IP channels;
- **Length**: indicates the size of the Modbus/TCP packet.

The rest of the fields could vary depending on the FC of the packet.

The Modbus protocol was built to work on serial networks in which every entity was considered trusted and the isolation of the network was considered enough to keep malicious actors out, this leads to considerable vulnerabilities. Some of these issues carry over in Modbus/TCP: the main problems we want to highlight are the *lack of authentication* for masters or slaves and the *lack of encryption* in the communication.

2.2 CoAP

The CoAP [17] is a service layer protocol used in IIoT scenarios and designed to support the communications of resource constrained entities such as sensors, microcontrollers, and embedded devices [8]. Messages are exchanged over UDP and the communications follow a request/response pattern modeled after HTTP GET, POST, PUT, DELETE methods. Each message starts with a fixed header which holds the following information:

- **Type**: indicates the type of message, *Confirmable* (CON) messages which must be acknowledged or rejected, *Non-Confirmable* messages which must never be acknowledged, *Acknowledge* (ACK) messages, and *Reset* (RST) messages used for rejections.
- **Code**: indicates the class and details of the message using the format *c.dd*. The class can be "0" for request messages, "2" for success responses, "4" for client error responses and "5" for server error responses.
- **Message Id**: unique value used to match requests with responses.

Following the header there is a variable length **Token** value which is used to match requests with responses along with the Message Id. This is followed by one ore more **Options** and an optional **Payload**. It's worth noting that the data of a response to a GET request could be held in two different places: in the option *Uri-query* (code "15") if the response is piggybacked on an ACK message or in the payload section if the response is an independent message.

3 RELATED WORK

In this section, we present a literature review of the methodologies conceived to use DFA for security, stressing those applied to ICSs.

3.1 Pattern DFA

A DFA [13] is a finite state machine which accepts an alphabet of symbols and produces only one result for each sequence. It can be defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where $\mathbf{Q}$ is the set of states, $\mathbf{\Sigma}$ is the alphabet, $\delta : \mathbf{Q} \times \mathbf{\Sigma} \Longrightarrow \mathbf{Q}$ is the *transition function*, $\mathbf{q_0} \in \mathbf{Q}$ is the initial state and $\mathbf{F} \subseteq \mathbf{Q}$ is the set of final states.

In [6], *Goldenberg and Wool* describe an approach using DFA for anomaly detection in ICSs. We refer to this work as *Pattern DFA*. The *Pattern DFA* models the periodic pattern of communication between a Human Machine Interface (HMI) and a PLC. It is built automatically after a training phase during which query and response packets are captured and analyzed and it differs from a basic DFA because *it does not have final states* since its intent is to check that there are no variations in the expected periodic pattern of packets. Moreover the results and output of the DFA are related to the *transitions* that the packet traffic produces (and not so much to the states).

We will next summarise how *Goldenberg and Wool* describe the anatomy of this DFA (which states and transitions define it) and how they automatically generate it from captured data.

Anatomy: each *Pattern DFA* has its own alphabet Σ built during the initial training phase. Its symbols are defined by parts of the Modbus/TCP packet:

$$(Q, FC, RN, BC), \tag{1}$$

where:

- **Q**: is the query/response flag.
- **FC**: is the Function Code.
- **RN**: is the Reference Number (this value is only specified in query packets and will always be "0" for responses).
- **BC**: is the byte count.

The DFA states represent the situation after the *Pattern DFA* receives query or response packets and each one have *normal* and *exception* transitions as defined by the transition function δ:

- **Normal**: this transition is triggered when the packet received is the next one in the expected pattern. If the DFA is

in state s_i, after this transition it will be in state s_{i+1}. This transition can have subcases: a *Mismatch* happens when the TI of a response does not match the one of the query, an *Oversized packet* is a packet longer than 252 bytes.

- **Retransmission**: the packet received is the current packet in the expected pattern. This causes a self loop transition and is not considered an alert situation.
- **Miss**: the packet received is part of the pattern but it is not the expected one. This is often caused by packets being dropped and is not considered an alert situation. If the received packet is relative to state s_j in the pattern, the following state will be s_{j+1}.
- **Unknown**: the packet received is not part of the expected pattern. This situation causes an *unknown alert* to be raised. The next state will be reset to s_0.

Additionally, the system determines if the master/client IP address changed. If so, the security system will raise a *master/client IP changed* alert.

While exceptional states do not usually raise alerts, the system keeps track of the number of exceptional packets for each category and uses this data during the automatic generation of the DFA.

Generation: the system automatically generates a *Pattern DFA* after it captures and analyzes a number of packets during the training phase. The algorithm used for the generations is described in detail in [6].

Goldenberg and Wool discuss how this approach is not suitable to accurately model multi periodic traffic (i.e., communications in which there are different query/response patterns running at different speed). Using a single DFA to model such a scenario would require a large amount of states and it would lead to many false positives since the different patterns are most probably not in lockstep.

To tackle this representation challenge, they propose a multi-DFA model in which packets rejected as *unknown* by the standard *Pattern DFA* are passed along to a second level DFA which models the other pattern, and so on.

3.2 Statechart-based DFA

Kleinmann and Wool in [12] discuss an extension to the *Pattern DFA* which involves a *Statechart DFA* to model multi periodic traffic. The scenario upon which they base their solution is that of a multi-threaded HMI which runs different patterns of queries with different scheduling frequencies.

