Detecting Process-Aware Attacks in Sequential Control Systems

Oualid Koucham[1](✉), Stéphane Mocanu[1], Guillaume Hiet[2], Jean-Marc Thiriet[1], and Frédéric Majorczyk[3]

[1] Univ. Grenoble Alpes, CNRS, Gipsa-lab, 38000 Grenoble, France
{oualid.koucham,stephane.mocanu, jean-marc.thiriet}@gipsa-lab.grenoble-inp.fr

[2] CIDRE/Inria, CentraleSupélec, Cesson-sévigné, France
guillaume.hiet@centralesupelec.fr

[3] DGA/Inria, Rennes, France
frederic.majorczyk@supelec.fr

Abstract. Industrial control systems (ICS) can be subject to highly sophisticated attacks which may lead the process towards critical states. Due to the particular context of ICS, protection mechanisms are not always practical, nor sufficient. On the other hand, developing a process-aware intrusion detection solution with satisfactory alert characterization remains an open problem. This paper focuses on process-aware attacks detection in sequential control systems. We build 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. We develop an efficient specification mining algorithm and use filtering rules to handle the large number of mined properties. Furthermore, we introduce the notion of activity and discuss its relevance to both specification mining and attack detection in the context of sequential control systems. The proposed approach is evaluated in a hardware-in-the-loop setting subject to targeted process-aware attacks. Overall, due to the explicit handling of process variables, the solution provides a better characterization of the alerts and a more meaningful understanding of false positives.

1 Introduction

Cyber attacks represent a growing concern for industrial control systems (ICS) [1]. On one hand, ICS are increasingly connected to traditional information systems. This trend has been spurred, among other reasons, by the adoption of commodity hardware and software components, as well as the convergence towards TCP/IP solutions [11]. On the other hand, a majority of industrial systems lack security mechanisms, having historically relied on isolation from traditional information systems. In the singular context of ICS, protection mechanisms are not sufficient, nor always practical. For instance, hardware constraints hinder the

B.B. Brumley and J. Röning (Eds.): NordSec 2016, LNCS 10014, pp. 20–36, 2016.
DOI: 10.1007/978-3-319-47560-8_2

use of measures such as encryption to ensure confidentiality or integrity [5]. Any latency within the low layers of industrial systems can affect the real-time constraints and perturb the functioning of the control loops. Despite recent efforts geared towards developing suitable protection mechanisms [15], ICS remain particularly vulnerable, highlighting the need for appropriate detection measures. In this paper, we are concerned with developing such a detection solution.

As opposed to traditional information systems, ICS are cyber-physical systems interacting with a physical process. Taking into account this aspect is paramount to the detection of targeted attacks relying on advanced knowledge of the process [17]. Noteworthy examples include the highly sophisticated Stuxnet attack [12]. ICS are characterized by a duality between continuous behavior as traditionally represented by differential equations, and sequential behavior where control follows sequences of discrete steps. Our focus is on the latter aspect. This paper presents an anomaly-based intrusion detection approach to detect process-aware *sequence* attacks targeting a particular class of systems, namely sequential control systems. Sequence attacks aim to put the process in a critical state by a malicious temporal ordering of commands or messages [6]. Examples of such attacks include *exclusion* attacks where two states should not happen simultaneously (an open valve and a running motor at the same time for instance), or *wear* attacks where components' lifetime is reduced through malicious manipulations (by, for example, repeatedly opening and closing a valve) [17]. We restrict ourselves to *qualitative* sequence attacks where only the temporal ordering matters.

General overview. We build on results from runtime verification and specification mining to automatically infer and monitor process specifications. The specifications are represented by sets of temporal safety properties [2] over states and events corresponding to sensors and actuators. The properties are synthesized as monitors which report violations on execution traces. Filtering rules allow handling the large number of mined specifications. Mining and monitoring can also be done per *activity*, a notion which captures the different subprocesses and functioning modes of a sequential system. A subprocess refers to a phase in the operation of the system. For instance, a sequential system might go through a start phase, a shutdown phase, and several intermediate phases. An activity can also distinguish between manual or automatic modes of functioning. Compared to prior work on process-aware intrusion detection [4,22], this work focuses on the sequential aspect of control systems, covers more expressive properties through a suitable formalism, and discusses a solution to alleviate the effort of manually writing process specifications. In contrast with sequence-aware solutions targeting communication patterns within ICS [6,29], the proposed approach explicitly handles process variables. This leads to improved alerts characterization, and a better understanding of false positives.

Contributions. All in all, we make the following contributions:

– We propose an approach to detect process-aware sequence attacks targeting sequential control systems by leveraging results from runtime verification and specification mining

– We suggest a number of filtering rules to handle the large size of inferred specifications, and introduce the concept of activity while discussing its relevance within sequential control systems
– We evaluate our solution in a hardware-in-the loop setting and analyze its performance and limitations

The paper is organized as follows. Section 2 provides an overview of prior work on intrusion detection within ICS. Section 3 discusses background concepts pertaining to ICS, runtime verification and specification mining. Section 4 presents our approach including our specification mining algorithm Sect. 4.2 and filtering rules Sect. 4.3. Section 5 evaluates the approach and discusses its limitations.

