INTERNATIONAL COMPUTER SCIENCE INSTITUTE

Through the Eye of the PLC: Towards Semantic Security Monitoring for Industrial Control Systems

Dina Hadziosmanovic[*], Robin Sommer[ǂ§], Emmanuele Zambon[*], and Pieter Hartel[*]

TR-13-003

August 2013

Abstract

Attacks on industrial control systems remain rare overall, yet they may carefully target their victims. A particularly challenging threat consists of adversaries aiming to change a plant's ***process flow***. A prominent example of such a threat is Stuxnet, which manipulated the speed of centrifuges to operate outside of their permitted range. Existing intrusion detection approaches fail to address this type of threat. In this paper we propose a novel network monitoring approach that takes process semantics into account by (1) extracting the value of process variables from network traffic, (2) characterizing types of variables based on the behavior of time series, and (3) modeling and monitoring the regularity of variable values over time. We implement a prototype system and evaluate it with real-world network traffic from two operational water treatment plants. Our approach is a first step towards devising intrusion detection systems that can detect semantic attacks targeting to tamper with a plant's physical processes.

* University of Twente, Drienerlolaan 5, 7522 NB Enschede, Netherlands
ǂ International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, California, 94704
§ Lawrence Berkeley National Lab, 1 Cyclotron Rd, Berkeley, California, 94720

This work was partially supported by funding provided to ICSI through National Science Foundation grants CNS-1032889 and CNS-1314973. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the National Science Foundation.

1. INTRODUCTION

Industrial control systems (ICS) monitor and control physical processes, often inside critical infrastructures like power plants and power grids; water, oil and gas distribution systems; and also production systems for food, cars, ships and other products. As these environments differ significantly from traditional IT systems, they also face quite unique security challenges.

Off-the-shelf intrusion detection systems prove a particularly ill fit. Classic signature matching requires precise patterns of anticipated intrusions—an unrealistic assumption in a setting where attacks remain rare overall, yet may carefully target their victims—and existing behavioral approaches fail to incorporate the domain-specific context of operation in these specialized environments. While a few network intrusion detection systems now include ICS-specific protocol support, their capabilities remain limited to finding network-level attacks, such as protocol violations and buffer overflow exploits. They fail, however, to address the fundamentally different threat of adversaries aiming to change a plant's *process flow*. A prominent example is Stuxnet [12], which manipulated the speed centrifuges to operate outside of their operational range eventually causing hardware damage.

Our work targets finding such *semantic* attacks: malicious actions that drive an ICS into an unsafe state without exhibiting any obvious network-level red flag. At a high-level our approach derives behavioral models of a plant's state from past activity, which then facilitates monitoring of future changes for unexpected deviations. Our starting hypothesis is that ICSs—with their narrow focus, a relatively small and homogeneous set of actors, and mostly automated activity—display an overall regularity that we can exploit for finding a stable baseline separating common activity from attacks. However, since our approach reconstructs process data, the generation of behavioral models becomes more complex by the inevitable data noise in real environments and the existence of semantically different types of process variables. As such, a significant part of our contribution indeed concerns *validating* the regularity assumption.

We focus on operating at the network-layer where we can passively collect a comprehensive picture of a plant's activity without interfering with its operation. More specifically, we monitor the network traffic of programmable logic controllers (PLCs), extracting updates of process variables from their communication with other devices. As PLCs represent the interface between a plant's operators and the field devices (e.g., a pump), any state changes—including malicious commands—go through them to take effect. Their network traffic hence provides an ideal vantage point for monitoring.

We present a prototype implementation of our approach that focuses on *Modbus*, a common legacy ICS protocol. In Modbus, each PLC defines a *memory map* representing an internal table of process variables (typically a few thousand). Modbus' data model proves particularly challenging due to its flat structure and the frequent programmer practice of misusing its simple structure (e.g., arbitrary addressing of variables), which makes it harder to characterize process variables accurately.

In the training phase, our system extracts memory map operations from raw network packets and constructs a corresponding time series for each process variable. We then deploy different techniques to derive variable-specific prediction models as the basis for the detection phase. When future memory operations deviate from a model's forecast, the system flags them for manual inspection as potentially undesirable activity. We evaluate our prototype with a particular focus on the accuracy of the behavioral models, aiming to understand where they work well, yet also cases that fail to yield a stable baseline. By using our testbed setup, we demonstrate how proposed techniques perform in the detection of semantic attacks. To evaluate our approach in a real world environment, we use packet traces from two operational water treatment plants recorded over two-week periods. We find that in these environments we can indeed retrieve and capture models of 98% of all used process parameters. The vast majority of these parameters refers to configuration changes that indeed stay constant over time. For the remaining parameters that performed with limited capabilities we perform an in-depth analysis to identify, isolate and address challenges that represent root causes for poor performance.

Overall, our work represents a step towards monitoring process semantics of ICSs that goes beyond scanning for low-level artifacts that an attack may, or may not, produce. While in our case study we focus on a particular protocol, and on a specific industry sector, we believe that our results generalize to other settings as conceptually ICS equipment tends to operate similarly across environments. While we clearly do not put our ambition to replace existing control mechanisms in ICS automation, we believe that our approach demonstrates the feasibility of supplementing network monitoring approaches with the specific context information required for meaningful interpretation of network communication in the ICS domain. We also note that our approach operates orthogonally to any built-in security and safety mechanisms a plant may have, augmenting them with an independent, process-aware, perspective. To the best of our knowledge, this is the first work that aims at understanding real-life ICS processes at such level of details from the network perspective.

We structure the remainder of this paper as follows. In §2 we provide background on the ICS domain. Section §3 discusses the feasibility of reconstructing process semantics from network traces and presents our threat model. In §4 we present our approach along with a testbed scenario (§4.1) that we use for demonstrating the capabilities of the presented techniques. §5 summarizes implementation details, and in §6 we present our results. Finally, in §7 we discuss our findings, and in §8 we present related work.

2. BACKGROUND

A typical ICS includes a number of common components: human machine interface, supervisory infrastructure, field devices, and communication infrastructure. The *human machine interface (HMI)* provides the interface for operators to monitor and manage the industrial process in the field. The *supervisory infrastructure* consists of a group of both general-purpose and embedded computing devices that facilitate communication between operators and field devices (e.g., translating operator instructions to low-level field commands, ICS servers performing automated process field requests, user management and authentication). *PLCs* are embedded devices that run custom code to control specific *field devices* (e.g., regulate the speed of a pump); depending on the size of the setup, multiple PLCs may work jointly to automate the process under control, with each PLC con-

trolling multiple field devices. Finally, the *communication infrastructure* provides the substrate to connect the other components.

Communication. Conceptually, we commonly find two semantic groups of network communications between ICS components: *(i)* process awareness, and *(ii)* process control. *Awareness* propages status information about the controlled process across devices. In particular, the ICS supervisory infrastructure requests regular updates from the PLCs to the HMI to report the current plant status to its operators. In addition to escalating critical updates for timely reaction, awareness also collects trending data for long-term process analysis. PLCs also propagate awareness information across themselves to ensure that each device learns sufficient information about critical variables before entering the next process stage (e.g., PLC 1 might require information about the state of a field device connected to PLC 2 before starting a subsequent process stage).

Process Control is generally exercised in one of two ways: *(i)* by PLCs according to their embedded logic; and *(ii)* by operator commands that override such internal logic. Note that in either case it is the PLC that carries out the action, and hence will reflect the process change as updates to its internal state.

