From System Specification to Anomaly Detection (and back)

Davide Fauri
Eindhoven University of Technology
d.fauri@tue.nl

Daniel Ricardo dos Santos
Eindhoven University of Technology
d.r.dos.santos@tue.nl

Elisa Costante
SecurityMatters
elisa.costante@secmatters.com

Jerry den Hartog
Eindhoven University of Technology
j.d.hartog@tue.nl

Sandro Etalle
SecurityMatters
Eindhoven University of Technology
s.etalle@tue.nl

Stefano Tonetta
Fondazione Bruno Kessler
tonettas@fbk.eu

ABSTRACT

Industrial control systems have stringent safety and security demands. High safety assurance can be obtained by specifying the system with possible faults and monitoring it to ensure these faults are properly addressed. Addressing security requires considering unpredictable attacker behavior. Anomaly detection, with its data driven approach, can detect simple unusual behavior and system-based attacks like the propagation of malware; on the other hand, anomaly detection is less suitable to detect more complex *process-based* attacks [26, 35] and it provides little actionability in presence of an alert. The alternative to anomaly detection is to use specification-based intrusion detection (see, e.g., [28, 30, 33, 45]), which is more suitable to detect process-based attacks, but is typically expensive to set up and less scalable. We propose to combine a lightweight formal system specification with anomaly detection, providing data-driven monitoring. The combination is based on mapping elements of the specification to elements of the network traffic. This allows extracting locations to monitor and relevant context information from the formal specification, thus semantically enriching the raised alerts and making them actionable. On the other hand, it also allows under-specification of data-based properties in the formal model; some predicates can be left uninterpreted and the monitoring can be used to learn a model for them. We demonstrate our methodology on a smart manufacturing use case.

CCS CONCEPTS

• **Security and privacy** → **Intrusion detection systems**; • **Computer systems organization** → *Embedded and cyber-physical systems*;

KEYWORDS

Intrusion detection, specification, anomaly detection, industrial control system

Partially funded by EU H2020 CITADEL (nr 700665), ITEA3 APPSTACLE (nr 15017) and IDEA-ICS project by NWO.
CPS-SPC'17, November 3, 2017, Dallas, TX, USA

1 INTRODUCTION

Industrial control systems (ICS) drive critical infrastructures and other cyber-physical processes, such as smart manufacturing. Traditional safety issues for such systems are further complicated with new security concerns triggered by increased interconnectivity and observed attacks, such as Stuxnet, Dragonfly, and BlackEnergy (see, e.g., [36]). Consider a remotely controlled smart manufacturing plant. The added connectivity and lower physical supervision yield additional possibilities for attacks. Thus, in addition to known possible failures, one must also consider unpredictable behavior, such as that caused by an attacker.

Cyber attackers can compromise ICS through, e.g., backdoors or holes in the network perimeter, vulnerabilities in common protocols, man-in-the-middle attacks, and malware. Industrial control systems also suffer from a special class of process-based attacks, where the attacker modifies the status of the system to damage the integrity of the physical process. Real examples of such attacks include the rapid opening and closing of a water valve to cause a pipe burst, or the opening of multiple breakers in a power distribution system to cause an outage [44].

Process-based attacks are of crucial importance for ICS, as demonstrated by a large interest from the research community [13, 15, 25, 28, 30, 33, 45, 46]. Detecting these attacks is challenging because the commands and the parameters used are within the norm and a stealthy attacker might even tamper with sensor readings; what causes the damage has to do with a combination of, e.g., the timing and order of commands and stored measurements.

Many works that try to address this problem rely on the presence of an accurate specification (computer model) of the actual physical behavior of the system [13, 45]. By feeding the appropriate parameters and measurements in the computer model, one can identify when the physical system actually deviates from the expected behavior. The disadvantages of specification-based systems are rooted in its operational costs: obtaining such a complete specification (which can involve timing, control logic, dynamical systems, etc.) may take weeks of work for each system to be monitored and therefore makes this approach hardly practical of the vast majority of ICS systems. In addition, specification-based systems (by definition) do not scale well to cover large scale systems and cope with difficulty with system changes, while practical experience suggests that, even in industrial control systems, system changes are frequent [20].

The alternative to a (physical) specification-based approach is to use a learning-based approach. The idea is to (a) sniff the network

traffic of the system, (b) identify the variables (parameters) in it, (c) extract features and make a model of (learn) their "normal" behavior and finally (d) compare the behavior of the running system with the one prescribed by the model. Monitoring the network for anomalies using a semantics and context-aware approach can be applied with great success in ICS systems [51], offering the ability to detect attacks/unwanted deviations with low cost (false positives). To achieve this, one needs to know what to monitor and what forms relevant context information. Two disadvantages of anomaly detection systems are: (a) they detect only the simpler process-based attacks and (b) they lack *actionability* [20], in the sense that alerts typically give very little information to act upon; this is due to the semantic gap between alerts and their operational interpretation [41]. An alert is considered actionable when it is meaningful to a security operator, who can take appropriate measures based on the information presented.

In this paper, we propose to combine formal specification with learning as a new methodology to overcome the semantic gap between network anomalies and actionable alerts, without incurring in the high costs of a full-fledged specification-based system.

We do so by leveraging a lightweight logical system specification. With our methodology, system specification is simplified by allowing the use of uninterpreted predicates that, instead of being fully specified, are learned through network monitoring. The use of such predicates alleviates the burden on the operator who has to provide the specification. Detected anomalies are evaluated by an operator: false positives lead to an update of the detection model, whereas true positives lead to an updated specification.

The crux of the approach is a mapping between elements of the formal specification (logical variables and events) and network traffic fields. This is done manually by the operator using his/her background domain knowledge and the help of automatic classification and visualization techniques. A simple example is a packet sniffed from a Modbus communication channel that reads value 30 from register $r1$. The network traffic alone does not provide any meaning to the number 30. It may be that 30 is an anomalous value that was rarely or never seen in $r1$ during the training of the model and therefore an alert will be raised. But if the operator does not know what this value means, there is little that he/she can do to identify it as a real attack or to fix the problem. On the other hand, if register $r1$ is mapped to process variable *temperature*—defined in the system specification—then the operator can decide if the alert is a true or false positive and act on it. In fact, the operator can use true alerts to update the specification of the system with a range of normal temperatures for the system.

Besides providing actionable alerts, such a lightweight specification can also enhance the selection of features to be monitored. In a real ICS there may be thousands of variables involved in the process. Selecting which of those to monitor is a daunting task. The specification can be used in this scenario in three ways: (i) the user may select to monitor only the variables that are defined in the specification; (ii) the user may choose to monitor all possible variables, in which case only those defined in the specification will provide actionable alerts; or (iii) the user may choose to monitor only the variables defined in the specification together with other correlated variables. We leave the last option for future work and explore the first two in this paper.

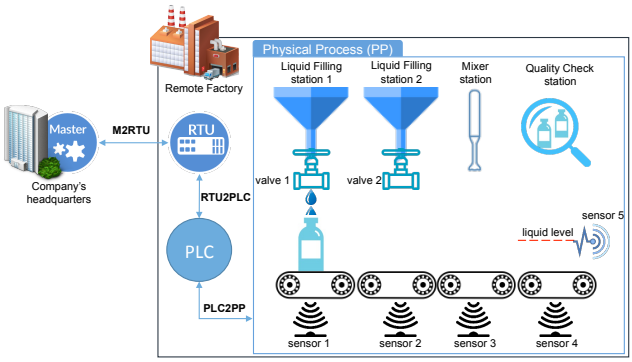

Figure 1: Smart manufacturing use case

In this way, system specification and anomaly detection strengthen each other; system specification is simplified and can be used to synthesize modelling setting (locations, features, context) for the anomaly detection. To the best of our knowledge, this is the first attempt to integrate formal system specification and anomaly detection. Related work (e.g., run-time verification [21] and policy monitoring [4, 5]) may use similar specification languages, but does not consider data-driven anomaly detection and interpret all first-order symbols on the system execution.

The rest of this paper is organized as follows. In Section 2, we present a motivating scenario that is used throughout the paper. In Section 3, we give an overview of our approach, sketching the specification and monitoring framework. In Section 4, we describe the proposed method for combining the formal specification and anomaly detection in more details. In Section 5, we present the implementation and an initial experimental evaluation of our approach. In Section 6, we discuss related work. Finally, in Section 7, we provide some conclusions and ideas for future work.

