Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2016) 1–14

Procedia Computer Science

Exploiting Traffic Periodicity in Industrial Networks

Rafael R. R. Barbosa[a], Ramin Sadre[b], Aiko Pras[c]

[a] *3D Hubs*
[b] *Université Catholique de Louvain*
[c] *University of Twente*

Abstract

Industrial Control System (ICS) networks play a major role in the operation of large industrial facilities. Due to the polling mechanism typically used to retrieve data from field devices, ICS traffic exhibits strong periodic patterns. In this paper, we present a novel approach that uses message repetition and timing information to automatically learn traffic models that capture these periodic patterns. We demonstrate the feasibility of our approach based on three traffic traces collected at real-world industrial networks. We present two practical applications for the learned models. First, we discuss how they can be used in a intrusion detection system. The learned models can be seen as a whitelist of the valid commands and frequency at which they are sent, and thus could be used to detect data injection and denial of service attacks. Second, we discuss how the models can be used for the generation synthetic traffic traces, which in turn can be for testing intrusion detection systems and for evaluating performance of ICS devices, among other purposes.

Keywords:
ICS, SCADA, Modeling, IDS

1. Introduction

Industrial Control System (ICS) networks are commonly deployed to aid the operation of large industrial facilities. Historically, ICS networks were composed by special-purpose embedded devices communicating through proprietary protocols. Since the last decade, however, a trend toward adopting the TCP/IP protocol stack and using commercial off-the-shelf devices has been observed.

Despite the usage of TCP/IP, the communication in ICS networks differs significantly from what can be observed in the Internet or a office LAN. In an ICS network, data is continuously retrieved from field devices (Programmable Logic Controllers, PLCs), so that a real time view of the supported industrial processes can be established. Typically, data is retrieved through an automated and periodic polling process run by the ICS server(s), while manual interventions, such as sending commands to field devices, are rare. As a consequence regularities arise: the set of communicating nodes is stable and the traffic exhibits strong periodic patterns [1]. Well-known characteristics of Internet traffic [2], such as self-similarity or heavy-tailedness, are not generally present [3].

In this paper, we present a novel and automated approach called *PeriodAnalyser* to learn models of ICS traffic. Having traffic models for ICS networks can be beneficial for several application areas. For example, the trend toward TCP/IP based protocols has made ICS networks more vulnerable to the same threats that plague traditional IT networks [4]. In the presence of a large number of zero-day attacks, anomaly-based intrusion detection for ICS is of special interest [5]. In IT networks, anomaly-based approaches typically present a high false-alarm rate due to the

enormous variability of network traffic [6]. In contrast, the regularities present in ICS networks make anomaly detection very promising. The models learned by *PeriodAnalyser* can be seen as a whitelist of all valid commands and the frequency at which they are sent, therefore providing protection against a number of attacks, including data injection and denial of service. Traffic models are also needed to generate synthetic traffic traces. Such traces, combined with attack traffic, could be used as ground truth for testing and evaluating IDS. Alternatively, the synthetic traces could be used for evaluating the performance of ICS devices.

To the best of our knowledge, we are the first to propose an automated approach that *directly* exploits periodicity in ICS traffic. We use message repetition and timing information to detect periodic cycles in the traffic, thus addressing the limitations of existing approaches, which are discussed in Section 2. We validate our approach using traffic traces collected at operational ICS networks: two water treatment facilities and one electric and gas utility. Although we focus on two protocols present in our datasets, Modbus[1] and MMS[2], the principles presented here should be applicable to other control protocols as well.

The remainder of this paper is organized as follows. In Section 2 we discuss existing approaches to learn periodic patterns and their limitations. We then describe our novel approach in Section 3 and evaluate it using three real-world traffic traces in Section 4. In Section 5, we discuss possible application areas. Finally, in Section 6, we present our conclusions and propose future work.

2. Related Work

We start by noting that the problem of periodic traffic pattern identification is different from the classical problem of cycle detection, such as the one solved by Floyd's algorithm [7]. The latter considers the problem of (efficiently) detecting that a (large) sequence has become periodic under the assumption that the sequence is *perfectly* periodic.

The problem considered in our work is similar to the problem of periodicity detection in temporal data studied by the *data mining* community [8]. In this context, the problem consists in finding repeating patterns in a time series of symbols from a certain alphabet, typically represented as a string, such as `abcdabyzabfg`. For instance, the approach proposed by [9] allows for *imperfect* patterns, i.e., patterns that do not reoccur in every cycle, and *partial* patterns, i.e., only a subset of the pattern is periodic. The string `ab` is considered periodic in the example above.

The difficulty in applying such methods to network traffic is to define a proper time bin size (i.e., the interval between two symbols) for creating the time series. If the chosen time bin size is too large, unrelated network messages end up in the same time bin and hence are represented by only one symbol, obfuscating the periodic pattern. A very small time bin size could be used to avoid this problem to some extent[3], however this solution is not efficient since it results in many empty time bins and long symbol strings.

Spectral analysis has been more commonly used to uncover periodicity in network traffic (e.g., [10, 11, 12]). Van Splunder [13] proposes an approach based on the variance of packet inter-arrival times to detect network traffic periodicity. In a previous work [14], we investigated the idea of detecting anomalies in the periodic behavior of ICS traffic using tools such as Discrete Fourier Transforms (DFT) or Autocorrelation Functions (ACF). The main shortcoming of such methods is the so-called *semantic gap* caused by the fact that these approaches operate on a signal generated from the observed network packets (for example, the number of packets or bytes sent per time interval). While it is relatively easy to detect periodic activities in the signal, little insight is provided on which packets caused the periodic behavior [9].

The work by Goldgenberg and Wool [15] is more closely related to our own. Based on the observation that the traffic exchanged between a Human-Machine Interface (HMI) and a PLC in an ICS network consists of requests for the same values being sent periodically, they proposed to model Modbus traffic by means of *Deterministic Finite Automata* (DFA). The automaton captures the *order* in which requests and their respective responses are normally exchanged and triggers alarms when an unexpected transition, i.e., an unexpected sequence of two messages, is observed. The proposed approach is able to automatically learn the automaton from a training set. However, since

[1] Modbus Messaging On TCP/IP Implementation Guide V1. 0a.