The main idea of this solution is to use a different *Pattern DFA* to model the traffic produced by each thread and use a *Pattern Selector* ϕ to differentiate the traffic among the DFA. Whenever a packet *pkt* is received the system calls the function $\phi(pkt)$ and selects the DFA based on the result: if $\phi(pkt) = \emptyset$ then the packet is not part of any DFA alphabet and the system raises an **unknown** alert. If $\phi(pkt) = \{A\}$ then the packet is part of the alphabet of the *Pattern DFA A* which is selected to run it. Finally, if $\phi(pkt) = \{A1, A2, \ldots\}$ then the packet is part of multiple DFA alphabets and the system gives it to the DFA for which its time of arrival is closest to the expected one.

In order to identify the correct DFA among a set of candidates, the time of arrival of a packet must be compared to the expected arrival time of the next packet for each DFA. Therefore, the *Pattern DFA* is extended to keep track of time intervals between states. This extension closely resembles the one we propose in Section 4.2 although it must be noted that in our proposal the interval effectively *augments the symbols* of the *Pattern DFA*'s alphabet and is used to detect timing anomalies.

Training Phase: the *Statechart DFA* is built by splitting the training data into different channels and using the algorithm described in [6] on each of them to generate the respective *Pattern DFA*.

The main challenge of this phase, in our opinion, is the **splitting** operation, since there is no way of automatically discerning a channel from another by simply looking at the packets' contents.

We propose a solution to this problem in Section 4.3.

3.3 Further Works

Markman et al. in [14] describe the structure of the communication of a SCADA system for a water control facility as a series of bursts of queries interspersed with silence. Their system checks the time interval between queries and divides the bursts based on a threshold (packets with small intervals are part of the same burst). They argue that these bursts have *semantic meaning* (i.e., they are meant to contain a certain pattern of queries and are not the result of a buffering process).

Byres et al. in [3] use the *attack tree* representation technique to detail possible real-world threat scenarios for SCADA systems based on Modbus/TCP. According to this work, the attacks we will analyze in the following sections fall under the category of *support goals* which means that they are steps taken in order to achieve some other goal. This highlights how AMON can be useful for an early detection of potential threats and how it can help identifying attacks before the final payload execution.

Faisal et al. in [5] propose a *specifications based* IDS. Specifications are lists of allowed behaviors which are taken from design documents and manuals. They consider this approach valid for Modbus/TCP communications because of the simplicity of the protocol. This approach is indeed optimal whenever a precise specification for the behavior of a network is available and AMON can enforce particular specifications if given a specially crafted training dataset. However, this level of precision is not always realistic. Moreover, in very complex scenarios which employ lots of different Modbus/TCP functions, the precision of the approach could decrease.

Kirat and Vigna in [11] use Longest Common Subsequence (LCSS) algorithms (a category closely related to LRS algorithms) to collect evasion behavior signatures from malware. Their technique involves calculating a *diff* of two system call traces, one taken from a sandboxed environment (which is characterized by evasion behavior) and one taken from a normal environment (with the execution of some kind of payload). This information allows to identify the point where the malware behavior diverges. In their approach, they experience some problems relative to the fact that the LCSS is not always the most meaningful one, we address similar issues as detailed in Section 3.2.

Hajji et al. in [7] use a DFA to detect anomalous behavior in Europay-Mastercard-Visa (EMV) transactions. The DFA described in this work is built over a Transition State Graph which models a secure transaction. This is similar to the approach of the *Integrity*

DFA (cfr. Section 4.4) in the sense that DFA can work to "parse" some data (the fields of a Modbus/TCP packet or a series of operations and states in a transaction) and alert deviations from a defined path.

Becchi and Crowley in [1] discuss the difficulties of implementing signature based intrusion detection by translating regular expressions in DFA and propose an hybrid deterministic and non deterministic finite automaton model. This approach is different from the one we used since we attempt to build a system able to adapt automatically to different scenarios by using an anomaly based technique.

Branch et al. in [2] use a time-dependant DFA to detect *DoS* attacks. A time-dependant DFA defines time constraints on the state transitions and treats a symbol which is in the correct place in a sequence but does not respect the time constraints as a symbol not part of the sequence (thus resetting the DFA to its first state). This idea is similar in principle to the extensions we made to the *Pattern DFA* (cfr. Section 4.2) since it takes time intervals into consideration. However, our approach extends the alphabet of the DFA in order to keep its functions as simple (and fast) as possible. Another fundamental difference is in the fact that the time-dependant DFA described in [2] represent signatures for specific attacks while we take an anomaly detection approach and thus use DFA to represent normal traffic patterns in the connection.

We believe AMON can stand out among these works because of its anomaly based approach united with a very detailed modeling of the *safe* state of the communication which takes into consideration aspects like *data variations* and *packet exchange intervals*.

4 AMON

In this section, we present AMON: a DFA based IDS for industrial network traffic monitoring. AMON is built by extending the *Pattern DFA* (cfr. Section 3.1) and *Statechart DFA* (cfr. Section 3.2). This extension is done both by augmenting the capabilities of the two approaches and by combining them with the *Integrity DFA* described in Section 4.4.

In the following Subsections we first describe the system and adversary models for our work. Subsequently, we present our extensions and the new *Integrity DFA* in detail for the Modbus/TCP protocol. Finally, we show how to translate these concepts to CoAP and how to combine everything in the final IDS scheme.

4.1 System and Adversary Models

In this section, we describe the system and adversary models used for the development of AMON.