2 MOTIVATING SCENARIO

To illustrate our approach, we use a remotely controlled medicine bottle filling plant, as shown in Figure 1. Bottles on a belt encounter two *liquid filling stations*, a *mixer*, and a *quality check*. Each station has a *sensor* to detect the presence of the bottle. The filling stations also have actuators to open and close the filling tube *valves*. At the quality check, the volume of liquid is measured with an *ultrasonic sensor*. The process is remotely controlled and monitored from the *company headquarters* through an internet connection (the *M2RTU* communication channel in the Figure) to a Remote Terminal Unit (*RTU*) at the plant, which connects (using the *RTU2PLC* channel) to a Programmable Logic Controller (*PLC*) that drives the actuators and sensors in the *physical process* (using the *PLC2PP* channel).

This industrial process comprises the following steps:

(1) The system operator, who is in the headquarters, connects to a Human Machine Interface (HMI), where a process window displays a picture of the system and the states of sensors and actuators. The operator can define the setpoints of the actuators (milliliters for ingredients and seconds for the mixer) and start or stop the remote operation. When the operator sends the setpoints to the RTU and presses the start button, the PLC starts the process.

(2) When a bottle activates a position sensor, the PLC stops the belt and starts the corresponding actuator (valve or mixer).
(3) When the bottle is at an ingredient position, the PLC opens the ingredient's valve and fills the bottle with the quantity defined by the setpoint.
(4) When the bottle is at the mixer position, the PLC activates the mixer for as many seconds as defined by the setpoint.
(5) When the bottle is at the quality check position, an ultrasonic sensor measures the quantity of liquid in the bottle. The measurement is sent to a Supervisory Control and Data Acquisition (SCADA) system (not shown in Figure 1 for the sake of simplicity), which decides whether to keep the bottle or not, according to the setpoints defined by the operator. For instance, if the setpoints for the two ingredients are 40ml and 50ml, then the acceptable measurement is 90ml.
(6) The process continues indefinitely. Employees go to the remote plant every three days to collect all the bottles produced.

The plant uses mainly Modbus/TCP [42] to connect the devices. In the Modbus network, one device called Master (the RTU in this example) requests information and the others, called Slaves, supply it (the PLC in our example). The Master can also write information to the Slaves. This information is stored in *registers* and *coils*, which have unique addresses per device. The protocol has four object data types: *discrete inputs* are 1-bit read-only, *coils* are 1-bit read-write, *input registers* are 16-bit read-only, and *holding registers* are 16-bit read-write. Reading, writing, and other operations are indicated by function codes, e.g., 1 is a "Read coils" operation, 4 is "Read Input Registers", 15 is "Write Multiple Coils", and 16 is "Write Multiple Holding Registers". To define the setpoints, for instance, the RTU sends a network packet to the PLC containing the function code 16, the address of the register supposed to hold this information, and the value of the setpoint.

Other devices and protocols used in the implementation of this scenario are described in Section 5.

Attacks. The three days unsupervised time span mentioned in the last step of the process description gives an attacker the opportunity to exploit vulnerabilities of the system and to launch attacks that may disrupt the plant production. An attacker can, for instance, subvert the process to produce bottles with too much or too little liquid. This can be implemented as man-in-the-middle attacks that tamper with the communication between the operator and the RTU, the RTU and the PLC or the PLC and the physical process.

An attack aiming to overflow a bottle can be implemented by learning the setpoints sent by the operator to the RTU, storing them, and modifying them with new values that overflow the bottles. As a result, the PLC receives incorrect setpoints and the actuators that control the valves of the two ingredients stay open for longer than they are supposed to. The attacker can also modify information sent by the PLC to the RTU so that the SCADA system receives wrong process information and the operator is unaware of the attack.

While the quality check sensor may confirm the right amount of liquid (assuming that its reading is correctly sent to the SCADA system), it cannot determine how much of each ingredient is included. Therefore, another attack may change the ratio of the ingredients, since incorrect mixtures could be poisonous. This can be implemented via Modbus similarly to the overflow attack, but switching the first setpoint with the second one or vice-versa.

Thus, we would like to ensure that *Valid* amounts of both liquids are used. In Section 4, we detail how this example is specified and how we can learn what is a *Valid* bottle by using our framework.

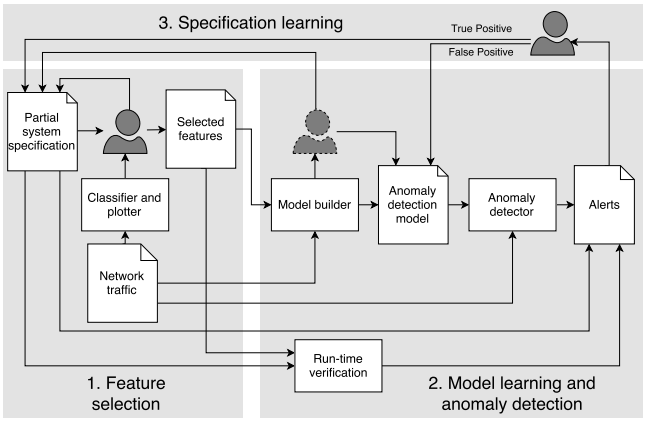

Figure 2: Overview of the approach. Note that the intervention of human operators in steps 1 and 3 is mandatory, while in step 2 it is optional.

3 SOLUTION OVERVIEW

Our methodology, shown in Figure 2, is divided in three phases, described in details below. Notice that throughout this paper, we use the term 'process variable' in different contexts. A process variable can be a logical variable defined in the system specification or a specific memory location of a specific process device. The difference should be clear in the context.

1. Feature selection. The objective of the first phase is to refine the captured "raw" *network traffic* into a set of semantically meaningful variables, represented by the *selected features*. These variables should be mapped to the events and variables described in the specification.

To do so, we first parse the captured traffic using a packet dissector (such as Wireshark[1]) to extract the process variables contained in the application layer. We assign to each variable a unique identifier, consisting of a combination of the IP address of the device where the variable was observed, the protocol and type of variable (e.g., 16-bit Modbus register) and the memory address that the variable occupies within the device.

Since industrial network communications are mostly machine-to-machine, the representation of process variables is usually optimized for efficient transmission and storage, rather than understandability. As such, variables are associated with only a memory address or an encoded label instead of a descriptive name that relates to the variable meaning. On the other hand, the provided specification usually gives only a high-level description of the process, not mentioning how each variable is encoded.

[1] https://www.wireshark.org/

To link the low-level and high-level elements, we ask for the human intervention of a domain expert. We first automatically *classify* the variables into [19]: *sensors*, which represent measurements of physical quantities related to the industrial process; *counters*, whose values are cyclic counters modulo a maximal value; *constants*, which always hold the same value; and *others*, which do not match any of the three previous categories. Then we *plot* the values of the process variables over time, highlighting the significant changes as potential events, e.g., the flipping of a binary variable or a sudden change in a real-valued variable. We then let the expert assign a label to these variables and events, according to the list of terms contained in the specification. Additionally, the expert can choose to define a transformation of two or more variables into a *derived variable*, capable of expressing new information about the system state. For example, variables $ingr1$ and $ingr2$ can be combined as $total = ingr1 + ingr2$ to provide the total volume of liquid in a bottle. These combinations may or may not be foreseen in the specification.

In some cases, an under-specified predicate in the specification might only indicate *which* variables are involved, but not *how* to combine them. For example, the composition of the final product can be defined as an uninterpreted function of other variables: $composition = f(ingr1, ingr2)$. Following [51], we define *compound variables* as tuples of single variables. Then we can obtain compound variables from such uninterpreted functions, e.g., $(ingr1, ingr2)$ in the case above.

We thus obtain two intermediate results: a mapping from the unique identifier to the semantical meaning of each process variable and a set of transformations that combine single variables into derived and compound ones. Already in this first phase, the domain expert has the possibility of improving the initial specification by using the knowledge gained from the classification and visualization of the variables. For instance, if the *total* variable was not foreseen in the specification, it may be added as a feature at this point and used by the expert to formulate suitable properties to add to the specification.

2. Model learning and anomaly detection. During the first phase, the domain expert performs a more 'qualitative' evaluation, looking at the behavior of variables in general, e.g., the domain expert did not need to know how many milliliters are in a bottle, but only that the liquid should not spill out. During the learning phase, we exploit the 'quantitative' information contained in the network traffic to learn a semantically meaningful model of normality based on the features selected in phase 1.

To do so, we apply the mapping and set of transformations to the process variables extracted from the network. The resulting variables can then be processed by the learning module of an anomaly detection system, which forms the *model builder* component.