2 Related Work

Intrusion detection work in ICS can be classified into two broad categories: (i) approaches which seek intrusion manifestations solely in the cyber part [7,21,30], and (ii) approaches which take into account the physical process [4,14,22]. We are interested in attackers whose objective is the disruption of the underlying physical process. These attacks represent a challenge to traditional intrusion detection approaches. Thus, we argue that a knowledge of the physical process is essential to the detection of sophisticated process-aware attacks, and to the understanding of false positives. In this paper, we present a process-oriented intrusion detection solution.

A majority of the approaches found in the literature are anomaly-based, i.e. they try to detect any significant deviation from a reference behavior. These solutions often rely on assumptions about the simplicity of ICS protocols, the stability of the network's structure, or the regularity of the communications. Compared to signature-based intrusion detection, anomaly-based approaches have the crucial advantage of potentially detecting novel attacks. However, while ICS exhibit certain regularities relative to traditional systems, investigations on real-world data show that these assumptions are not always justified [6]. Moreover, anomaly-based approach, especially when relying on machine learning techniques, exhibit some drawbacks [26] such as the number of false positives, and the poor characterization of the alerts. This can lead to wrong reactions by the operators, or to a loss of confidence in the IDS alerts. As a result, some effort is needed to better characterize the alerts and handle false positives. Our approach attempts to address some of these issues.

Within the literature, the work closest to ours include the sequence-aware approaches developed in [6,29], and the process-aware approaches developed in [4,22]. Caselli et al. [6] adopt a Markov chain-based solution relying on communication patterns to detect *sequence* attacks. Sequence attacks are defined as malicious/erroneous ordering or timing of commands or messages. We argue that such attacks, in the scope of sequential control systems, are better detected by focusing on process variables instead of network communications. In the same vein, Yoon et al. [29] propose a probabilistic suffix tree-based approach to model communication patterns under a high predictability assumption. Mitchell et al.

[22] rely on manually written behavior rules to detect process-aware attacks. Carcano et al. [4] develop ISML, a language for describing critical states. While similar to our solution in terms of process awareness, both approaches require manual expression of the behavior rules and are not suitable for detecting all malicious ordering of events. This is because both approaches rely exclusively on propositional logic formulae to express behavioral rules or critical states. Such formalism cannot represent general ordering constraints. Schumann et al. [25] propose R2U2, a framework for the runtime monitoring of security properties in unmanned aerial systems. Our approach focuses on sequential control systems and discusses the automatic generation of properties.

3 Background

This section discusses the necessary background concerning ICS, runtime verification and specification mining.

3.1 Industrial Control Systems

ICS are hierarchical systems consisting of multiple components which interaction achieves an industrial objective [27]. Among these components, Programmable Logic Controllers (PLC) are of particular interest to our approach. Operating at the cyber-physical frontier, PLC execute *control logics* to regulate the physical process. This is realized through a scan cycle that includes: (i) reading inputs from sensors, (ii) executing the control logics, (iii) transitioning to new stable states, and (iv) writing outputs to actuators. Due to their critical role, PLC constitute an ideal target for process-aware attacks.

The IEC61131-3 standard [16] defines five programming languages for programmable controllers: (i) Ladder diagram, (ii) Function Block Diagram, (iii) Sequential Function Chart (SFC), (iv) Instruction List, and (v) Structured Text. In this paper, we focus on SFC which is a graphical language representing the control logic as a series of steps and transitions. SFC is especially suitable for processes exhibiting a step by step behavior [16]. This is the case of sequential control systems which are the focus of our approach.

3.2 Runtime Verification

Runtime verification [19] is a verification technique which aims at checking whether a run of a system satisfies or violates a given correctness property. In our case, a run of a system consists of a possibly infinite sequence of sets of logical propositions. Each position in the sequence represents the current state of sensors and actuators. In practice, during runtime, we only have access to finite prefixes of runs. Monitors are devices which take as input such a finite prefix, and yield a verdict belonging to a truth domain, indicating the status of the property on the trace. Using a monitor, we would like to check whether an execution satisfies a given correctness property. Thus, our aim is to detect sequence attacks using monitors synthesized from high-level correctness properties, and expressed in a formalism suitable for representing ordering constraints.

States. Let AP be a finite set of atomic propositions about sensors and actuators in the process. A state s is an element of 2^{AP}.

Linear temporal logic. Our main goal is the detection of sequence attacks involving the ordering of messages or commands. To formally represent the normal ordering relationships between states, a suitable formalism is required. Linear temporal logic [23] augments propositional logic with operators able to express ordering relationships. The syntax of LTL over the alphabet $\Sigma = 2^{AP}$, which we write $LTL(\Sigma)$, is defined as follows:

$$\varphi :: p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi U \varphi \mid X\varphi,\ p \in AP$$

We define Σ^ω (resp. Σ^*) as the set of infinite (resp. finite) sequences over Σ. Let $\varphi, \varphi_1, \varphi_2 \in LTL(\Sigma)$ be LTL formulae, $i \in \mathbb{N}$ a position, and $w(i)$ the ith element of the infinite sequence $w \in \Sigma^\omega$. LTL formulae can be inductively interpreted over elements in Σ^ω as follows:

$$\begin{aligned}
w, i &\models p \in AP &&\iff p \in w(i)\\
w, i &\models \neg\varphi &&\iff w, i \not\models \varphi\\
w, i &\models \varphi_1 \vee \varphi_2 &&\iff w, i \models \varphi_1 \vee w, i \models \varphi_2\\
w, i &\models \varphi_1 U \varphi_2 &&\iff \exists k \in \mathbb{N}, k \geq i.\ w, k \models \varphi_2 \wedge \forall i \leq j < k.\ w, j \models \varphi_1\\
w, i &\models X\varphi &&\iff w, i+1 \models \varphi
\end{aligned}$$

We also define $\Diamond\varphi \equiv trueU\varphi$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$. Here, $\neg$ and $\vee$ are, respectively, the negation and logical OR operators. The remaining logical operators ($\wedge, \Rightarrow, \Leftrightarrow$) can be derived as usual.