[2] ISO/IEC 9506 - Manufacturing Message Specification.

[3] Note that TCP is allowed to merge multiple application Protocol Data Unit (PDU) into a single segment, causing these PDUs to be observed at identical times.

only a single automaton is used per connection, a connection carrying requests sent with different periods will result in large models. Moreover, small fluctuations that change the relative order of messages are not captured by the model. The authors propose a two-level approach which reduces (but does not eliminate) the problem. A second limitation is that the model only captures the order of messages but not their inter-arrival times. For instance, the approach cannot distinguish between the case where a certain sequence of requests is sent every 10 minutes and the case where the same requests are sent every 10 milliseconds.

More recently, Caselli et al. [16] proposed a general approach to detect so called *sequence attacks*, that is, crafting a certain sequence of "valid" events (e.g., network messages, log entries and variable values) in a way that it has an adverse impact on the system. The proposed Sequence-aware Intrusion Detection System (S-IDS) models the normal behavior using Discrete-Time Markov Chains (DTMCs). Anomalies are detected when unknown states are reached or unlikely or unknown transitions are taken. Although periodic behavior can be indirectly captured as a sequence of states, complex periodic patterns (multiple periods with possible drifting because of timing variation etc.), as addressed by our work, would either result in large models or compact models that only provide a fuzzy description of the real process.

3. PeriodAnalyser

In this section we describe *PeriodAnalyser*, our novel approach to learn periodic traffic patterns. Although our approach is not based on DFAs, it has in common with [15] that communication to and from PLCs is modeled as a series of requests (and their responses).

PeriodAnalyser is composed by three modules. First, the monitored network traffic is preprocessed by the *Multiplexer*, which separates the traffic into different flows. Then, the *Tokenizer* transforms each packet into a protocol-independent format, called *token*. Finally, each *token* is then processed by the *Learner* with the goal to identify and characterize periodic activities (called *cycles* in the following). The information of all identified cycles constitutes the model of the communication in the ICS network.

We acknowledge that the information captured by the *Learner* might (partially) already be known to the ICS system operators or, alternatively, it could be extracted from configuration files [17]. Consequently, one could make use of this information instead of learning it from the network traffic, eliminating the need for the *Learner* module. However, in our experience with real-world utilities, this information is not readily available or it is incomplete at best.

Before we explain the three modules in Sections 3.2 to 3.4, we first discuss the different requirements to *PeriodAnalyser* that have influenced its design in Section 3.1.

3.1. Requirements

In the following we present the requirements to our approach.

Multiple cycles. Multiple activities with different periods can happen on the same network connection. For instance, consider that the requests R_1, R_2, R_3 and R_4 are sent by the server to the field device every 1 s, 1 s, 1 s and 2 s, respectively. In our approach, two cycles should be identified: one cycle with a period of 1 s consisting of the requests R_1, R_2 and R_3, and one cycle with a period of 2 s containing only the request R_4.

Periodicity at different aggregation levels. Periodic patterns can be observed on "short-lived" or "long-lived" network connections. An ICS application might establish a new TCP connection for each cycle of the periodic message exchange, resulting in many short-lived connections, or it might establish one long-lived connection that persists for several minutes or hours.

Request reordering. The order of requests inside a cycle iteration is not necessarily fixed. For instance, iterations R_1, R_2, R_3 and R_1, R_3, R_2 should be considered equal. The lack of order can be caused, for instance, by a multithreading implementation of the application issuing the requests, where multiple threads compete for access to the network interface.

Table 1. Mapping Modbus and MMS PDUs to tokens

Protocol	PDU	Message Identifier	Pair Identifier
Modbus	request	*length field, unit identifier, function code, data*	*transaction ID*
	response	–	*transaction ID*
MMS	confirmed request	*confirmedServiceRequest* and optional fields	*invokeID*
	confirmed response	–	*invokeID*
	unconfirmed request	*unconfirmedService* and optional field	None
	initiate request	"initiate"	"initiate"
	initiate response	–	"initiate"
	conclude request	"conclude"	"conclude"
	conclude response	–	"conclude"

Timing variations. Small variations in the cycle timing should be expected. Delays and jitter can be either introduced by the network [18] or by the server generating the requests, for instance the application can be preempted if the load on the server is high. Besides delays, packets can be lost, either by the network or by the measurement equipment.

3.2. Multiplexer

The objective of this module is to filter the ICS protocol packets, discarding the non-ICS traffic, and to multiplex the packets into different *flows*.

In this work, we consider two protocols used in ICS environments: Modbus/TCP and MMS. The filtering task is performed by identifying all packets that contain a Modbus or MMS application header. Both protocols use TCP as transport layer and we also discard packets that do not carry application data, such as TCP ACK packets or TCP handshaking packets.

We observed two types of TCP connections in our datasets: long-lived connections containing requests sent at periodic intervals and short-lived connections established (and tore down) at periodic intervals. After filtering, packets belonging to the first type are grouped to a flow using the 5-tuple *(Server Address, IP Protocol, Server Port, Client Address, Client Port)* as grouping key. For the latter type, we use the 4-tuple *(Server Address, IP Protocol, Server Port, Client Address)* to aggregate such connections to one single flow. In practice, we detect short connections based on their duration. In our datasets, short connections typically have a duration below one second.

3.3. Tokenizer

The goal of this module is to transform the packets that have been filtered and grouped by the Multiplexer to a common protocol-independent format, which is later used by the Learner module. As a consequence, the Tokenizer and the Multiplexer are the only modules that need to be adapted in order to support new protocols.

For each observed packet, the Tokenizer outputs a tuple formed by its time stamp and a *token*. If the packet contains a request message the token consists of two parts: a *message identifier*, used to identify the request, and a *pair identifier*, used to pair request and response messages. If the packet carries a response message, the token consists only of the *pair identifier*. Remember that, whereas we expect requests (e.g., "get pump pressure") to be periodically repeated, the contents of the responses (e.g., the pressure value) are likely to vary considerably.

The token is based on application header fields, therefore its generation is protocol dependent. Table 1 lists the protocol fields extracted from Modbus and MMS PDUs to generate the *message identifier* and *pair identifier* that form a token.