The parameters learned from the data, namely a set of bins and detection thresholds (more details in Section 3.2), which form the *anomaly detection model*, can be reviewed by the domain expert. This expert can then update the specification according to the learned parameters (e.g., by adding a constraint saying that the level of $ingr1$ should never be lower than a certain threshold, observed during the learning process), or tune the anomaly detection model to include rare but valid values.

We note that, in the previous phase, the human intervention was necessary for inferring a labeling of the process variables. During the learning phase, instead, the human involvement is optional. Nevertheless, it helps in adding information about the process that cannot be reliably learned from the training data. It also helps in keeping the specification up to date with regard to the actual process, since they may be at odds.

The result of this phase is a set of detection thresholds against which the single and derived variables should be compared. This can be applied to the real-time monitoring of the process. From a live network capture, we extract and combine the same process variables as we did from the training dataset, and compare them against the thresholds. Values that exceed the thresholds are labeled as anomalous and an *alert* is raised for each type of anomaly. Since the cause of an anomaly is given by a specific feature in the model and the feature can be mapped back to an element of the specification, the alert is enriched with the information in the specification, e.g., the variable name and possible constraints associated to it.

Although not the focus of this paper, parallel to the anomaly detection there may be a *run-time verification* component (see, e.g., [21]) with monitors synthesized directly from the specification and the mapping from network to specification elements. The specification is used in this case to prevent known faults or attacks. The alerts raised by this component are actionable by default, since they are generated from the specification, which clearly defines the context and the variables involved in the alert.

3. Specification learning. In the last phase, the actionable alerts raised by the anomaly detector can be either true or false positives. The usual cause of a false positive is the appearance of a rare but valid value which was not present in the training data nor in the specification. It is up to the human operator to classify each alert as a true or false positive. This task is made easier by two facts. First, since the anomaly detector does not monitor every possible process variable, but only those features selected with the help of the system specification, the number of false positives should decrease so that the user does not have to manually classify an unreasonable number of alerts. Second, since the alerts are presented in a meaningful way, with the attached semantics, it is easier for the operator to determine whether an alert is a true or a false positive.

Those alerts that are classified as true positives can be used to refine the specification, by adding new constraints on the process or giving an interpretation to under-specified predicates. Those alerts that are classified as false positives can be used to tune the parameters of the anomaly detection model, such as increasing a threshold or widening the bins.

With an updated specification, the fault detection mechanism is capable of generating new monitors from the added properties. There could also be a reconfiguration component that synthesizes a safe system configuration from the specification. Such (semi)-automated reconfiguration of the system to satisfy the updated specification is one of the research topics of the CITADEL[2] project, where an adaptation plane can trigger the reconfiguration of the system's operational plane.

[2] http://www.citadel-project.org/

In the remainder of this Section, we present the specification and anomaly detection framework used throughout this paper.

3.1 Specification framework

System properties language. We derive features from system properties that are expressed in a formal language. This language is based on a fragment of First-Order Linear-time Temporal Logic (LTL) [34], extended with Event Freezing operators to compare terms at different points of time, and explicit notion of time (see [43] for a detailed description). The first-order symbols are interpreted using a background theory (as in, e.g., [22]).

The system is described with a set V of variables (representing the status of the system), a set E of events (such as open a valve and send a message), and a first-order signature Σ (e.g., arithmetic operators and/or user-defined functional symbols). The properties are built from these sets with the following grammar:

$$\begin{aligned} \phi &:= p(t, \ldots, t) \mid \phi U \phi \mid \phi S \phi \\ t &:= v \mid e \mid f(t, \ldots, t) \mid time \mid t@next(\phi) \mid t@last(\phi) \end{aligned}$$

where p and f represent, respectively, a predicate and a function symbol in the signature Σ; v a variable in V; e an event in E; $time$ the time; $t@next(\phi)$ and $t@last(\phi)$ the value of the term t at the next and at the last state in which ϕ holds, respectively; U the LTL temporal operator "until"; and S its past dual operator "since". "Always" (G) can be encoded in this as usual (i.e., $G\phi := \neg F \neg \phi$, where $F\phi := true U \phi$). Typical properties are invariants of the form $G\phi$, such as:

$$G(bottle\_ok \rightarrow Valid(ingr1, ingr2))$$

The semantics of the properties is defined in terms of the execution traces of the system. A trace is a sequence $s_0, e_0, s_1, e_1, \ldots$ where, for all $i \geq 0$, s_i is an assignment to the variables in V and $e_i \in E \cup \mathbb{R}_0^+$ such that the sequence series $\sum_{i, e_i \in \mathbb{R}_0^+} e_i$ is diverging. The partial sum $\sum_{i \leq j, e_i \in \mathbb{R}_0^+} e_i$ represents the time at the j-th state of the trace and so the value of the time symbol $time$.

A property is interpreted over a first-order structure, which assigns an interpretation to the signature symbols, and over a trace that represents the evolution of the system variables and events (the interpretation of the signature symbols is rigid, i.e. it does not vary along the trace). Some signature symbols are primitive and interpreted by the background theory. Other symbols are *uninterpreted* and may be used to model abstract concepts. We typically use infix notation of interpreted symbols and function notation for uninterpreted ones. So, for example, $v_1 * v_2$ represents the standard multiplication between two numbers, while $Valid(v_1, v_2)$ is uninterpreted. For the scope of this paper, we use a question mark to highlight that a symbol is uninterpreted and write $?Valid(v_1, v_2)$.

Fault detection specification. In the design of high-assurance systems, after the system requirements are specified, safety or security engineers identify the possible hazards or vulnerabilities that may lead to a system failure. Fault-detection, isolation and recovery (FDIR) components are then added to the design to address them. An FDIR design specifies a *diagnosis condition*, capturing the fault that must be detected, and a *monitoring condition*, giving the events and variables that must be monitored [6]. In fact, only a subset $V_{obs} \subseteq V$ of variables and $E_{obs} \subseteq E$ of events are observable and can be used by the monitor to determine when the diagnosis condition occurred. Therefore, the diagnosis condition should be *diagnosable* [39], i.e. enough events/variables of the system must be observable to detect any occurrence of the diagnosis condition. See [9, 10] for the formal characterization and [7] for automated synthesis of these observables.

3.2 Anomaly detection

Anomaly detection learns models of normal behavior to be able to detect unknown faults or attacks. Subsequent behavior triggers a comparison with the model and alerts the operator of any deviations. However, effective anomaly detection needs to reason at the application level in a context-aware manner. White-box monitoring [16, 17] has been successfully applied in the ICS setting [51] and creates semantical models that are context-aware. The application semantics is reconstructed from the messages and used to build user understandable alarms and models; for example capturing the normal commands sent and the ranges used for their arguments.

Reconstructing semantical information from messages requires an in-depth protocol parser to extract information and a semantic model to interpret the extracted information. For instance, in Modbus/TCP, one needs to parse the application layer to obtain function code 16 and then map this to the "Write Multiple Holding Registers" operation. Two limiting factors to extracting the semantics are i) protocol expressiveness; some protocols use the same code for multiple concepts and ii) availability of automatically importable system specifications.

If specification documents are available, they are often expressed in natural language, requiring lengthy and expensive manual import operations. Here, we limit the required effort by leveraging the fault detection specification. We still need to specify how concepts from the formal language manifest themselves in the ICS network traffic, however, we only need this for those concepts actually used in the monitoring specification, not for every possible message. Further automation of this (for example by linking concepts to the PLC ladder logic) is an interesting venue for future work.

Information needed to build semantical concepts may be distributed over multiple events. We use the approach of [17] to build features spanning multiple messages: a group of messages is formed by filtering events between a (logical) start and an end condition. A feature extracts a value (e.g., a duration) from the resulting sequence of events.

White-box framework. The starting point for our anomaly detection model is a set of *messages* within the system. Network traffic consists of normal messages (the majority) and malicious messages. What constitutes normal traffic typically depends on the context, e.g., which user is sending a message and what is the destination of the message.

Directly estimating the distribution of normal messages to distinguish them from malicious messages is infeasible. Therefore, we consider the *features* of a message that capture its relevant properties. Examples of features are the destination IP address of a message, the function being called on that destination, an argument to that function, etc. We consider *numerical* features (e.g., integers) and *nominal* features (e.g., names). Besides these *elementary features*, we also consider *compound features*, which are tuples of numeric

and/or nominal values, and *derived features*, which are relations between elementary features (such as the sum of two numerical features). Together, all features form a *feature vector*, which extracts *value vectors* out of the messages. Take as an example a Modbus message from 192.168.0.30:1500 to 192.168.0.100:80 that calls function 5 with argument 15. If we have a feature vector ('connection', 'function_code', 'parameter_1'), that would yield the value vector ((192.168.0.30:1500, 192.168.0.100:80), 5, 15).