Events. In sequential control systems, we are often interested in expressing properties involving *events* such as *rising* ($\uparrow$) or *falling* ($\downarrow$) edges. Such events can be expressed in LTL [24]:

$$a^{\uparrow} \equiv \neg a \wedge Xa \qquad a^{\downarrow} \equiv a \wedge X\neg a$$

Monitoring and finite semantics. As discussed above, monitors only have access to finite but expanding prefixes. However, LTL formulae are interpreted over infinite sequences. This mismatch restricts the class of *monitorable* LTL formulae [2]. Monitorability refers to the capacity of a monitor, after any finite number of observations, to still detect the violation/satisfaction of a property after, at most, a finite number of additional observations. Formally, an LTL formula φ is monitorable if for every finite word $u \in \Sigma^*$, there exists a finite word $v \in \Sigma^*$ such that for any infinite word $w \in \Sigma^\omega$, uvw either satisfies or violates φ [2]. In this work, we are interested in a particular class of monitorable formulae called safety properties. Informally, a safety property states that "something bad should never happen". The formula $\Box\neg(valve_1 \wedge valve_2)$ is a safety property stating that $valve_1$ and $valve_2$ should never be simultaneously open.

In practice, monitors can be synthesized as finite state automata from LTL formulae. Such an automaton recognizes minimal bad prefixes of a safety property. Minimal bad prefixes are finite sequences which cannot be extended to satisfy the safety property, and which do not contain any other bad prefix [8]. If a safety property is violated on an infinite sequence, then it has already been violated on some finite prefix. In our case, a monitor is a finite state automaton which recognizes, as early as possible, such a prefix and reports a violation. Constructing a monitor usually requires translating the LTL formula into a Büchi automaton which accepts all infinite sequences satisfying the formula (see [28] for a formal definition). A nested depth-first-search allows the identification and removal, from the Büchi automaton, of all states which cannot initiate an accepting run. The resulting automaton can then be treated as a finite state automaton with all states accepting, and used as a monitor [8].

3.3 Process Specification Mining

Specification patterns. While LTL provides a suitable formalism to characterize safety properties pertaining to states and events ordering, expressing specifications directly in terms of formulae remains tedious. As properties grow in complexity, writing accurate and correct formulae becomes a difficult task. Thus, several works [10,24] have looked at *specification patterns* that express commonly occurring properties. By relying on such specifications patterns, we can give meaning to properties and, in our particular case, to violations of safety properties. Another advantage of using specification patterns is controlling the nature of properties to be monitored to the class of safety properties.

We base our work on a subset of Dwyer's patterns augmented with events [10,24]. Dwyer's patterns and classification are the result of an extensive review of the literature for recurring specifications. We restrict ourselves in this paper to *absence*, *universality*, *precedence* and *response* monitorable patterns. Absence patterns state that a certain event or state never occurs during the execution of the system. Universality patterns state that a certain event or state always holds during the execution of the system. Precedence and response patterns express relationships between two events or states where the occurrence of one is a necessary condition for the occurrence of the other.

Moreover, we can specify *scopes* which restrict the portion of the execution where the pattern should hold. Five scopes are defined: (i) a *global* scope, (ii) a scope starting *after* an event/state, (iii) a scope ending *before* an event/state, (iv) a scope *between* two events/states, (v) a scope starting *after* a first event/state and lasting *until* the eventual occurrence of a second event/state. All scopes are left-closed and right-open. Readers are referred to [10] for more details. In the rest of this paper, we will express specification patterns as predicates over events/states. The predicate name captures the nature and scope of the pattern. For instance, the predicate $absence\_between(X, Y, Z)$ refers to the absence pattern concerning event/state Z between events/states X and Y. An instantiation of a pattern is a mapping of placeholders to propositions.

Mining specifications. The problem of specification mining can be expressed as follows: given a finite set of specification patterns and a finite set of execution traces of a system, find all instantiations that are valid on the traces. Several works have explored this issue based on a variety of patterns [18,20]. Usually one is required to explore the space of all possible instantiations (permutations) and test the validity of each instantiation on the traces. While the size of the search space can be significant, recent work [18] has shown that using memoization and selective treatment of the traces can significantly reduce the complexity of the task even when dealing with general LTL formulae. However, the number of valid mined specifications can still remain significant, especially due to the introduction of events. Section 4 presents our mining algorithm and filtering rules to handle this issue.

4 Attack Detection Approach

4.1 General Overview

Our approach proceeds in two stages: a *mining and filtering* stage, and a *detection* stage. In the first stage (Fig. 1), specifications expressed as a set of temporal properties ($\{Spec^1_{LTL}, \ldots, Spec^m_{LTL}\}$) are mined from execution traces of the system by relying on specification patterns. When using activities, execution traces are divided depending on the current activity using the activity recognizer, and mining is done per activity. In all cases, the resulting raw specifications undergo a set of filtering rules to reduce their number.