System Model: we consider ICS networks where a centralized SCADA system monitors physical processes of industrial plants. We design an hybrid network configuration where a gateway acts as *protocol translator* between legacy protocols and IIoT-ready ones. This hybrid system model allows to face upcoming industrial network implementations, where the legacy devices are coupled with recent IIoT-ready devices. Legacy industrial components are used along with innovative ones to contain re-engineering costs. We assume that AMON runs on secured hardware and cannot be compromised.

Adversary Model: we assume the attacker to have some degree of access to the network, this could be caused by direct physical access to a control room or by machines being wrongly exposed to the internet. Some of these intrusion vectors are detailed in [3].

We consider the following attack scenarios in this work:

- A *MITM* enabled by an ARP spoofing attack which can masquerade a malicious host. In this case, the attacker has direct connectivity to the central switch in the control room zone network and is able to inject packets.
- A *DoS* attack performed by flooding a network node with requests. In this case, the attacker obtains control over a HMI machine and is able to directly query the server.
- A *Buffer Overflow* attack exploited by sending packets in which the *Length* field is mismatched with the actual packet size.

4.2 Extended Pattern DFA

The *Pattern DFA* described in Section 3.1 will detect different anomalies in the communication like unusual or unusually formed queries, sequences of packets which do not reflect the normal behavior or an entity which is no longer working (because it will not send its share of the packets). However, there are some attacks and anomalies which would not be noticed:

- A *DoS* attack which attempts to flood the server by sending a high number of packets while keeping the expected pattern.
- A *MITM* attack performed via ARP spoofing.
- Integrity violations (e.g., response to a query with wrong *TI*).

To account for some of these possibilities we extended the *Pattern DFA* alphabet (Eq. 1) with an indicator of frequency. A symbol in the alphabet will thus be a 5-tuple of the form:

$$(Q, FC, RN, BC, \Delta t), \tag{2}$$

where, for query packets, Δt is a measure of the time interval from the preceding queries (we expect responses to immediately follow the specific queries so the value is always "0" for them). This value must be rounded to account for a level of variation which is expected in the normal communications of a physical network.

For each *Pattern DFA* transition we define a **Wrong query interval** variant which is indicated by the "*" symbol (e.g., a *normal** packet is a normal packet with wrong query interval). Moreover, we keep a $TI : RN$ map for each query and we use it to set the *RN* value for the corresponding response. This allows to correctly identify responses when queries on different registers but with the same *FC* are present in the pattern. We extend the check on IP address also to MAC address adding new *master/slave MAC address changed* alerts (*MstMAC*, *SlvMAC*). This ensures detection of ARP spoofing attacks.

4.3 Extended Statechart DFA

The main issue we want to tackle with this extension is the **splitting** of the packets into different channels to feed them to the correct *Pattern DFA* during detection. We propose an LRS-based algorithm to extract multiple repeating patterns from the packets' stream captured during the training phase. The *Statechart DFA*

function pseudocode can be found in Algorithm 1, the LRS function pseudocode in Algorithm 2, their symbols are detailed in Table 1.

Algorithm 1 Statechart DFA Generation

```
1: procedure SC_Gen(Seq, Pattern_List)
2:     Sub ← LRS(Seq)
3:     if (Len(Sub) > 0) then
4:         Cleared_Sub ← CS(Seq, Sub)
5:         Find_Sub(Sub, Pattern_List)
6:         Find_Sub(Cleared_Sub, Pattern_List)
7:     else if (Len(Seq) > 0) then
8:         Add Seq to Pattern_List
```

Algorithm 2 Longest Repeating Subsequence

```
1: procedure LRS(Seq)
2:     n ← Len(Seq)
3:     RST ← Empty_Table(n+1, n+1)
4:     IT ← Empty_Table(n+1, n+1)
5:     for (i ← 1 to n + 1) do
6:         for (j ← 1 to n + 1) do
7:             if (Seq[i] = Seq[j] AND i ≠ j) then
8:                 RST[i][j] ← RST[i-1][j-1]
9:                 Append(RST[i][j], Seq[i])
10:                IT[i][j] ← IT[i-1][j-1]
11:                Append(IT[i][j], i)
12:            else if (Len(RST[i][j-1]) > Len(RST[i-1][j])) then
13:                RST[i][j] ← RST[i][j-1]
14:                IT[i][j] ← IT[i][j-1]
15:            else
16:                RST[i][j] ← RST[i-1][j]
17:                IT[i][j] ← IT[i-1][j]
18:    LRS ← FNO(IT)
19:    Return LRS
```

Algorithm 3 Find Non Overlapping

```
1: procedure FNO(IT)
2:     Vertical ← FF_Vert(IT)
3:     Horizontal ← FF_Hor(IT)
4:     Remove Horizontal indexes from Vertical
5:     Return Vertical
```

Algorithm 1 starts by finding the LRS in all of the traffic captured during the training phase. The actual LRS in a long sequence of symbols could have overlapping parts as shown in the following example:

*ABCD***ABCD***ABCD*

In this example the LRS is *ABCDABCD*, the bold characters highlight the symbols which overlap (albeit at different indexes on the respective subsequences).

We want to avoid this occurrence to improve the overall performance of the algorithm (if a simple subsequence like *ABCD* was repeated n times we would need to run the LRS algorithm for $n-1$ times in order to let the base pattern emerge). We also want to avoid another problem: if a long sequence of repeating characters is interspersed with the symbols from another repeating sequence, the LRS will be prefixed with repeating occurrences of the first character of the subsequence. An example is the sequence in which the subsequence *ABCD* is interspersed with elements of the subsequence *EFGH*:

*ABCD***E***ABCD***F***ABCD***G***ABCD***H**

If this sequence is repeated up to three times, the LRS algorithm will extract the subsequence *ABCDABCDABCDABCD* which can then be reduced to *ABCD*. However, if the sequence is repeated four or more times the LRS algorithm will start prefixing the results with a number of occurrences of the first character *A*. This is indeed expected behavior for a generic string but does not adapt well to the query/response communication patterns we want to model.