Malicious messages may have the same values as normal messages on some features. However, if we choose the right features, they will have values that are rare among normal traffic on at least one of the features. For features that can take many different values, such as numeric features, individual values may be rare even for normal traffic. Yet rare values should indicate that a message is likely an attack. Therefore, we group values into *bins* and consider the occurrence of bins rather than individual values. For nominal features we usually consider the binning where each bin consists of a single element, for numeric features we usually consider bins that are ranges $[v_l, v_u)$ and for compound features we consider bins that are the product of such bins. Our fundamental assumption is that for attack messages a feature will yield bins with low probability. We are therefore looking for *anomalies*, messages where the probability of some feature is below a given *threshold*.

Our main goal is to find anomalies along with their causes so they can be presented to an operator who can then evaluate and address them. We could define anomalous messages as those yielding an anomalous value on one of the features. However, not all entities on a system behave the same. Thus, what is normal for one entity may be anomalous for another. To address this we introduce *profiles* which are conditional distributions describing the normal traffic per entity. We use a *profiling feature*, which we assume is the first feature in a feature vector, to determine the 'entity' for a message. In this way we can for example make a profile for each source IP, for each destination IP or for each connection. A message is anomalous when the entity involved is anomalous or when one of the features of the message is anomalous for that entity. We call such feature a *cause* of the anomaly.

One can use a single global threshold for all features and bins, a threshold per feature, or even per individual bin. The choice of a profiling feature is based on its ability to distinguish between sets of behavior. In industrial control systems, devices tend to follow different behavior patterns, which makes device identifiers suitable profiling features.

4 FROM SPECIFICATION TO MONITORING AND BACK

In this section, we describe and illustrate the proposed methodology to enhance monitoring with anomaly detection. System properties that contain uninterpreted predicates provide the context and features for the anomaly detection. We focus on invariant properties $G\phi$. We again use the bottle filling plant scenario and assume that there is only one bottle on the belt at a time. This avoids complicating formulations to take into account multiple opening/closing of the valves and linking these to the right bottle.

A key property in our bottle filling scenario is that the mixture should be valid once the filled and mixed bottle passes the quality check (event *bottle_ok*), which can be expressed as:

$$G(bottle\_ok \rightarrow ?Valid(ingr1, ingr2)) .$$

Since an invariant should always hold, we can assume that $?Valid(ingr1, ingr2)$ is normally true within *context bottle_ok*; its negation is an *anomaly*. We thus aim to find anomalous values for the *diagnosis features* $ingr1$ and $ingr2$, i.e. the parameters of the uninterpreted predicates, within this context. A diagnosis feature t might not be directly observable. Therefore, the next step is to define a monitoring specification capturing the diagnosis features in terms of observable events. With context ψ we require that the system model satisfies $G(\psi \rightarrow t = f(O))$ for some term f over observable features $O \subseteq V_{obs} \cup E_{obs}$.

In our scenario, the messages on channels M2RTU, RTU2PLC, and PLC2PP are the observables and we need to find a formal relationship between them and the diagnosis features ($ingr1, ingr2$). In the system model we can prove that, when *bottle_ok*, the ingredient amounts are exactly those set remotely through a message to the PLC. Formally:

$$G(bottle\_ok \rightarrow ingr1 = setpoint\_1@last(RTU2PLC.msg)) ,$$
$$G(bottle\_ok \rightarrow ingr2 = setpoint\_2@last(RTU2PLC.msg)) .$$

Using this relationship to obtain the monitoring features means monitoring the RTU2PLC channel for anomalous setpoint values. An anomalous mixture being set, e.g., due to an attacker manipulating the communication before it reaches the PLC, will result in an alarm.

When an alarm is raised, it is analyzed to determine whether it is a false or true alarm. From this feedback, we learn a formula ψ to refine our specification. In case of a false alarm, the anomaly detection model gets updated. In case of a true alarm, we refine the system specification to disallow it in this context. For example, assume that we get a true alarm which states that $ingr1 = 0$, which is unusual, and the operator, realizing neither ingredient should be 0, generalizes the feedback to $\psi = (ingr1 = 0 \vee ingr2 = 0)$. We then update the system with invariant:

$$G(\neg(bottle\_ok \wedge (ingr1 = 0 \vee ingr2 = 0))) .$$

The system implementation can be refined by adding an FDIR component that monitors for one of the ingredients not being present in the bottle and, in that case, discards it.

With the monitoring specification above, only the setpoint values are validated, trusting that the PLC will use these correctly. If we are concerned that the PLC itself may be compromised, we should validate the actual commands being sent to the physical process. We can prove on the system model that, when the bottle passes the quality check, the amount of the first ingredient is equal to the time passed between opening and closing the first valve, multiplied by a flow rate (e.g., $10ml/s$):

$$\begin{aligned} G(bottle\_ok \rightarrow ingr1 = (\quad & time@last(open1) \quad - \\ & time@last(close1) \quad) * 10) , \\ G(bottle\_ok \rightarrow ingr2 = (\quad & time@last(open2) \quad - \\ & time@last(close2) \quad) * 10) . \end{aligned}$$

This monitoring specification uses multiple observable events to compute the diagnosis feature. The anomaly detection thus considers the group of messages specified by the filter ($open1 \vee close1 \vee open2 \vee close2$) from $open1$ to $bottle\_ok$.

To link abstract events with Modbus messages, the anomaly detection uses the following mapping:

$$open1 = PLC2PP.msg.Modbus.\langle regdst = valve1\_id, fc = 16, val = 1\rangle$$
$$close1 = PLC2PP.msg.Modbus.\langle regdst = valve1\_id, fc = 16, val = 0\rangle$$

and similar for the other events, where *regdst* indicates the destination register to be accessed, *fc* is the function code of the operation (16 is a *write* operation), and *val* shows the value to be set. The valves have one value that can be written, indicating its state: open (1) or closed (0). We thus still need to extract the semantic meaning for some network events, but only need to do this for those events and variables actually used in the monitoring specification.

5 EVALUATION

We evaluated our approach using simulated datasets from our bottle filling plant example. The datasets consist of a partial specification of the operation of the plant and two types of network traffic captures. The first capture type is taken from a standard execution of the simulated process; the second type is taken after introducing a malicious attacker in the network, who alters the process variables sent to the control device.

The white-box anomaly detection approach that we use [17] has already been shown to be feasible in this setting [51], giving good trade-offs between false positives and detection rate when applied on simulated cases as well as on data from real-world applications.

Quantitative analysis in this scenario is not very meaningful: as stated before, we use an already validated model learning and anomaly detection engine. Therefore we can avoid computing standard anomaly detection metrics such as accuracy and false positive ratio. On the other hand, the scalability of our proposed framework cannot be evaluated on our small scale simulator, and is one of the goals left for future work.

Instead, we perform a qualitative analysis of the methodology. Our use case is simple but not a 'toy example'; it is realistic enough that, apart from potential scalability issues, results can be translated to deployed ICS systems. This allows us to do a realistic qualitative evaluation showing the feasibility and benefits of our approach of combining lightweight specification with anomaly detection.

After explaining the experimental setup (Section 5.1), we divide our evaluation in three steps. First, we select the features to use in our white-box anomaly detector and evaluate how the partial specification improves the selection over a naive baseline (Section 5.2, "**From specification to feature selection**"). Second, we train the anomaly detector and test it against our simulated attacks to the process: we assess how the high-level knowledge from the specification improves the actionability of the resulting alerts (Section 5.2, "**From specification to anomaly detection**"). Third, we evaluate how the results of our anomaly detection help us complete the under-specified predicates of the partial specification (Section 5.2, "**From anomaly detection to specification**").

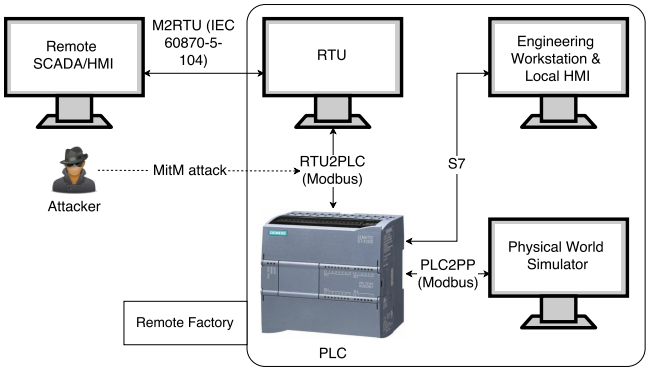

Figure 3: Overview of the simulator

5.1 Experimental setup

Simulator. Figure 3 shows an overview of the simulator we used, which has five components modeling different parts of the bottle filling plant in Figure 1.