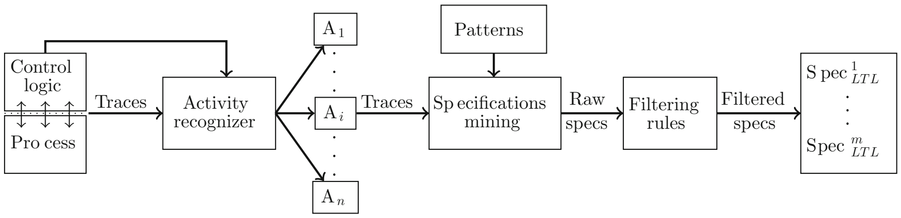

Fig. 1. First stage: mining and filtering the specifications

The traces are assumed to be free of malicious activity and representative of the normal behavior of the system. However, the representativeness is not guaranteed as the mining operates on a finite window. This can be an important source of false positives. While this limit is common to all approaches based on a learning phase, we would like to better characterize false positives (and alerts in general) by giving them meaning with respect to the process behavior, i.e. higher semantics in terms of the process. For instance, for a given alert, we would like to report on the concerned actuators/sensors, the process stage during which the alert was raised, and the reason why a violation represents an illegitimate action with respect to the process's normal behavior.

A first level of characterization is achieved by relying on specification patterns which reflect common safety properties expressed directly in terms of sensor/actuator states and events. A second level of characterization is attained by the means of activities. An activity corresponds to a subprocess or to different modes of functioning within the sequential system. Activites can distinguish between different normal behaviors within the process, while reducing the complexity of the mining phase. To define and distinguish the activities in the traces, we require a high level expression of the control logic. In our work, we derive the activity recognizer from control logic expressed as SFC. In practice, steps in the SFC are assigned to activities, and the activity recognizer interprets the SFC using its formal semantics [3]. The task of assigning activities to steps is left to an expert or a developer. As future work, we intend to explore heuristics which can guide the expert and automatically suggest activities assignments.

The monitors, synthesized from the mined and filtered properties, report violations during the detection phase (Fig. 2). When using activities, the activity recognizer dynamically identifies the current activity, and only the relevant property monitors (i.e. those pertaining to the current activity) read the trace to detect the violations and output their verdicts.

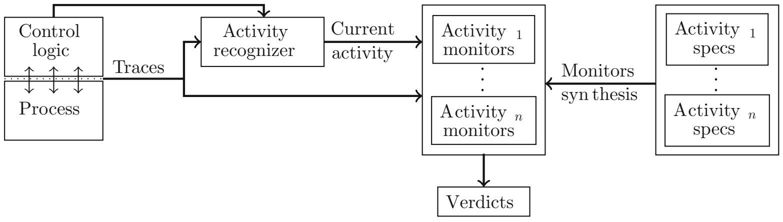

Fig. 2. Second stage: detecting specification violations

Threat model. We assume that the attacker's objective is the disruption of the physical process using qualitative sequence attacks. We also assume that the measurements sent by the sensors are correct. This means that we do not handle false data injection attacks i.e. injection of bad measurements. As we rely directly on process variables, no assumptions are made on the trustworthiness of the PLC if a proper logging mechanism is available at the field level. However, we still require the presence of a secure channel for sending alert notifications.

4.2 Mining Process Specifications

In this section, we present our mining algorithm which carefully walks through the search space to find valid properties which could have been violated on the mining traces. This constraint is captured by the notion of *falsifiability*. A falsifiable property with respect to a trace is a property which can be violated on the trace. Falsifiability is especially relevant with regards to pattern scopes. Execution traces arising from sequential control systems are highly structured due to the execution of specific control logics. As such, they contain a relatively limited number of scopes. All properties which refer to non-existent scopes are not falsifiable. Since they specify constraints on non-existent scopes, one cannot check their violation. Consider for instance the property $universality\_after(valve1, motor1)$. It corresponds to the following LTL formula: $\Box(valve1 \Rightarrow \Box motor1)$. The antecedent of the implication refers to the scope. If $valve1$ is not true at any position on the trace, the implication becomes vacuously true and the formula is not falsifiable. In addition, properties such as $absence\_before(valve1, motor1)$ and $absence\_between(valve1, valve2, motor1)$ will also be vacuously true on the trace since all these formulae involve implications with false antecedents ($\Diamond valve1$ for the first formula and $valve1 \wedge \Diamond valve2$ for the second formula). By checking the falsifiability of the initial property, we can ignore other scope-related formulae.

Thus, the main idea is to partition the space of possible instantiations in terms of scopes, then check their falsifiability with respect to their scopes in order to potentially bypass other scope-related properties. In practice, for each type of scope, we instantiate a monitor called an *auxiliary monitor* which checks whether the property is falsifiable on the traces. An auxiliary monitor essentially makes sure that the scope pertaining to a property instantiation actually occurs on the traces. As an example, for the property $universality\_after(valve1, motor1)$, we synthesize an auxiliary monitor from the formula $\Diamond end \rightarrow \neg(\neg valve1 \ \mathcal{U} \ end)$. Here, *end* represents a special symbol which is appended to the traces and is used to adapt LTL's infinite semantics to the finite mining traces [9]. The property is violated if $valve1$ does not occur on the trace. When mining, we start with single scopes (*after* or *before*) as they affect both single and double scopes (*after until* and *between*). For instance, if an *after* property involving $valve1^{\uparrow}$ as a scope is not falsifiable on the execution traces, then all *after*, *before*, *after until* and *between* properties involving the $valve1^{\uparrow}$ scope will not be falsifiable.