To handle this problem we use the *Find Non Overlapping* (*FNO*) function which is able to cut the overlapping parts of a subsequence by looking at its index table *IT*. This function is described in Algorithm 3: in the *IT* table the indexes relative to the first occurrence of the complete LRS in the last row partially overlap with the ones relative to the first occurrence in the last column: by removing the overlapping indexes we will then remove the overlapping part of the subsequence.

After this step, if the LRS is not empty we run the algorithm again using it as an input in order to find potential nested patterns. Finally, we clear the original sequence from the subsequence's symbols in order to let other patterns emerge and we look for LRS in this cleared subsequence (*Cleared_Sub*).

A sequence which has no LRS and is not empty is considered a pattern and is added to the *Pattern_List*.

Table 1: Definitions for Algorithm 1, 2, and 3.

LRS	Longest Repeating Subsequence Algorithm
Len	Returns length of sequence
CS	Removes Sub's states from Seq
Seq	Initial sequence of states
Pattern_List	Global list of patterns
RST	Repeating Subsequence Table
IT	LRS Index Table
FNO	Find Non Overlapping subsequence
FF_Vert & FF_Hor	Find first vertical or horizontal occurrence of the subsequence in the IT table

4.4 Integrity DFA

We designed the *Integrity DFA* (shown in Figure 1) to parse the fields of each incoming packet. This Section describes it in detail.

Training Phase: the *Integrity DFA*'s training phase consists of checking the FCs used during normal communication and storing them in a list we will call *Accepted_FCs*. Moreover, response packets are analyzed in order to learn the average variation of the data which is returned: in many physical scenarios (e.g., water distribution systems) we expect to be able to predict a variation threshold for the responses of subsequent queries (i.e., a water tank can not be emptied instantly). An average data variation measure is saved for each FC, RN couple (*FCRN*), this allows us to distinguish between queries with the same FC on different registers.

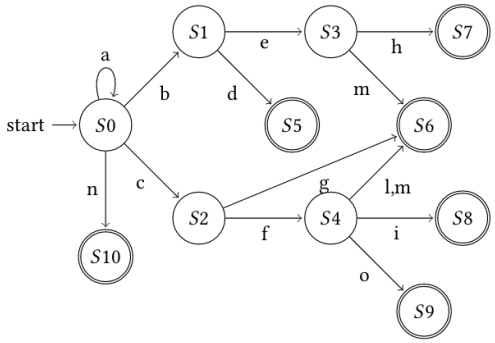

Figure 1: The *Integrity DFA*.

Anatomy: in order to perform some integrity checks we update a *Transactions* map ($TI : FC$) for every query. Moreover, we keep track of the latest data for every *FCRN* identifier in order to calculate the data variation for each response. The states of the *Integrity DFA* are:

- **S0**: starting state, a packet arrived. The DFA will remain in this state if non Modbus/TCP packets arrive (transition **a**).
- **S1**: the packet is a Modbus/TCP query.
- **S2**: the packet is a Modbus/TCP response.
- **S3**: the TI of the query is valid (i.e., this is not a retransmission).
- **S4**: this is the response to a pending query.
- **S5**: this query does not have a unique TI and is not valid $\Longrightarrow$ **Alert**.
- **S6**: this packet presents some anomalies:
  - the FC of this packet is not part of the accepted FCs (i.e., is not in *Accepted_FCs*) $\Longrightarrow$ **Alert**.
  - the FC of this response packet doesn't match the one in the query packet $\Longrightarrow$ **Alert**.
  - the TI of this response doesn't match any pending query $\Longrightarrow$ **Alert**.
- **S7**: this query packet is accepted.
- **S8**: this response packet is accepted.
- **S9**: this response packet presents an *anomalous data variation* $\Longrightarrow$ **Alert**.
- **S10**: there is a mismatch between the size indicated in the *Length* field of the packet and its actual size $\Longrightarrow$ **Alert**.

A detailed description of the transitions is in Table 2.

4.5 CoAP integration

The previous Sections described the application of the Extended *Pattern*, *Statechart*, and *Integrity DFA* to the Modbus/TCP protocol. As part of our work we extended these approaches to CoAP: in order to do so a simple translation is required (we found that only one Modbus/TCP field relevant to the DFA has no correspondence in CoAP).

The field translations are listed in Table 3, the symbol $\emptyset$ is used when there is no correspondence for a particular field, to indicate a particular option by name we use the notation *Options[Option-Name]*.

Table 2: *Integrity DFA* Transitions.

a	Not a Modbus/TCP packet
b	This is a query packet: save TI and FC in *Transactions* map
c	This is a response packet
d	The TI of this query packet is already present in the *Transactions* map
e	This is not a retransmitted query
f	The TI of this response is in the *Transactions* map
g	The TI of this response is not in the *Transactions* map
h	The FC of this query is part of the accepted ones
i	The FC of this response corresponds with the one saved in the *Transactions* map.
l	The FC of this response does not correspond with the one saved in the *Transactions* map.
m	The FC of this query is not part of the accepted ones
n	The size of the packet and the *Length* field are mismatched
o	The difference between the data of this response and the last seen data for its *FCRN* identifier is greater than expected

Table 3: Modbus/TCP to CoAP translations.