The Physical Process shown in Figure 1 is implemented by the *Physical World Simulator*, which simulates the conveyor belt, the bottles, and the sensors and actuators of the manufacturing process. The simulator automatically advances the conveyor belt every half second; it also keeps track of the position of each bottle on the belt, as well as the volume of liquid contained in it. The sensors and actuators are managed by the PLC via Modbus/TCP thanks to a Modbus client contained in the simulator. The *Engineering Workstation* was not shown in Figure 1 because it is not part of the main plant process, but is used to program the control logic, download it to the PLC using the S7 protocol [40], and to display the local HMI that is placed in the local area of the plant. It is a Windows XP machine running Siemens TIA Portal v12[3] and is directly connected with the PLC. The *PLC* used in this simulator is a Siemens S7-1200, which communicates with the RTU via Modbus/TCP. The *RTU* component is the same as the one shown in Figure 1: it is used to connect the remote SCADA with the PLC. The RTU is located in the factory and it controls only one PLC; its main task is to translate the messages that the remote SCADA sends by using the IEC 60870-5-104 [27] protocol into Modbus/TCP messages and vice versa. The last component is the *Remote SCADA/HMI*, used to display the process to the remote operators: in our use case, it resides in the company's headquarters.

Overall, most key components in our simulator are physical devices or systems actually used in practice, such as the PLC or the engineering stations running actual SCADA management software. Only the physical process, as well as some parts of the underlying network that connect to it, are simulated. The resulting simulator is therefore not just a collection of scripts: it is a significant cyber-physical system.

Datasets. We captured five datasets from the simulation, as shown in Table 1. Column 'Dataset' shows the name of the dataset; '#Messages' reports the number of messages in the dataset; and 'Time' is the time span of each dataset. The first three datasets

[3] http://www.industry.siemens.com/topics/global/en/tia-portal/Pages/default.aspx

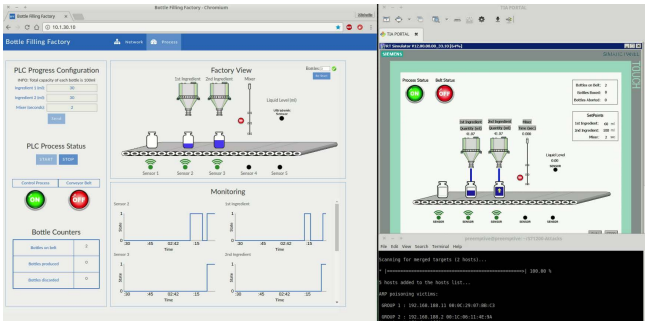

Figure 4: Overflow attack. The remote HMI (left) shows the setpoints sent by the operator (`setpoint_1` = 30, `setpoint_2` = 30). The local HMI (right) shows the setpoints manipulated by the attacker (`setpoint_1` = 60, `setpoint_2` = 100).

represent the normal operation of the plant, producing 1, 10, and 80 bottles, respectively. The last two datasets have the injected attacks described below.

Attacks. To create traffic captures that can be used to test our detection capabilities, we implemented two man-in-the-middle attacks that tamper with the Modbus communication between the PLC and the RTU (RTU2PLC channel in Figures 1 and 3). Both attacks intercept the original communication, learn the setpoints sent by the operator to the PLC, store them, and modify them with new values. As a result, the PLC receives incorrect setpoints and controls the actuators accordingly. The attacker then modifies information sent by the PLC to the RTU so that the SCADA system receives wrong process information and the operator is unaware of the attack.

The first attack is demonstrated in Figure 4: it aims to overflow a bottle by raising the setpoints of the two valves, controlling them to stay open for longer than they are supposed to. These changes have a combined effect of a larger spillage of product than what each bottle can contain, potentially causing economical damage due to wasted ingredients, or disrupting the production line.

The second attack aims to alter the quality of the final product, by modifying the setpoints of the two valves so that the total volume of liquid is the same as before, but the ratio of ingredients in a bottle is changed. The attacker selects the new setpoints from within a range of previously observed values, to avoid them from being detected as anomalous. Changing the composition of the final product can cause delayed economical damages such as product recall costs, or lost revenue due to inferior product quality.

Table 1: Datasets used in the experiments

Dataset	#Messages	Time
normal-1bottle	17034	60 s
normal-10bottles	38702	143 s
normal-80bottles	267291	1027 s
modbus-overflow	77511	214 s
modbus-ratio	77511	214 s

Implementation. Figure 5 shows the implementation of our methodology based on the elements described in Figure 2. We capture the traffic of the local network within the remote factory: specifically, we consider the traffic being sent to and from the PLC. This means that we are downstream from the man-in-the-middle attacks, and see the modified, malicious values that are injected. We use pyshark[4], a Python binding for tshark[5], to parse the network traffic captures and extract the process variables from the application layer of Modbus packets. We store the normal traffic and attack traffic datasets in a tidy data format [50], where each row is a single observation of a process variable, consisting of a low-level unique identifier, a timestamp, and the observed value.

To assist the human operator in assigning a high-level label to the process variables, we classify them and visualize their values in an interactive plot, shown in Figure 6. The operator can explore, reorder, and compare the process variables before labeling them: the assigned labels are then used instead of the unique identifiers.

We exploit this label-based indexing to easily define Python functions to select and (optionally) combine process variables from the dataset. An example is shown in Figure 5 (top right), where `total_liquid_in_bottle` is defined as the sum of `setpoint_1` and `setpoint_2`.

We base our implementation of the model learning and anomaly detection modules on previous work [51]. After learning the bins and detection thresholds for each process variable, we store the results in a text file structured in JSON. This choice allows for human inspection and modification, while still being understandable by our detection engine. An excerpt of the text file is shown in Figure 5 (bottom right), where the bin for `total_liquid_in_bottle` is the interval [50, 100].

Finally, we modify the alerts shown on screen by the anomaly detector to be more descriptive. We do so by referencing the anomalous process variables by their high-level label, by mentioning which threshold is being violated, and by adding optional custom messages that can better explain the context of the alert. An example of such an alert is shown Figure 5 (middle right), where the operator can clearly see that the bottles may be overflowing.

5.2 Experiments

From specification to feature selection. To provide a naive baseline for feature selection, we first attempted to classify and label process variables using only low-level information gathered from the network traffic data. Out of 26 detected process variables, we managed to confidently assign a high-level label to only 4. Of those identified variables, the binary variable `system_on` represents the on/off state of the whole bottling line; and the other three counter variables `bottles_started`, `bottles_on_belt`, and `bottles_done`, represent the total number of bottles that started being processed, those that are being processed at the moment (i.e. on the moving belt), and the total number of bottles that completed the process, respectively. For some of the other variables, we noticed that the timings of their change points are usually correlated: this observation suggested a logical ordering of operations, but was not sufficient to meaningfully label the involved variables.

[4] https://github.com/KimiNewt/pyshark
[5] https://www.wireshark.org/docs/man-pages/tshark.html

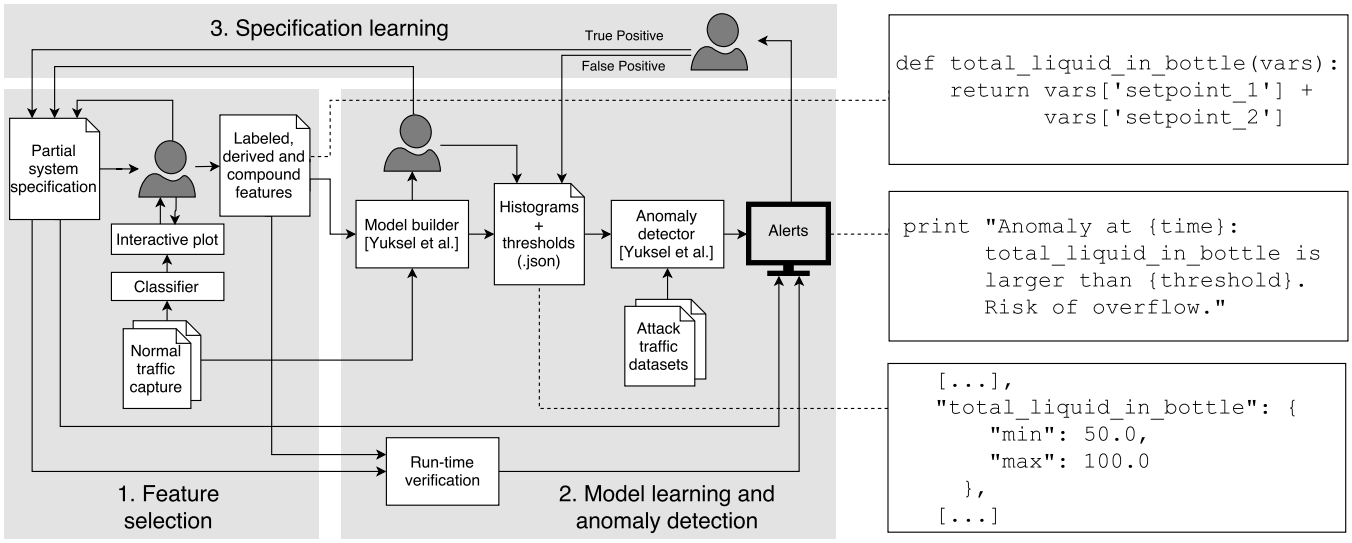

Figure 5: Implementation. The boxes on the right show an example of label-based combination of process variables (top), an example of actionable alert message (middle), and an example of easy to edit parameters for the anomaly detector (bottom).