3.4. Learner

The Learner processes the output of the Tokenizer flow by flow with the goal to identify all cycles inside a flow. To this end, the sequence of requests in a flow is split into *candidate iterations*. If N identical candidate iterations

```
Input  : A tokenized flow and parameters N, ϵ, dur_thr
Output: A list of tuples (request set, dur_min, dur_max, dur_std)
/* Step 1:  group requests */
count request occurrences ;
group requests with same counter ±ϵ;
for each group do
    for each subset in group do
        /* Step 2:  find candidates */
        for each request in subset do
            candidates ← N repeating cycles;
            /* Step 3:  test candidates */
            dur_min, dur_max ← minimum and maximum candidate duration;
            if dur_max − dur_min < dur_thr
                dur_std ← cycle duration standard deviation;
                store (request set, dur_min, dur_max, dur_std);
                continue to next subset;
            else
                reset candidates;
        end
        ignore remaining subset requests as non-periodic
    end
end
```

Algorithm 1: The Learner algorithm

are observed, the algorithm concludes that there is enough evidence for a cycle. Unlike [15], we do not require that requests are sent in a fixed sequence to identify candidates. Instead, we assume that repeated requests delimit candidates: once a repeated request is observed a new candidate starts. This requires that the same request does not appear in different cycles. Issuing the same request in different cycles, i.e., with different frequencies, is inefficient and should not happen in an ICS system in practice.

For instance, consider the following sequence `abcdcdbadcbaabcd` of periodically sent requests `a`, `b`, `c`, and `d`. The second appearance of `c` marks the begin of a new candidate. As a result, we obtain four candidates `abcd`, `cdba`, `dcba` and `abcd`, each containing all four requests. If we started to capture traffic at a later moment, say, at the first `d`, the first candidate would be `dc`, the second candidate would be `dba`, and the third and fourth would be `dcba` and `abcd`, respectively. Following our definition, only the last two candidates would be considered as identical (and identified as a cycle if $N = 2$), while the first two would be ignored. This illustrates that the algorithm is able to synchronize itself to the cycle.

Once we have identified a group of N identical candidates, we verify if they show a regular timing behavior. To this end, we calculate the duration of each candidate. The duration of a candidate is defined as the time between its first request and the first request of the following candidate. If the difference between the minimum and maximum duration of the candidates is above a threshold dur_{thr}, the candidates are rejected since it indicates that the grouping of the requests might have occurred by chance. In our experiments, we found that a dur_{thr} of 1 s worked well. Such a delay variation should not happen in most healthy ICS network even with highly fluctuating network latencies.

We propose a pre-processing step to handle multiple cycles with different periods and non-periodic messages more efficiently. It consists in grouping requests according to their number of occurrences. The idea is that requests sent with the same frequency will occur roughly the same number of times in the training data. In case multiple cycles are present in a flow, their requests should have different numbers of occurrences and, thus, be separated at this step. Similarly, non-periodic requests are likely to have a different number of occurrences from periodic requests. Note, however, that even messages in the same cycle might have a different number of occurrence, for instance, if the duration T of the training set is not a multiple of the cycle duration, or if requests are lost. We introduce a tolerance ϵ to deal with this issue. Requests with the same number of occurrences, more or less ϵ, belong to the same group. In our tests, we set ϵ to 1.

Algorithm 1 shows the pseudo-code for the Learner. The algorithm can be split into 3 steps:

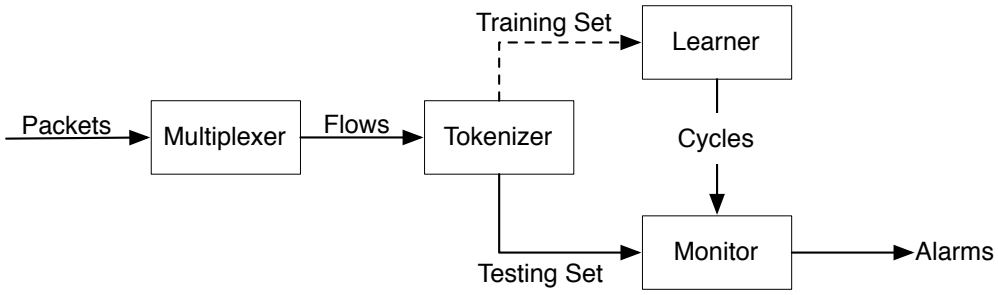


Figure 1. *PeriodAnalyser* evaluation approach

1. **Group Requests.** Count the number of occurrences of each request, and form groups of requests with the same number of occurrences $\pm\epsilon$.

2. **Find Candidates.** Search each group for cycle candidates. A candidate starts as an empty set. Requests are added to the candidate, until a request is repeated, triggering the creation of a new candidate. If N identical candidates are found, proceed to step 3. If it fails, iteratively test all subset combinations of requests in the group. Requests failing all subsets are reported as non-periodic. The message order in a candidate is irrelevant.

3. **Test candidates.** Calculate the minimum duration dur_{min} and the maximum duration dur_{max} of all candidates, and verify if the difference is above a threshold dur_{thr}. If the test succeeds, return the set of requests in the cycle, its duration range dur_{min}, dur_{max}, and the standard deviation of its duration dur_{std}. If it fails, go back to step 2 and find another candidate set.

The time complexity of the algorithm is exponential with the number of messages of the largest group identified in the first step of the algorithm, as all subset combinations are tested. Suppose n is the size of the largest group. In the worst-case, where all requests are not periodic, the algorithm has to test $\binom{n}{n}+\binom{n}{n-1}+\binom{n}{n-2}+\cdots+\binom{n}{1} = 2^n-1$ combinations. Note, however, that we do not test all requests combinations blindly. In step 1, initial groups are formed by grouping requests with roughly the same number of occurrences. Furthermore, non-periodic requests might be discarded when searching for candidates in step 2. Keep in mind that we assume that periodic request-response pairs represent the majority of the traffic. In practice we always found the Learner to finish after a relatively small time. In the worst tested case, it took 100 ms to learn 30 min of traffic on a 2.66 GHz Intel Core 2 Duo.