Modbus/TCP	CoAP
Transaction Identifier	Message Id + Token
Function Code	Code
Reference Number	Options[Uri-Path]
Length	$\emptyset$

4.6 Analysis and integration

AMON is the result of the combination and extension of the *Pattern DFA*, *Statechart* and *Integrity DFA*, in this Section we explain the reasons for this choice.

Combination: as seen in the previous sections the extended *Pattern DFA* and *Integrity DFA* can detect different threats in a Modbus/TCP communication and could be employed by themselves as IDS. However, because of their focus on different aspects of the communication (i.e. patterns and packet contents), we think they should be employed together in order to compensate and synergize their behavior. In Table 4, we show that there are some cases in which the combination of the analysis of the two automata generates new results.

The *Integrity DFA* is able to identify a packet classified as *Normal* by the *Pattern DFA*. Such packet could still have some suspicious aspects, like a strange TI, a mismatch between the Length field value and the actual content or some strange variation of response data. These scenarios are marked with (*) in Table 4. A packet accepted by the *Integrity DFA* could actually be seen as *Miss* or *Unknown* packet by the *Pattern DFA*. These scenarios are marked with (**) in Table 4.

Workflow: on a practical side, the training phase for AMON consists of the combination of the training phases of the modules with some additions necessary for the extensions described in Section 4.2.

Table 4: *Ext. Pattern* and *Integrity DFA* synergies.

Ext. Pattern DFA	*Integrity DFA*
Normal, Oversized, Mismatch	Will detect TI problems(*). Will detect data length problems(*). Will detect anomalous data variations(*).
Retransmission	A response packet will end in S6. A query will end in S5.
Miss	A response could end in S6. A query could be accepted(**).
Unknown	Based on the reason for the unknown transition the *Integrity DFA* could end in every state: S5 and S6 would be integrity alerts. S9 and S10 would be accepted(**).

The resulting training data includes the extended pattern description (cfr. 4.2), the set of authorized FCs and the expected data variations.

After the training phase, the detection phase starts: AMON analyzes each packet sequentially first through the extended *Pattern DFA* and then through the *Integrity DFA*. Each module produces a result state which is collected and analyzed to detect anomalous situations (cfr. Figure 2). The *Detector* module looks at *precise sequences* as well as *percentages of states* in order to decide if a situation warrants an alert. In order to find relevant sequences of states, we applied a LRS algorithm to find the longest repeating substring (which differs from a subsequence because it requires all the characters to be sequential) in the result states' sequence collected during some attack tests. More details can be found in Section 5.

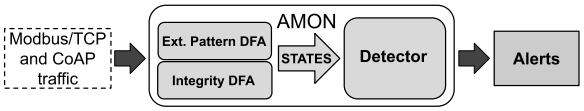

Figure 2: AMON Detection Workflow.

5 EVALUATION

This section describes the tests we ran to validate AMON against a series of attacks and the scenario in which these attacks took place.

5.1 Simulation Scenario and Tools

Testing security solutions for industrial processes is a delicate issue since actual real world attack implementations could endanger critical assets and infrastructures. For this reason, we created a test environment using Mininet, a tool which allows the simulation of a network in a virtualized environment. With this approach it is possible to avoid using expensive testbeds. Moreover, a new network configuration can be easily and quickly prototyped.

For this work, we developed a simulation environment for an hypotetical industrial scenario: a water tower which gradually empties by filling up two consumers with different consumption rates. This scenario is implemented with an hybrid network configuration which involves a server and a client configured to communicate respectively with the CoAP and Modbus/TCP protocols. The CoAP server mimics a PLC connected to three sensors, one for the water tower and the other for the consumer tanks. A gateway stands in between the server and the client and enables the communication by keeping a local database of the server's values, which is periodically updated through CoAP requests, and by providing a Modbus/TCP interface to the client. This configuration models a realistic scenario in which legacy components (i.e., the Modbus/TCP infrastructure) are used along with innovative ones to contain re-engineering costs: new sensors able to communicate over CoAP are installed in the system while the legacy Modbus/TCP HMI-PLC configuration allows communication by installing the Modbus/TCP-CoAP gateway. All the elements of the system are implemented in Python: the CoAP server and the gateway client use the CoAPthon library [18], the Modbus/TCP client and gateway server use the uModbus library[2].

AMON runs on a fourth host which passively receives a copy of each packet directed to or coming from the server through Switch Port for Analysis (SPAN) also known as port mirroring. This is configured through the Open vSwitch[3] *ovs-vsctl* tools. We represent the network topology conceived for the tests in Figure 3.

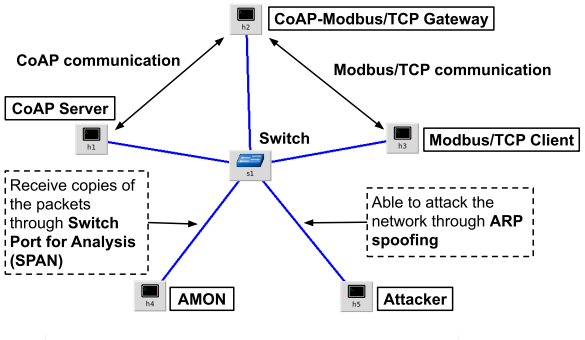

Figure 3: Network configuration for the attack tests.