We then used the information contained in the partial specification as an aid to labeling. We compared the logical ordering that we inferred previously, with the high-level concept of events in the specification. As a result, we could reliably label 22 out of 26 process variables: this is presented in Table 2. We could not label three constant variables, as their value remained set to 0 across all the training data; and a binary variable, whose behavior was not reflected in the specification.

Finally, we used the invariant properties of the specification as guidelines to select and combine specific process variables into *derived features*. For example, the *bottle_ok* property relates to the total amount of ingredients in a bottle: we therefore created the `total_liquid_in_bottle` feature, defined as the sum of the two setpoints for *ingr*1 and *ingr*2. Additionally, to test the under-specified predicate $?Valid$, relative to the quality of the final product, we combined the setpoints for the volume of the two ingredients in a (`setpoint_1`, `setpoint_2`) compound feature, to which we assigned the `composition` label.

From specification to anomaly detection. We trained the anomaly detector using only labeled features: that is, we used only the subset of process variables that were assigned a high-level label during the feature selection, plus the variables derived from them. We tested the trained anomaly detector on the first attack dataset, where the attacker overflows a bottle by incrementing the two setpoints on registers R-000 (which we labeled `setpoint_1`) and R-001 (which we labeled `setpoint_2`) to the values 60 and 100.

The detector raised three alerts: one for each setpoint, and a third for the `total_liquid_in_bottle` feature. All alerts returned by the anomaly detection system were of the type shown in the upper right box of Figure 5.

Within each alert, there is important contextual information such as the time when the anomalous event happened, which process variable triggered the anomaly (identified using its high-level label), if the violated threshold was an upper or lower bound, and the value of said threshold. Moreover, a custom message indicates the likely consequence of the anomalous event. All this contextual, actionable information helps the security operator to determine where and when the anomaly happened in the production workflow, assess the possible impact of such an anomaly, and rapidly decide whether to flag it as a false positive or not.

From anomaly detection to specification. Finally, we show how alerts raised by our trained anomaly detector can help in completing the partial specification. To do so, we tested our detector on the second attack dataset. In this case, the attacker modifies `setpoint_1` and `setpoint_2` to alter the ratio between *ingr*1 and *ingr*2, with the intent to alter the quality of the final product. A naive anomaly detector would not have raised any alert, as neither process variable individually assumes anomalous values.

Our anomaly detector, though, raised an alert on the `composition`=(`setpoint_1`,`setpoint_2`) compound feature. Since we defined this compound feature starting from an under-specified predicate, the raised alert did not contain enough information to say what is wrong.

By reviewing the alert and inspecting the anomalous values of the compound feature, we recognized the anomaly as a true positive, and from our domain knowledge we inferred that the feature `composition` should be related to the ratio of the ingredients in a bottle (i.e., `composition = setpoint_1/setpoint_2`). Following the method shown in Section 4, we could add the property $G(\neg(bottle\_ok \wedge (ingr1/ingr2 \neq k)))$ to the specification, where k is a constant defining the ratio of ingredients. The value $k = 0.75$ was learned from the histogram of the `composition` compound feature, which showed that the ratio `setpoint_1`:`setpoint_2` is always 3:4. Note that in general the expert can use values directly from domain knowledge in addition to those learned by the model.

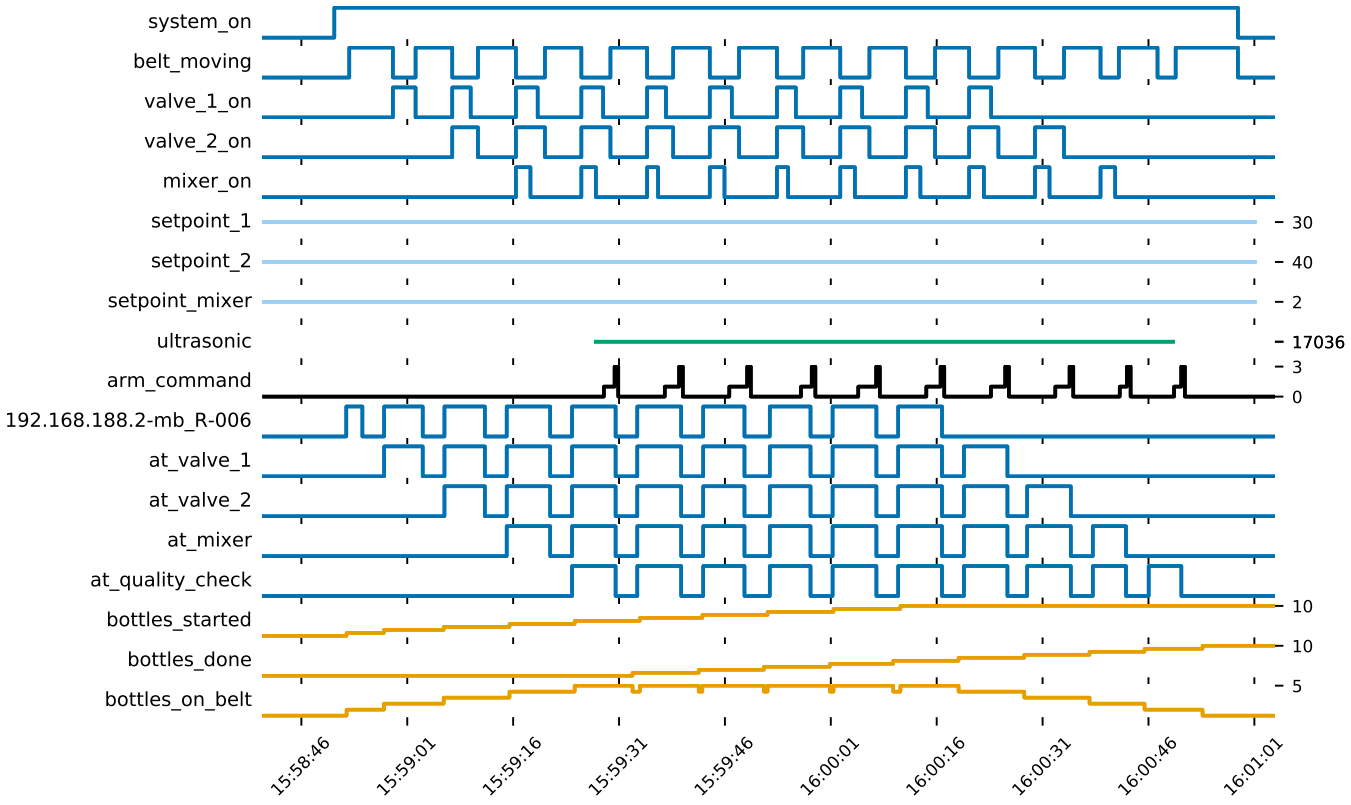

Figure 6: Interactive plot of the process variables. The plot lasts for 10 consecutive bottle filling operations. The x axis is the time and the y axis shows the value of each register (labeled with a variable name). Binary variables are shown in dark blue, constant variables in light blue, sensor measurements in green, counter variables in orange, other variables in black. The operator could not assign a meaning to the variable 192.168.188.2-mb_R-006, which retains its original, uninformative identifier.