4. Evaluation

The goal of this section is to evaluate our approach using datasets collected in real ICS networks. The central questions are whether empirical ICS traffic follows our assumptions on periodicity and whether *PeriodAnalyser* is able to accurately learn its characteristics. Furthermore, we study how much input data the Learner needs and whether flows identified as periodic change their behavior later.

To perform the evaluation, we propose a new module, called *Monitor*, that is able to detect deviations in the traffic from the cycle models learned by *PeriodAnalyser*. Figure 1 depicts our approach. The traffic is first preprocessed by the *Multiplexer* and the *Tokenizer*. During a training phase, part of the dataset is processed by the *Learner* with the goal to identify and characterize the cycles. Once the training phase has been concluded, the Monitor uses this cycle information to detect deviations in the remainder of the dataset. Obviously, the Monitor works like an *anomaly detector*, with the cycle information describing the normal (expected) communication behavior, and we will indeed discuss in Section 5.1 how *PeriodAnalyser* and the Monitor can be used to build an anomaly-based intrusion detection system.

In the following, we describe our empirical datasets in Section 4.1 and the Monitor in Section 4.2. In Section 4.3, we verify whether our assumptions regarding connections to field devices are valid, that is, if the flows present in our

Table 2. Datasets overview

name	# hosts	duration	avg. pkts/s	avg. KB/s
Modbus	45	13 days	504.1	82.5
MMS 1	14	10 days	28.7	5.1
MMS 2	11	1.5 days	137.8	24.0

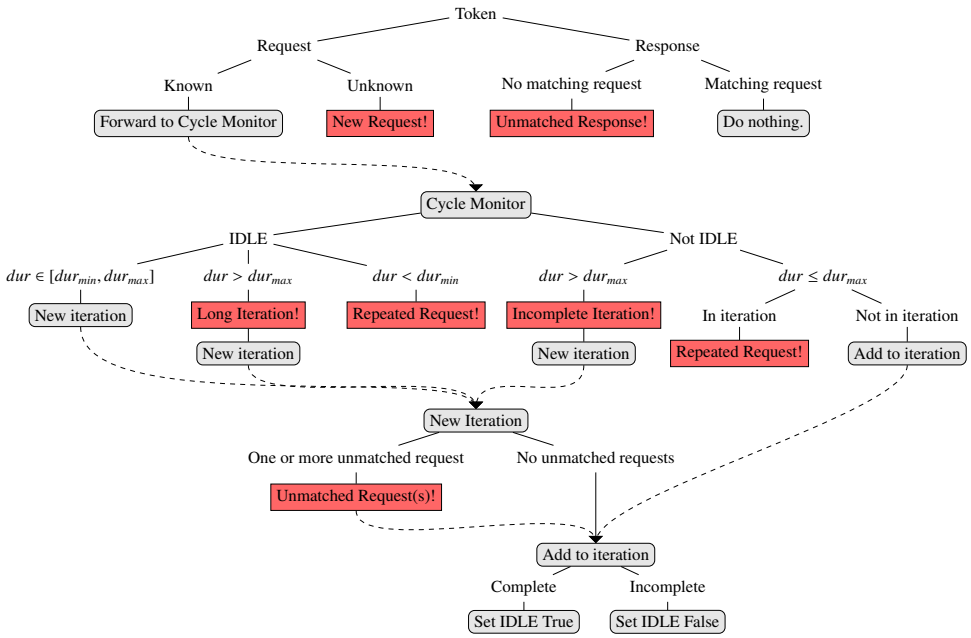


Figure 2. Decision graph for the Monitor algorithm

dataset consist only of a series of periodic requests and their responses. In Section 4.4, we study the impact of the number of required candidates N in the Learner on the number of deviations detected by the Monitor.

4.1. Datasets

For our analysis, we use network traffic traces collected at three real-world ICS networks: 2 operated by water treatment facilities and one gas utility. One location uses Modbus and the other two use MMS. The duration of the datasets range from 1 to 13 days of data and each dataset contains 11 to 45 hosts. An overview of the datasets is shown in Table 2.

We split each flow in the datasets in two parts: a training and a testing set. Unless stated otherwise, the training set consists of the first 30 min of a flow, and the testing set is the remainder of the traffic. Considering that all datasets contain more than one day of data, and that the expected periodicity is in the order of a few seconds, this choice is a good compromise between a sufficient number of samples and the length of the datasets.

4.2. Monitoring periodic behavior changes

We describe the *Monitor* that observes the network traffic in the testing set and raises an alarm if its behavior does not match the cycles discovered by the Learner in the training set.

Fig. 2 shows the Monitor algorithm as a decision graph. Each cycle identified previously by the Learner is monitored by an independent *cycle monitor*. Hence, the first step of the Monitor algorithm is to identify to which cycle an incoming packet (represented by a token) belongs. If the token is an unknown request (i.e., not belonging to any cycle), a *New Request* alarm is raised. If the token is a response without a previously received matching request (identified by the pair identifier), an *Unmatched Response* alarm is raised.

Table 3. Number of flows per type

Dataset	Periodic	Periodic w/ new requests	Non-periodic
Modbus	19	4	1
MMS 1	9	0	14
MMS 2	15	0	5

A cycle monitor can be in two states: idle or not idle. If it is idle, it means that all requests belonging to the cycle have been observed in the current iteration and that the currently processed request is considered as the potential beginning of a new iteration. The *duration* of the current iteration is defined as the time between the first request of the current iteration and the arrival of the currently processed request. If the duration is greater than the allowed maximum dur_{max} the *Long Iteration* alarm is raised. If the duration is less than the allowed minimum dur_{min} the *Repeated Request* alarm is raised because the request is regarded as a repetition of an already observed request of the current iteration. Finally, if the duration is between dur_{min} and dur_{max}, a new iteration starts. At this point, *Unmatched Request* alarms are raised if there are requests for which no corresponding responses were observed.

If the cycle monitor is not idle, the current cycle has not yet completed. The monitor tests whether the current iteration's duration (as defined above) is exceeding the allowed maximum duration dur_{max}. If yes, an *Incomplete Iteration* alarm is raised because the iteration has failed to complete in time. If not, it is tested whether the request has been already seen in the current iteration and, if so, a *Repeated Request* alarm is raised.