Typically, we have many traces at our disposal. When running the auxiliary monitors, we can enforce several policies depending on the number of violations recorded. In all of our scopes, except for the *after until* case, we require that the auxiliary monitors report no violations on all the traces, i.e. that the properties are falsifiable on all the traces. This has the advantage, with regards to our notion of activity, to naturally restrict the scopes to the variables pertaining to the activity for which the mining occurs. For the *after until* case, we require the property to be falsifiable in at least one of the traces. The goal is to limit the cases where an *after* property is valid, and which lead to several corresponding scope-irrelevant *after until* properties to become valid.

Algorithm 1 outlines our mining procedure. For each instantiation, we retrieve the verdicts of the main and auxiliary monitors (line 7). If the main monitor reports a violation, then the property is false (and falsifiable), so we move to the next instantiation in the same scope. Else if the auxiliary monitor raises a violation, we blacklist the current scope and all related scopes before moving to

Algorithm 1. Specification mining

```
Data: $Tr$ : Finite set of execution traces, $I$ : Finite set of property instances
Result: Set of valid properties
1  $\Pi = partition\_by\_scopes(I)$;
2  $blacklisted\_scopes = \{\}$;
3  $valid\_properties = \{\}$;
4  foreach $type \in \{after, after\_until, before, between, global\}$ do
5      foreach $scope \in \Pi(type) \setminus blacklisted\_scopes$ do
6          foreach $instance \in scope$ do
7              $(verdict\_inst, verdict\_aux) = check\_instance(instance, Tr, type)$;
8              if $verdict\_inst = \bot$ then $instance \leftarrow invalid$;
9              else if $verdict\_aux = \bot$ then
10                 $blacklisted\_scopes \cup = \{scope\} \cup affected\_scopes(type, scope)$;
11                 break;
12 foreach $type \in \{after, after\_until, before, between, global\}$ do
13     foreach $scope \in \Pi(type) \setminus blacklisted\_scopes$ do
14         foreach $instance \in scope$ do
15             if $instance$ is valid then $valid\_properties \cup = instance$;
16 return $valid\_properties$;
```

the next scope. The function *check_instances* returns a verdict depending on the falsifiability policy on the set of traces and the type of the instantiation.

4.3 Specifications Filtering Rules

In order to deal with the important number of specifications generated after the specification mining phase, we use a set of filtering rules. These rules are based on the semantics of Dwyer's patterns as discussed in Sect. 3.3. The idea is to find logical dependencies between mined properties based on their scopes and events/states relationships. The general form of these logical dependencies is:

$$\frac{\psi_1, \quad \psi_2, \quad \forall \sigma \in \Sigma^\omega, \sigma \models \psi_2 \Rightarrow \psi_1}{filter(\psi_2)}$$

In this rule, ψ_1 and ψ_2 are valid properties on the traces. The premise $\forall \sigma \in \Sigma^\omega, \sigma \models \psi_2 \Rightarrow \psi_1$ represents the fact that, for all infinite sequences σ, whenever property ψ_2 is satisfied, then property ψ_1 is also satisfied. In other words, by keeping track of violations of ψ_1, one can indirectly detect violations of ψ_2.

Logical dependencies arise in the case of Dwyer's patterns due to the interplay between scopes and states/events. Suppose we have mined the properties $universality\_after(valve_1, motor_1)$ and $absence\_after(valve_1, motor_1^\uparrow)$. The first property states that "motor$_1$ stays on after a state where valve$_1$ is open" while the second property states that "motor$_1$ is never started after a state where valve$_1$ is open". On all infinite sequences when the first property is satisfied, the second property will be satisfied. This is due to the fact that for $motor_1$ to be started after $valve_1$ is on, it needs to be off at some point. However, this is impossible due to the first property. Note that the converse is not true. There exists an infinite sequence where the second property is satisfied but not the first: a sequence where, after a state in which $valve_1$ is on, $motor_1$ goes off but never on. Note also that the second property is informative. The violation of the second property, in conjunction with the violation of the property $absence\_after(valve_1, motor_1^{\downarrow})$, can be symptomatic of a *wear attack* on $motor_1$. The case sketched above generalizes to the following rule:

$$\frac{absence\_after(X, Y^{\uparrow}), \quad universality\_after(X, Y)}{filter(universality\_after(X, Y))}$$

We can formally prove that $\forall \sigma \in \Sigma^{\omega}, \sigma \models \psi_2 \Rightarrow \psi_1$ for given LTL properties ψ_1 and ψ_2 by referring back to their semantics defined in Sect. 3.2. In our case, to systematically verify such logical relationship, we build the Büchi automaton corresponding to the formula $\psi_1 \vee \neg\psi_2$, and check that it accepts all possible infinite words i.e. the formula is valid [28]. We have identified and verified a non-exhaustive set of 20 rules which represent logical dependencies between patterns. Their identification relies on observations about: (i) inclusion relationships between scopes, and (ii) the interplay between events and states within the same scope such as in the example above.

5 Evaluation

In order to evaluate our solution, we have implemented the process shown in Fig. 3 in a hardware-in-the loop setting including a real PLC and a simulation of the process. We acknowledge that a thorough evaluation would require real data from an operational plant. However, getting such data is difficult due to the particularly sensitive context of ICS. Publicly available datasets[1] are often too simple, including few sensors/actuators. Studies which use real datasets are often limited to network trace files, while we require the availability of control logic for a comprehensive analysis. Yet, we believe that this evaluation can shed some light on the advantages and limitations of the proposed solution.