From the examples shown in the paper and the experiments, we learned that a *Valid* bottle is one where the amount of neither ingredient is 0, the total amount in a bottle (defined as the sum of both ingredients) is in a valid range $t_l \leq total \leq t_u$ and the ratio of ingredients is constant, as shown in the following invariants:

$$G(\neg(bottle\_ok \wedge (ingr1 = 0 \vee ingr2 = 0)))$$
$$G(\neg(bottle\_ok \wedge (total < t_l \vee total > t_u)))$$
$$G(\neg(bottle\_ok \wedge (ingr1/ingr2 \neq k)))$$

where the constants $t_l = 50$, $t_u = 100$, and $k = 0.75$ were learned from the network data and the variables $ingr1$ and $ingr2$, as well as the event *bottle_ok* were mapped to network fields as shown in Figure 6 and formalized in Section 4.

6 RELATED WORK

Several ICS-specific intrusion detection systems that rely on the payload of network traffic are available in the literature [15, 24, 25, 28, 30, 45]. These solutions adopt different techniques: detecting variations in the length or contents of the payload to identify threats such as buffer overflow or injections [8, 37, 48, 49]; detecting deviations from the protocol specification by using a policy-based [32] or a learning-based approach [53, 54] to identify attacks that e.g., exploit protocol vulnerabilities; and detecting process attacks by

Table 2: Mapping of the registers/coils to labels

coil	label	coil	label
C-000	system_on	C-003	valve2_on
C-001	belt_moving	C-004	mixer_on
C-002	valve1_on	C-005	?

register	label	register	label
R-000	setpoint_1	R-010	at_quality_check
R-001	setpoint_2	R-011	bottles_started
R-002	setpoint_mixer	R-012	bottles_done
R-003	ultrasonic	R-013	?
R-004	?	R-014	bottles_on_belt
R-005	arm_command	R-015	pre_valve1
R-006	?	R-016	infra_valve1_valve2
R-007	at_valve1	R-017	infra_valve2_mixer
R-008	at_valve2	R-018	infra_mixer_qc
R-009	at_mixer	R-019	post_quality_check

analyzing the trend of physical variables [25, 45] or sequences of events [15].

There are specification-based approaches that: monitor the values of known critical variables stored in a controller's memory [28]; adopt the same state model used during the controller design phase to detect possible attacks by validating the behavior of variables against implicit physical constraints [45]; verify power measurements against the known physical behavior of a transformer [30]; or apply model-based prediction to verify whether, given a certain command, the system can reach an insecure state [33]. Caselli et al. [14] automated the development of specification rules for networked control systems by using available documentation, but their solution was implemented for BACnet-based building automation systems.

On the other hand, learning-based approaches leverage the predictability of the control and communication flows of an ICS to learn a prediction model. In [24], the authors propose a passive Modbus scanner that can gradually learn the patterns in communication and control flows. Attempts towards single variable monitoring were proposed in [15, 19, 25], where distinct techniques (thresholding windows, Markov chains and autoregression, respectively) were used to learn the behavior of a process. In the first two works, human inspection is required to validate the role and criticality of each variable, whereas in [25] heuristics are applied for variable classification. Finally, as far as we know, all the available approaches create models per single variable, missing the opportunity of exploiting compound and derived variables as we did in our work.

The white box anomaly detection framework was developed originally for database systems [17]. The framework uses an anomaly-based engine that automatically learns a model of normal user behavior, allowing it to flag when insiders carry out anomalous transactions. It was also integrated into a hybrid framework for data loss prevention and detection that combines signature-based and anomaly-based solutions [18]. It exploits an operator's feedback on alerts to automatically build and update signatures of attacks that are used to block undesired transactions before they can cause any damage. A major difference between the database and ICS settings is that database commands are text-based, instead of binary, and much easier to interpret than network fields in ICS protocols. Therefore, the alerts raised in [17] are immediately meaningful, whereas for ICS we need to assign a meaning by using external information.

The white-box approach was applied to ICS in [51]. The main challenge was to derive elementary and compound features from ICS protocols. This was done by extracting features from ICS protocol fields, as in this work. However, in that work almost all protocol fields become elementary features and compound features are constructed from sequences of correlated elementary features. A complementary solution to feature selection was discussed in [52], where two metrics were shown, namely stability and granularity. Stability helps finding features that yield rare values in normal traffic; for a feature F, stability is measured as the likelihood that a validation value has also been seen in the training. Granularity is the fraction of information (measured using Shannon entropy) retained when going from attribute to feature.

Basin et al. have many works using run-time verification for policy monitoring (see, e.g., [1–5]). Different from our methodology, these works assume a complete specification of the system and its properties and do not employ anomaly detection. Koucham et al. [29] built on results from runtime verification and specification mining to automatically infer and monitor process specifications. Such specifications are represented by sets of temporal safety properties over states and events corresponding to sensors and actuators. The properties are then synthesized as monitors which report violations on execution traces. The technique is similar to the ones by Basin et al., with the difference that specification can be mined from execution traces. In our work, we do not assume the existence of execution traces or logs, instead we use raw network data. However, using specification mining [31] (or even process mining [47] as in [35]) after we are able to map semantical information to network data could be an interesting future development.

7 CONCLUSIONS AND FUTURE WORK

This paper shows how formal specification and anomaly based monitoring can be combined to overcome the semantic gap between network anomalies and actionable alerts. We also show that this combination is to the benefit of both, since the specification can be refined with the data gathered from network traffic.

We have evaluated our approach on an example smart manufacturing setting. We have specified the example scenario, developed a simulation environment with common attacks, captured network traffic, built a white-box anomaly detection model, used it to detect attacks in the form of actionable alerts and to learn predicates that strengthened the initial specification.

The work presented in this paper has been conducted within the CITADEL project, which is focused on the development of adaptive systems for critical infrastructure protection. The smart manufacturing scenario is a simplification of a real use case of the CITADEL project. We are currently working on specifying and gathering network traffic of the real system. The approach of the project is based on a formal specification of the system, which enables the analysis of safety and security properties based on rigorous formal techniques. A fundamental problem is the monitoring of failures and, in particular, communication failures, either caused by some faulty hardware components of the network or by malicious attacks. Monitors are traditionally synthesized automatically from a formal (automata-based or logic-based) specification of the system [21]. However, specifying system properties or the faults that may occur is challenging and may be not feasible in the face of unknown behavior, such as attacks. With the methodology shown in this paper, fault detection is extended to include monitoring for network anomalies.

7.1 Future work

As mentioned in Section 5, our simulator is too simple to properly test the scalability of our approach. For this reason, as future work we aim for further validation against a more complex, real world use case, such as the Secure Water Treatment (SWaT) testbed [23], which has been formally specified [38]. This would also be important to showcase more kinds of specification properties that we are able to learn from the anomaly detection. We also aim at extending the approach to more general under-specified properties, extracting

from the temporal structure of the property the (temporal) context in which the uninterpreted predicates should hold.

Moreover, we plan to automate as much as possible the mapping between network fields and elements of the specification. We have begun exploring the use of association rule mining to learn relationships between process variables. We envision the use of ontologies as an intermediate step. One scenario that could immediately benefit from the use of ontologies is building automation systems, using the BACnet protocol[6]. There are well-known ontologies for BACnet[7] and the protocol carries variable labels along with the data. Given the limits of automation, we also intend to pursue the use of advanced visualization techniques (e.g., [11, 12]) to assist domain experts in mapping process variables to high-level labels.

Finally, one of the main goals of the CITADEL project is the implementation of an embedded platform that includes mechanisms to monitor its operation and its interaction with the environment, and mechanisms to use dynamic reconfiguration capabilities to maintain safe and secure operation. In the case of CITADEL, bridging the semantic gap as discussed in this paper has the potential of producing alerts that can be *automatically* handled by the system to bring it to a new safe configuration.