Finally, the request is added to the current iteration, and the state of the monitor changes accordingly. Initially, a cycle monitor starts as not idle.

We discussed in Section 3.1 that small timing variations might occur. In order to make the Monitor more robust, the cycle monitors use more relaxed thresholds dur_{min} and dur_{max} than the ones determined by the Learner:

$$\begin{aligned} dur_{min} &= dur_{min,learned} - max(min_{thr}, 2dur_{std}), \\ dur_{max} &= dur_{max,learned} + max(max_{thr}, 2dur_{std}). \end{aligned} \tag{1}$$

The values of min_{thr} and max_{thr} will be discussed in Section 4.4.

4.3. Validating the communication model

The objective of our first test is to verify the basic assumption that ICS traffic consists of a series of periodic requests and responses that can be represented by our cycle model. In the following, we make a distinction between three types of flows: flows that do not show any periodic behavior in the training set, flows that are periodic in the training set but present new requests in the testing set, and flows that are periodic in training and testing sets. To perform this task, we test whether the Learner can find two identical cycle candidates cycles ($N = 2$). Additionally, we look for *New Requests* alerts raised by the Monitor. Note that in this test we are not interested in timing-related issues reported by the Monitor, such as *Long Iteration*.

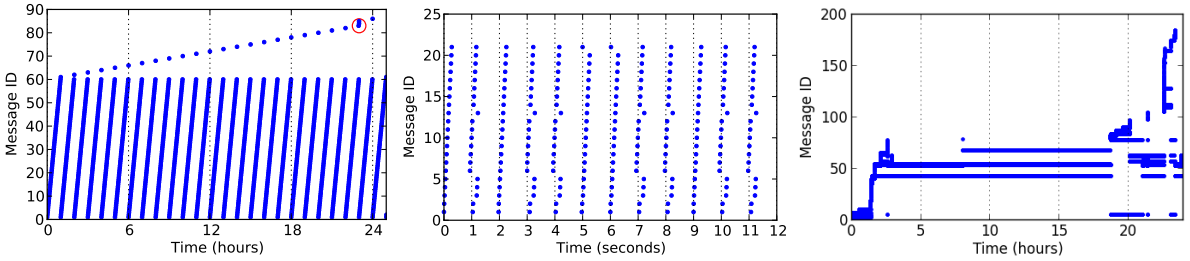
Table 3 shows the results of the test. Most flows observed in our datasets fit the assumption that the traffic consists of a series of periodic requests and responses. In the Modbus dataset, a single flow is reported as non-periodic. In the combined MMS datasets, 24 out of 43 flows are periodic. The high number of non-periodic flows could be expected given the more advanced controls supported by MMS, in comparison to Modbus. This difference reflects the scenarios for which these protocols were designed to be used. Modbus is a typical Supervisory Control and Data Acquisition (SCADA) protocol, supporting only simple supervisory commands, while MMS is a typical Distributed Control System (DCS) protocol supporting more advanced commands, which enable a more tight integration with closed control loops used in the process. Nonetheless, over half of the MMS flows consist only of a series of periodic requests, showing that our algorithm can be used to monitor a considerable number of flows in DCS environments.

Among the periodic flows in the Modbus dataset, four contain *New Requests*: two are likely to be caused by manual activity, and two are clock synchronization flows. The flows with suspected manual activity are mostly periodic, except for short intervals when up to three new (read) requests are observed. We note that false alarms

generated by human activity could be avoided by correlating information from other sources, such as logs from ICS servers (if available) or resource management systems.

The clock synchronization flows contain write messages for values from 0 to 59 issued hourly, representing each minute, and from 0 to 23 issued daily, representing each hour. Such behavior obviously cannot be learned using the original 30 min training set and requires at least 25 h of training data (see Fig. 3a) [4] with $N = 1$. However, this would not eliminate all *New Request* alarms: In addition to the hourly and daily write messages, we identify a weekly pattern of requests writing values from 0 to 7 (the first one marked with a red circle in Fig. 3a). Other requests repeat after 9 days (not shown), suggesting an even longer cycle, but our dataset is not sufficiently large to confirm this behavior.

Three Modbus flows show a periodic pattern where the order in which requests are sent is not fixed. An example is shown in Fig. 3b. This lack of order could have been caused by a multi-threaded application, as anticipated in Section 3.1. Such periodic patterns cannot be correctly modeled by the approach proposed in [15], as it assumes that the order at which requests are issued is fixed.



(a) Clock synchronization flow (b) Flow with unordered requests (c) Flow reading unconstrained addresses

Figure 3. An extract of three flows observed in our datasets

A typical non-periodic flow in the MMS datasets is exemplified in Figure 3c. A remarkable aspect of these flows is the presence of long periods of "well-behaved periodicity", for instance between hours 10 and 15. Applying the Learner (and later the Monitor) to these periods reveals perfectly periodic patterns. We observe that the learned requests are *confirmed requests* issuing read commands to an *unconstrained address*. The new requests are also read commands, but to a slightly modified address. We assume that some of the new requests simply replace the old without breaking the periodic pattern. Unfortunately, the semantics of this address is not defined by the standard, i.e., it is vendor-dependent. Verifying this assumption is left as future work.

Finally, Table 4 provides an overview of the different types of periodic behavior observed in our datasets by further classifying each periodic flow in our dataset according to three characteristics. First, flows are classified as containing single or multiple cycles. Second, flows are single or multi request per cycle. For a flow to be considered multi request, it is sufficient that at least one of its cycles contains more than one request. Third, single or multi connection per flow. Single connection flows represent on long-lived connection with periodic requests, while multi connection flows represent periodic short-lived connections.

The majority of periodic flows consist of a single cycle with a single periodic request over a single long lived connection. However, all other combinations are also present in the datasets. This diversity in periodic behavior clearly demonstrates that our requirements *Multiple cycles* and *Periodicity at different aggregation levels* discussed in Section 3.1 are needed.