The normal traffic pattern for this network consists of a polling cycle of CoAP requests which every second queries for the values of the three tanks ($t1, t2, t3$) in order to keep the gateway local database updated. The Modbus/TCP client sends queries for the values of $t1$ and $t2$ (every 2 seconds) and for the value of $t3$ (every 10 seconds).

5.2 Attack Tests

To test the validity of our approach, we implemented a series of attacks which explicitly attempt to sidestep the Extended *Pattern DFA* and *Integrity DFA* detection capabilities.

We considered two categories of attacks:

- **Malicious client**: the attacker obtains the control of a client and is able to query the server directly. The attacks and results are described in Subsections 5.2.2 and 5.2.3.
- ***MITM* with *ARP spoofing***: a simple ARP spoofing attack can deviate the client/server communications through a malicious host. This could lead to the obfuscation of physical parameters of the system's sensors or to the execution of arbitrary operations by the actuators. The attacks and results are described in Subsections 5.2.4, 5.2.5, and 5.2.6.

[2]https://github.com/AdvancedClimateSystems/uModbus/
[3]http://www.openvswitch.org/

We conducted each attack separately against the Modbus/TCP and CoAP links.

To evaluate these tests we check the **capability** of detecting an ongoing attack, the **time** required, the emergence of a particular **state sequence** which can be used as an attack signature by the *Detector* (and its coverage of the complete sequence of states produced) and finally the emergence of particular **percentages** of states (to use along with the attack signature to get more attack details). We represent *Integrity DFA* states with the concatenation of the states traversed by the packet, for example the state S0S1S3S7 indicates an accepted query.

5.2.1 Normal Traffic. We tested the behavior of the system under normal traffic to ensure the avoidance of false positives. The state percentages for this test are listed in Table 5. The results highlights that the Extended *Pattern DFA* only produces *normal* states with a small percentage of delayed *normal** packets. The *Integrity DFA* accepts every query (*S0S1S3S7*) and response (*S0S2S4S8*).

Table 5: Normal Traffic.

State	Modbus/TCP	CoAP
normal	99.24%	99.61%
normal*	0.76%	0.39%
S0S1S3S7	50%	50%
S0S2S4S8	50%	50%

The LRS of states emerged during this test covers 96.42% of the states produced: *S0S1S3S7, normal, S0S2S4S8, normal*. This is the expected pattern of states since it shows an accepted/normal query followed by an accepted/normal response.

5.2.2 Standard DoS. In this attack, a malicious client floods the server with a specific query in order to impede its functions. The query is one of the accepted queries collected during the training phase (requesting the value relative to the tank *t1*) in order to avoid the raising of *unknown* and *integrity* alerts.

Modbus/TCP link attack: the state percentages for the attack on the Modbus/TCP link are presented in Table 6. The Modbus/TCP states show a high percentage of *miss** statuses since the query flooded is part of the accepted pattern. The CoAP traffic shows increased delays caused by the ongoing attack.

Table 6: Standard DoS results: Modbus/TCP.

State	Modbus/TCP	CoAP
normal	50%	61.11%
normal*	0.04%	38.89%
miss*	49.96%	0%
S0S1S3S7	50%	50%
S0S2S4S8	50%	50%

The LRS of states emerged during this attack covers 99.93% of the states produced: *S0S1S3S7, normal, S0S2S4S8, miss**. This sequence of states is semantically correct for the *Standard DoS* attack in this scenario and can be used to quickly identify it.

CoAP link attack: the CoAP link attack is somewhat different from the Modbus/TCP one because in this case it's not the CoAP client running on the gateway which performs the attack but a third malicious client. We chose this configuration because we consider the gateway as part of the safe system (malicious exploitation of the gateway would lead to different possible attacks). The state percentages for the CoAP link attack are listed in Table 7. This attack affects the quality of the CoAP link communications (but not so greatly the Modbus/TCP communications) and produces a wider range of different states.

Table 7: Standard DoS results: CoAP.

State	Modbus/TCP	CoAP
normal	97.83%	12.79%
normal*	2.17%	7.04%
miss*	0%	14.93%
miss	0%	10.66%
retransmission*	0%	48.61%
mismatch	0%	5.97%
S0S1S3S7	50%	80.81%
S0S2S4S8	50%	19.19%

The LRS of states emerged during this attack covers 49% of the states produced: *S0S1S3S7, MstMAC, retransmission**. This sequence of states shows that the IDS is able to recognize the external nature of the attack (the *MstMAC* state) and can be used to identify it.

5.2.3 Smart DoS. A skilled attacker could attempt to fool the *Pattern DFA* by performing a DoS attack using the normal communication pattern to flood the server.

Modbus/TCP link attack: the state percentages for this attack on the Modbus/TCP link are listed in Table 8. As for the *Standard DoS*, the attack disturbed the CoAP communication link. The Modbus/TCP link states are divided between *normal* and *normal** because the pattern is respected, the percentage of delayed packets is vastly increased with respect to the normal traffic. We use this information to complement the LRS of states described below. Specifically, we recognize a *Smart DoS* attack only after that at least 20% of the states are *normal**.

Table 8: Smart DoS results: Modbus/TCP.

State	Modbus/TCP	CoAP
normal	71.5%	62.5%
normal*	28.5%	37.5%
S0S1S3S7	50.02%	50%
S0S2S4S8	49.98%	50%

The LRS of states emerged during this attack covers 86% of the states produced: *S0S1S3S7, normal, S0S2S4S8, normal*, S0S1S3S7, normal, S0S2S4S8, normal*. This is not a very unusual sequence of states even for a normal communication so we combine it with percentage information to decide if an alert should be raised.