5.1 Process Description

The process [13] in Fig. 3 represents a typical sequential system. The goal is to produce a mixture of products following a certain recipe. The process involves two stages. In the first stage, two weighted products are introduced successively in the tank *T1* via the valves *vp1* and *vp2*. The required weights are indicated by sensors *p1* and *p2*. A mixer actuated by motor *m1* performs the primary mixing. After 50 s, and if *TP* is empty as indicated by the sensor *tpvid*, the mixture can be cleared out from *T1* through the valve *vt1*. In a second stage, a

[1] https://sites.google.com/a/uah.edu/tommy-morris-uah/ics-data-sets.

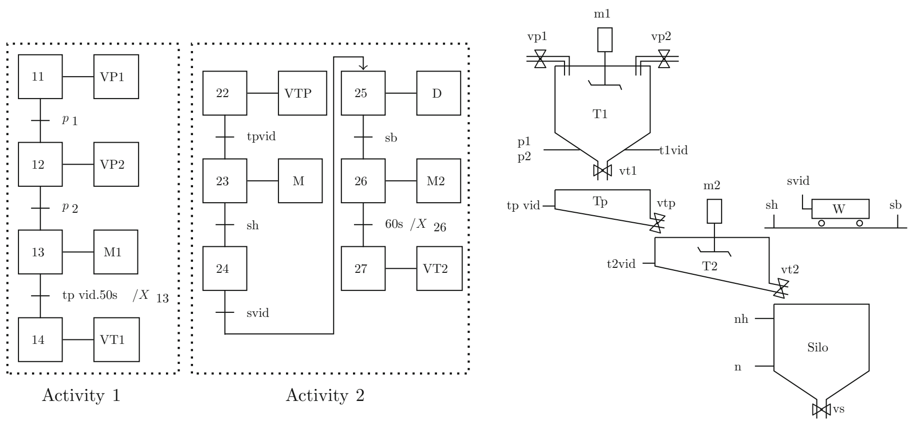

Fig. 3. Example of a sequential process and its control logic expressed using SFC [13]

product carried by the wagon *W* is added to the primary mixture. Sensors *sb* and *sh* indicate the position of the wagon. Actuators *m* and *d* (not shown in the figure) are responsible for the wagon's movement. A mixer actuated by motor *m2* performs the secondary mixing, which lasts for 60 s. Finally, the end product is drained to the *silo* through valve *vt2*. The valve *vs* allows emptying the *silo*. We would like to keep the final product level in the *silo* between the levels indicated by *nb* and *nh*. When the level reaches *nb*, a new production cycle is started until the level reaches *nh*. Figure 3 shows part of the SFC implementing the control logic for this process. In total, the process contains 20 actuators and sensors.

Activity decomposition. Following the process description, we can identify two main activities as shown in Fig. 3. We also use a default activity to mark all the coordinating steps which are outside these activities.

5.2 Experimental Setup

We evaluate our approach in a hardware-in-the-loop setting. The process is simulated in OpenModelica[2] while the control logic is implemented in a Schneider M580 PLC. A Human-Machine Interface (HMI) allows monitoring the process status and send commands. The HMI-PLC communication relies on the Modbus protocol. The monitors, the specification miner, and the activity recognizer are implemented in C++. To synthesize the monitors from LTL formulae corresponding to patterns, we use the Spot library[3]. Filtering rules are implemented in Prolog and take as input the predicates resulting from the mining phase.

Attacks. We perform a total of 15 process-aware sequence attacks during the simulation to test our solution. The attacks are carried by sending malicious

[2] https://www.openmodelica.org.
[3] https://spot.lrde.epita.fr.

commands to the PLC. We also define a number of manipulations which the operators are allowed to perform. For instance, operators can manipulate the valve *vp1* only before the weight *p1* is reached. Moreover, some actions are allowed without any restrictions such as the manipulation of *vs*. More importantly, not all of these behaviors appear in the attack-free traces used in the inference stage. This allows us to evaluate our solution with respect to false positives. The attacks involve malicious ordering of commands such as simultaneous opening of *vp1* and *vp2*, or opening *vt1* before the end of the first mixing phase. Table 1 summarizes some allowed behavior and attacks performed.

Data collection. Data is collected at two levels: (i) at the level of the HMI-PLC channel as Modbus network traces (pcap files), and (ii) at the level of the process simulation which produces a timestamped log of the values taken by the sensors and actuators throughout the simulation. We collect two separate datasets: (i) a legitimate dataset in which the process runs for 20 min without any attacks but with manual intervention of a human operator who performs actions within the allowed behavior, and (ii) a dataset spanning 40 min with interventions of a human operator and containing process-aware sequence attacks. The parameters of the simulation, such as the flow rates, are chosen so that the process completes several times the various stages during the recording window. All our tests are run offline using the recorded datasets.

Table 1. Examples of allowed behavior and sequence attacks performed on the process

Allowed behavior	Performed attacks
• Manipulating *vp1* before *p1* is reached	• Manipulating *vp1* after *p1* is reached
• Manipulating *vp2* after *p1* is reached and before *p2* is reached	• Manipulating *vp2* before *p1* is reached or after *m1* is started
• Manipulating *vt1* after *m1* is stopped	• Opening *vt1* before *m1* is stopped

5.3 Results

Process specification mining. We apply our proposed specification mining algorithm on traces per activity. Inference is performed on an Intel Dual Core i5 2.4 GHz machine with 4 GB of RAM running Linux kernel 4.4.5. We evaluate the mining algorithm in terms of 3 measures: (i) the monitors overhead, (ii) the runtime efficiency, and (iii) the number of mined properties.