Process Variables. Inside the PLC, two components determine the process control: *(i)* the process control code, and *(ii)* the transient state in the form of process variables. The code consists of logic that regulates connected field devices, and drives interaction with the external infrastructure. For example, the code defines the procedure for filling in a tank, along with necessary preconditions that need to be satisfied (e.g., the water level and pressure). PLCs are typically programmed in derivatives of languages such as Pascal and Basic.

The PLC's process variables characterize the current operation state. Examples of typical variables include the setpoint for a physical process, the current value of a valve sensor, and the current position in a cycle of program steps. Process variables serve as input to the PLC code. For example, a variable value representing a high pressure level might trigger the start of a draining stage. Likewise, the PLC carries out operator commands by writing into corresponding variables. For example, a command to open a valve would update a variable that the program code is regularly checking; once it notices the update, it outputs the corresponding analog signal to the physical device.

Within the device, process variables map directly to PLC memory cells. At the network-level, ICS protocols define corresponding *network representations* to refer to variables as part of commands, e.g., to specify the target variable for a read operation. In our work we focus on analyzing the Modbus protocol. Modbus represents process variables in the form of a PLC-specific *memory map*, consisting of 16-bit *registers* and 1-bit *coils*. Some vendors also deploy variations of the default specification, such as combining 2 or 4 registers to hold 32-bit or 64-bit values, respectively. The layout of a Modbus memory map remains specific for each device instance, and is generally determined by a combination of device vendor, programmer, and plant policies (see, e.g., [25] for generic vendor version, which typically act as a starting point).

Further common ICS protocols include Profinet, EtherNET/IP, MMS, DNP3, and the IEC 6X series, which all use different network representations. For example, Profinet defines *slots*, *subslots*, and *channels*; while MMS uses *objects* to address process variables [18].

3. RECONSTRUCTING PROCESS SEMANTICS

At the core of our work lies the assumption that one can infer process semantics from ICS network traffic. PLCs hold the relevant status information that we strive to monitor, yet due to their closed and embedded nature it remains challenging to tap their information directly. However, PLCs exchange comprehensive status information with the ICS server (that updates HMI) on a regular basis, and in turn the HMI issues control commands to the PLCs to initiate process changes. We find both activities reflected at the network level in the form of requests and replies that report and manipulate PLC process variables, encoded in their corresponding network representation. Hence, a network monitor following this communication can derive an understanding of the controlled process that, in principle, is a superset of the information available in the HMI's perspective. For example, except exchanging values of relevant process parameters from the field, HMI and PLCs exchange a set of internal variables that are critical for process infrastructure but are not explicit part of the field process (e.g., program counters, timers, process stages). By contrast to the process field parameters, these variables are not directly monitored by operators. Crucially, malicious commands have to traverse the network. This implies that typically either their cause or their consequences will likewise be reflected as changes to process variables. As a trace of the attack cause, we will see the attack stage when a command modifies variables that control the process. As a trace of the attack consequence, variables tracking the current process state will begin reflecting the malicious update. There are a number of practical challenges to overcome for reliably reconstructing process semantics from the network layer, including the need for gaining access to tapping points that provide broad coverage, technical ambiguities with interpreting protocols, and semantic context required for interpreting observations. However, a number of ICS-specific properties come to our advantage here. First, these networks tend to be relatively small with few key devices, and hence they exhibit significantly reduced complexity and variety compared to more general environments. Furthermore, the supervision process tends to repeat frequently, often with update cycles in the order of the seconds, which enables the monitor to keep its state up-to-date with little delay. Finally, the interpretation of process variables remains challenging, yet consistent over time as changes to the PLC configuration remain rare. Nevertheless, as in any real-world environment, we expect to find a significant level of irregularities that may mislead a detector and need to accommodate that in our approach.

Threat Model. We consider the scenario that an unauthorized user achieves unfettered access to one or more systems inside an ICS environment. We focus on attacker activity *after* they have already gained access, since the initial break-in typically proceeds similar as in regular IT environments (e.g., stealing credentials, exploiting vulnerable software exposed to the outside world, or accessing backdoors intended for maintenance). We focus our analysis on attacks that, for a successful execution, need to deviate at least one

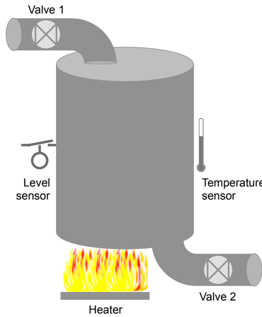

Figure 1: Process setup of the testbed environment

variable controlled by PLCs that is part of the awareness or control flow (i.e., thus is visible in network exchanges). In practice, such attacks represent activity that is legal at the protocol level, yet violates semantics constraints that a process imposes, including both semantically incorrect messages (e.g., conflicting commands) and operations that lead the site into an undesirable state (e.g., a command to start a pump when it must remain shut). Plant operators confirm that an ICS, although designed with safety constraints in mind, cannot address misuses on the logical and functional level of operation [19]. For a more comprehensive discussion on the context of attacks we target, including a survey of known Modbus-based attacks, we refer to Appendix A.

Finally, we assume that it is feasible to obtain a complete copy of all network packets from the observed system, which is technically straight-forward to set up in local area networks without impacting ongoing operations (e.g., via physical wire taps, hubs, or SPAN ports on switches).

4. APPROACH

We now present our approach to detect attacks in ICS that aim at manipulating process variables. It consists of three main phases: *(i) extraction* distills current variable values out of network traffic; *(ii) characterization* divides the observed process variables into three categories that we examine separately; and *(iii) modeling and detection* derives behavioral models for each variable and reports when new observations deviate from what they predict. We discuss these phases individually in §4.2–4.4, after first introducing a small testbed setup in §4.1 that we use for illustration.

4.1 Testbed Scenario

To illustrate our approach, we set up a small testbed environment consisting of one PLC and one HMI workstation. We base the design on a demonstration kit from a known ICS vendor that models a simple water tank setup. The controlled process comprises six plant components (see Figure 1): a tank, a heater, two valves, a level sensor and a temperature sensor. The process consists of three operations which are repeated continuously: tank filling, water heating, and tank draining.

Although in a real-world environment an ICS server would collect data from the PLC and the HMI would use the process data collected by the ICS server, to simplify our architecture we configure the HMI to request process updates directly from the PLC once per second. The HMI collects nine variables: two for measurements (i.e., tank level and water temperature), five for control (i.e., valve 1 and 2 status, heater status, tank level setpoint and water temperature setpoint), and two for reporting (i.e., tank level and water temperature high/low alarms).

Table 1: Testbed PLC memory map

Reg.	Name	Type	Desc
HR0010	V1On	bool	Status of valve 1
HR0011	V2On	bool	Status of valve 2
HR0012	HeaterOn	bool	Status of the heater
HR0020	TankLevelSP	fixpoint	SP tank level (L)
HR0021	TankLevel	fixpoint	Level of the tank (L)
HR0022	TempSP	fixpoint	SP water temp.
HR0023	Temp	fixpoint	Water temp (celsius)
HR0030	TankLevelAl	enum	Alarms tank level
HR0031	TempAl	enum	Alarms water temp.