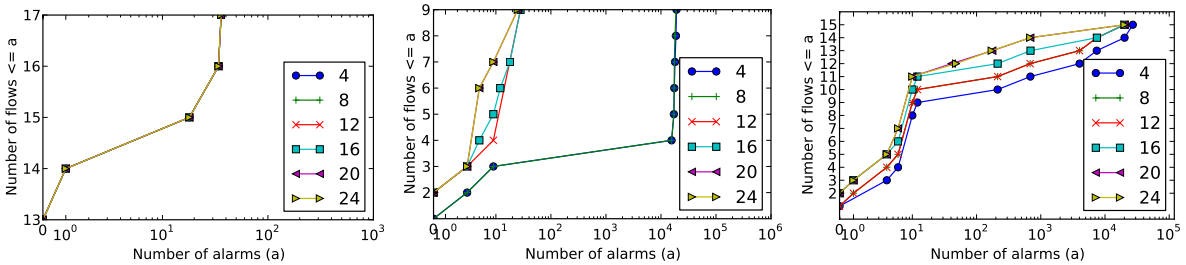
4.4. The impact of N

In this section we study the impact of the number of required candidates N on the learned traffic models. We expect that by using more candidates when learning patterns we will be able to estimate better values for the thresholds dur_{min} and dur_{max}, and thus generate less false alarms due normal variations in the duration of a cycle.

[4] Although the pattern has a 24 h period, the Learner algorithm requires one additional hour to detect a repeated request, which delimits the end of a candidate.

Table 4. Number of flows per type

Cycle	Request	Connection	Modbus	MMS 1	MMS 2
Single	Single	Single	14	2	4
Single	Multi	Single	3	2	0
Single	Multi	Multi	0	5	2
Multi	Multi	Single	2	0	9



(a) Modbus (b) MMS 1 (c) MMS 2

Figure 4. The effect of N in the MMS datasets

In our analysis, we plot the number of alarms a (x-axis) generated by the Monitor vs. the number of flows that generate a or less alarms (y-axis) for a given value of N. For ease of visualization, we show only results for $N = 4, 8, \ldots, 24$. We consider only alarms that are time-based, i.e., a is the sum of the following alarms: *Long Iteration*, *Incomplete Iteration*, and *Repeated Request*. Other alarm types are not relevant with respect to N.

In the *Modbus* dataset the number of alarms is independent of N, that is, all characteristics of the cycles can be reliably learned with $N = 2$ (e.g., Fig. 4a). Only three flows present alarms, of which two also present *New Request* alarms discussed in the previous section. The time related alarms are caused by an "extra" iteration that does not fit the periodic behavior. *Repeated Request* alarms are generated for every request sent in this extra iteration. In addition, these requests cause the Monitor to become out of synchronization with the periodic pattern, which, in turn, causes the Monitor to generate a few (false) alarms until it re-synchronizes to the periodic pattern. We speculate that this extra iteration is triggered by a manual access. In a real-world deployment using *PeriodAnalyser* to detect anomalies in the ICS traffic (as discussed in Section 5.1), such alarms could be aggregated into fewer alarms.

The number of candidates N has a large impact on the number alarms in the *MMS 1* dataset, as it can be observed in Fig. 4b. For $N = 4$ and $N = 8$ (the lines overlap), five flows present more than 10^4 alarms. These alarms seem to be caused by high network delays, therefore it is necessary to use more than 8 candidates to obtain good values of the cycle duration thresholds. Most delay variability is captured when using $N = 12$, but the number of alarms reduces until $N = 20$. At this point, most flows present 10 or less alarms. Two of the flows still present more than 20 alarms with $N = 20$. No improvements are observed using $N > 20$ (the lines for $N = 20$ and $N = 24$ overlap).

Fig. 4c shows the results for *MMS 2*. Increasing N continuously reduces the number of alarms, until $N = 20$ and no improvements are observed using $N > 20$ (again, the lines for $N = 20$ and $N = 24$ overlap). Even with $N = 20$, four flows present a very high number of alarms, up to 10^4.

In Table 5, we show the total number of alarms in three different scenarios: $N = 4$, $N = 20$ and manually adjusting the min_{thr} and max_{thr} to reduce the number of alarms. As discussed above, the alarms observed in the Modbus dataset are caused by real anomalies, i.e., deviations from periodical behavior, and cannot be eliminated by adjusting the parameters. In *MMS 1*, however, it is possible to reduce the number of alarms to 4 by manually adjust min_{thr} and max_{thr} to 1 s. Considering that the period of the learned cycle in these flows is around 21 s, the choice seems reasonable.

The alarms in *MMS 2* are caused by very large variations in the iteration duration; the average duration is approximately 1 s, however, durations of 0.5 s are also common in these flows. By setting min_{thr} to 0.7 s, we still observe 309

Table 5. Number of alarms in different scenarios.

Scenario	Modbus	MMS 1	MMS 2
$N = 4$	89	108599	59704
$N = 20$	89	76	21126
Manual	-	4	309

alarms. Finally, we note that, together these flows contain hundreds of TCP retransmissions, and the largest number of *Unmatched Request* and *Unmatched Response* in our datasets. This suggests that the high number of alarms is caused by a bad link quality.

5. Discussion

In this section we discuss practical applications of *PeriodAnalyser*. In Section 5.1, we discuss how it can be used as an intrusion detection system (IDS). We present how the alarms generated by the Monitor can be used to detect real-world attacks. In Section 5.2, we describe how *PeriodAnalyser* can be used to generate synthetic traces, which in turn can be for testing intrusion detection systems and for evaluating performance of ICS devices, among other purposes.

5.1. PeriodAnalyser as an IDS

As already indicated in Section 4, *PeriodAnalyser* and the Monitor can be used as an anomaly-based IDS. First, the Learner builds the cycle-oriented normality model of the ICS communication from a training dataset. Then, the Monitor detects deviations from the learned behavior in live network traffic. Alarms raised by the Monitor indicate anomalies. It is however important to note that not all variations in the periodic behavior are relevant from a security perspective. For example, the time related alarms might only indicate performance issues, not security threats. For instance, most of the alarms observed for the MMS 2 dataset are caused by small variations in the duration of an iteration, which are likely caused by normal network delays. Optionally, time based alarms could be disabled in such cases.