Monitors overhead. As mentioned in Sect. 3.2, the monitors are derived from Büchi automata which can lead to a double exponential space blow-up with respect to the formula's size [2]. The monitors we generate do not represent pathological cases. All the monitors we synthesized have a size less or equal than 25 states. This is in fact another motivation for using patterns: we can control the patterns in terms of monitorability and size of the associated monitor. Moreover, only the mapping differs between instantiations of the same pattern. This reduces the memory-overhead required for the mining task.

Runtime efficiency. We measured the runtime efficiency of our specification miner for both activities. Our proposed algorithm spends on average 45 s for the first activity, and 55 s for the second activity. This is reasonable as mining is performed once on the training traces. We notice however that our solution performs worse on the second activity compared to the first one. This is mainly due to the presence, in the second activity, of more sensor and actuator variables. Another remark is that the algorithm's performance deteriorates when faced with unstructured execution traces such as randomly generated traces. However, this does not apply to sequential control system as they follow specific control logics.

Number of properties. Out of 407820 possible instantiations, the mining algorithm returns 7206 properties for the first activity, and 16269 properties for the second activity. We also apply our filtering rules to the mined properties. The filtering rules take into account: (i) the logical relationships identified in Sect. 4.3, (ii) the actual sensors and actuators involved in each activity. With regards to the second set of rules, an interesting feature of the mined properties is that their scopes involve sensors/actuators which are specific to the activity in question. This is due to the falsifiability policy we impose which naturally restricts the scopes. The filtering results in 719 properties for the first activity, and 1908 properties for the second activity.

Comparison with Texada. We also experimented with Texada [18], an efficient general LTL specification miner. We mine the patterns using its map mapper. It is worth noting that although Texada can omit vacuous properties, the runtime overhead becomes significant (over 10 min for both activities using the linear mapper). Texada's map miner spends little over 1 min for both activities. In contrast, the number of properties returned by Texada is an order of magnitude bigger. When comparing the mined properties, in the cases where our notion of falsifiability matches that of Texada, the mined properties are similar.

Attack detection. We evaluate the detection capabilities of our solution by running the inferred monitors on the malicious execution traces. Table 2 reports some violations recorded and their interpretation. All 15 performed attacks were detected by the monitors. Their interpretation relies on two key elements: (i) the activity during which the violation is reported, and (ii) the pattern corresponding to the property violated. However, as expected and discussed in Sect. 4.1, we obtain some false positives. The main recorded case of false positives was relative to the manipulation of *vt1*. As the operator does not manually interfere with *vt1* during the learning phase, we infer properties such as $absence\_global(vt1^{\downarrow})$. This property holds in the absence of manipulations, since *vt1* is the last action performed in activity 1, and *t1vid* signals the end of the activity. Knowing that an operator is allowed to manipulate *vt1* at some point during the activity, this property is too restrictive. Note that we also mine properties such as $absence\_before(p2, vt1)$ which violations would correspond to an attack.

In addition to delivering high semantics in terms of alerts' understanding, we can also deactivate monitors which do not correspond to properties we want to ensure. For instance, the property $absence\_global(vt1^{\downarrow})$ which causes a false positive can be deactivated, as it clearly concerns a legitimate action. The easiness

Table 2. Examples of raised alerts and their corresponding interpretation

Alert	Type	Properties violated	Interpretation
Alert 1 (act. 1)	TP	$absence\_between(m1^{\uparrow}, p1^{\downarrow}, vp2^{\uparrow})$ $absence\_between(m1^{\uparrow}, p2^{\downarrow}, vp2^{\uparrow})$	*vp2* opened after starting *m1* (attack)
Alert 3 (act. 1)	FP	$absence\_global(vt1^{\downarrow})$ $absence\_after\_until(m1^{\downarrow}, p1^{\downarrow}, vt1^{\downarrow})$	*vt1* closed after *m1* is stopped (legitimate action)
Alert 5 (act. 2)	TP	$absence\_before(m2^{\downarrow}, vt2^{\uparrow})$	*vt2* opened before the end of the mixing task (attack)

* TP: True Positive, FP: False Positive

with which one can alter the learned behavior is due to the inference of multiple properties which individually concern a limited set of sensors/actuators. Note also that one can analyze a priori the inferred properties by performing queries over variables which might cause false positives. For instance, since the valve *vs* can be opened any time during the execution of the system, one can query the inferred properties to ensure that no property restricts the usage of *vs*.

One issue we encounter when running our monitors is the possibly consequent number of violations raised for each attack. In our experiments, some attacks can produce as much as 30 violations. While these violations do not represent false positives, their number can render their analysis arduous. Moreover, some properties are more pertinent. Further work is needed to handle this issue through a correlation stage which can summarize and prioritize pertinent violations.

6 Conclusion

In this paper, we presented an approach for the detection of process-aware sequence attacks in sequential control systems. We used runtime monitors to report violations of process specifications expressed as sets of safety temporal properties. We also developed a mining algorithm to alleviate the cost of writing specifications. The notion of activity within sequential systems was introduced to improve mining and attack detection. Finally, we evaluated our approach in a hardware-in-the-loop setting subject to process-aware attacks. The evaluation results show that we are able to detect such attacks while achieving a good understanding of false positives. Our main goal for future work is the addition of a correlation stage to deal with the important number of raised violations.