SP: setpoint

Table 1 shows the relevant parts of the PLC's memory map. The PLC's code implements the following logic:

```
while ( true )
  V1On        := ( TankLevel < TankLevelSP && !V2On);
  HeaterOn    := ( Temp < TempSP && !V1On && !V2On);
  V2On        := ( Temp >= TempSP && TankLevel > 0 && !V1On );
  TankLevelAl := 0;
  if ( TankLevel > hthreshold)    -> TankLevelAl := 1;
  if ( TankLevel > hhthreshold )  -> TankLevelAl := 2;
  if ( TankLevel > hhhthreshold ) -> TankLevelAl := 3;
  TempAl      := 0;
  if ( Temp > hthreshold )        -> TempAl      := 1;
  if ( Temp > hhthreshold )       -> TempAl      := 2;
  if ( Temp > hhhthreshold )      -> TempAl      := 3;
```

For the sake of simplicity we did not include safety constraints in our process logic, except for the alarming system. This also simplifies the illustration of our two attack scenarios: *(i)* changing the level setpoint to overflow the tank; and *(ii)* sending tampered measurements information to PLC to trigger process changes.

4.2 Data extraction

The data extraction phase is a preprocessing step that distills the values of process variables out of the ICS network traffic. It consists of two subparts: *(i)* parsing the application-layer network protocol to extract the relevant commands, including all their parameters; and *(ii)* constructing *shadow memory maps* inside the analysis system that track the current state of all observed process variables, providing us with an external mirror of the PLCs' internal memory.

For this work we focus on parsing Modbus, in which each command comes with a set of parameters as well as a data section. Basic parameters include function/sub-function codes that define the operation, an address reference specifying a memory location to operate on, and a word size giving the number of memory cells affected. The data section includes the actual values transmitted, i.e., the current value for a *read* operation and the intended update value for a *write*. We maintain shadow memory maps by interpreting each command according to its semantics, updating our current understanding of a PLC's variables accordingly.

The following (simplified) commands from our testbed setup (see §4.1) illustrate the extraction step.

```
Time 1: PLC 1, UID: 255, read variable 21, value: 10
Time 2: PLC 1, UID: 255, read variable 21, value: 14
Time 3: PLC 1, UID: 255, read variable 21, value: 18
```

After processing the commands, the shadow memory map will report 18 as the current value for variable 21. Looking more closely at variable 21, we know from the testbed configuration that it corresponds to the *tank level* and, hence, will reflect three distinct types of behavior: an increasing trend during filling, a constant value during heating, and a decreasing trend during the draining phase. Indeed, our extraction step confirms this expectation: The following list represents a small excerpt from variable 21's values, as extracted from actual network traffic in the testbed:

```
...490,492,494,496,498,500,500,500,500,500,500,
...496,492,488,484,480,476,472,468,464,460,456, ...
```

We have confirmed that these extracted values indeed match what the PLC stores internally over time.

4.3 Data characterization

Next, we perform a *characterization* phase that separates variables into different categories based on the knowledge obtained during focus groups sessions with plant engineers. In general, PLC process variables fall into four groups: *(i) control*: variables for configuring plant operation (e.g., device setpoints, configuration matrix); *(ii) reporting*: variables for reporting alarms and events to operators through HMI or other PLCs (e.g., pump load is too high); *(iii) measurement*: variables reflecting readings from field devices and sensors (e.g., current tank level, current water flow), *(iv) program state*: variables holding internal PLC state such as program counters, clocks, and timeouts. While the character of variables varies significantly with their groups, we observe three cases that suggest specific models for predicting future behavior: most variables either *(i)* change continuously, and gradually, over time; *(ii)* reflect attribute data that draws from a discrete set of possible values; or *(iii)* never change. The first is typical, e.g., for sensor measurements, program state and reporting tend to use the second, while the third is typical for process settings (e.g., setpoints). Unfortunately there is no definite resource to directly tell what type of data a variable reflects—recall from §2 that memory maps are specific to each PLC instance. Thus, we apply heuristics to categorize process variables according to the behavior we observe. In our testbed we can directly cross-check if the results indeed match the configuration. In the actual environment we examine later in §6, we compare our results with labels extracted from PLC project files.

During discussions with engineers, we learned that reporting variables are encoded in bitmaps which, depending on the number of distinct reporting events, appear as a discrete set of 2^k values. We use this information to build heuristics that allows us to distinguish between attribute, continuous and constant series. For us, a series that consists of only 2^k, where $k = 0..8$ discrete values in the whole training set is considered as an attribute series. A special case of an attribute series with $k = 0$ represents constant series (i.e., the whole dataset consists of only one distinct value). A series that consists of more than 2^k distinct values is considered as a continuous series. In our testbed environment, we set the parameter $k = 3$ (i.e., series with up to 8 values are considered as attribute). We run the characterization on the network traffic of over 2h of operation and classify the 9 variables as 4 constant, 3 attribute and 2 continuous series.

4.4 Data modeling and detection

Once we distinguish between constant, attribute and continuous time series, we can proceed with building behavioral models.

Modeling. To model constant and attribute data, we derive a set of expected values (e.g., enumeration set for attribute data, one observed value for constant series). To model continuous data, we leverage two complementary techniques,(*autoregression modeling* and *control limits*, to capture the behavior of a series and understand operational limits. Autoregressive model represent a common technique to capture the behaviour of correlated series, such as sucessive observations of an industrial process [31]. An autoregressive model of order p states that x_i is the linear function of the previous p values of the series plus a prediction error term [7]:

$$x_i = \phi_0 + \phi_1 x_{i-1} + \phi_2 x_{i-2} + ... + \phi_p x_{i-p} + \epsilon_i$$

where $\phi_1, ..., \phi_p$ are suitably determined coefficients and ϵ_i is a normally distributed error term with zero mean and non zero variance σ^2. There are several techniques for estimating autoreregressive coefficients (e.g., least squares, Yule Walker, Burg). We choose to use Burg's method, as it has proven as a reliable choice in control engineering, a field closely related to ICS processes [16]. To estimate the order of the model, we use the common Akaike information criterion [27]. Using the autoregressive model, we can make one step estimation for future values of the process variable underlying the time series. This way, the model can be used to detect stream deviations. However, as for any regression, a set of small changes can take the stream outside of operational limits without exhibiting regression deviations [31]. To address this, we use a complementary strategy, namely Shewart control limits [31]. This is a common technique used for controlling mean level and preventing the system shift from normal operation. The control limits represent a pair of values $\{L_{min}, L_{max}\}$ that define the upper and lower operation limit of the process variable. Typically, the limits are calculated as values that are three standard deviations from the estimated mean.

Detection. For constant and attribute series we raise an alert if a value in a series reaches outside of the enumeration set. To detect deviation in continuous series, we raise an alert if the value *(i)* reaches outside of the control limits or *(ii)* produces a deviation in the prediction of the autoregressive model. More specifically, for estimating the deviation in the autoregressive model, we compare the residual variance with the prediction error variance. The residual variance describes the deviation of the real stream from the stream predicted by the model during training. The prediction error variance describes the deviation of the real stream from the stream predicted by the model during testing. A prediction error variance that is significantly higher than the residual variance implies that the real stream has significantly deviated from the estimated model, thus we raise an alert. We apply this techniques since it is commonly used for the detection of anomalies during instrument operation in control engineering [16]. To estimate the "significance" of deviation we use hypothesis testing (see §5).