The datasets used in Section 4 are attack-free and, to our knowledge, no empirical ICS traffic dataset containing malicious traffic has been published so far. Hence, we base the following discussion on some of the general attack classes proposed in [19] and, where available, we refer to concrete types of attacks taken from an open access list of real-world attacks signatures used in the Quickdraw Intrusion Detection System[5]. The list includes signatures to protect Modbus TCP[6] and DNP3[7], two well-know standards for SCADA communication.

Information Gathering attacks may precede other attacks and are an attempt by the attacker to gather as much knowledge as possible of the target system. A typical way to acquire this information is through *scans*, like the *Modbus TCP - Function Code Scan*. In general, attacks of this type require a large number of parameters (addresses, ports, function codes, etc.) to be tested. Therefore, they are likely to make use of requests not observed during the learning phase, triggering *New Request* alarms. Even if it is possible to craft an attack using the learned periodic requests, the scan would still be identified as *Repeated Requests*.

Denial of Service attacks prevent a legitimate user to access a service or reduce its performance. For example, the *DNP3 - Unsolicited Response Storm* attempts to overload a DNP3 server by sending a number of unsolicited response packets, normally used to report alarms. Again, if the attack uses new requests, they will be immediately identified. The attacker could attempt to repeat the messages from a cycle rapidly, but that would also generate a large number of *Repeated Requests*. While a small number of repeated requests do not pose a security threat, a large number indicates a possible attempt to overload the receiver by rapidly repeating requests from a cycle. Note that such attacks are not detected by the approach proposed in [15] (see Section 2).

[5] http://www.digitalbond.com/tools/quickdraw/
[6] http://www.modbus.org/
[7] http://www.dnp.org/

Network attacks manipulate the network protocols. For instance, the *Modbus TCP - Clear Counters and Diagnostic Registers* attack uses a single packet with a specific code function to clear counters and diagnostics registers in a server, in an attempt to avoid detection. The *Modbus TCP - Slave Device Busy Exception Code Delay* attack consists of answering every request with the "device is busy" message preventing a response timeout. Such attacks rely on the use of uncommon requests, which would readily be identified as *New Requests*. In general, we observe that *New Request* alarms are a powerful protection against *data injection attacks*.

Buffer Overflow attacks try to gain control over a process or crash it by overflowing its buffer. For example, the *Modbus TCP - Illegal Packet Size, Possible DoS* attack sends a single packet with an illegal packet size, exploiting a bug in the implementation of the protocol stack. This class of attacks relies on forging invalid packets, which are by definition not commonly used, thus also generating *New Request* alarms.

The alarms created by the Monitor can also be used to detect anomalies that are not or only indirectly related to a malicious activity. *Long Iteration* alarms indicate the sender has become inactive. However, SCADA applications are designed to deal with delays [20], therefore accepting delays larger than those learned from the data might be acceptable, or, even desirable. Finally, while a low number of *Incomplete Iteration* and *Unmatched Request/Response* alarms is not a serious threat, likely a performance problem in the network or probe causing packet loss, a high number indicates that the receiver has become inactive. We note, however, that the threshold at which these alarms become a security issue is dependent on the environment. For instance, water processes are considerably slower than electric processes, so the former might also tolerate larger delays.

In summary, a powerful line of defense created by our approach is the protection against data injection: only requests commonly used during normal behavior might be used to perform attacks, otherwise *New Request* alarms are issued. Furthermore, attacks that can be created with such requests still need to be sent in a way that does not considerably differs from normal periodic behavior, otherwise they will cause *Repeated Request* alarms.

5.2. Synthetic trace generation

In the previous section, we discussed how the models learned by *PeriodAnalyser* can be used to identify anomalies in observed sequences of messages for the purpose of intrusion detection. Conversely, the models can also be used to *output* sequences of message. The goal would be to generate synthetic message traces that are realistic, i.e., that have properties close to what can be observed in a real system.

As any other models, the models obtained by *PeriodAnalyser* are abstractions of the reality and, hence, represent the behavior of a real system only to a certain degree. The tokenizer module abstracts the protocol-dependent information by extracting specific fields from requests and responses to generate protocol-independent tokens (see Table 1). While enough information is kept to uniquely identify requests, and match requests to responses, the contents of the responses are mostly ignored. In other words, the model can be used to captures how often the water level of a tank is checked, but not the actual water level.

In the *PeriodAnalyser* models, a learned cycle carries information on the source and destination of the exchanged messages, the number and types of requests that are allowed in the cycle, and the duration's minimum, maximum and standard deviation. Consequently, a learned model enables us to generate synthetic traces that reflect the randomness of the processes observed in the real system: we can permute the order of the requests and their timing according to the learned cycle information.

Such synthetic traces are useful for various purposes. In the context of security and IDS, a big challenge for researchers is the lack of public labeled traces since operators of ICS are not willing to publish measurements from their systems due to their security-sensitive nature. Synthetic traces generated by our models can be combined with attack traffic in order to obtain a ground truth for testing and evaluating IDS. The fact that we do not model the contents of the response messages is not necessarily a drawback. Many IDS, for example all flow-based IDS [21] but also, to some extent, the one discussed in Section 5.1, do not rely on deep packet inspection.

A second application area are load generators as used, e.g., for the performance evaluation of ICS components in a simulator. For this kind of application, empirical data traces are often not sufficient since they can only reflect the behavior of one specific existing system. In contrast, our model-generated synthetic traces are more scalable and flexible: For example, we can "clone" the cycle model of a PLC with different source and destination addresses, in this way simulating a larger ICS network. Furthermore, we can vary the duration distributions, for example to simulate the behavior of an ICS built on top of a high-latency network.

6. Conclusions and Future Work

In this work, we propose *PeriodAnalyser*, an automated approach to learn traffic models from ICS network measurements. Our novel approach addresses the limitations of existing approaches by directly exploiting periodicity in ICS traffic. We avoid the "semantic gap" of spectral analysis by identifying which messages generate the periodic behavior. Also, we directly capture timing information, making it possible to identify and monitor the frequency at which messages are exchanged, which allows our approach to automatically learn multiple periods and therefore adapt to different ICS environments.