CoAP link attack: as in the *Standard DoS*, we conducted this attack from a third malicious host. The state percentages for the CoAP link attack are listed in Table 9.

Table 9: Smart DoS results: CoAP.

State	Modbus/TCP	CoAP
normal	90.91%	13.94%
normal*	9.09%	4.85%
miss*	0%	70.91%
miss	0%	3.64%
retransmission	0%	1.21%
mismatch	0%	3.03%
S0S1S3S7	50%	85.45%
S0S2S4S8	45.45%	14.55%
S0S2S4S9	4.55%	0%

The LRS of states emerged during this attack covers 69% of the states produced: *normal, MstMAC, miss**. This sequence of states shows how the attack actually disrupts the normal pattern of communication (i.e., the obfuscation does not work) as we have a large number of *miss* states.

5.2.4 Traffic sniffing. In this attack the MITM simply receives and forwards all the packets of the communication without modifying them.

Modbus/TCP link attack: we present the state percentages for this attack on the Modbus/TCP link in Table 10. The attack causes delays in the communication (increased *normal** rate), but no other alert states.

Table 10: Traffic Sniffing: Modbus/TCP.

State	Modbus/TCP	CoAP
normal	91.0%	99.05%
normal*	9.0%	0.95%
S0S1S3S7	50%	50%
S0S2S4S8	50%	50%

The LRS of states emerged during this attack covers 78% of the states produced: *S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal*. This sequence of states correctly identifies the ongoing attack since it highlights an accepted query received from a different client (*MstMAC*) and an accepted response from a different server (*SlvMAC*). This sequence is common to most ARP spoofing attacks, so we will use other details to differentiate them.

CoAP link attack: the state percentages for the CoAP link attack are listed in Table 11. The attack causes more delays on this link as the higher percentage of *normal** packets shows.

Table 11: Traffic Sniffig: CoAP.

State	Modbus/TCP	CoAP
normal	97.56%	72.84%
normal*	2.44%	27.16%
S0S1S3S7	50%	50%
S0S2S4S8	50%	50%

The LRS of states emerged during this attack covers 64% of the states produced: *S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC*. This sequence of states is very similar to the Modbus/TCP link one and the same observations we made apply also to this attack.

5.2.5 Buffer overflow attempt. Depending on its implementation, a server could have *Buffer Overflow* vulnerabilities and an attacker could try to exploit them by appending raw bytes to a packet. This will result in a mismatch between the length indicated in the packet fields and the actual size of the packet. This attack was only tested on the Modbus/TCP link since on CoAP packets there are no equivalent *Length* fields.

Modbus/TCP link attack: we list the state percentages for the attack on the Modbus/TCP link in Table 12. The Extended *Pattern DFA* states show the disturbance caused by the ongoing attack. The *Integrity DFA* states correctly highlight the presence of packets bigger than expected (*S0S10*), the presence of *integrity alerts* (*S0S2S6*) is caused by the fact that the overflown queries' TI are not saved, causing the relative responses to raise the alerts.

Table 12: Buffer Overflow: Modbus/TCP.

State	Modbus/TCP	CoAP
normal	45.38%	98.89%
normal*	1.87.0%	1.11%
retransmission	4.42%	0%
miss*	4.72%	0%
retransmission*	4.32%	0%
miss	39.29%	0%
S0S1S3S7	1.47%	50%
S0S2S4S8	1.47%	50%
S0S10	48.53%	0%
S0S2S6	48.53%	0%

The LRS of states emerged during this attack covers 64% of the states produced: *S0S10, SlvMAC, S0S2S6, MstMAC, normal, S0S10, SlvMAC, miss, S0S2S6, MstMAC, miss*.

5.2.6 Replay. A way to perform a *Replay* attack is to change the data fields of a packet before forwarding it to the client. In this way, the packet will be accepted by the client and it will respect the pattern.

Modbus/TCP link attack: the state percentages for the attack on the Modbus/TCP link are listed in Table 13. This attack raises a few *Anomalous data variation* alerts when the replay begins. We can use this small percentage to identify and differentiate it from a simple *Traffic Sniffing*. We identify a *Replay* attack if we see between 0% and 10% of *S0S2S4S9* states.

Table 13: Replay: Modbus/TCP.

State	Modbus/TCP	CoAP
normal	92.41%	99.69%
normal*	7.59%	0.31%
S0S1S3S7	50%	50%
S0S2S4S8	48.71%	50%
S0S2S4S9	1.29%	0%

The LRS of states emerged during this attack covers 78% of the states produced: *S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal*. This is equal to most of the ARP spoofing states sequences so we must use percentage information to raise a precise alert.

CoAP link attack: the state percentages for the CoAP link attack are listed in Table 14.

Table 14: Replay: CoAP.

State	Modbus/TCP	CoAP
normal	97.12%	64.14%
normal*	2.88%	35.86%
S0S1S3S7	50%	50%
S0S2S4S8	47.12%	8.48%
S0S2S4S9	2.88%	1.52%

The LRS of states emerged during this attack covers 69.5% of the states produced: *S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal**. The interesting part of this sequence is the higher impact of the attack on the timings of the packets, hence the presence of a *normal** state.

5.2.7 Attack Detection. The presence of anomalous states in the DFA results can provide the evidence of malicious activities in the communications. We developed the AMON's *Detector* module to identify specific ongoing attacks, by keeping a list of LRS signatures and percentages of states. This module is responsible for raising the alerts and it is fundamental to filter the noise of the intermediate level of information provided by the DFA states.