To illustrate the detection capabilities we test our approach on two semantic attacks crafted for the process operating in our testbed. The first attack effectively consists of a command that changes the tank level setpoint (HR0020 in Figure 2). As a result, the tank filling phase (HR0021 in

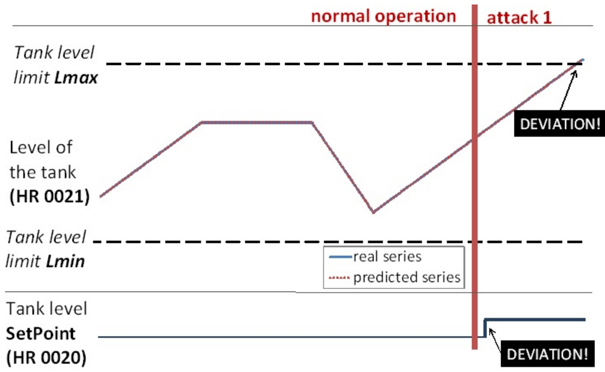

Figure 2: Illustration of the configuration change

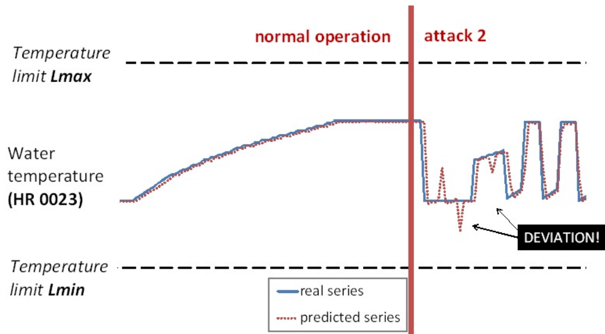

Figure 3: Illustration of measurement tampering

Figure 2) continues until the water level overflows the tank capacity. Results show that this attack is detected as *(i)* a deviation in setpoint variable and *(ii)* a value reaching maximal control limit $Lmax$. The second attack consists of a set of commands that set tampered information about the temperature level measurement (HR0023 in Figure 3). As a result, the tank filling phase is terminated early, the heating process starts and then immediately stops and the draining process starts. As a consequence, both the heater and the boiler may get damaged. In this second scenario no alarm is generated by the PLC and thus presented to operators. Our results show that this attack is detected as a deviation in autoregressive model.

Our results demonstrate known capabilities of both approaches [31]. In particular, an autoregressive model is effective for detecting sudden changes (e.g., detection of the second attack). However, a sufficiently slow deviation (i.e., slower than the model order p), can still take the system beyond specification limits without triggering an alarm (e.g., the first attack was not detected by the autoregressive model). On the other hand, control limits are generally a good strategy for maintaining the process mean level (thus, can detect the process drift in the first attack). However, control limits cannot detect a deviation that is within the defined limits of operation (e.g., cannot detect the second attack).

5. IMPLEMENTATION

We implement a prototype of our approach using a combination of Bro [24] and custom C++ code. Bro performs the initial *data extraction* step. We develop a Modbus analyzer for Bro that extracts the main protocol commands from network traffic and makes them available to scripts written in Bro's custom policy language. We leverage Bro's BinPAC [23] parser generator to automatically generate much of the Modbus-specific code from a corresponding grammar. Our Modbus analyzer is fully integrated into Bro, and is scheduled to be part of the next public release. We also add a custom analysis script to Bro that records each Modbus command into an ASCII-based log file that we then process with external code implementing the subsequent characterization, modeling, and detection phases.

For the *data characterization* we test different values for k in the range [2..8] to distinguish between attribute and continuous series. For our tests with real environments we choose $k = 3$ since for this value our preliminary analysis showed the least number of mismatches for attribute series. For the *data modeling* of continuous series, we build the autoregressive model and derive process control limits. We leverage an open source implementation of the autoregressive model[1]. To derive control limits, we implement Shewart control limits following the description in [31]. For each continuous series we derive and estimate of the behaviour on both the autoregressive and control limit model. Finally, for the *detection* of deviations in continuous series by using the autoregressive model, we use two variance hypothesis tests (commonly known as F-test). For both tests we set $p = 0.05\%$ as significance level.

6. EVALUATION

Our work represents a first step towards accurately modeling ICS processes from a network vantage point. As such, we are primarily interested in understanding properties of the setting that impact security monitoring at the semantic level, and less in specific true/false positives rates. With that perspective, we evaluate our memory map modeling with two overarching objectives: *(i)* understand the degree to which our approach can successfully predict typical process behavior; and *(ii)* gain insight into the underlying activity that improves, or weakens, accuracy. In the following we first present the two real-world environments in §6.1 that we use for the evaluation. We evaluate the characterization and modeling phases independently and describe the corresponding methodology and results in §6.2 and §6.3 respectively. We do not analyze in depth the data extraction here as it constitutes primarily a pre-processing step which proved to work without problems also with the real-world traffic.

6.1 Environments and data sets

Our data comes from two real-life water treatment plants that serve a total of about one million people in two urban areas. They are part of a larger system of over 30 sites controlled by one company. The two plants are of comparable size, and they perform semantically similar tasks (e.g., water pumping, purification, ozone treatment). However, their setups still look quite distinct. They deploy different numbers of PLCs (3 vs. 7), and chose different strategies to divide processes among them. The PLC memory maps differ both between the two environments, and also among PLCs of the same site. While both plants use equipment from the same (well-known) vendor, they deploy different software versions.[2]

[1]Available at https://github.com/RhysU/ar.git

[2]According to agreements with the two sites, we cannot

We have access to one 3 day and one 14 day long packet trace from the two plants, respectively, both containing the complete network traffic captured from the mirroring port of the switches that connect the different PLCs and the ICS servers. The traces include 64GB and 101GB of network traffic, respectively, with bandwidths varying between 9 and 360 packets/sec during the recorded periods. We find two ICS protocols and six non strictly ICS protocols in use. The ICS protocols are Modbus, which is used for communication between PLCs and from PLCs with ICS servers and a vendor proprietary protocol which is used for communication between ICS server and HMI. The non ICS protocols are VNC, SSL, FTP, HTTP SMB and DCOM and are used by the servers and workstations in the network. The non ICS traffic is an negligible fraction of the overall network trace. Of the 20 and 28 hosts active in the two traces, 7 and 11 receive or send Modbus messages. While we see the vendor proprietary protocol in use among hosts that are part of the supervisory infrastructure, we observe only Modbus for all communication involving PLCs. We see three types of Modbus messages in the traces: *read multiple registers* (function code 3), *write multiple registers* (16), and *parallel reading and writing on multiple registers* (23). In the following, we focus our discussion on the Modbus traffic. According to the plant operators, there were no security or operational incidents during out measurement periods. To present our test results, we select 5 PLCs taken from both plants, namely: all 3 PLCs from the first plant, and 2 PLCs from the second plant. The second plant operates with 7 PLC in total, of differing complexity: The number of process variables goes as low as 135 for three of them and as high as 3500 in one; the remaining four PLCs operate on approximately 2200 variables. We select two PLCs that representative the most complex and the most simple PLC setup, respectively.

In addition to the traces, the plant operators provided us with *project files* that describe each PLC's memory map layout, exported by PLC programming environments in the form of CSV files holding information on addressing, data type, and process role for each process variable defined by a PLC. In practice, such project files closely resemble the information shown by the example memory map in Table 1.