REFERENCES

[1] D. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, and H. Mantel. 2014. Scalable Offline Monitoring. In *RV*. Springer, 31–47.
[2] D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. 2011. MONPOLY: Monitoring Usage-control Policies. In *RV*. Springer, 360–364.
[3] D. Basin, F. Klaedtke, S. Marinovic, and E. Zalinescu. 2012. Monitoring Compliance Policies over Incomplete and Disagreeing Logs. In *RV*. Springer, 151–167.
[4] D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring of Temporal First-order Properties with Aggregations. In *RV*. Springer, 40–58.
[5] D. Basin, F. Klaedtke, and S. Müller. Policy Monitoring in First-Order Temporal Logic. In *CAV*. Springer, 1–18.
[6] B. Bittner, M. Bozzano, A. Cimatti, R. De Ferluc, M. Gario, A. Guiotto, and Y. Yushtein. 2014. An Integrated Process for FDIR Design in Aerospace. In *IMBSA*. Springer, 82–95.
[7] B. Bittner, M. Bozzano, A. Cimatti, and X. Olive. 2012. Symbolic Synthesis of Observability Requirements for Diagnosability. In *AAAI*. AAAI Press, 712–718.
[8] D. Bolzoni, S. Etalle, and P. Hartel. 2006. POSEIDON: a 2-tier anomaly-based network intrusion detection system. In *IWIA*. IEEE, 10 pp.–156.
[9] M. Bozzano, A. Cimatti, M. Gario, and S. Tonetta. 2014. Formal Design of Fault Detection and Identification Components Using Temporal Epistemic Logic. In *TACAS*. Springer, 326–340.
[10] M. Bozzano, A. Cimatti, M. Gario, and S. Tonetta. 2015. Formal Design of Asynchronous Fault Detection and Identification Components using Temporal Epistemic Logic. *LMCS* 11, 4 (2015).
[11] B.C.M. Cappers and J.J. van Wijk. 2015. SNAPS: Semantic network traffic analysis through projection and selection. In *VizSec*. IEEE, 1–8.
[12] B.C.M. Cappers and J.J. van Wijk. 2016. Understanding the context of network traffic alerts. In *VizSec*. IEEE, 1–8.
[13] A.A. Cárdenas, S. Amin, Z. Lin, Y. Huang, C. Huang, and S. Sastry. 2011. Attacks against process control systems: risk assessment, detection, and response. In *ASIACCS*. ACM, 355–366.
[14] M. Caselli, E. Zambon, J. Amann, R. Sommer, and F. Kargl. 2016. Specification Mining for Intrusion Detection in Networked Control Systems. In *USENIX Security*. USENIX Association, 791–806.
[15] M. Caselli, E. Zambon, and F. Kargl. 2015. Sequence-aware Intrusion Detection in Industrial Control Systems. In *CPSS*. ACM, 13–24.
[16] E. Costante, J. den Hartog, M Petković, S. Etalle, and M. Pechenizkiy. 2014. Hunting the Unknown - White-Box Database Leakage Detection. In *DBSec*. Springer, 243–259.
[17] E. Costante, J.I. den Hartog, M. Petković, S. Etalle, and M. Pechenizkiy. 2017. A white-box anomaly-based framework for database leakage detection. *JISA* 32 (2017), 27–46.
[18] E. Costante, S Etalle, D. Fauri, J.I. den Hartog, and N. Zannone. A Hybrid Framework for Data Loss Prevention and Detection. In *S&P WRIT*. IEEE, 324–333.
[19] N. Erez and A. Wool. 2015. Control variable classification, modeling and anomaly detection in Modbus/TCP SCADA systems. *IJCIP* 10 (2015), 59–70.
[20] Sandro Etalle. 2017. From Intrusion Detection to Software Design. In *ESORICS*. Springer, 1–10.
[21] Y. Falcone, K. Havelund, and G. Reger. 2013. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*. IOS Press.
[22] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. 2007. Combination Methods for Satisfiability and Model-Checking of Infinite-State Systems. In *CADE*. Springer, 362–378.
[23] J. Goh, S. Adepu, K.N. Junejo, and A. Mathur. 2016. A Dataset to Support Research in the Design of Secure Water Treatment Systems. In *CRITIS*.
[24] J. Gonzalez and M. Papa. Passive scanning in Modbus networks. In *ICCIP*. Springer, 175–187.
[25] D. Hadžiosmanović, R. Sommer, E. Zambon, and P.H. Hartel. 2014. Through the Eye of the PLC: Semantic Security Monitoring for Industrial Processes. In *ACSAC*. ACM, 126–135.
[26] D. Hadžiosmanović, D. Bolzoni, and P.H. Hartel. 2012. A log mining approach for process monitoring in SCADA. *IJIS* 11, 4 (2012), 231–251.
[27] IEC 60870-5-104 2006. *Transmission protocols - Network access for IEC 60870-5-101 using standard transport profiles*. Standard. IEC.
[28] W. Jardine, S. Frey, B. Green, and A. Rashid. 2016. SENAMI: Selective Non-Invasive Active Monitoring for ICS Intrusion Detection. In *CPS-SPC*. ACM, 23–34.
[29] O. Koucham, S. Mocanu, G. Hiet, J. Thiriet, and F. Majorczyk. 2016. Detecting Process-Aware Attacks in Sequential Control Systems. In *NordSec*. Springer, 20–36.
[30] G. Koutsandria, V. Muthukumar, M. Parvania, S. Peisert, C. McParland, and A. Scaglione. 2014. A hybrid network IDS for protective digital relays in the power transmission grid. In *SmartGridComm*. IEEE, 908–913.
[31] C. Lemieux, D. Park, and I. Beschastnikh. 2015. General LTL Specification Mining. In *ASE*. IEEE, 81–92.
[32] H. Lin, A. Slagell, C. Di Martino, Z. Kalbarczyk, and R.K. Iyer. 2013. Adapting Bro into SCADA: Building a Specification-based Intrusion Detection System for the DNP3 Protocol. In *CSIIRW '13*. ACM, 5:1–5:4.
[33] H. Lin, A. Slagell, Z. Kalbarczyk, P. Sauer, and R. Iyer. 2017. Runtime Semantic Security Analysis to Detect and Mitigate Control-related Attacks in Power Grids. *IEEE SG* (2017), 1–1.
[34] Z. Manna and A. Pnueli. 1995. *Temporal verification of reactive systems: safety*. Springer.
[35] D. Myers, K. Radke, S. Suriadi, and E. Foo. 2017. Process Discovery for Industrial Control System Cyber Attack Detection. In *IFIP SEC*. Springer, 61–75.
[36] Repository of Industrial Security Incidents (RISI). 2015. Online Incident Database. (2015). Retrieved August 2, 2017 from http://www.risidata.com/Database
[37] D. Perdisci, R.and Ariu, P. Fogla, G. Giacinto, and W. Lee. 2009. McPAD: A multiple classifier system for accurate payload-based anomaly detection. *Computer Networks* 53, 6 (2009), 864–881.
[38] M. Rocchetto and N.O. Tippenhauer. 2017. Towards Formal Security Analysis of Industrial Control Systems. In *ASIACCS*. ACM, 114–126.
[39] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D.C. Teneketzis. 1996. Failure diagnosis using discrete-event models. *IEEE CST* 4, 2 (1996).
[40] Siemens (Ed.). 2013. *Cpu-cpu communication with Simatic controllers*.
[41] R. Sommer and V. Paxson. 2010. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *S&P*. IEEE, 305–316.
[42] A. Swales. 1999. *Open modbus/tcp specification*. Tech. Report. Schneider Electric.
[43] S. Tonetta. 2017. Linear-time Temporal Logic with Event Freezing Functions. In *GandALF*. Open Publishing Association, 195–209.
[44] Trend Micro. 2016. First malware-driven power outage reported in Ukraine. (Jan. 2016). http://www.trendmicro.com/vinfo/us/security/news/cyber-attacks/first-malware-driven-power-outage-reported-in-ukraine
[45] D.I. Urbina, J. Giraldo, A.A. Cardenas, N.O. Tippenhauer, J. Valente, M. Faisal, J. Ruths, R. Candell, and H. Sandberg. 2016. Limiting the Impact of Stealthy Attacks on Industrial Control Systems. In *CCS*. ACM, 1092–1105.
[46] A. Valdes and S. Cheung. Communication pattern anomaly detection in process control systems. In *HST*. IEEE, 22–29.
[47] W. van der Aalst. 2016. *Process Mining*. Springer.
[48] K. Wang, J.J. Parekh, and S.J. Stolfo. 2006. Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In *RAID*. Springer, 226–248.
[49] K. Wang and S.J. Stolfo. 2004. Anomalous Payload-Based Network Intrusion Detection. In *RAID*. Springer, 203–222.
[50] Hadley Wickham. 2014. Tidy data. *Journal of Statistical Software* 59, 10 (2014).
[51] Ö. Yüksel, J. den Hartog, and S. Etalle. 2016. Reading between the Fields: Practical, Effective Intrusion Detection for Industrial Control Systems. In *SAC*. ACM, 2063–2070.
[52] Ö. Yüksel, J.I. den Hartog, and S. Etalle. 2016. Towards Useful Anomaly Detection for Back Office Networks. In *ICISS*. Springer, 509–520.
[53] E. Zambon. 2014. *CRISALIS D6.1 Protocol-aware approaches*. Technical Report. EU FP7.
[54] E. Zambon, M. Caselli, and M. Almgren. 2015. *CRISALIS D6.4 Network-Driven Analysis tools*. Technical Report. EU FP7.

[6] http://www.bacnet.org/
[7] http://project-haystack.org/ and http://bacowl.sourceforge.net/