Table 15 and Table 16 summarize the signatures, the state percentages (not necessary for every attack, the symbol $\emptyset$ is used when the signature is sufficient to identify the attack), and show the interval of time between the beginning of the attack and the raising of the alert that we observed during the tests. These results show how AMON is able to combine the data produced by various network communications anomalies into effective alerts in a quick and responsive way.

Table 15: Modbus/TCP: Signatures and Timeliness.

Signature	Percentages	Δt	Alert
S0S1S3S7, normal, S0S2S4S8, miss*	$\emptyset$	18ms	Standard DoS
S0S1S3S7, normal, S0S2S4S8, normal*, S0S1S3S7, normal, S0S2S4S8, normal	$normal* > 20\%$	74ms	Smart DoS
S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal	$\emptyset$	1.3s	Traffic Sniffing
S0S10, SlvMAC, S0S2S6, MstMAC, normal, S0S10, SlvMAC, miss, S0S2S6, MstMAC, miss	$\emptyset$	57ms	Buffer Overflow
S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal	$0\% < S0S2S4S9 < 10\%$	2s	Replay

Table 16: CoAP: Signatures and Timeliness.

Signature	Percentages	Δt	Alert
S0S1S3S7, MstMAC, retransmission*	$\emptyset$	295ms	Standard DoS
S0S1S3S7, MstMAC, retransmission*	$\emptyset$	62ms	Smart DoS
S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC	$\emptyset$	622ms	Traffic Sniffing
S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal*	$0\% < S0S2S4S9 < 10\%$	2.7s	Replay

6 CONCLUSION

In this work, we developed AMON, an IDS framework based on DFA for Modbus/TCP and CoAP traffic monitoring. The results obtained implementing AMON in a simulated hybrid industrial scenario show the usefulness of the conceived security system. In future work, we will extend AMON considering real IIoT testbeds implementations. Moreover, we will develop active features and evolve AMON to an Intrusion Prevention System.

ACKNOWLEDGMENTS

This work was supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the LOCARD project (G.A. no. 832735). This work is also supported by a grant of the Italian Presidency of the Council of Ministers and by CyBrain Srl.

REFERENCES

[1] Michela Becchi and Patrick Crowley. 2007. A hybrid finite automaton for practical deep packet inspection. In *2007 ACM CoNEXT conference*. ACM, 1.
[2] Joel Branch, Alan Bivens, Chi Yu Chan, Taek Kyeun Lee, and Boleslaw K Szymanski. 2002. Denial of service intrusion detection using time dependent deterministic finite automata. In *Proc. Graduate Research Conference*. 45–51.
[3] Eric J Byres, Matthew Franz, and Darrin Miller. 2004. The use of attack trees in assessing vulnerabilities in SCADA systems. In *Proceedings of the international infrastructure survivability workshop*. Citeseer, 3–10.
[4] Anton Cherepanov. 2017. WIN32/INDUSTROYER: a new threat for industrial control systems. *White paper, ESET (June 2017)* (2017).
[5] Mustafa Faisal, Alvaro A Cardenas, and Avishai Wool. 2016. Modeling Modbus TCP for intrusion detection. In *2016 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 386–390.
[6] Niv Goldenberg and Avishai Wool. 2013. Accurate modeling of Modbus/TCP for intrusion detection in SCADA systems. *International Journal of Critical Infrastructure Protection* (2013), 63–75. https://doi.org/10.1016/j.ijcip.2013.05.001
[7] Tarik Hajji, Noura Ouerdi, Abdelmalek Azizi, and Mostafa Azizi. 2018. EMV Cards Vulnerabilities Detection Using Deterministic Finite Automaton. *Procedia Computer Science* 127 (2018), 531–538.
[8] Markel Iglesias-Urkia, Adrián Orive, and Aitor Urbieta. 2017. Analysis of CoAP implementations for industrial internet of things: a survey. *Procedia Computer Science* 109 (2017), 188–195.
[9] Dragos Inc. 2017. TRISIS Malware: Analysis of Safety System Targeted Malware. https://dragos.com/blog/trisis/TRISIS-01.pdf.
[10] Stamatis Karnouskos. 2011. Stuxnet worm impact on industrial cyber-physical system security. In *IECON 2011-37th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 4490–4494.
[11] Dhilung Kirat and Giovanni Vigna. 2015. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 769–780.
[12] Amit Kleinmann and Avishai Wool. 2015. A statechart-based anomaly detection model for multi-threaded SCADA systems. In *International Conference on Critical Information Infrastructures Security*. Springer, 132–144.
[13] Mark V Lawson. 2003. *Finite automata*. Chapman and Hall/CRC.
[14] Chen Markman, Avishai Wool, and Alvaro A Cardenas. 2017. A New Burst-DFA Model for SCADA Anomaly Detection. In *Proceedings of the 2017 Workshop on Cyber-Physical Systems Security and PrivaCy*. ACM, 1–12.
[15] Modbus. 2004. MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b3.
[16] Modbus. 2004. Modbus messaging on TCP/IP implementation guide. *v1.0b* (2004).
[17] Zach Shelby, Klaus Hartke, Carsten Bormann, and B Frank. 2014. RFC 7252: The constrained application protocol (CoAP). *Internet Engineering Task Force* (2014).
[18] Giacomo Tanganelli, Carlo Vallati, and Enzo Mingozzi. 2015. CoAPthon: Easy development of CoAP-based IoT applications with Python. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, 63–68.