6.2 Data characterization

Recall that the goal of variable characterization is to apply to each variable (register) the most appropriate technique in the modeling step based on the variable semantics. For example, we want to use a set of values for setpoints and alarms, while we want to use autoregressive model and control limits for measurements and counters. In §4.3 we propose a characterization heuristic that separates registers into constant, attribute and continuous time series based on the value of the variables over time. Another approach to data characterization leverages the semantic information contained in the project files. In fact, project files contain as human readable text the semantics of a specific process variable (e.g., variable X is the "throughput rate of pipe Y"). Under the assumption that project files are available, this approach would in theory achieve the characterization goal without the need to process the network traffic. In this section we aim to evaluate the practical applicability and quality of results of our heuristic-based approach compared to the extraction of the same information from project files. To this end, we translate the semantic information of project files into labels that define what we expect the characterization phase to return. We assign the labels semi-automatically by constructing a table that maps keywords commonly found in the descriptions into the three categories. For example, we consider descriptions including "measurement", "counter", or "usage" to indicate variables holding continuous values. On the other hand, words like "command" or "alarm" suggest attribute data, and "configuration" indicates a process variable of generally constant value. In total, we identify 24 keywords, which allow us to classify all PLC variables defined in the project files. We then run our heuristic-based algorithm on the network traffic exchanged by three PLCs from the first plant during three days and we calculate an average percentage of variables belonging to the specific series across three PLCs. Our results show that the characterization phase classifies 95,5% of all variables as constant series, 1,4% of variables as attribute, and 3,1% as continuous series. Further analysis with different batches shows that these results remain consistent over time for intervals longer of one day. Table 2 shows the comparison with the classification we derive independently from the PLC project files. Our analysis shows that the retrieved information from project files covers only 35% of all observed variables in all three PLCs. As we found out, the main reason for such small coverage are implicit definitions of multiple variables (e.g., PLC programmers use tailored data structures to define a range of variables by only defining the starting variable in project files). We now analyze the results of the comparison. We see an excellent match for constant variables. However, only about half of the continuous variables match, and even less in the attribute category. Closer inspection reveals two main reasons for the discrepancy. First, ambiguities in the project file mislead the keyword-based heuristic. Generally, the descriptions are not standardized but depend on the PLC programmer, and hence keywords sometimes overlap. For example, one PLC has several fields that include the description "ControlForAlarm". Yet, we consider the keyword "control" to indicate a constant variable, and "alarm" to suggest an attribute series. While this example could be addressed easily, similar ambiguities would remain. This difficulty shows that in practice it is not so easy to extract meaningful semantic information from project files, as initially assumed.

Table 2: Comparison of obtained characterizations against the labels from project files

Type of data stream	Matched process variables (in %)		
	PLC1a	*PLC1b*	*PLC1c*
Constant	96.2	95.0	97.0
Attribute	33.3	20.0	40.1
Continuous	44.3	56.7	68.3

The second cause of mismatches is that variables that, according to the PLC configuration, contain attribute or measurement data, in practice exhibits a constant behavior. For example, in PLC1b a variable describing the measurement level of a specific tank always remains constant, and hence the characterization step classifies it as such.

Although both approaches show advantages and pitfalls, our heuristic-based approach is the only one that allows us to characterize all the variables. In general, we would be in

name the equipment vendor.

favor of combining the two approaches, providing our heuristics additional context information. However, for this initial work, we chose not to do so in order to *(i)* understand the step's capabilities on its own, and *(ii)* use the PLC information as a cross-check. Furthermore, as our analysis shows, integration would raise a different challenge due to the inherent ambiguities.

6.3 Data modeling

To evaluate our modeling approach, we examine how well its predictions capture common plant behavior. In a first step, we measure the number of deviations that the models report on network traffic representing typical plant operations. In a second, more interesting step, we then dig deeper into these results and focus on understanding the underlying reasons and situations in which our approach *(i)* indeed models process activity correctly; and *(ii)* fails to capture the plant's behavior, flagging benign deviations as alarms. Our objective here is to gain insight into the capabilities and limitations of our approach, as well more generally into potential and challenges of modeling process activity at the semantic level.

We point out that we do not evaluate the detection rate (i.e., true positives), due to the inherent difficulty of achieving meaningful results in a realistic setting. As actual attacks are rare, we cannot expect our traces to contain any malicious activity (and as far as we know, they do not). However, it also remains unrealistic to inject crafted attacks into the traces; we would be limited to trivial cases like those already demonstrated in §4.4 (which our detector would find for the same reasons as discussed there). On the other hand, we cannot inject more complex attack data sets, like from simulations carried out elsewhere, as any ICS activity has little meaning outside of its original environment (e.g., recall how Stuxnet tailored its steps to its specific target setup; interpreting that activity inside a different setting would make little sense). Hence, we see more value in our semantic analysis of capabilities and limitations than measuring detection rates on unrealistic input.

We now start with measuring the number of deviations that the models report. Generally, we consider our approach to generate an "alert" on a process variable when, at any time during testing a batch of data, an observation deviates from the prediction—i.e., when observing an unexpected value for constants and attributes, or a value outside the autoregressive/control limit models for a continuous time series. We perform 3-fold cross validation using the *rolling forecasting procedure* [17] on a set of 3-day batches of data extracted from the two plant's network traces. The rolling forecasting procedure is a common technique for performing cross validation in time series and implies two key modifications compared to the traditional cross validation procedure: *(i)* the training length is increasing through different folds and *(ii)* the training set never includes data occurring after test set (i.e., the model should not train on the data that is later than test data). The first two batches of data come from the first plant, each representing a randomly chosen continuous range from the first and the second weeks, respectively. The third batch represents the complete trace from the second plant (recall that we only have 3 days of network trace available from that plant). At a technical level, a 3-day batch size gives us a reasonable volume of data suitable for processing repeatedly with our implementation. At an operational level, operators confirm to us that one day matches a typical PLC work cycle.

In Table 3 we summarize the results of the testing across different behavior models. For each pair of category and PLC, we compute the percentage of variables deviating, showing mean and standard deviation over the batches.

In the following, we discuss the three categories separately.

Table 3: Testing model capabilities

	Deviating variables across different types of series (mean %/ st.dev)		
	Constant	*Attribute*	*Continuous*
PLC 1a	0.5/0.29	19.05/0.2	57.49/5.78
PLC 1b	0.31/0.04	19.80/1.4	44.64/5.41
PLC 1c	0.14/0.02	19.55/4.0	37.58/2.98
PLC 2a	0.64/0.0	26.92/0.0	63.63/0.0
PLC 2b	0.0/0.0	0.0/0.0	0.0/0.0

6.3.1 Constant series

Our results show that by far the most variables that we classify as constant indeed stay stable over time. Examing the small number of deviating variables in this category, we observe two main causes for false positives: *(i)* configuration changes, and *(ii)* misclassifications of the variable type. The former typically relates to a previously unobserved status change of specific field device. For example, in PLC1 we find a pump device that is enabled only after about 40 hours of normal operation. In another similar, but more extreme case, we observe a burst of alarms: 60 variables all trigger at the same time even when the training was longer than two days. Upon closer inspection we find them all to belong to a "configuration matrix", a large data structure that defines an operation mode in terms of a set of values controlling multiple devices simultaneously. As it turns out, it is a single packet from the HMI that performs a "multiple register write", triggering the deviation for all of them. We verified that this occurrence represents the only time that the operators change the matrix over the two weeks interval that our traces cover. As such, it is a significant yet rare change that one could either whitelist or decide to keep reporting as a notification.