We evaluate our approach using three real-world traffic traces, covering two protocols: Modbus and MMS. Our results show that most flows of SCADA protocols such as Modbus can be accurately modeled by our approach. Although more variation is encountered in DCS protocols, such as MMS, still a large number of flows consist only of periodic requests, which can also be accurately modeled by *PeriodAnalyser*.

We consider our approach and the resulting models as useful for various application areas. For intrusion detection systems, the models deliver two layers of protection. First, by learning all polling requests issued to field devices, we create a whitelist that prevents data injection attempts by third parties. Second, by learning the rate at which these requests are sent, we can prevent denial of service attacks that attempt to overload a field device with (whitelisted) requests. In addition, the learned rate can also be used to detect whether a specific request (or a set of requests) is not sent after a long period of time, for instance, in case the process responsible for issuing the request(s) becomes inactive. Our models also carry enough information to generate meaningful synthetic traffic traces. Such traces can be used as ground-truth to test intrusion detection systems or as scalable load generators for the performance evaluation of ICS designs.

Further research efforts will aim improving the Learner and Monitor, by testing if certain assumptions can be made about the periodic request cycles, for instance by checking if requests are sent in a fixed order. Also, the approach should be extended to deal with other ICS protocols, such as DPN3[8] and IEC 60870-5-104.

7. Acknowledgements

This work has been partially supported by the European Commission through projects FP7-SEC-607093-PREEMPTIVE and FLAMINGO, a Network of Excellence project (318488), both funded by the 7th Framework Program.

References

[1] R. R. R. Barbosa, R. Sadre, A. Pras, A first look into SCADA network traffic, in: 2012 IEEE Network Operations and Management Symposium, Vol. 17, Springer, IEEE, 2012, pp. 518–521. doi:10.1109/NOMS.2012.6211945.

[2] S. Floyd, V. Paxson, Difficulties in simulating the Internet, IEEE/ACM Transactions on Networking 9 (4) (2001) 392–403. doi:10.1109/90.944338.

[3] R. R. R. Barbosa, R. Sadre, A. Pras, Difficulties in Modeling SCADA Traffic: A Comparative Analysis, Passive and Active Measurement: 13th International Conference, Pam 2012, Vienna, Austria, March 12-14, 2012, Proceedings 7192 (2012) 126. doi:10.1007/978-3-642-28537-0_13.

[4] G. N. Ericsson, Cyber Security and Power System Communication - Essential Parts of a Smart Grid Infrastructure, IEEE Transactions on Power Delivery 25 (3) (2010) 1501–1507.

[5] M. Cheminod, L. Durante, A. Valenzano, Review of Security Issues in Industrial Networks, IEEE Transactions on Industrial Informatics 9 (1) (2013) 277–293. doi:10.1109/TII.2012.2198666.

[6] R. Sommer, V. Paxson, Outside the Closed World: On Using Machine Learning for Network Intrusion Detection, in: 2010 IEEE Symposium on Security and Privacy, IEEE, 2010, pp. 305–316. doi:10.1109/SP.2010.25.

[7] D. E. Knuth, The art of computer programming, vol. 2: Seminumerical algorithms, revised edition (1969).

[8] S. Laxman, P. S. Sastry, A survey of temporal data mining, Sadhana 31 (2) (2006) 173–198. doi:10.1007/BF02719780.

[9] J. Han, Y. Yin, Efficient mining of partial periodic patterns in time series database, in: Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337), IEEE, 1999, pp. 106–115. doi:10.1109/ICDE.1999.754913.

[10] O. Argon, Y. Shavitt, U. Weinsberg, Inferring the Periodicity in Large-Scale Internet Measurements, in: INFOCOM, 2013 Proceedings IEEE, IEEE, 2013, pp. 1–9.

[11] A. Broido, E. Nemeth, Spectroscopy of private DNS update sources, in: Proceedings the Third IEEE Workshop on Internet Applications. WIAPP 2003, IEEE Comput. Soc, 2003, pp. 19–29. doi:10.1109/WIAPP.2003.1210282.

[8] IEEE 1815-2012 - Electric Power Systems Communications-Distributed Network Protocol.

[12] G. Gu, J. Zhang, W. Lee, BotSniffer: Detecting botnet command and control channels in network traffic, in: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08), San Diego, California, USA, 2008, pp. 1–18.
[13] J. van Splunder, Periodicity detection in network traffic (2015).
URL https://www.math.leidenuniv.nl/scripties/MasterVanSplunder.pdf
[14] R. R. R. Barbosa, R. Sadre, A. Pras, Towards periodicity based anomaly detection in SCADA networks, in: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012), IEEE, 2012, pp. 1–4. doi:10.1109/ETFA.2012.6489745.
[15] N. Goldenberg, A. Wool, Accurate modeling of Modbus/TCP for intrusion detection in SCADA systems, International Journal of Critical Infrastructure Protection (2013) 1–20doi:10.1016/j.ijcip.2013.05.001.
[16] M. Caselli, E. Zambon, F. Kargl, Sequence-aware intrusion detection in industrial control systems, in: Proceedings of the 1st ACM Workshop on Cyber-Physical System Security, CPSS '15, ACM, New York, NY, USA, 2015, pp. 13–24. doi:10.1145/2732198.2732200.
URL http://doi.acm.org/10.1145/2732198.2732200
[17] H. Hadeli, R. Schierholz, Leveraging determinism in industrial control systems for advanced anomaly detection and reliable security configuration, in: Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on, 2009, pp. 1–8. doi:http://dx.doi.org/10.1109/ETFA.2009.5347134.
[18] J. Zhang, A. Moore, Traffic Trace Artifacts due to Monitoring Via Port Mirroring, in: 2007 Workshop on End-to-End Monitoring Techniques and Services, IEEE, 2007, pp. 1–8. doi:10.1109/E2EMON.2007.375317.
[19] S. Hansman, R. Hunt, A taxonomy of network and computer attacks, Computers & Security 24 (1) (2005) 31–43. doi:10.1016/j.cose.2004.06.011.
[20] D. Bailey, E. Wright, Practical SCADA for industry, Newnes, 2003.
[21] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, B. Stiller, An Overview of IP Flow-Based Intrusion Detection, IEEE Communications Surveys & Tutorials 12 (3) (2010) 343–356. doi:10.1109/SURV.2010.032210.00054.