References

1. Common cyber security vulnerabilities in ICS. Technical report, U.S DHS (2011)
2. Bauer, A.: Monitorability of omega-regular languages. CoRR abs/1006.3638 (2010)

3. Bauer, N., Huuck, R., Lukoschus, B., Engell, S.: A unifying semantics for sequential function charts. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) Integration of Software Specification Techniques for Applications in Engineering. LNCS, vol. 3147, pp. 400–418. Springer, Heidelberg (2004). doi:10.1007/978-3-540-27863-4_22
4. Carcano, A., Coletta, A., et al.: A multidimensional critical state analysis for detecting intrusions in SCADA systems. IEEE Trans. Ind. Inf. **7**(2), 179–186 (2011)
5. Cárdenas, A., Amin, S., et al.: Challenges for securing cyber physical systems. In: Workshop on Future Directions in Cyber-physical Systems Security, July 2009
6. Caselli, M., Zambon, E., Kargl, F.: Sequence-aware intrusion detection in industrial control systems. In: Proceedings of the 1st ACM Workshop CPSS, pp. 13–24 (2015)
7. Cheung, S., Skinner, K.: Using model-based intrusion detection for SCADA networks. In: Proceedings of SCADA Security Scientific Symposium, pp. 127–134 (2007)
8. d'Amorim, M., Roşu, G.: Efficient monitoring of ω-languages. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 364–378. Springer, Heidelberg (2005). doi:10.1007/11513988_36
9. De Giacomo, G., Masellis, R.D., Montali, M.: Reasoning on LTL on finite traces: insensitivity to infiniteness. In: Proceedings of AAAI 2014, pp. 1027–1033 (2014)
10. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of ICSE (1999)
11. Dzung, D., Naedele, M., Von Hoff, T.P., Crevatin, M.: Security for industrial communication systems. Proc. IEEE **93**, 1152–1177 (2005)
12. Falliere, N., Murchu, L.O., et al.: W32.Stuxnet Dossier-Symantec security response. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf. Accessed June 2016
13. Foulard, C., Flaus, J.M., Jacomino, M.: Automatique pour les classes préparatoires, 1st edn. Hermés-Lavoisier, Paris (1997)
14. Hadziosmanovic, D., Sommer, R., et al.: Through the eye of the PLC: towards semantic security monitoring for industrial control systems. In: Proceedings of ACSAC (2014)
15. ISO/IEC 29192 - Information technology - Security techniques - Lightweight cryptography. Standard, ISO, Geneva, Switzerland (2012)
16. John, K.H., Tiegelkamp, M.: IEC 61131–3: Programming Industrial Automation, 2nd edn. Springer, Heidelberg (2010)
17. Larsen, J.: Breakage-Black Hat (2008). https://www.blackhat.com/presentations/bh-dc-08/Larsen/Presentation/bh-dc-08-larsen.pdf. Accessed June 2016
18. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining. In: Proceedings fo ASE 2015, pp. 81–92 (2015)
19. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebraic Program. **78**(5), 293–303 (2009)
20. Li, W., Forin, A., Seshia, S.A.: Scalable specification mining for verification and diagnosis. In: 47th ACM/IEEE DAC, pp. 755–760 (2010)
21. Lin, H., Slagell, A., Di Martino, C., et al.: Adapting bro into SCADA: building a specification-based intrusion detection system for the DNP3 protocol. In: Proceedings of CSIIRW 2013, pp. 1–4 (2013)
22. Mitchell, R., Chen, I.R.: Behavior rule specification-based intrusion detection for safety critical medical cyber physical systems. IEEE Trans Depend. Sec. Comp. **12**(1), 16–30 (2014)
23. Pnueli, A.: The temporal logic of programs. In: Proceedings of SFCS 1977, pp. 46–57. IEEE Computer Society, Washington, DC (1977)

24. Puaun, D.O., Chechik, M.: On closure under stuttering. FAC **14**, 342–368 (2003)
25. Schumann, J., Moosbrugger, P., Rozier, K.Y.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 233–249. Springer, Heidelberg (2015). doi:10.1007/978-3-319-23820-3_15
26. Sommer, R., Paxson, V.: Outside the closed world: on using machine learning for network intrusion detection. In: 2010 IEEE S&P, pp. 305–316 (2010)
27. Stouffer, K., Falco, J., Scarfone, K.: Spp. 800–82 Rev 2. Guide to Industrial Control Systems (ICS) Security. NIST (2015)
28. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Banff Higher Order Workshop 1995 (1996)
29. Yoon, M.k., Ciocarlie, G.F.: Communication pattern monitoring: improving the utility of anomaly detection for industrial control systems. In: SENT (2014)
30. Zimmer, C., Bhat, B., et al.: Time-based intrusion detection in cyber-physical systems. In: Proceedings of First ACM/IEEE International Conference on CPS, pp. 109–118 (2010)

http://www.springer.com/978-3-319-47559-2

Secure IT Systems
21st Nordic Conference, NordSec 2016, Oulu, Finland,
November 2-4, 2016. Proceedings
Brumley, B.B.; Röning, J. (Eds.)
2016, X, 269 p. 43 illus., Softcover
ISBN: 978-3-319-47559-2