The second cause for false alarms represents shortcomings of our data classification phase. For example, during one of the folds in PLC2a we find that it misclassifies 9 out of 15 measurement variables representing aggregated flow information as constant due to a lack of activity during the training period. Similarly, in the same fold we find that 7 out of 18 device statuses (thus assumed attribute data) are misclassified as constants. This is because in both cases the values remain constant for more than 20 hours, yet then change during the testing period. Interestingly, we see several such situations that first trigger an alert for a configuration change (e.g., the status of a filter in PLC1b changes for 15mins after it has spent 21 hours in a previous state), followed by a burst of further ones reflecting the change being in effect now (i.e., variables representing activity linked to that filter start to deviate from constant behavior: status, volume, throughput/hour, total throughput). In this case, the two main causes of errors are hence related.

Discussions with the plant operators confirm that daily *cleaning* activities on that PLC might cause such sudden changes for a short amount of time. The misclassification in

this case was avoided in the next fold with a longer training interval, confirming that when training spans the corresponding work cycle, the modeling of constants indeed works as expected.

6.3.2 Attribute data series

Our test show a stable, but reasonably high number of deviating attribute series across all tested folds. By sampling a subset of deviating variables, we find that the main cause for mismatches in this model concerns continuous variables misclassified in the data characterization phase: due to slow process character some of them exhibit only a limited number of distinct values during a training interval, and are thus wrongly labeled as attribute data (e.g., variables describing time information in the form of date and hour). Apart from this scenario, the targeted variables (thus commands and alarms) are captured correctly for training longer than one day. However, we note that a blind spot for our current attribute models are alarm/command *sequences*. With attribute data, sequences carry important semantic information as such variables often encode the current process state within a series of steps (e.g., alarm type X raised to operator, operator acknowledged, alarm cleared, state normal). Since some alarms require operator acknowledgment, a sequence that, e.g., omits that feedback would be suspicious. For such variables, we attempted to apply the continuous models as an alternative, but they only further reduced the accuracy. That however is not surprising: for attribute data ordering matters, yet typically not the actual timing (e.g., an operator may acknowledge an alarm at any time). Hence, we consider a sequence-based analysis of process states as a promising extension of our current attribute model.

When examining the variables that describe attribute data in detail, we discover further structure that our modeling does not currently key on, yet which we consider a promising venue for exploiting in the future. During focus group sessions with plant engineers, we learn that alarms and commands are typically encoded in bitmaps, and we indeed find this reflected in the network traffic. For example, for a variable that the PLC project file refers to as *"various status notifications from PLC3 to server"*, we observe a series of what, at first, appears like an arbitrary set of values: 40960, 36864, 34816. However, when aligned in binary format, the values map to:

```
1010 0000 0000 0000
1001 0000 0000 0000
1000 1000 0000 0000
```

This representation reveals patterns of bits that are constant (e.g., the first bit indicates that PLC1c is active). If we integrated this structure into the characterization step, we would be able to refine the attribute modeling significantly. In other words, some variables require a different granularity than just their numerical value for capturing their semantics.

6.3.3 Continuous data series

We now summarize results from the two models considering continuous time series: control limits and autoregression. We observe that autoregressive model generally alerts more frequently than control limits. In fact, the control limits contribute to only 28% of all deviations in continuous series.

Control Limits model.

Our results show that, apart from the overlap with autoregressive model, control limits report additional 3% series as deviating. Our inspection reveals that these variables represent series that are increasing trends during the whole available trace. For such variables, approaches in statistical process control commonly accept that the series should be modeled according to their regression nature only, and not on the control limits. This means that these variables should be whitelisted in this model. An alternative approach would be to obtain absolute process limits (e.g., from process engineers) and enforce those limits for series control.

Autoregressive model.

Our results show that the autoregressive model has no difficulty in modeling internal process stage and counter variables. Differently from alarms and commands, which occur in relation to human interaction, these variables are connected to the automatic process behavior with highly correlated and regular sequences of values which are straightforward to capture. Of all the deviations, we only find one related to a counter variable (which delayed an increment for 2 seconds): since the behavior of this counter was extremely regular in the training data, the model detector was not tolerant against the delay.

The remaining deviations refer to measurement variables. By inspecting them more closely, we distinguish three groups. A first group of deviations refers to variables that autoregression fails to model well, independently of the training interval. This group accounts for $\sim 80\%$ of the deviations. We believe the autoregression model fails to model the behavior of these variables because we observe the same variables reported consistently across all folds and batches. By sampling deviating variables, we find out that 70% of them have a presumably random behavior with high oscillations. The remaining 30% behave as series that are nearly constant (or slow trends) whose deviation is captured when a sudden peak occurs. To understand the semantics of this behavior, we look into project files. It turns out that, according to the project files, all these variables correspond to *floating point* measurement values of the same set of field devices (e.g., measurement from devices concerning purification in PLC2a). In Modbus, floating point values are represented by a set of registers. The vendor specification for our PLCs states that a single precision floating point is encoded according to the IEEE 754 standard [3] in *two* registers, which represent the actual value as a product of sign, exponent and mantissa. In Figure 4 we show how these three components are projected onto a pair of registers. To understand how this specification relates to our series, we find a pair of registers in project files that are labeled as a higher and lower register of the same floating point value. When reconstructed, the resulting value represents a value with an increment in range of 10^{-4}. Independently, the two variables describing the behavior of the two registers look quite different. In particular, while one variable looks pseudo random (this illustrates the noisy fraction behavior of RegisterB from Figure 4), the second variable looks nearly constant (this illustrates the exponent part of the RegisterA in Figure 4). We acknowledge that, depending on the measurement noise, some registers containing (half of a) floating point variable are not suitable to be modeled raw. Our current tests show that, in the analyzed environments, this refers to approximately %50 of all measurements (since the same variables are consistently alerted over all batches). To address this problem, we would need to reconstruct the floating-point

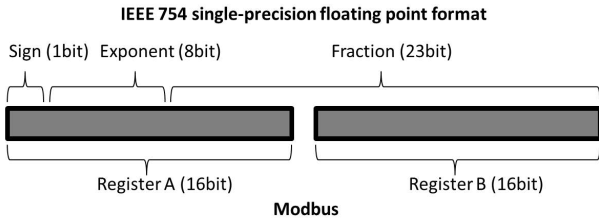

Figure 4: Representation of single precision floating point in Modbus

values as part of the data extraction phase. Unfortunately, it is technically challenging to find a unique approach to identify the two halves of a floating point variable. Vendors use different approaches and even within the same vendors, programmers might follow different conventions. For example, in the analyzed PLC project files we observe the use of at least three different conventions: use consecutive pairs of registers with *(i)* the register with even address as the upper register, *(ii)* the register with the odd address as the upper register, or *(iii)* for a set of variables, put all upper registers first and then all the lower registers.

By observing the peaks in the percentage of deviating variables across different folds, we find the second group of deviations. In more detail, we wanted to understand if the deviating variables refer to multiple field devices (e.g. one variable per field device) or to a few field devices (e.g. multiple variables per field device). We find out that in all analyzed cases, all the deviating variables correspond to a limited set of devices (e.g., a peak of 9 deviating variables in PLC 2a semantically describe different aspects of only one field device, a pump). We also find several situations in which multiple variables are linked together and hence exhibit similar (even identical) behavior. For example, we see 10 ozone filters whose flow is described by the same autoregressive model, and whose deviation occurred at the same time and thus resulted in a peak of deviations in one PLC.

In either case, a more sophisticated model could aggregate variables by incorporating more process context information into the detection approach, for example by extracting information from project files or performing a vertical analysis of variables, scanning for patterns of similar behavior or grouping together variables that refer to the same device.

Finally, the third group of deviations is related to variables that behave differently over time. For example, the value remains nearly constant for 20 hours, then it fluctuates for 15 minutes and then it reverts to a constant value. We believe this group of variables is the same group which was mischaracterized as constant series, as we described in §6.3.1. The data series of these variables is not stationary in a wide sense, and thus it is generally not well suited for autoregressive modeling. To address this, we envision the adoption of multivariate modeling approaches.

7. DISCUSSION

In this section we discuss the evaluation results as well as other aspects that relate to the applicability of our approach, namely the threat model and generalization to the domain and other industrial control protocols.

Putting our findings into perspective, we show that we could reliably monitor 98% of the process control variables used in two real-world plants. 95% of these variables are configuration settings, and according to plant operators changes in configurations indeed represent one of the most direct threats for plant control. In fact, our tests confirmed that we could detect a (although legitimate) configuration change happening at one of the plants during the monitoring period. The remaining 2% of the variables are still challenging to model with the presented approaches. However, when digging into the causes, we could isolate a number of reasons for the deviations, rooted not only in the models themselves but sometimes also in the data characterization and extraction phases. Specifically, we see mismatches between *(i)* training periods and activity cycles; *(ii)* data representation and process semantics; and *(iii)* chaining and cluster effects that cause individual deviations to propagate to a large number of variables.

At a higher level, our findings provide a perspective on ICS environments that may be unintuitive to security researchers. We find a common assumption that ICS activity follows regular patterns that should be straight-forward to model with approaches like the one we deploy in this work. However, we show that when looking at the core of the process control, inevitably, the real world is more complex than one might assume, exhibiting plenty of irregularities, semantic mismatches, and corner-cases that need care to get right. This is a well known challenge in the process control community: operational safety typically requires intensive manual work on understanding and estimating the process behavior before enforcing any controls.

In relation to our threat model, we acknowledge that our approach does not explicitly detect PLC code updates. However, a PLC code update is an unusual event which involves issuing special commands to the PLC (function codes in Modbus). It is therefore trivial to detect such events by extracting the command from application layer messages and whitelisting the ones that are used. We also note that some process manipulation attacks remain outside of what our approach can conceptually find. By gaining control of a PLC, Stuxnet recorded the value of measurement variables during normal operation and replayed the recorded values after triggering the process variation to hide its traces. If the replayed values emulate the normal pattern over time perfectly, it will not be possible to detect the anomaly by any of our models. However, in case that the tampered measurement is not accurate over a long period of time (e.g., because the period of sampling was too short), our approach would still have a chance to detect the attack.

We believe that more extensive tests could be conducted with data coming from other environments, and in particular from other industrial domains. We are confident that our approach is applicable to other environments since we do not use any assumptions that are specific to the water treatment only. We believe that the choice of focusing on the Modbus specification for designing our approach is beneficial for extending its use to other (more recent) industrial control protocols. In fact, the data model of Modbus is generic and only defines two types of process variables (registers and coils). This makes decoding Modbus messages easy, yet renders it hard to extract meaningful semantics (see the problem with floating point values discussed before). Other industrial control protocols (e.g. DNP3, MMS and IEC104) define a much more structured data model, with a complete set of variable types (booleans, integers,

floating points, etc.) and more fine grained variable semantics (e.g. measurements, setpoints, alarms, etc.). With more information extracted from protocol messages the impact of errors in the characterization step would decrease, and in some cases it might not be needed at all.

Summarizing, we believe that our approach lays the ground for detecting critical attacks on industrial control systems that current approaches can fundamentally not find.

8. RELATED WORK

Statistical process control is a well established field which focuses on modeling and monitoring parameters in industrial processes. Researchers use various techniques (e.g., time series analysis, outlier detection, control limits, feedback adjustments) to model and validate safety of industrial processes [31, 7]. In computer science, a set of prior work focuses on understanding ICS communication patterns, showing that communication flows indeed reflect the regular and (semi-)automated character of process control systems [5]. Other efforts focus on analyzing security threats in ICS. For example, some authors analyze protocol vulnerabilities [6, 1, 4], explore the lack of compliance to protocol specifications in different PLCs [8, 28]. To address security threats some efforts exploit communication patterns for anomaly-detection [20, 29] However, the effects that one can find at the flow-level remain limited; detecting semantic process changes requires inspection of the application layer. Consequently, some authors propose to parse network protocols for extracting information that can highlight changes to the process environment. For example, authors perform partial protocol parsing to enumerate functionality that Modbus clients use, aiming to detect unexpected deviations in requests sent to PLCs [10] and interpret events on a higher level [15], fingerprint and monitor current device configuration remotely [22, 26]. Düssel et al. [11] propose using application syntax (not *semantics*) for network-based anomaly detection; they use Bro to parse RPC, SMB, NetBIOS services inside process control networks, but do not further examine ICS-specific protocols. In terms of classic IDS signatures, DigitalBond provides Snort preprocessors that add support for matching on Modbus/DNP3/EtherNetIP protocol fields [2]. McLaughlin in [21] proposes a host-based approach for analyzing PLC behavior; they use a set of methods to reconstruct PLC configuration and process safety interlocks from PLC program code to build semantically harmful malware.

To the best of our knowledge there are only two prior efforts that extract and analyze process data values from network traffic. First, Fovino et al. [13] track current values of selected critical process parameters and thereby maintain a virtual image of the process plant that they then use to detect predefined undesirable system states. Their method of annotating critical parameters and states requires manual, intensive involvement of plant experts and thus remains expensive and likely incomplete. Second, Gao et al.[14] use neural networks to classify between normal and tampered process variables that are under injection attacks. Both works validate their approaches in controlled testbed environments. By contrast, we perform an unsupervised modeling on real world environement and provide in depth discussion of semantics that influence our results.

9. REFERENCES

[1] Project basecamp. http://www.digitalbond.com/tools/basecamp/. [accessed May 2013].

[2] Quickdraw SCADA IDS. http://www.digitalbond.com/tools/quickdraw/. [accessed May 2013].

[3] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, Aug. 2008.

[4] Common cybersecurity vulnerabilities in industrial control systems. U.S. Department of Homeland Security, 2011.

[5] R. R. Barbosa, R. Sadre, and A. Pras. Difficulties in modeling SCADA traffic: a comparative analysis. In *Proceedings of the 13th international conference on Passive and Active Measurement*, PAM'12, pages 126–135, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] C. Bellettini and J. L. Rrushi. Vulnerability analysis of SCADA protocol binaries through detection of memory access taintedness. In L. J. Hill, editor, *Proc. 8th IEEE SMC Information Assurance Workshop*, pages 341–348. IEEE Press, 2007.

[7] G. E. P. Box and G. Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990.

[8] E. Byres, D. Hoffman, and N. Kube. On shaky ground - a study of security vulnerabilities in control protocols. In *NPIC HMIT*, 2006.

[9] A. Carcano, I. N. Fovino, M. Masera, and A. Trombetta. Critical information infrastructure security. chapter Scada Malware, a Proof of Concept, pages 211–222. Springer-Verlag, Berlin, Heidelberg, 2009.

[10] S. Cheung, B. Dutertre, M. Fong, U. Lindqvist, K. Skinner, and A. Valdes. Using model-based intrusion detection for SCADA networks. In *Proceedings of the SCADA Security Scientific Symposium*. Digital Bond, 2007.

[11] P. Düssel, C. Gehl, P. Laskov, J.-U. Busser, C. Stormann, and J. Kastner. Cyber-critical infrastructure protection using real-time payload-based anomaly detection. In *Proceedings of the 4th international conference on Critical information infrastructures security*, CRITIS'09, pages 85–97, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier- symantec security response. [online], 2011.

[13] I. N. Fovino, A. Carcano, M. Masera, A. Trombetta, and T. Delacheze-Murel. Modbus/DNP3 state-based intrusion detection system. In *Proceedings of the 2010 24th IEEE International Conference on Advanced Information Networking and Applications*, AINA '10, pages 729–736, Washington, DC, USA, 2010. IEEE Computer Society.

[14] W. Gao, T. Morris, B. Reaves, and D. Richey. On SCADA Control System Command and Response Injection and Intrusion Detection. In *eCrime Researchers Summit (eCrime)*, 2010.

[15] J. Gonzalez and M. Papa. Passive scanning in Modbus networks. In *Critical Infrastructure Protection*, volume 253 of *IFIP International Federation for Information Processing*, pages 175–187, 2007.

[16] M. Hoon. Parameter Estimation of Nearly non-Stationary Autoregressive Processes. Technical report, Delft University of Technology, 1995.

[17] R. J. Hyndman and G. Athanasopoulos. Forecasting: principles and practice. An online textbook. [last accessed May 2013].

[18] P. International. Profinet application layer service definition, 2004. V. 1.95.

[19] R. Langner. *Robust Control System Networks*. Momentum Press, 2011.

[20] O. Linda, T. Vollmer, and M. Manic. Neural network based intrusion detection system for critical infrastructures. In *Neural Networks, IJCNN International Joint Conference on*, pages 1827 –1834, june 2009.

[21] S. McLaughlin. On dynamic malware payloads aimed at programmable logic controllers. In *Proceedings of the 6th USENIX conference on Hot topics in security*, HotSec'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[22] P. Oman and M. Phillips. Intrusion detection and event monitoring in SCADA networks. In *Critical Infrastructure Protection*, volume 253 of *IFIP International Federation for Information Processing*, pages 161–173. Springer US, 2007.

[23] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for Writing Application Protocol Parsers. In *ACM Internet Measurement Conference*, 2006.

[24] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[25] Schneider Electric. *Modbus Protocol and Register Map for ION Devices*, 70022-0124-00 edition.

[26] R. Shayto, B. Porter, R. Chandia, M. Papa, and S. Shenoi. Assessing the integrity of field devices in Modbus networks. In *Critical Infrastructure Protection II*, volume 290 of *IFIP International Federation for Information Processing*, pages 115–128. Springer Boston, 2009.

[27] N. Sugiura. Further analysts of the data by Akaike' s information criterion and the finite corrections. *Communications in Statistics - Theory and Methods*, 7(1):13–26, 1978.

[28] A. Treytl, T. Sauter, and C. Schwaiger. Security measures for industrial fieldbus systems - state of the art and solutions for IP-based approaches. In *Proceedings IEEE International Workshop on Factory Communication Systems*, pages 201 – 209, 2004.

[29] A. Valdes and S. Cheung. Communication pattern anomaly detection in process control systems. In *Proceedings of International Conference on Technologies for Homeland Security*, Waltham, MA, May 11–12, 2009. IEEE.

[30] C. Vulnerabilities and Exposures. CVE-2010-4709 - heap-based buffer overflow in automated solutions Modbus/TCP Master OPC Server. http://www.cvedetails.com/cve/CVE-2010-4709/, 2011. [accessed November 2012].

[31] G. B. Wetherill and D. W. Brown. *Statistical process control : theory and practice*. Chapman and Hall, London, New York, 1991.

APPENDIX

A. THREAT MODEL

To better define the intrusions we are targeting, we survey plausible ICS attacks carried out over the Modbus TCP protocol as a case study. We classify the attacks according to the level of analysis they require for detection.

In Table 4 we aggregate the attacks into three categories. Basic *Level 1* attacks operate at the IP or TCP level, such as manipulating packet sizes. *Level 2* attacks manipulate fields of the Modbus payload, such as breaking the protocol conventions or transmitting values outside the range of either the protocol specification or what the receiving side supports (e.g., invalid function codes). Finally, *Level 3* attacks

represent activity that is legal at the protocol level, yet violates semantic constraints that a process imposes, including both semantically incorrect messages (e.g., conflicting commands) and operations that lead the site into an undesirable state (e.g., a command to open a pump when it must remain shut). The Level 3 attacks represent the most challenging target for the detection, and we focus our effort on their detection.

Caracano et al. [9] present a proof-of-concept malware that aims at diverting process flow in ICS. The malware performs simple value manipulations and generates Modbus packets that constitute legitimate commands. More specifically, the malware tracks the status of current values and crafts packets that will invert coil values or set registers to their maximum/minimum allowed value according to the protocol specification. Such attacks can potentially lead to fatal consequences on critical process variables. However, the precise effect on the process remains specific to each environment.

The most relevant example of a real-world Level 3 attack is Stuxnet. By attacking PLCs, this malware crossed a boundary as the first publicly known malware that injected *semantically meaningful commands* into a highly-specific plant environment. Stuxnet managed to divert process by generating malicious yet technically valid control commands that changed the behaviour of centrifuges.

Generally, by performing a Level 3 attack, an attacker can do the following damage to the system:

1. Impact operator awareness (e.g., corrupt the flow and content of commands sent towards HMI to prevent the normal reaction). Such attacks typically involve sending forged alarms/events, or incorrect measurements.
2. Divert the process (e.g., change critical parameters that will deviate the process). Such attack may involve actions such as sending a "stop" command to a PLC, or changing control variables.

Although some semantic attacks do not have specific requirements with respect to timing (e.g., a previously unseen *stop PLC* command can be considered as dangerous any time), we generally view the ICS as a time-dependent system, thus the attacks which include value manipulation do take timing into consideration.

Table 4: Summary of attacks against Modbus implementations

Level	Impact	Attack description	Example
1	Data integrity	Corrupt integrity by adding or removing data to the packet.	Craft a packet that has a different length than defined in parameters or is longer than 260 bytes (imposed by the spec.) [2].
2	Reconnaissance	Explore implemented functionalities in PLC.	Probe various FC and listen for responses and exceptions [2].
	System integrity	Exploit lack of specification compliance.	Manipulate application parameters within spec. (e.g., offset) or outside of spec. (e.g., illegal FC) [2, 8, 30].
		Perform unauthorized use of an administrative command.	Use FC 8-0A to clear counters and diagnostics audit [2].
	Denial of service	Perform MITM to enforce system delay.	Send exception codes 05, 06 or FC 8-04 to enforce Listen mode [2].
		Perform unauthorized use of administrative command.	Use FC 8-01 to restart TCP communication [2, 8].
3	Process reconnaissance	Explore structure of memory map.	Probe readable/writable points and listen for exceptions to understand process implementation details [2].
	Process integrity	Perform unauthorized change of process variable.	Write inverted read values. Write maximal or minimal data values allowed per data point [9].

FC: Function code defining the type of functionality in Modbus. *MITM: Man-in-the-middle attack.*